論文 / 著書情報
Article / Book Information

題目(和文)	共有メモリ型及び分散メモリ型並列プロセッサに対するコンパイル技術の研究
Title(English)	
著者(和文)	佐藤真琴
Author(English)	Makoto Satoh
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第8077号, 授与年月日:2010年3月26日, 学位の種別:課程博士, 審査員:前島 英雄
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第8077号, Conferred date:2010/3/26, Degree Type:Course doctor, Examiner:
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

東京工業大学

学位論文

共有メモリ型及び分散メモリ型並列プロセッサに対するコンパイル技術の研究

大学院総合理工学研究科

物理情報システム専攻

佐藤 真琴

平成22年3月

目 次

図 目 次

表 目 次

第1章　序論

1.1　本論文の背景と目的

今日，コンピュータは我々の生活に根付き，あらゆるところで使われている．例えば，家庭や職場にあるパソコンやサーバー，ゲームマシン，携帯通話機，クーラー等の家電機器，自動車，電車や飛行機の制御，製造工場におけるロボットの制御等，枚挙にいとまがない．

これらのコンピュータの中で，いわゆるスーパーコンピュータと言われている高性能計算機が存在する．これら計算機の主要な目的は計算機による実世界のシミュレーションである．また，コンピュータの歴史を紐解いて見ると，コンピュータのそもそもの開発目的はこの計算機シミュレーションの実現であった．それは弾道計算のシミュレーションであったり，気象計算であったりした．このようにして始まった計算機シミュレーションは現在，益々，応用範囲を広げ，重要さを増し，様々な形で我々の暮らしに役立っている．例えば，天気予報，創薬，航空機設計，自動車安全設計，ビル・橋梁設計などが挙げられる．

一方，計算機シミュレーションはその時代で使われる最高の計算機を使ってなされてきており，コンピュータの性能と二人三脚で発展してきた．例えば，少ないメモリ，少容量の記憶媒体，及び貧弱な計算能力の場合でも近似手法を駆使して計算を行ってきた．これら計算機資源の制約は現在，かなり緩和されてはいるが，計算能力が増すにつれて，より精密な計算精度を追求したり，従来，現実的な時間で計算が不可能であった新分野の計算が可能になったりしている．さらに，安全安心な社会の実現や産業競争力の強化には巨大で複雑な計算が必要とされている．このため，より高速な計算能力や巨大で複雑なアプリケーションプログラムへの対応が必要とされている．

より高速な計算機を実現するために，これまで半導体の微細加工技術の進展によりCPUの周波数を向上させる取り組みがなされてきた．これによって現在のCPUの周波数は2GHzから3GHzオーダーに達している．しかしながら，半導体の微細加工技術はほぼ限界に達しており，今後，CPUの周波数向上により計算機の処理性能を格段に向上させることは望めなくなってきた．そこで，多数のCPUを同時に実行させる並列処理が重要になってきた．現在，スーパーコンピュータはいうに及ばず，家庭用PCでもマルチコアが使われている．また，組込みプロセッサでもマルチコアが製品化されている[6, 4]．今後もマルチコアやクラスタなどの並列計算機は益々利用が広がると考えられる．

しかしながら，並列計算機があってもそれを効率良く使わなければ処理性能は向上しない．従って，プログラムを効率良く並列化するための，並列向きのプログラミング言語とそれを効率の良いコードに変換する並列化コンパイル技術が重要となる．

一方，アプリケーションプログラムは益々，巨大化・複雑化するため，トータルな開

発期間は益々増加し続けている．開発期間とは，プログラムの開発，テスト・デバッグ，実行，改良，テスト・デバッグ，··· といったサイクルを表す．巨大で複雑なプログラムではこれらの要素すべてに膨大な時間を要し，近い将来，限界に達すると思われる．このため，低レベルなプログラミング言語では生産性を高めることが困難であり，高レベルな並列プログラミング言語よるプログラミングの生産性向上が重要になる．

本論文の目的は，共有メモリ型並列プロセッサ及び分散メモリ型並列プロセッサを対象に，高レベルなプログラミング言語で書かれたプログラムを効率よく並列化するために並列化コンパイラ技術を高度化することである．

以下，共有メモリ型並列プロセッサ及び分散メモリ型並列プロセッサに対するプログラミング言語とそのコンパイラの課題について詳細に述べる．

1.2 共有メモリ型並列プロセッサに対するプログラミング上の課題と研究対象

共有メモリ型並列プロセッサに対するプログラミング言語としては Fortran 等の通常の逐次型言語および OpenMP[5] に代表される並列化指示文体系が科学技術計算分野では良く使われる．

OpenMP の長所は理解するのが容易であり，段階的に並列化できる点である．段階的な並列化とは，1つのループだけ並列化する，といったことができることである．これによって，プログラマは実行時間が長いループのみを並列化することができるので，少ない作業時間で大きな効果を手に入れることができる．OpenMP の短所は，OpenMP 指示文を元のプログラムに挿入するために，プログラマがそのプログラムを解析して並列化可能か否かを調べる必要があることである．この作業は一般に困難で，エラーを引き起こしやすい．特に，科学技術計算分野のプログラムではループが処理時間の90%以上を占めると言われており，そのループ中から関数やサブルーチン（以降，手続きと総称する）を呼出すという，プログラム中で最も長い実行時間を要する部分の解析は特に困難である．

Fortran 等の通常の逐次型言語の長所はプログラマが並列化作業を行う必要がない点である．一方，短所は並列化可否や並列化後の効率がコンパイラの解析能力や変換能力に依存するという点である．特に，ループ中から手続きを呼出すという，プログラム中で最も長い実行時間を要すると考えられる部分の並列化は従来の製品コンパイラでは行っていなかった．

そこで，本論文では，プログラム生産性と効果の点から，逐次型言語である Fortran において手続き呼出しを含むループを効率的に自動並列化することを研究対象として選んだ．

1.3 分散メモリ型並列プロセッサに対するプログラミング上の課題と研究対象

分散メモリ型並列プロセッサに対するプログラミング言語としては，Fortran 等の通常の逐次型言語，Message Passing Interface (MPI)[2, 3] に代表されるメッセージパッシング型言語，および High Performace Fortran (HPF)[1] に代表されるデータ並列型言語が挙げられる．

MPI に代表されるメッセージパッシング型言語は大抵，Single Program Multiple Data（SPMD）型と呼ばれる方法で記述する．SPMD とは，プロセッサ番号を変数として用いることにより，一つのプログラムの中で全てのプロセッサのプログラムを記述する方法である．したがって，MPI では，各プロセッサにデータを分割配置し，それら分割されたデータを各プロセッサで計算し，必要なら他のプロセッサからプロセッサ間通信によってデータを授受すると言ったプログラミングを行う．

MPI の長所は並列に関わる全ての処理をプログラマが記述するため，処理性能を向上させやすいことである．このため，科学技術計算分野で現在，最もよく使われている．一方，MPI の短所は上記のようなプリミティブな作業をプログラマが行う必要があるため，生産性が非常に低いことである．

HPF に代表されるデータ並列型言語とは，逐次プロセッサ向けの従来のプログラミング言語にデータを分散させる指示文や計算を分散させる指示文を追加した言語である．これらの指示文からプロセッサ間通信などを生成するのは並列化コンパイラの仕事になる．

HPF の長所は MPI に比べて生産性が高い点である．一方，HPF の短所はコンパイル技術が未成熟であり，複雑なプログラムに対して効率の良いプログラムを出力することがしばしば困難になる点である．

Fortran 等の通常の逐次型言語については，分散メモリ型並列プロセッサ向けに効率よくコンパイルすることが非常に困難で，まだ模索段階である．

そこで，本論文では，プログラム生産性の点から，データ並列型言語である HPF に対するコンパイル技術を高度化することを研究対象として選んだ．

1.4 本論文の概要

本論文は以下の各章から構成される．

第 2 章では，共有メモリ型並列プロセッサを対象として，Fortran 言語で書かれた逐次プログラムから高並列かつ高速なプログラムを出力する手続き間自動並列化コンパイラ技術，特に，メモリ消費量の低減を狙った変数プライベート化技術を提案すると共に，手続き間定数伝播の効果を評価する．

ループ並列化では，ループを単にプロセッサ台数分に分けるだけでなく，変数プライベート化やリダクション並列化等のプログラム変換を必要とする場合が非常に多い．これらのプログラム変換では，一般に，メモリ消費量が増加したり，プロセッサ間通信が増えたりする．メモリ消費量が増加する代表的な変換が変数プライベート化である．Fortran 言語にはコモン変数と呼ばれるグローバル変数があり，大規模なデータはコモン変数とすることが多い．並列化に伴ってコモン変数のプライベート化が必要に

なれば，メモリ消費量が増大する．さらに従来は複数のコモン変数が連続的に並んだコモンブロックをプライベート化の単位とし，本来，プライベート化が不要な変数までプライベート化するため，メモリ消費量は非常に大きかった．共有メモリ型並列プロセッサでは複数のプロセッサで限られたメモリを共有するため，大規模な科学技術計算ではメモリ消費量低減が課題となる．

一方，従来の手続き間自動並列化コンパイラでは，手続きインライン展開または手続きクローニングと呼ばれる処理と手続き間定数伝播と呼ばれる処理を組み合わせて，コンパイル中に計算できる式は極力計算することによってプログラムを簡単化し，コンパイラの解析精度を上げていた．しかしながら，手続きインライン展開または手続きクローニングと呼ばれる処理はコード量を爆発的に増加させ，その結果，実行時間が大幅に長くなる場合がある．このため，このような処理は解析のためにだけ使い，解析の結果，あるループが並列化可能とわかった場合にはこれらの処理を適用したコードを捨て，適用前のコードに並列化変換を適用することが多かった．このため，並列化は適用されたが，十分に効率的なコードが出力されたとは言えなかった．

そこで，本章ではコモン変数に対してメモリ消費量を低減するような変数プライベート化技術を提案する．また，手続きクローニングが適用されたコードを並列化することにより，適用前のコードを並列化するより，より効率の良いコードが出力できる場合があることを示す．

本提案手法では，コモンブロック中のコモン変数に対して，プライベート化が必要な変数，及び，その変数のみをコモンブロックから取り出してよいかをプログラム全体に渡って解析し，そうして良い場合に，取り出した変数のみから構成されるコモンブロックを作成した．これにより，プライベート化不要な変数に対して全プロセッサ数分のコピーを作る必要がなくなった結果，メモリ消費量を低減することができた．また，手続きクローニングと手続き間定数伝播を適用したコードに並列化を適用して得たオブジェクトコードを詳細に解析することにより，効率の良いコードが生成できた原因を示した．提案手法をコンパイラに実装し，ベンチマークプログラムで評価した結果を報告する．

第3章では，分散メモリ型並列プロセッサを対象として，データ並列言語の一種であるHPF言語が持つ高生産性な2レベルデータマッピング指示文をコンパイラで精密に解析する手法を提案する．

プログラムは通常，異なる参照パターンを持ついくつかのフェーズからなり，各フェーズで最適なデータ分散は異なる．このようなプログラムの例として，3つの次元毎に異なる最適データ分散を持つ，3次元FFTやADI法がある．3次元FFTは密度汎関数法等で使われ，様々な物理・化学計算で使われる他，医療画像処理分野における高速3次元画像処理で使われる．ADI法は偏微分方程式の差分法の一つであり，工学分野や金融工学におけるオプションの効率的な評価等に使われる．一つの配列に対してこのような各フェーズごとに異なるデータ分散を与えることをデータ再分散と呼ぶ．データ再分散は実アプリケーションプログラムで非常に頻繁に使われるが，従来のコンパイラはこれを十分にサポートしてこなかった．本章ではHPF言語における高生産なデータ再分散をサポートし，しかもできるだけ正確に解析することを目的とする．即ち，REALIGN指示文中のalign-targetとして通常の配列を許すプログラムも解析可能な新しいデータマッピング解析を提案する．本提案手法では，通常の配列であるalign-targetにある配列をREALIGNした時にその配列のデータ分散が変化することを表現できる

ような 4 つのデータフロー解析と上記 REALIGN により配列は最終的にどの template に ALIGN しているかを解析可能なデータマッピング解析を組み合わせ，プログラムの任意の位置で，どの配列がどの template にどのような ALIGN 方法で ALIGN しているかを解析した．また，データフロー解析のあいまい性を緩和するために，4 つのデータフロー解析のうちの 1 つを解析精度を高める目的で使用した．提案手法をコンパイラに実装し，ベンチマークプログラムで従来手法と比較評価した結果を報告する．

第 4 章では，分散メモリ型並列プロセッサを対象として，データ並列言語の一種である HPF 言語で書かれたプログラム中の計算を，複数のプロセッサに効率的に分散させるための条件とその時のコード生成手法を 3 つ提案する．HPF 言語には規則的データ分散と不規則データ分散がある．分子動力学で使われ，たんぱく質の構造解析など創薬分野で役立っている固有値計算は行列要素がランダムに分布しており，規則的データ分散で処理される．一方，不規則データ分散は疎行列に対して用いられることが多い．HPF に対するコンパイル技術は規則的データ分散に対してもまだ不十分なため，本章ではデータ分散を規則的なものに限定し，これに対して HPF コンパイラの処理の一つである計算分散を研究対象とした．

4.1 節では一般的な規則的データ分散に対して，各ループ繰返し単位で見たとき，ループ中の文が同一のプロセッサで処理される場合に従来より高速なテーブル参照コード，及び，最適な計算分散を行う条件とその時のコード生成手法を提案する．

HPF の特徴として 2 つの複雑なデータ分散指示文がある．1 つはデータ連続参照による処理高速性とプロセッサ間の負荷分散を調整可能な block-cyclic 分散である．2 つ目は ALIGN 指示文を用いた 2 レベルマッピングである．プログラマはこれらを用いることにより，高生産に並列プログラミングが可能であるが，並列化コンパイラがこれらから効率の良いコードを出力するのは非常に困難である．そこで，本章では HPF の規則的データ分散を定式化し，ある条件の下で最適な計算分散コードを生成することを目的とする．このため，図式を用いた一般的な枠組みを提案し，この枠組みを用いることにより，従来より高速なテーブル参照コードを導く．さらに，ある特別な条件が成り立つ場合にグローバルなインデックスとローカルなインデックスを対応させる写像が良い性質を満たすことから最適なコードが出力できることも示す．提案手法をベンチマークプログラムに人手で適用し，評価した結果を報告する．

4.2 節では規則的データ分散の中からブロックデータ分散を例に取り，各ループ繰返し単位で見たとき，ループ中の複数の文が必ずしも同一のプロセッサで処理されない場合に最適な計算分散を行う条件とその時のコード生成手法を提案する．以下の 2 つの場合に効率的な計算分散を行うことは困難であった．

(a) 同じループ制御変数が異なる次元に現れる場合，

(b) 複数のループ制御変数が一つの次元に現れる場合．

また，以下の 2 つの場合に計算分散によって生成されたループ中の if 文を削除するのは困難であった．

(c) 1 つのプロセッサに対して複数の文の計算分散後のループ範囲が異なる可能性がある場合，

(d) 1 つの文の計算分散後のループ範囲が複数のプロセッサで異なる可能性がある場合．

そこで，本章では上記計 4 つの場合に対して効率的な計算分散手法を与える．前者 2 つに対しては，文の実行プロセッサをみつけ，各次元の制約に関するディオファントス方程式を解くことによってその文の計算分散後のループ繰返し範囲を求め，多重ループを内側ループから外側ループに向かって変換することによって解決した．後者 2 つに対しては，同じ条件式を持つ if 文が同じループの属するように元のループをループ分割することにより，分割後のループでは if 文が不要になるようにした．提案手法をコンパイラに実装し，ベンチマークプログラムで評価した結果を報告する．

第 5 章では，本論文全体を総括し，今後の課題を述べる．

参考文献

[1] High Performance Fortran Forum: High Performance Fortran 2.0　公式マニュアル (1999).

[2] http://www.mpi-forum.org/

[3] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.2, High Performance Computing Center Stuttgart (HLRS), Sept. 2009.

[4] http://www.necel.com/magazine/ja/vol_0081/vol_0081_1.html

[5] http://www.openmp.org., OpenMP Fortran Application Program Interface Ver.1.0, Oct. 1997.

[6] http://japan.renesas.com/fmwk.jsp?cnt=multi_core_landing.jsp&fp=/products/mpumcu/multi_core/

第2章　手続き間並列化コンパイラ及びコモン変数プライベート化

2.1　概要

本章は，Fortran 言語で書かれた逐次プログラムを入力して高並列かつ高速なプログラムを出力する手続き間自動並列化コンパイラ WPP の特徴と性能評価，特に，メモリ消費量の低減を狙った変数プライベート化技術，及び，最適化の一種である手続き間定数伝播の評価に対する詳細な解析結果について述べる．

4 台から 128 台程度の比較的少数のプロセッサ台数で容易にプログラムを高速化可能な共有メモリ型並列計算機（SMP）は，科学技術計算分野において，比較的小規模な並列プログラムの実行計算機として，あるいは大規模なクラスターの最小構成単位としてよく使われている．1プロセッサ上で動作するプログラムを SMP 上で動作するプログラムに変換する作業は並列化と呼ばれ，その手段として OpenMP[10] に代表される並列化指示文体系が科学技術計算分野では良く使われる．ここで，並列化指示文とは C，C++，Fortran などのプログラミング言語で書かれた逐次プログラム中の並列化したい個所に挿入するコメント形式の文のことであり，例えば OpenMP コンパイラは OpenMP プログラムを入力してこの指示文の指示する内容に従って該当個所を並列コードに変換する．

OpenMP の指示文自体は理解するのが容易である．しかしながら，逐次プログラム中に OpenMP 指示文を挿入するためには，そのプログラムを解析して並列化可能か否かを調べることが必要であり，これは一般に困難で，エラーを引き起こしやすい作業である．実際，もし，ユーザが高並列で高速な OpenMP プログラムを得たい場合には，ユーザはしばしば対象となるプログラムを深く調査して，変数に複雑な並列属性を指定したり，並列スレッド間で同期を取るために同期指示文を挿入したりしなければならない．また，単に指示文を挿入するだけでは並列化できないプログラムに対しては，並列化を促進するために対象プログラムにプログラム変換を適用しなければならない．このようにユーザにとって逐次プログラムを並列化する作業は困難である．

一方，計算機の計算能力の向上に伴い，アプリケーションプログラムは益々複雑化，大規模化してきている．そのようなプログラムではループ中で実行されるプログラム行数も大規模であることが多く，そのため，ループ中に手続き呼出し（以降，Fortran 言語におけるサブルーチンと関数を総称して手続きと呼ぶ）を含むことも少なくない．このような手続き呼出しを含むループはプログラム中で長い実行時間を占めると思われるため，コンパイラはこのようなループを最適化することが重要である．

しかしながら，従来のコンパイラは手続きを単位としてコンパイルし，1つの手続きのコンパイル中は他の手続きの情報は参照できない．即ち，手続き呼出しを含むループをコンパイルしている時，コンパイラはその手続きの情報は持っていないため，例

えば，そのループが並列化可能か否かの解析をすることができなかった．OpenMP の指示文はこのような場合にループに対して並列化を指示できるため大変有用であるが，上記で述べたような複数手続きにまたがるループ並列化解析をユーザが行わないといけないため，一般に非常に負荷の高い作業となる．

このような問題を解決する一つの解として上記に述べた作業を自動的に行う自動並列化コンパイラ Whole Program Parallelizer (WPP) [1, 14, 7, 17, 16, 15, 12, 13] を開発してきた．即ち，WPP は逐次 Fortran プログラムを入力し，それに対して手続き間解析，手続き間自動並列化，及び，プログラム最適化変換を適用し，高並列かつ高速な OpenMP プログラムを出力する．ここで，手続き間並列化とは手続き呼出しを含むループを並列化することである．WPP はプログラム中で多くの実行時間を占めると思われるこのような手続き呼出しを含むループを並列化できるので，高い並列化率を達成することができる．以上に述べたように，WPP は SMP 上でプログラムを高速に実行することを第一の目的として開発されてきた．

一方，WPP は高速化のためにメモリを大量に消費する変換も行う場合がある．その代表的なものが変数プライベート化技術である．SMP では複数のプロセッサで限られたメモリを共有するため，大規模な科学技術計算ではメモリ消費量低減も重要である．そこで Fortran 言語において大規模データを置くことが多い COMMON ブロックに対して消費メモリ量を低減しながらプライベート化を行う技術を開発してきた．これによって，限られたメモリ量でもプログラムを並列化することができる．

また，WPP を使ってプログラムの評価を進めるにあたって，手続き間定数伝播と並列化を組み合わせることにより，スーパーリニアと呼ばれる，プロセッサ台数倍以上の処理速度向上が得られる場合や 1 プロセッサ上でもプログラムを高速化できることがあった．本章ではこの原因を探り，本技術が有効となるプログラムパターンの一つを明らかにする．これによって，どのような条件が成立したときに本技術を適用すべきかという一つの指針を得ることが出来き，本技術の適用のための前処理によってしばしばプログラム量が爆発的に増加するという現象を緩和しながら，プログラムを高速化することができる．

本章の残りは以下となる．2.2 節は WPP の構成と特徴を示す．2.3 節は手続き間定数伝播と手続きクローニングを説明する．2.4 節は選択的 COMMON 変数プライベート化を説明する．2.5 節は評価結果について述べる．2.6 節で関連研究について述べ，最後に 2.7 節でまとめを述べる．

2.2　WPP の構成

図 2.1 は WPP の構成を示す．WPP は Fortran77 で書かれた逐次プログラムを入力し，OpenMP 指示文を含む並列化されたソースプログラムまたは日立製スーパーコンピュータ SR8000 向け並列化オブジェクトコードを出力する．入力プログラムにはまず手続き間スカラー変数解析が適用され，各手続き又は各ループがその中で各スカラー変数をどのように READ/WRITE するかを解析する．次に，手続き間配列解析が適用され，各手続き又は各ループがその中で各配列のどの添字範囲をどのように READ/WRITE するかを解析する．これらの解析結果を受けて，手続き間並列化はプログラム中で最も並列化効果の高いループを見つけ出して，そのループを並列化する．

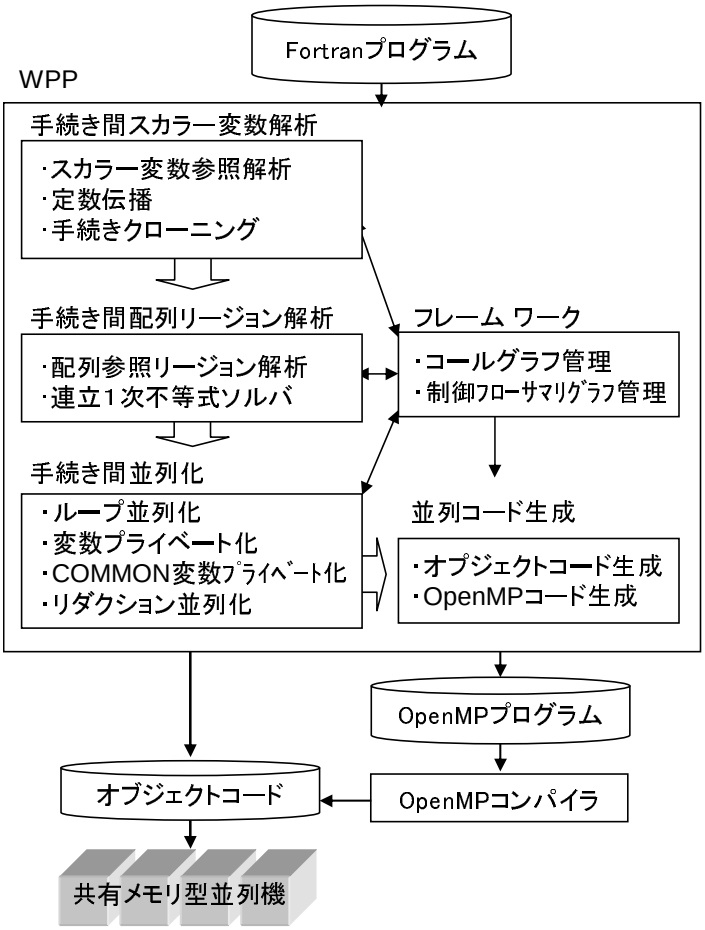


図 2.1: WPP の構成.

これら 3 つの処理部が扱うコールグラフや複数手続きにまたがるデータの管理は，フレームワークが行う．並列化した結果は，並列コード生成部において，OpenMP ソースプログラム又は SR8000 向けオブジェクトコードの形で出力され，後者は日立 SR8000 上で実行される．OpenMP ソースプログラムは，そのプログラムを実行したい共有メモリ型並列機向けの OpenMP コンパイラでコンパイルされてオブジェクトコードに変換され，実行される．

2.3 WPP の特徴

引き続き図 2.1 を用いて WPP の特徴を述べる．

2.3.1 手続き間スカラー変数解析

手続き間スカラー変数解析は各手続き又は各ループに対して 5 つの変数参照集合 MOD_V，$KILL_V$，USE_V，$EUSE_V$，及び，$LIVE_V$ を決定する．各々の集合は各手続き又は各ループがどのようにスカラー変数を READ するか WRITE するかを手続き又はループ全体でまとめたものである．表 2.1 にこれらの集合の意味を記述する．表

表 2.1: スカラー変数に対する参照集合.

種別	意味
MOD_V	値が定義される可能性のある変数の集合
$KILL_V$	値が定義されることが確実である変数の集合
USE_V	値が使用される可能性のある変数の集合
$EUSE_V$	1 つの手続きや 1 つのループ繰返しにおいて 値が定義される前に使用される可能性のある変数の集合
$LIVE_V$	ループや手続きの出口で LIVE である変数の集合

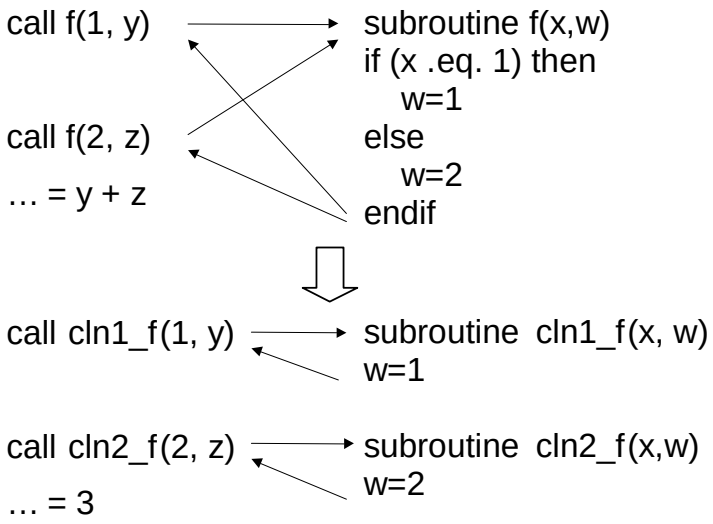


図 2.2: 手続きクローニングによって促進された手続き間定数伝播.

中,「定義」とは値を WRITE することを表し,「使用」とは値を READ することを表す．また,「LIVE」とは指定個所における変数の値がそれ以降のプログラムで「使用」される可能性があることを表す．これらの集合は WPP の並列化フェーズで，手続き呼出しを含むループが並列化可能か否かを決定するのに利用される．

手続き間スカラー変数解析は，さらに，プログラムに手続きクローニングや手続き間定数伝播を適用する．ここで，手続き間定数伝播とは，コンパイル時にある変数の値が定数になるとわかるとき，その情報を手続き呼出し先や手続き呼出し元に伝播させて，プログラム全体に渡ってコンパイル時に計算できるものは計算してしまうという処理である．一方，手続きクローニングとは，1 つの手続きに対して名前の異なるクローン（コピー）を作る処理である．この処理のみの適用は意味をなさないが，手続き間定数伝播と組み合わせることにより効果を持つ．例えば，ある手続きが複数の呼び出しを持ち，1 番目の引数が定数値 0 または 1 と取る場合があるとする．この場合，1 番目の引数は 2 つの値を持つのでこのままでは手続き間定数伝播は適用できない．そこで，定数値 0 を取る場合は元の手続きを，定数値 1 を取る場合はその手続きのクローン手続きを呼び出すとすると，各々の手続きにおいて 1 番目の引数は常に同じ値の定数となるので，手続き間定数伝播が適用できる．これらはプログラムを簡略化したり，プログラムが持つデータ依存を簡略化するため，後のフェーズの手続き間

表 2.2: 配列変数に対する参照集合.

種別	意味
MOD_A	値が定義される可能性のある配列添字の集合
$KILL_A$	値が定義されることが確実である配列添字の集合
USE_A	値が使用される可能性のある配列添字の集合
$EUSE_A$	1 つの手続きや 1 つのループ繰返しにおいて 値が定義される前に使用される可能性のある配列添字の集合
$LIVE_A$	ループや手続きの出口で LIVE である配列添字の集合

表 2.3: 各リージョンに対する繰返し種別.

種別	意味
CUR	i 番目のループ繰返しにおける配列参照リージョン
$PREV$	1 番目から $(i-1)$ 番目までのループ繰返しにおける 配列参照リージョンの和集合
ALL	全ループ繰返しにおける配列参照リージョンの和集合

配列解析における解析精度向上や様々なコンパイラ最適化を促進するのに役立つ.

図 2.2 は手続きクローニングと手続き間定数伝播を適用し，さらに，コンパイル時計算を行うことによって，プログラムの簡単化が実現できることを示した例である．手続き f に手続きクローニングを適用した結果，手続き cln1_f と cln2_f が生成され，各クローン手続きの引数 x の値を各々，1, 2 として手続き間定数伝播を適用した結果，サブルーチン f 中の if 文がコンパイル時に評価でき，プログラムが簡単化される．手続きクローニングを適用しないと手続き f の 1 番目の引数が複数の値を持つため，手続き間定数伝播が適用できないことがわかる．他のシステムでは解析時にのみ手続きクローニングを適用し，コード生成時にはクローンを生成しない．ところが WPP ではコード生成時にもクローニングを適用するので，最適化がより促進される．

2.3.2 手続き間配列解析

手続き間配列解析は各手続きと各ループに対して 5 つの配列添字参照集合 MOD_A, $KILL_A, USE_A, EUSE_A$，及び $LIVE_A$ を決定する．各々の集合は各配列に対する配列添字集合（以降，**配列参照リージョン**と呼ぶ）であり，各手続きがどのように配列要素を READ するか WRITE するかを手続き全体やループ全体でまとめたものである．表 2.2 にこれらの集合の意味を記述する．各集合の意味は各集合の名称中の ‘A’ を ‘V’ に変えた集合の，表 2.1 における説明中の「変数」を「配列添字」に置き換えたものになる.

各ループに対して，これらの配列参照リージョンはさらに 3 種類のリージョン，CUR, PREV, 及び，ALL から構成される．表 2.3 にこれらの意味を示す.

WPP では多次元配列の配列参照リージョンを連立 1 次不等式で表現し，これを数式処理ルーチンを用いて解くことにより，精密な解析を行う．例えば，ストライド付き

リージョンや三角形状リージョンを近似せず正確に表現することができ，また，実引数と仮引数の配列で次元の異なるものに対して正確なリージョン変換を施すことが出来る．

2.3.3 手続き間並列化

手続き間並列化は上記2つの解析で得られた手続き間解析情報を使って，さらに並列化独自の解析や変換を使って手続き呼出しを含むループを並列化する．まず，上記2つの解析情報のみを使う並列化手法としてDOALL並列化がある．DOALL並列化はデータ依存を全く持たないループを並列化する手法である．その判定条件は以下となる．即ち，あるループがDOALL並列化可能であることはループ中やそのループ内から呼び出される手続き内で参照される任意のスカラー変数wと配列xに対して以下の式が成立することである．

$$w \notin MOD_v \quad \cap \quad EUSE_v, \tag{2.1}$$

$$MOD_A(x, PREV) \quad \cap \quad EUSE_A(x, CUR) = \Phi, \tag{2.2}$$

$$MOD_A(x, PREV) \quad \cap \quad MOD_A(x, CUR) = \Phi. \tag{2.3}$$

一方，DOALL並列化だけでは並列化が適用できるループが限られるため，手続き間並列化解析では様々な並列化手法を適用している．以下にこれらの概要を述べる．

1. **手続き間変数プライベート化**
   本変換は，スカラー変数や配列が各ループ繰返しで定義された後に使用される，即ち，作業領域として使われることを手続き間にまたがって見つけ出し，それらの変数のコピーをスレッドごとに持たせることにより，スレッド間のデータ依存を解消する変換である．
2. **手続き間-初期値/終値/条件付終値-保障**
   これらの変換は，手続き間変数プライベート化が適用される場合において，特に以下の処理が必要なときに適用される：スレッドごとに持たせた変数のコピーに初期値設定が必要な時，ループの最終繰返しを実行するスレッドの変数コピーに設定される値を親スレッドに戻すことが必要な時，及び，必ずしもループの最終繰返しを実行するスレッドでなく，最後に変数コピーに値を設定するスレッドにおいて，その値を親スレッドに戻すことが必要な時．これらを適用することにより，元のプログラムの意味を変えることなく，手続き間変数プライベート化が適用できる．
3. **選択的 *COMMON* 変数プライベート化**
   本変換は，COMMONブロックからプライベート化可能な変数を抜き出し，元のCOMMONブロックをプライベート化変数からなるCOMMONブロックとそれ以外の変数から成るCOMMONブロックに分ける．本変換は並列化可能なループの数を増やすだけでなく，単純にすべてのCOMMON変数をプライベート化するのに比較してプログラムのメモリ使用量を大幅に削減する．
4. **手続き間リダクション並列化**
   本変換は，手続き間をまたがって，総和(SUM)，総積（PRODUCT），最大値/最

小値（MAX/MIN），及び，最大値位置/最小値位置（MAXLOC/MINLOC）というリダクションと呼ばれるパターンの計算を並列実行可能な形に変換する．

2.3.4 OpenMP プログラム生成

WPP における並列コード生成部は以下の特徴を持つ．

- ループのみを並列化する．
- fork/join 並列化モデルにおける fork 処理と join 処理を並列化ループとは独立に出力できる．そのため，2 つ以上の連続ループを 1 組の fork 処理と join 処理の中で実行することができる．

これによって，fork/join に伴うバリア同期オーバーヘッドを減らすことが出来る．OpenMP 仕様もこのような並列ループ (=for 指示文）と fork/join 処理（= parallel 指示文）を分割して指定できるので，WPP の中間語はそのままで効果的な OpenMP プログラムに変換することが可能である．さらに，以下の最適化は生成される OpenMP プログラムに反映される．

- 手続き内ループ構造変換
- 手続きクローニング
- 手続き間定数伝播

一方，WPP は以下が適用された並列化オブジェクトコードを出力することが可能だが，以下は OpenMP 1.0 の仕様に含まれないため，WPP が生成する OpenMP プログラムには適用されない．

- （手続き間）条件付き終値保障
- （手続き間）配列リダクション並列化
- （手続き間）選択的 COMMON 変数プライベート化
- DOACROSS 型ループ並列化

2.4 手続き間プライベート化

変数プライベート化は、そのままでは並列化できないループを並列化可能にするための変数（スカラ変数、配列）に対する変換機能の一つである．プログラムでは、高速化のため、同じ計算は最初の 1 回だけ行なって一時変数に保存し、後はその値を利用するのが一般的である．ところが、一時変数が現れるループをそのまま並列化して実行すると、各プロセッサが各自の計算結果をその一時変数に上書きするため、後で利用する際には必ずしも自プロセッサが書いた値とは等しくなく、結果が不正になり得る．そこで、一時変数に対しプロセッサごとに個別の領域を割り当てて，各プロセッ

real A(100)	**real** $A(100), l\_A(100, np)$
L1: **do** $j = 1, 100$	L1: **do** $j = 1, 100/np$
do $i = 1, 100$	**do** $i = 1, 100$
$A(i) = \cdots$	$l\_A(i, mype) = \cdots$
$\cdots = A(i)$	$\cdots = l\_A(i, mype)$
enddo; **enddo**	**enddo**; **enddo**
(a) 適用前	(b) 適用後

図 2.3: ローカル変数に対するプライベート化の適用例.

サは元の一時変数でなく，この個別領域へ読み書きすることで一時変数への上書きを防ぎ，結果不正を避ける．これを変数のプライベート化と呼ぶ.

図 2.3 はプライベート化の適用例である．図 2.3 (a) のループ L1 を図 2.3 (b) では np 個のプロセッサで並列実行し，同時に，図 2.3 (a) における配列参照 $A(i)$ を図 2.3 (b) では配列参照 $l\_A(i, mype)$ とすることによって，np 個のプロセッサ $mype = 0, \cdots, np-1$ が個別領域に値を読み書きできるようにしている．別の実現方法としてプロセッサごとのローカルメモリやスタック等に、その領域を実行時に割り当てる方法 [20] がある．ローカル変数のプライベート化にはこの方法を用いている．

以下では，手続き間プライベート化技術本来の説明と選択的 COMMON 変数プライベート化技術に固有な説明を分けて説明する．

2.4.1　ローカル変数に対する手続き間プライベート化

ローカル変数に対するプライベート化は手続き内のローカル変数や手続きの仮引数に対して適用される．以下，手続き間並列化の枠組みにおける，その適用条件，アルゴリズム，適用例を説明する．

適用条件　ローカル変数のプライベート化はスカラ変数と配列とに分けて処理する．スカラ変数に対しては以下の条件を満たすものをプライベート化する．これは手続き内解析と同じである.

- 定義がある
- ループ運搬フロー依存はない.

配列に対しては以下の条件を満たすものをプライベート化する.

$$MOD_A(x, CUR) \neq \Phi, \tag{2.4}$$

$$MOD_A(x, PREV) \cap EUSE_A(x, CUR) = \Phi, \tag{2.5}$$

$$MOD_A(x, CUR) \cap USE_A(x, PREV) \neq \Phi$$
$$\vee \quad MOD_A(x, PREV) \cap MOD_A(x, CUR) \neq \Phi. \tag{2.6}$$

最初の 2 つはスカラ変数の条件を配列用に書き直したものである．最後はループ運搬逆依存またはループ運搬出力依存があることを示す.

アルゴリズム 各フェーズでは以下の処理を行う.

a）並列性解析フェーズ

- 変数が上記の条件を満たすか否か解析し、これらの条件を満たす変数にプライベート属性を付加する.

b) 並列化コード生成フェーズ

- 並列化が決定したループに対し、そのループ内に現れるプライベート変数の宣言を，プロセッサ番号（またはスレッド番号）を表わす次元を追加した高次元配列の宣言とし，プライベート変数への参照を，プロセッサ番号（またはスレッド番号）を表わす次元の添字を自プロセッサ番号とした高次元配列への参照に変換する.

例 再び，図 2.3 を用いて説明する．ループ L1 は配列 A を除いて並列化条件を満たしているとする．図 2.3 (a) においてループ L1 に対して以下が成立することは容易にわかる.

$$MOD_A(A, CUR) = \quad MOD_A(A, PREV) = \quad USE_A(A, PREV) = [1:100],$$
$$EUSE_A(A, CUR) = \qquad\qquad \Phi.$$

よって、A は配列に対するプライベート化適用条件を満たし、プライベート化可能である．したがって、WPP は配列 A にプロセッサ番号を表わす次元を追加した配列 l_A, l_B を作成し、並列ループ内の配列 A への参照を配列 l_A への参照に置き換える．図 2.3 (b) はこの結果を表したものである．ここで，np はプロセッサ数、$mype$ はプログラムを実行するプロセッサ番号を表わす.

2.4.2 COMMON 変数のプライベート化

課題の説明 COMMON 変数とは Fortran 言語におけるグローバル変数のことであり，以下の性質を持つ.

- COMMON 変数はいくつかをまとめて COMMON ブロックとし，COMMON ブロックに名前を付けることができる.
- COMMON ブロックにまとめられた変数はその並び順に連続領域に割付けられる.
- 異なる手続きにある同じ名前を持つ COMMON ブロックの先頭アドレスは同じ.
- 異なる手続きにある同じ名前を持つ COMMON ブロックにおいて，変数名，変数の宣言，変数の並び順，COMMON ブロック全体サイズは異なってよい.

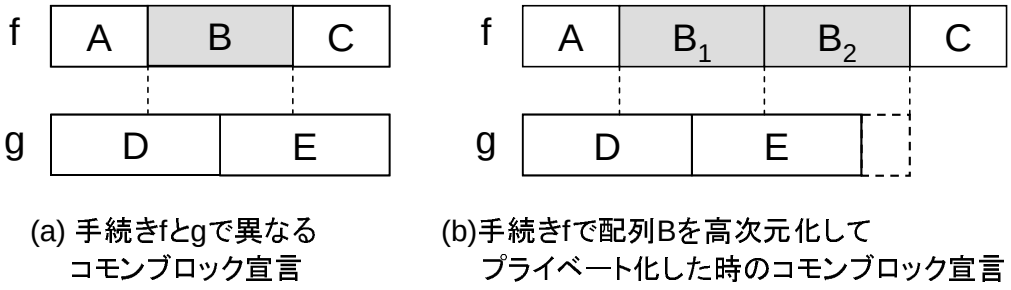


図 2.4: COMMON 変数プライベート化の課題.

例えば，1 つのプログラム中の関数 f, g, h に存在する名前 X を持つ COMMON ブロックとして以下のようなものが許される．

```
f:    common /X/ A(50), B(100), C(50)
g:    common /X/ D(100), E(100)
h:    common /X/ F(500)
```

COMMON 変数のプライベート化には以下の課題が存在する．

- 異なる手続きで COMMON ブロック中の変数の宣言が異なる場合、ある COMMON 変数にプライベート化を適用すると他手続きの COMMON 変数に影響がでる可能性がある．

図 2.4 は上記課題を説明した図である．図 2.4 (a) は関数 f と g において異なる宣言を持つ COMMON ブロック X において COMMON 変数が連続領域に割付けられている様子を示す．図 2.4 (b) は関数 f において配列 B(100) を高次元化する手法でプライベート化した様子を表す．図では B(1:100,1) を B_1，B(1:100,2) を B_2 と表している．但し，プロセッサ数は 2 と仮定する．この場合，B_2 の一部が配列 E と重なっているので，B_2 内の重なった部分に書き込みがあると配列 E に誤った値が書き込まれてしまう．配列 E は B_1 と一部が重なっていないといけないので，配列 E を他の部分へ移動させることも配列 B を他の部分へ移動させることもできない．

COMMON 変数には上記のようなことが起こり得るため、COMMON 変数のプライベート化は COMMON ブロック全体に対して適用するのが一般的である．例えば，OpenMP では個々の COMMON 変数に対してプライベート化は適用できず，COMMON ブロック全体に対して適用する．ところが、この方法は本来プライベート化不要な変数・配列にもプロセッサ台数分の領域を割り当てるため多くのメモリが必要になるという問題がある．

対策方法　上記で説明した課題を解決するために、以下の方針で対策を行う．

- ある COMMON 変数をプライベート化した場合，他の手続きの COMMON 変数に影響を与えるか否かを解析する．

- 影響を与えない場合，その変数を COMMON ブロックから取り出してプライベート化し，その変数のみを含む新規の COMMON ブロックを作成する．

さらにプライベート化の適用範囲を拡大するために以下の対策を取った.

- 影響可否の解析範囲を並列化したいループとその中から呼び出される手続きに限定する.
- ループ内に手続き呼出しを含まないループに対しては，そのループを含む手続き内のローカル変数としてプライベート化する.

上記 2 項目により異なる並列ループ間でプライベート化される COMMON 変数同士のアドレスに重なりがあっても各々別の領域でプライベート化できるのでプライベート化の適用範囲を拡大することが出来る．ここで，上記 2 項目に挙げられた COMMON 変数を区別して，前者をグローバルプライベート COMMON（GPC），後者をローカルプライベート COMMON（LPC）と呼ぶ.

このとき，変数の GPC 化適用条件は次のようになる.

- 影響可否解析範囲内において同じ COMMON ブロックの同じオフセットに必ずプライベート化候補の変数または配列の先頭が配置され，それらの変数同士は変数サイズまたは配列全体サイズが互いに等しい.

アルゴリズム 各フェーズでは以下の処理を行う.

a）並列性解析フェーズ

- プログラム全体を解析し、COMMON ブロック及びオフセットごとに COMMON 情報を作成する.
- COMMON 変数が出現するループごとに属性（GPC 候補または LPC 候補）を判定する.

b）並列化決定フェーズ

- 同オフセットの全 GPC 候補のサイズが一致するならこれらを GPC とする.
- 同オフセットの全 GPC 候補のサイズが不一致ならループは並列化しない[1].

c) 並列化コード生成フェーズ

- GPC に対して新規の COMMON ブロックと高次元化した COMMON 変数領域を確保し、並列ループ内の GPC への参照をこの領域への参照に置き換える.
- LPC は対して高次元化した手続きローカルな領域を確保して、並列ループ内の LPC への参照をこのローカル領域への参照に置き換える.

[1] 本来は COMMON ブロック全体をプライベート化すべきだが，そこまでの実装はしなかった.

```
      common /X/A(100),B(100)
L1:   do j = 1,100
        do i = 1,100
          call sub1(i)
          ... = A(i)
      enddo; enddo
      call sub2

      subroutine sub1(i)
      common /X/A(100),B(100)
        A(i) = ...

      subroutine sub2
      common /X/A(300)
L2:   do j = 1,100
        do i = 1,300
          A(i) = ...
          ... = A(i)
      enddo; enddo
```

(a) 適用前

```
      common /X/A(100),B(100)
      common /p_X/p_A(100,np)
L1:   do j = 1,100/np
        do i = 1,100
          call sub1(i)
          ... = p_A(i,mype)
      enddo; enddo
      call sub2

      subroutine sub1(i)
      common /X/A(100),B(100)
      common /p_X/p_A(100,np)
S2:   p_A(i,mype) = ...

      subroutine sub2
      real local_a(300)
L2:   do j = 1,100/np
        do i = 1,300
          local_a(i) = ...
          ... = local_a(i)
      enddo; enddo
```

(b) 適用後

図 2.5: COMMON 変数に対するプライベート化の適用例.

表 2.4: Origin2000 の性能諸元.

CPU	MIPS R10000
周波数	195 [MHz]
L1 / L2 キャッシュサイズ	32 [KB] / 4 [MB]
メモリサイズ/ノード	704 [MB]
全プロセッサ数/ノード	16

表 2.5: SR8000 の性能諸元.

CPU	PA-RISC ベース独自 CPU
周波数	250 [MHz]
L1 キャッシュサイズ	128 [KB]
メモリサイズ/ノード	8 [GB]
全プロセッサ数/ノード	8

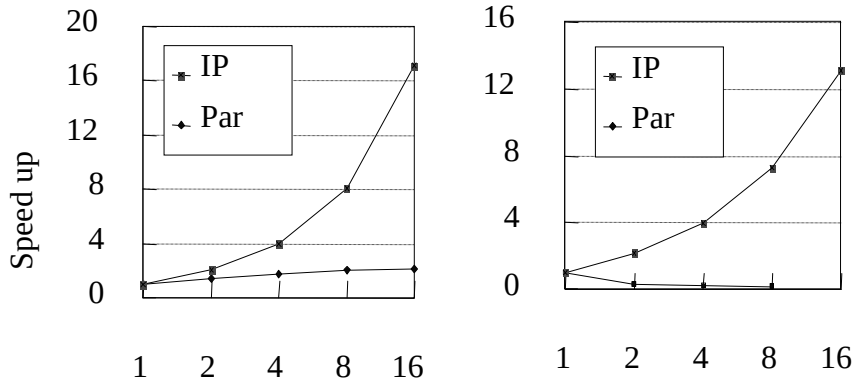
例 図 2.5 は、COMMON 変数に対するプライベート化の適用例である．ループ L1 は手続き間並列化可能、ループ L2 は手続き内並列化可能とする．この時、ループ L1 内の配列 A とループ L2 内の配列 A は同じ COMMON ブロックにあり、先頭からのオフセットが同じであるが、全体長が 400、800 となって一致しないので各々GPC、LPC となる．GPC に対して新規の COMMON ブロック p_com と、高次元化した COMMON 変数 $p\_A(100, np)$ を確保し、LPC に対しては、高次元化した COMMON 変数 $local\_A(100, np)$ を確保し、元の変数の参照をこれらの変数への参照に置き換える．

2.5 評価

SPECfp95 及び NPB2.3 benchmark suites から選んだいくつかのプログラムに対して，WPP を適用してそれらのプログラムの OpenMP プログラム (WPP-OpenMP) を出力し，これらの OpenMP プログラムを分散共有メモリ型並列計算機である SGI^{TM} Origin^{TM} 2000 上で実行した．Origin 2000 は，2CPU から成る共有メモリプロセッサ（SMP）を 1 ノードとする分散共有メモリ型マシンである．CPU は MIPS R10000(動作周波数 195 [MHz]) である．生成した OpenMP プログラムは MIPSpro Fortran でコンパイルした．コンパイルオプションは以下である．

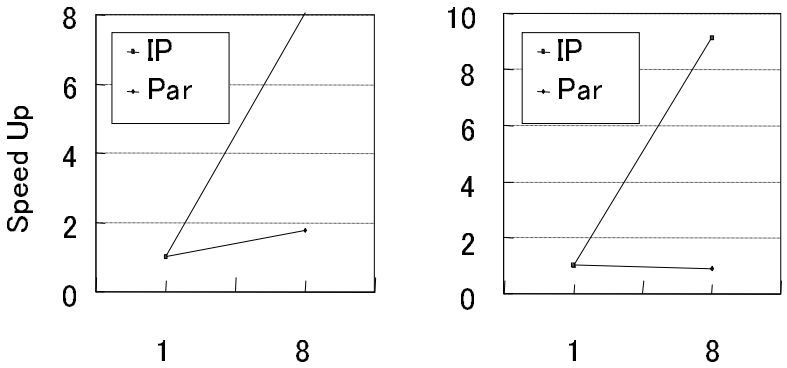
$$-mp - Ofast = ip27 - OPT : IEEE\_arithmetic = 3$$

次に，同じプログラムから並列化オブジェクトコード (WPP-native) を出力し，各 SMP ノードが 8 個の CPU から構成される SMP クラスターである日立 SR8000 上で実行した．表 2.4 は Origin2000 の性能諸元を，表 2.5 は SR8000 の性能諸元を示す．



(a) EP　　(b) turb3d

図 2.6: OpenMP プログラムの Origin2000 上での評価結果.



(a) EP　　(b) turb3d

図 2.7: オブジェクトコードの SR8000 上での評価結果.

2.5.1　WPP システムの処理性能評価

OpenMP コードの処理性能　図 2.6 は NPB 中のプログラム EP と SPECfp95 中のプログラム turb3d に対する WPP-OpenMP 版を Origin 2000 上で実行した結果を示す. Par 及び IP は各々，手続き内並列および手続き間並列化を適用した結果であることを示す．2 つのプログラムに対して IP 版はほぼリニアーな台数効果を得ていることがわかる.

EP: EP には内側に 2 つのループと手続き呼出しを含む主要ループが 1 つあり，実行時間のほとんどはこの主要ループによって占められている．WPP による手続き間並列化は，ループ内の手続き呼出しを解析することによってこの主要ループを並列化できるため，ほぼリニアーな台数効果を得ることが出来た.

一方，手続き内並列化では，主要ループは並列化できないものの，主要ループ中にある 1 つのループは並列化できた．しかし，台数効果は見込めず，どのプロセッサ台数を見ても処理性能はほぼ 2 倍である．これは上記並列ループの実効時間がプログラム全体の約半分の時間であるためと考えられる．これを確かめるため，Origin2000 上でプロファイルを取った．この結果，このループの実行比率はプログラム全体の 57%で

あることがわかり，上記予想が確認できた．

尚，現在の OpenMP1.0 仕様は配列リダクションを許していないため，EP 中に含まれる 10 要素からなる配列を 10 個のスカラー変数に書き換えた上でスカラー配列リダクションを適用した．この書き換えにより，WPP は手続き呼出しを含む，EP の主要ループの並列化が可能になった．

turb3d: turb3d には，6 つの手続き呼出しのみを含む 4 つの主要ループがあり，実行時間のほとんどはこれらのループによって占められている．これらの各ループは，ループ毎に同じ手続きを呼び出しているが，各呼出し毎に実引数の定数値が異なる．WPP による手続き間並列化では，同じ手続きでも実引数値が異なるごとに別のクローン手続きを作成することによって，これら実引数値を呼出し先手続きに伝播させ，if 文の削除，配列添字の単純化をコンパイル時に行なうことができた．更に，連立 1 次不等式を用いた手続き間配列解析により，ストライド参照の検出や異なる次元の引数配列に対応でき，4 つの主要ループ全てが並列化された．よって，手続き間並列化により，ほぼリニアーな台数効果を得ることができた．

一方，手続き内並列化では，主要ループは並列化できないものの，主要ループから呼び出される手続き内のいくつかのループは並列化できた．しかしながら，プロセッサ数が増加するにつれ，処理性能は低下した．これは，並列化できたループの粒度が小さく，プロセッサ数が増加するにつれ，並列化オーバーヘッドが増加したためと考えられる．

オブジェクト コード との処理性能比較 次に，WPP が出力するオブジェクトコードに対する評価結果を示す．この場合，OpenMP1.0 の仕様に縛られることがないので，配列リダクション並列化や DOACROSS パターン並列化等の様々な種類のループ並列化が適用できる．

図 2.7 は EP と turb3d に対する WPP-native 版を SR8000 上で実行した結果を示す．本評価で EP に対して何も書き換えを行わなかったが，EP には配列リダクション並列化が適用されて主要ループが並列化された．図 2.6 と図 2.7 は EP や turb3d に対して WPP が生成した OpenMP プログラムが WPP が生成するオブジェクトコードと同様な良い台数効果を与えたことを示す．特に turb3d に対しては，プロセッサ台数を超える性能向上効果，いわゆる，スーパーリニアな性能向上が見られた．

SPECfp95 に対する性能評価 図 2.8 は SPECfp95 に含まれるいくつかのプログラムの WPP-OpenMP 版を Origin 2000 上で評価した結果である．選択したプログラムは並列化によっては良い台数効果が得られるプログラムである．swim, hydro2d, mgrid, および turb3d に対する台数効果はほぼリニアーである．これらのプログラムのうち，turb3d のみに手続き間並列化が適用され，他のプログラムには手続き内並列化が適用された．

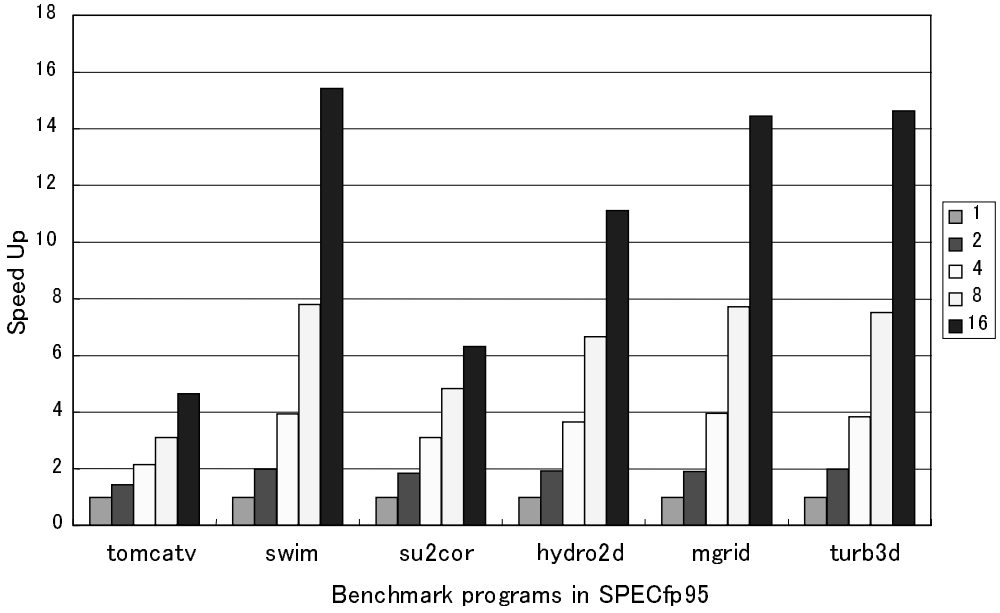


図 2.8: SPEC95 に対する Origin2000 上での評価結果.

表 2.6: SPECfp92 を用いた WPP の評価.

プログラム名	並列化率 [%]			サイズ [KB] / 率 [%]	GPC or LPC	IP PRIV	IP RED
	IP 無	IP 有	IPCOM				
doduc	33.7	41.7	46.0	5 / 10	GPC	○	
mdljdp2	6.5	6.5	79.8	12 / 99.9	GPC	○	○
nasa7	93.1	93.1	97.8	36 / 1.3	LPC		

2.5.2　COMMON 変数のプライベート化の評価

表 2.6 は，SPECfp92 ベンチマークプログラム集の中で手続き間並列化の効果が高いプログラムに対して WPP を適用し，SR2201 で評価した結果である．並列化率の欄の 1 列目は手続き内並列化の結果，2 列目は選択的 COMMON 変数プライベート化や選択的 COMMON 変数リダクション並列化（以降，これらを IPCOM と呼ぶ）を適用しない WPP の結果，3 列目は IPCOM を適用した WPP の結果を表す．4 列目は IPCOM が適用されたコモン変数の合計サイズ，及び，この値の IPCOM が適用されたコモンブロックの合計サイズにおける比率を表す．SPECfp92 は少し古いベンチマークプログラムなので，メモリ消費量が小さい点に注意すべきである．5 列目は IPCOM が適用されたコモン変数が GPC か LPC かを表す．6 列目と 7 列目は IPCOM が適用されたのは，手続き間プライベート化（IP PRIV) としてか，手続き間リダクション並列化（IP RED）としてかを表す.

表 2.6 より，IPCOM の効果はプログラムによってかなりの差があることがわかるが，メモリ消費量節約に一定の効果があることがわかった.

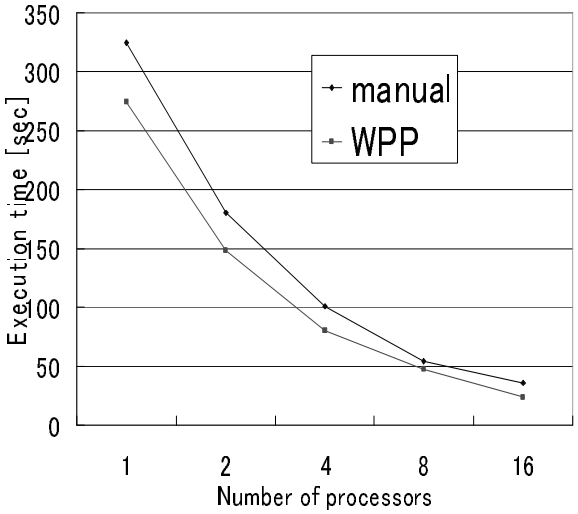


図 2.9: turb3d に対する OpenMP 人手挿入版と WPP 版との比較結果.

表 2.7: manual 版と WPP 版との実行時間の比較 [sec].

台数	1	2	4	8	16
manual	324.9	179.7	101.3	54.4	36.0
WPP-OpenMP	274.6	148.4	79.8	46.6	23.9
diff/manual	15.5%	17.4%	21.3%	14.4%	33.5%

2.5.3 turb3d の詳細評価

図 2.9 と表 2.7 は turb3d から WPP が自動生成した WPP 版 OpenMP プログラム（WPP-OpenMP）とその OpenMP プログラムにおける OpenMP 指示文を元の turb3d プログラムに挿入しただけの OpenMP 人手挿入版プログラム（manual）との Origin 2000 上での処理性能を比較した結果である．後者はオリジナルプログラムにループ並列化を指示する OpenMP 指示文を挿入したのみで，手続きクローニングや手続き間定数伝播は適用していない．一方，WPP-OpenMP にはループ並列化のほかに上記 2 つの最適化が適用されている．結果として，WPP-OpenMP は OpenMP 人手挿入版より

表 2.8: manual 版と WPP 版とのプロファイルの比較.

順位	手続き名	サンプル数		差	総和比
		manual	WPP		[%]
1	dcft	277351	232485	44866	86.5
2	dcopy	12423	12523	-100	90.3
3	lin	7363	7272	91	92.6
4	turb3d	5990	6032	-42	94.5
5	drcft	3812	3862	-50	95.7
総数	-	320787	274171	46616	100

も 1 から 16 プロセッサ上で 14.4%から 33.5%高速だった．

この原因を確かめるために，Origin 2000 の 1 プロセッサでプロファイルを取った．表 2.8 はその結果である．この表ではプロファイルによるサンプル数の多い手続きから順に 5 つだけ並べた．手続きの順位は manual でも WPP でも同一であった．但し，WPP 版については手続き dcft が 6 つのクローンに分かれているのでそれらのサンプル数の和を dcft の和とした．尚，サンプリングは 1[msec] 毎に実施される．表 2.8 における「総数」はプログラム全体のサンプル数を,「総和比」は順位 1 から順に manual 版のサンプル数を加えたものを manual 版の総数で割った比の値を表す.「総和比」からわかるように，5 つの手続きのサンプル数の合計はプログラム全体の約 96%に達するため，これらの手続きの挙動がプログラムの挙動をほぼ正確に反映していると考えられる．

表 2.8 より，手続き dcft とプログラム全体に対して，manual と WPP のサンプル数の差がほぼ同一で，手続き dcft のサンプル数がプログラム全体の約 9 割を占めていることから，WPP 版が高速であるのは手続き dcft が高速になったためと考えられる．また，手続き dcft のサンプル数の差は全体の 16%で，これはプロファイル非取得時の実行時間の差 15.5%とほぼ同じなので，プロファイルの結果はプロファイル非取得時の実行をほぼ反映していると言える．また，表 2.8 より，手続き dcft の WPP-OpenMP 版のサンプル数は manual 版より 16%だけ少なかったので，その分の命令数が減少したと考えられる．

dcft には外側ループ中に同じループ繰返し数の 2 つの内側ループを含む 2 重ループ構造が 1 つある．これらの内側ループについて，クローニングと手続き間定数伝播が適用された場合と適用されない場合の命令数は，1 番目のループが各々，11 命令と 14 命令，2 番目のループが各々，15 命令と 17 命令であった．両者の差は適用しない場合の命令数を基準とすると，1 番目のループが 21%，2 番目のループが 12%となり，平均は 16%となる．dcft にはこの他に，外側ループ外に 1 つ，2 つの内側ループの間に 1 つ，合計 2 つの手続き呼出しがある．これらの手続きは表 2.8 に現れないので，これらにおける実行時間は極めて短く，上記 2 つの内側ループの実効時間の差が manual と WPP との差になったと考えられる．

そこで，両者のオブジェクトコードを比較した結果，manual では同じ配列の異なる 2 つの参照に対して別のレジスタを使ってアドレス計算しているのに対して，WPP 適用コードでは 1 つのレジスタを使ってアドレス計算しており，この違いによってコード量の差が発生していることが確認できた．

具体的に図 2.10 を使って説明する．図 2.10 (a) は，dcft の外側ループとその第 1 の内側ループに対する manual コードである．第 2 の内側ループは第 1 の内側ループと類似しているため，省略した．7 行目と 8 行目にある X に対する 2 つの参照における添字 INDR と INDC は 2 行目から 6 行目の式から差が 1 であり，同じレジスタを使ってアドレス計算することが可能だとわかる．しかしながら，通常，コンパイラが差が定数であることを認識するためには 2 つの式が 1 次式でなければならないことが多い．1 次式でないパターンは少なく，また，どのようなパターンまで拡張すればよいかは対象とするプログラムによって様々なためである．本プログラムの場合，X の添字式は‘(I-1)*INC2X *2+(II-1)*2*INC1X+1’等となっており，‘変数 × 変数’を含む，かなり複雑な式であるため，2 式の差が 1 であることをコンパイラが認識できなかったと考えられる

```
 1: do I = 1,M
 2:   IBR = (I-1)*INC2X *2+1
 3:   IBC = (I-1)*INC2X *2+2
 4:   do II = 1,N
 5:     INDR = IBR+(II-1)*2*INC1X
 6:     INDC = IBC+(II-1)*2*INC1X
 7:     RA(II) = X(INDR)
 8:     RA(II+N) = X(INDC)
 9:   enddo
        ...
10:enddo
```

(a) manual におけるコード

```
 1: do I = 1,33
 2:   IBR = (I-1)*1 *2+1
 3:   IBC = (I-1)*1 *2+2
 4:   do II = 1,64
 5:     INDR = IBR+(II-1)*2*33
 6:     INDC = IBC+(II-1)*2*33
 7:     RA(II) = X(INDR)
 8:     RA(II+64) = X(INDC)
 9:   enddo
        ...
10:enddo
```

(b) WPP の適用結果

図 2.10: dcft に対する manual と WPP の適用結果の比較.

一方，図 2.10 (b) は，dcft の外側ループとその第 1 の内側ループに対する WPP 適用結果コードである．図 2.10 (a) と比較すると，手続き間定数伝播により，いくつかの変数が定数値となったことがわかる．本プログラムの場合，X の添字式は '(I-1)*1*2+(II-1)*2*33+1' 等となり 1 次式となっているため，2 つの異なる参照における添字式の差が 1 であると認識できたと考えられる．このようにして，手続き間定数伝播の結果，高速化されたと結論付けることができる．

WPP-OpenMP は 1 プロセッサ上でさえ OpenMP 人手挿入版よりも高速になったことは注目に値する．手続き間解析の応用として並列化を主眼に研究してきたが，手続き間解析は最適化にも効果があるという一つの例である．

2.6 関連研究

Illinois 大の Polaris コンパイラ及び Purdue 大の Polaris/OpenMP[11, 3] は解析フェーズにおいてプログラムにインライン展開を適用することによって広義の手続き間解析を行い，その解析結果から並列化 OpenMP プログラムを出力する．以下の機能も開発している．

- 手続きインライン展開
- 手続きクローニング

KAP^{TM} [9] も解析フェーズにおいてプログラムにインライン展開を適用することによって手続き間解析を行うが，その解析結果をインライン展開を適用する前のプログラムに反映させて並列化 OpenMP プログラムを出力する．

Stanford 大の SUIF コンパイラ [5, 6] は，狭義の手続き間解析による以下の解析を行なう．

- 手続き間スカラ変数参照解析
- 手続き間配列参照リージョン解析
- 手続きクローニング
- 手続き間定数伝播
- 手続き間並列化（プライベート化，リダクション認識）

一方，生成コードは ANL マクロを用いた並列 C ソースであり，OpenMP ソース生成機能は実現していない [6]．また，手続きクローニングは解析のためだけに行い，生成コードは手続きクローニング適用前のプログラムを並列化している [6].

IBM の XL Fortran v.7.1[2] では，以下のように積極的に手続き間解析を行なっている．しかし，手続き間自動並列化は未サポートと思われる．

- 手続きインライン展開
- 手続き間定数伝播
- 手続きクローニング
- 手続き間 Alias 解析
- コードマッピング（caller-callee の位置関係より）
- 大域変数マッピング（参照解析結果より）

SGI の MIPSpro Fortran 90 [18] も以下のように積極的に手続き間解析を行なっている．このコンパイラでは自動並列化の時，インライン展開は必須とあるので，狭義の手続き間並列化は未サポートと思われる．

- 手続きインライン展開
- 手続き間定数伝播
- 手続きクローニング
- 手続き間 Alias 解析
- デッド関数・デッド変数・デッド CALL 文の削除
- COMMON ブロック配列パディング（配列高次元化）

以上の 2 社は共に手続き間並列化をサポートしてない．このこととサポート項目より，手続き間解析の対象はスカラ変数のみであり，配列に対する参照リージョン解析は未サポートと予測できる．　以下の各社のコンパイラは手続きインライン展開のみサポートしている [19, 8, 4].

- Forte Fortran/HPC (Sun Microsystems Inc.)

表 2.9: 関連研究と WPP との比較.

コンパイル技術	WPP	SUIF	Polaris	KAP	IBM	SGI
手続き間解析（狭義）	○	○				
手続き間解析（広義）			○	○	○	○
手続きクローニング	○	△	○		○	○
配列リージョン解析	○	○				
手続き間並列化	○	○	○	○		○
OpenMP コード生成	○		○			

△：解析時のみ実施.

- Fortran Compiler for Linux (Intel)
- Visual Fortran (Compaq)
- Compaq Fortran for Tru64 UNIX/Linux Alpha/OpenVMS Alpha (Compaq)

表 2.9 は代表的なコンパイラの手続き間コンパイルシステムとしての比較をまとめたものである.

WPP はインライン展開を適用せずに入力プログラムを手続き間解析し，並列化された OpenMP プログラムを出力する自動並列化コンパイラである．プログラム解析時にのみインライン展開（KAP 等）や手続きクローニング（SUIF）を行い，コード生成時にはインライン展開や手続きクローニング前のプログラムを並列化するだけではプログラムを最適化するには不十分であると考える．即ち，turb3d プログラムの評価で見たように，高度に最適化されたコードを生成するためには，インライン展開や手続きクローニングのようなコードサイズの増加を許容する最適化が必要であり，元のプログラムに OpenMP 指示文を挿入するだけではこれは実現できないからである．WPP と SUIF コンパイラは多くの点で類似している．WPP の SUIF に対する特徴としては，OpenMP ソース生成，及び，選択的プライベート化が挙げられる．WPP は，Fortran 言語における COMMON 変数に対して，選択的プライベート化が適用できる唯一のコンパイラと考えられる.

2.7 まとめ

豊富な機能と高い並列化能力を持つ手続き間自動並列化コンパイラ WPP を開発した．WPP の特徴機能は，手続きクローニング，手続き間定数伝播，連立 1 次方程式を用いた配列参照リージョンの表現，選択的 COMMON 変数プライベート化，及び OpenMP プログラム生成である．SPECfp92 に選択的 COMMON 変数プライベート化を適用した結果，メモリ消費量削減に役立つことがわかった．また，SPECfp95 と NPB から選んだ 2 つのプログラムを SGI 社製 Origin 2000 及び日立製 SR8000 上で WPP を評価した結果，ほぼ台数に比例した性能を達成した．特に，SPECfp95 中の turb3d プログラムでは手続き間定数伝播の有無によって処理性能に差が認めれらた．また，手続き間定数伝播によって 1 プロセッサでも性能向上を確認した．これらの原因を探っ

た結果，プログラム中に含まれるパターン ‘変数 × 変数’ 中の一方の変数が手続き間定数伝播によって定数となる場合，コンパイラの最適化が働いて処理性能が向上することがわかった．

参考文献

[1] 青木雄一郎, 佐藤真琴, 飯塚孝好, 佐藤茂久, 菊池純男. 手続き間自動並列化コンパイラ WPP の試作 -実機性能評価-, 情報処理学会研究報告, 98-ARC-130, pp. 43-48, 1998.

[2] Bob Blainey. Performance Programming with IBM pSeries Compiler, SCICOMP4 (IBM SP Scientific Computing User Group) Oct. 2001.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, P. Tu, "Parallel Programming with Polaris", IEEE Computer, pp.78-81, Dec. 1996.

[4] http://www.compaq.com/fortran/docs/

[5] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. -W. Liao, E. Bugnion, M. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler", IEEE Computer, pp.84-89, Dec. 1996.

[6] M. Hall, S. Amarasinghe, B. Murphy, S. -W. Liao, and M. Lam, "Interprocedural Parallelization Analysis in SUIF ", ACM Transactions on Programming Languages and System, Vol. 27, No.4, pp.662-731, July 2005.

[7] 飯塚孝好, 佐藤茂久, 蓮見勝久, 菊池純男. 手続き間並列化コンパイラ WPP の試作 -現状と今後の課題-, 情報処理学会第 56 回 (平成 10 年前期) 全国大会講演論文集 (1), pp.280-281, 1998.

[8] http://developer.intel.com/software/products/compilers/f50/linux/finfo_ipo.htm

[9] Kuck and Associates, Inc., http://www.kai.com/

[10] http://www.openmp.org., OpenMP Fortran Application Program Interface Ver.1.0, Oct. 1997.

[11] Polaris/OpenMP, http://polaris.cs.uiuc.edu /

[12] http://www.rwcp.or.jp/trc/kikaku/activities/achievements/MP/hitachi/PDC-hitachi-e.html

[13] Research exhibits and brochures on the corner of RWCP (Real World Computing Partnership) at Supercomputing '99.

[14] 佐藤真琴, 青木雄一郎, 菊池純男. 手続き間並列化コンパイラ WPP の試作 -変数プライベート化技術-, 情報処理学会第 56 回 (平成 10 年前期) 全国大会講演論文集 (1), pp.284-285, 1998.

[15] 佐藤真琴、青木雄一郎、和田清美、飯塚孝好、菊池純男. 手続き間並列化コンパイラ WPP, 2000RWC シンポジウム報告集, pp. 67-70, 2000.

[16] Makoto Satoh, Yuichiro Aoki, Kiyomi Wada, Takayoshi Iitsuka, and Sumio Kikuchi. Interprocedural Parallelizing Compiler WPP and Analysis Information Visualization tool Aivi, Proc. of The Second European Workshop on OpenMP (EWOMP2000), pp. 38-47, 2000.

[17] 佐藤茂久, 蓮見勝久, 飯塚孝好, 菊池純男. 手続き間並列化コンパイラ WPP の試作 -定数伝播とクローニングの評価-, 情報処理学会第 56 回 (平成 10 年前期) 全国大会講演論文集 (1), pp.282-283, 1998.

[18] SGI's online-manual (IRIX 6.5 Man Pages: IPA(5)), http://www.sgi.com.

[19] http://www.sun.com/forte/developer/documentation/mr/READMEs/c.html#upd2a.

[20] P. Tu and D. Padua. "Automatic Array Privatization", Proc. of LCPC'93, pp. 500-521, LNCS Vol. 768, Springer, 1993.

第3章　HPFコンパイラにおける2レベルデータ分散の解析手法

3.1　概要

数千から数万の分散したCPUとメモリを高速なネットワークで接続する分散メモリ型並列計算機は今やスーパーコンピュータの主流の形態である．このような形態の並列計算機上でプログラムを効率的に実行するにはプログラムを分散メモリ型並列計算機向けに並列化することが必須である．このような並列化を実現するには以下に述べる3つのアプローチがある:

App 1: プログラマによる人手並列化,
App 2: プログラマとコンパイラの協調による半自動的な並列化，及び
App 3: コンパイラによる自動並列化.

App 1では，プログラマはMPIに代表されるメッセージパッシングライブラリを使って高度に効率的な並列プログラムを書くことが出来る．しかしながら，このプログラミング作業は時には数万を上回るプロセッサ間で発生するデータ通信を通信データ量や通信タイミングを調整しながら行うことになるため，非常な労力が必要になる．

App 2では，プログラマは High Performance Fortran (HPF) [12, 23]のようなデータ並列言語と呼ばれるプログラミング言語を使ってプログラムを書く．HPFのような言語は分散メモリ型並列計算機向け並列言語として十分なポテンシャルがある．例えば，HPFの拡張であるHPF/JA [14]を使って書かれたあるプログラムは地球シミュレータ上で14.9 TFlopsの処理性能を実現した[21]．しかしながら，一般に，HPFの複雑な言語仕様を使って書かれたプログラムから高度に効率的なオブジェクトコードをコンパイラが出力することは困難である．一つにはHPF言語が非常に複雑な言語となっており，その複雑さにコンパイル技術が十分追従できていないためである．また一つには，多くの研究者がHPFとその類似言語に対するコンパイル技術を長年研究してきた[2, 4, 11, 13, 14, 15, 16, 18, 20, 23, 24]が，比較的単純なプログラムに対してもHPF言語に対する最適化技術はまだまだ未成熟なためである．

App 3は，まだ研究段階であり，実利用に耐えられるレベルには到達していない．結局のところ，現在では，App 1が最もポピュラーである．しかしながら，もし，プログラマがApp 1にのみ頼り続けた場合，近い将来，その並列化作業の負担は限界に達するであろう．なぜなら，科学技術計算分野におけるアプリケーションは益々巨大化，複雑化しており，その並列化作業は益々困難になるためである．したがって, 今後, App 2がより重要になると考えられる．そのため，HPF及びその類似並列言語に対するコンパイル技術はさらに改良されなければならない．

HPFプログラムに対するコンパイル技術には様々な研究課題がある．本論文で選んだ研究課題は，データ再分散指示文を含むプログラムに対するデータマッピング解析

```
do i = L,U
  x(i) = y(i+1) + 1
enddo
```

(a) 例題プログラム

```
do i = L,U
  if (···) send (y(f(i)))
  if (···) recv (y(i+1))
  if (owner) x(i) = y(i+1) + 1
enddo
```

(b) 実行時解決コード

```
if (···) send (y(f(p) : f(q)))
if (···) recv (y(p : q))
do i = l,u
  x(i) = y(i+1) + 1
enddo
```

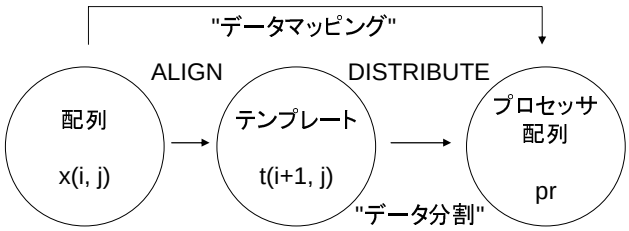
(c) ベクトル化通信コード

図 3.1: 正確なデータ分散情報の有る場合と無い場合の生成コード.

である．これはデータ再分散指示文を含むプログラム中の任意のプログラム点における任意の変数のデータマッピング状態を明らかにする解析技術である．この研究課題を選択した第一の理由はデータ再分散は実アプリケーションプログラムで非常に頻繁に使われるため[1]，HPFコンパイラがそれに対する効率的なコードを出力することは必要不可欠だからである．データ再分散を含む典型的なベンチマークプログラム例はNAS benchmark program suite [1]に含まれるSPやBTのようなAlternative Direction Integral (ADI)法を用いるプログラムや，同じsuiteに含まれるFTのような3次元Fast Fourier Transform (FFT)法を用いたプログラムである．ADI法は偏微分方程式の差分法の一つであり，工学分野や金融工学におけるオプションの効率的な評価等に使われる．3次元FFTは密度汎関数法等で使われ，様々な物理・化学計算で使われる他，医療画像処理分野における高速3次元画像処理で使われる．

この研究課題を選択した第二の理由は正確なデータマッピングを知ることはHPFコンパイラがプログラムを高度に最適化するときには本質的に重要だからである．このことを説明するために 図 3.1(a) のプログラムを考える．ここで，配列xとyはループ運搬データ依存をもたないと仮定する．もしコンパイラがこれらの配列の正確なデータマッピングを知らないとするとコンパイラは実行時解決コード (図 3.1(b))，または，動的コンパイルコードを出力することになる．前者は両配列のデータマッピングを比較し，もし，それらが異なればプロセッサ間データ通信を起動するコードである．このコードはループ繰返し毎に比較を実行し，もし異なれば配列要素毎の小さなデータサイズのプロセッサ間データ通信を起動するので，非常に効率が悪い．他方，後者は，コンパイラがコンパイル時にループに適用するのと同じシンボリック計算を実行時に適用するようなコードである．HPFプログラムに対する複雑なコンパイルプロセスは多大な実行時間を消費するため，このコードもまた効率の悪いものとなる．しかしな

[1]なぜなら，プログラムは通常，いくつかの異なる参照パターンからなり，各参照パターン間で最適なデータマッピングは異なるためである．



(a) 2 レベルデータマッピング

```
!HPF$ PROCESSORS  pr (np)
!HPF$ TEMPLATE t(M1, M2)
!HPF$ (RE)ALIGN x (i, j)  WITH  t (i+1, j)
                  alignee         align-target
!HPF$ (RE)DISTRIBUTE t (block, block) ONTO pr
                       distributee
```

(b) HPF のデータ（再）分散指示文

図 3.2: HPF のデータ分散.

がら，もしコンパイラが代入文の両辺における 2 つの配列のデータマッピングが同じあること，または，例え，異なったとしても両者のデータマッピングを正確に知っているとすれば，コンパイラはプロセッサ間データ通信のないコード，または，ループ直前で複数のデータを一括してデータ転送する効率的なベクトル化通信コードを生成できる． (図 3.1(c)) この時，図 3.1(c) に対する図 3.1(b) の実行時間比は数倍から数百倍程度になる．よって，正確なデータマッピングを知る解析は非常に重要である．

HPF におけるデータ再分散:

HPF [12, 23] は非常に良く知られたデータ並列言語である．プログラマはこの言語を使うことで，データをプロセッサにマッピングできる．即ち，データを分割して，各分割後データを各プロセッサの分散メモリに割付けることができる．図 3.2 (a) は，HPF で採用されている 2 レベルデータマッピングの概要を説明した図である．2 レベルデータマッピングは，ある配列 x を間接的にあるプロセッサアレイ pr にマッピングするのに使われる．即ち，まず，配列 x を**テンプレート** t にアラインし，次に t をデータ分割してプロセッサアレイ pr に割付ける．テンプレートとは，データマッピングを指示するためだけに使われる，実メモリに割付けられない仮想的な配列である．2 レベルデータマッピングの特徴は，直接的にデータ分割することに比べて様々なパターンのデータマッピングができることにある．

図 3.2 (a) におけるデータマッピングを実現する HPF 指示文を図 3.2 (b) に示す．第 1 行目は np 個の要素を持つプロセッサアレイ pr を宣言する文である．第 2 行目はテンプレート t の定義である．第 3 行目は i と j でパラメータ付けされた各配列要素 $x(i,j)$ がテンプレートの要素 $t(i+1,j)$ にアラインすること，即ち，両配列要素が同じプロセッサの分散メモリに割付けられること，を宣言する．第 3 行目において，x の位置に書かれる配列は *alignee* と呼ばれ，t の位置に書かれる配列は *align-target* と呼ばれる．第 4 行目はテンプレート t の全ての要素をデータ分割して論理プロセッサアレイ pr に割付ける．この指示文において，t の位置に書かれる配列は *distributee* と

呼ばれる．HPFではプログラムのデータ宣言部に第3行目，第4行目の指示文を書く時は各々，ALIGN,DISTRIBUTEと書き，プログラムの実行部にこれらを書く時は，各々，REALIGN，REDISTRIBUTEと書く．以降，両者を総称して(RE)ALIGN,(RE)DISTRIBUTEと書くこともある．[2]

本章の研究テーマに関連する事項として以下が重要である．

- distributeeとalign-targetはtemplateまたは通常の配列がなることができる．一方，
- distributeeはaligneeであってはならない．逆にaligneeはdistributeeであってはならない．

これらからわかることは，templateはdistributeeかつalign-targetとなるが，配列はdistributeeかつalign-target，aligneeのみ，aligneeかつalign-target，のいずれかになれるということである．最後の事柄が本章では重要である．最後の事柄に注意すると，ある配列x_0に対するデータマッピング指示文の一般形は以下となる．

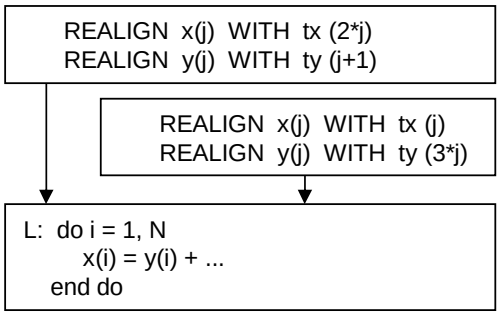
`!HPF$ DISTRIBUTE` $x_n(dist-kind)$ `ONTO PR`
`!HPF$ ALIGN` x_{n-1}`(i) WITH` $x_n(f_n(i))$
...
`!HPF$ ALIGN` x_1`(i) WITH` $x_2(f_2(i))$
`!HPF$ ALIGN` x_0`(i) WITH` $x_1(f_1(i))$

但し，x_nはテンプレートで，他のx_iは配列である．

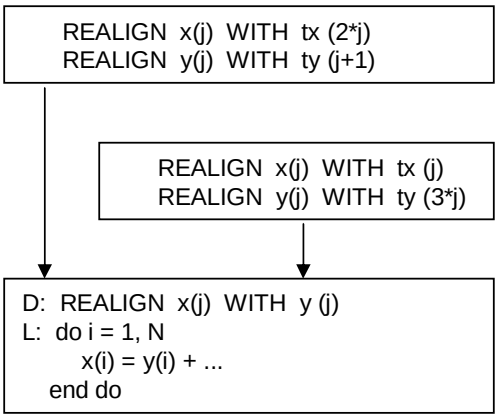
次に，データマッピング解析のターゲットであるデータ再分散指示文を考える．HPFには2種類のデータ再分散指示文 REDISTとREALIGNがある．REDISTはテンプレートのマッピングを変更する．さらに，このことにより，テンプレートにアラインする配列のマッピングをも変更する．(図3.2 (a)). 一方，REALIGNはaligneeを他のalign-targetにアラインさせたり，同じalign-targetに別の方法でアラインさせたりすることによって，aligneeのマッピングを変更する．(図3.2 (a)). REDISTはデータ分散を直接変更するので，データ再分散を指定するのに役立つことは明らかである．一方，REALIGNもaligneeのデータ分散を直接変更するが，これは特に，ある種のループの処理性能を向上させる時に便利である．例えば，図3.1(a)のループを再び考えよう．もし，プログラマがそのループ直前に‘`REALIGN x(i) WITH y(i+1)`’を挿入するなら，この指示文により，配列要素$x(i)$と$y(i+1)$は同じプロセッサにマッピングされることがわかるので，このループ中の代入文は同一のプロセッサで実行される．即ち，このループに対してはプロセッサ間データ通信は発生しない．

この場合のように，プログラマはREALIGN指示文中において，align-targetとして，テンプレートの他に通常の配列を指定することが出来る．align-targetとしてどちらを指定するかは，プログラマとコンパイラにとって大きな問題となる．即ち，align-targetとして通常の配列を指定することはコンパイラのデータマッピング解析を困難にする．一方，align-targetとしてテンプレートを指定することは，プログラマによるデータ再分散指定を困難にする．[3] 本章ではプログラマの負担を軽減する観点からこの課題を

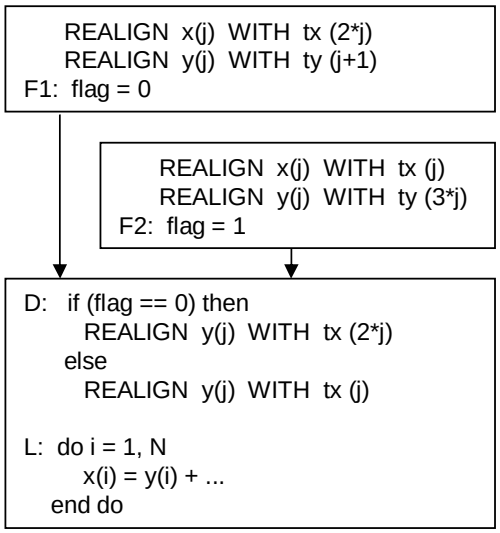
[2]本章では(RE)DISTRIBUTEをしばしば(RE)DISTと略す．
[3]詳しくは3.2節で述べる．



(a) 例題プログラムとフローグラフ



(b) align-targetが配列の場合



(c) align-targetがテンプレートの場合

図 3.3: align-target としてテンプレートまたは配列を用いたプログラム.

解決するために，align-target として通常の配列を許す REALIGN が含まれたプログラムに対する新しいデータマッピング解析を提案する.

本章の残りは以下のような構成である． 3.2 節は一般の HPF 指示文と制限された HPF 指示文に対して，プログラミングとコンパイラ解析の間のトレードオフについて議論する．3.3 節はデータフロー解析と分散記述子解析から構成されるデータ分散解析のアウトラインと解析の適用例を述べる．3.4 節はいくつかのベンチマークプログラムを用いて本データ分散解析を評価する．3.5 節で関連研究について述べ，3.6 節でまとめを述べる．付録では定理 1 を証明する.

3.2 課題の説明

本節では 3.1 節の最後で述べた課題について説明する．まず，第一に，REALIGN 指示文における align-target として template を用いるとプログラマビリティが悪化することを説明する.

図 3.3 (a) は例題プログラムとそのフローグラフを表す．この図において，2 つの基本ブロックがループ L を含む基本ブロックに合流しているため，配列 x と y に対して，

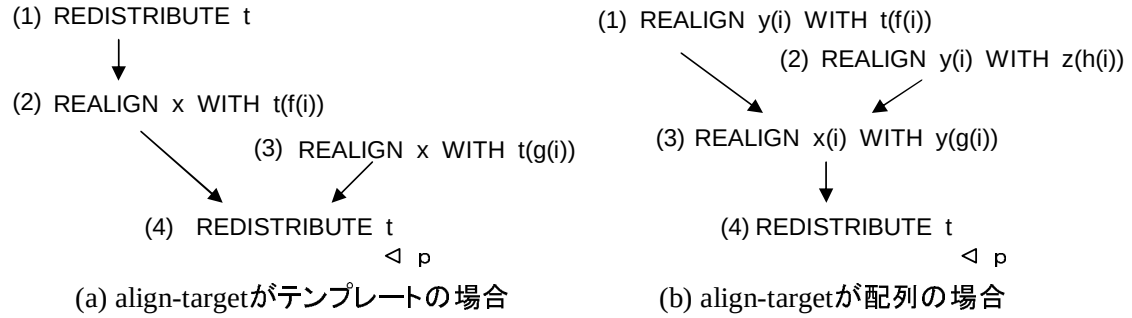


図 3.4: align-target としてテンプレートまたは配列を用いた再分散指示文.

各々，2つの異なるデータ分散がループ L 直前に到達していることがわかる．この場合，もし，ループ L 内において，コンパイラがプロセッサ間データ通信を生成しないようにしたいならば，ループ直前で配列 x を配列 y にアラインさせれば良い．これをコンパイラに指示するには以下の 2 つの方法がある．

方法 1: align-target が配列であるような REALIGN を用いる

方法 2: align-target が template であるような REALIGN を用いる

方法 1 の実現には図 3.3 (b) におけるように指示文 D を挿入するだけで良い．一方，方法 2 は，図 3.3 (c) におけるように指示文 D に始まる 5 行の文とその他に 2 つの文 F1 と F2 が必要である．これら 2 つの文はどのパスを通ってプログラムが実行されたかを知るのに用いられる．このように，align-target を template に制限することはプログラマビリティをかなり悪くする．さらに，このようなプログラムはプログラムを改造することによってループ L にさらに別のデータ分散が到達する場合，さらに別のフラグ変数を必要とするので，プログラムの保守性も同様に悪化する．

次に，第二に，REALIGN 指示文における align-target として一般の配列を用いるとコンパイラの解析が困難になることを説明する．

図 3.4 (a) は align-target が template であるような REALIGN 指示文を含むある例題プログラムに対するフローグラフである．この場合，データマッピング解析は非常に容易である．あるプログラム点における template t のデータマッピングはその点に到達する REDIST によって決定される．template に対するデータマッピングはこの指示文によってのみ変更されるからである．同様にして，あるプログラム点における alignee x のデータマッピングはその点に到達する REDIST と REALIGN によって決定される．例えば，指示文 (2) 直後のプログラム点における template t のデータマッピングは指示文 (1) によって決定されるデータマッピングと同じであり，指示文 (2) 直後のプログラム点における配列 x のデータマッピングはこれら 2 つの指示文によって決定される 2 レベルデータマッピングに等しい（図 3.2 参照). 同様にして，指示文 (4) 直後のプログラム点 p における配列 x の可能なデータマッピングの候補は 2 つの 2 レベルデータマッピング，指示文 (2) と (4) で決定されるもの，及び，指示文 (3) と (4) で決定されるもの，となる．

図 3.4 (b) は align-target が一般の配列であるような REALIGN を含むある例題プログラムのフローグラフである．この場合，データマッピング解析は非常に困難である．

例えば，指示文 (1), (3) により配列 x はプログラム点 p において template t にアラインするので，指示文 (4) 直後のプログラム点 p における配列 x の一つの可能なデータマッピングの候補は，指示文 (1), (3), 及び (4) によって決定される 2 レベルデータマッピングとなる．これは以下の指示文によるデータマッピングと同じになる．

`!HPF$ ALIGN` x`(i) WITH` $t(f(g(i)))$

ここで，template t の添字は 2 つの関数 $f(i)$ と $g(i)$ を合成したものである．この例で見たように，コンパイラは以下の 2 つを考慮しなければならない．

(a) ある点で template t にどのような変数がアラインするか？

(b) 配列 x は template t にどのようにアラインするか？

同様にして，プログラム点 p における配列 x のもう一つの別の可能なデータマッピングの候補は，指示文 (2), (3), 及び図にはない z に対する REALIGN と同じく図にはない z のテンプレートに対する REDIST によって決定される 2 レベルデータマッピングとなる．したがってコンパイラは以下の 2 つを考慮しなければならない．

(c) あるプログラム点で x はどのテンプレートにどのようにアラインするのか？

(d) どのようにテンプレートはデータ分散されるのか？

本研究の目的はプログラマに容易なプログラミング手法を提供することである．この目的を達成するためにその align-target が一般配列であるような REALIGN を含むプログラムに対するデータマッピング解析を提案する．

3.3 データマッピング解析

データマッピング解析はデータフロー解析とマッピング記述子解析から構成される．図 3.5 はデータマッピング解析の概略を表す．図 3.5 において Step 1 は 4 つの解析 (a) から (d) に対して各基本ブロックでのデータフロー集合を計算し，Step 2 は同じ 4 つのデータフロー集合を各データ再分散指示文まで伝播させる[4]．Step 3 は，各データ再分散指示文に対して，その指示文によってデータマッピングが変化させられる変数の集合，言い換えると，そのマッピング記述子（MD)が変化させられる変数の集合，を求め，変化後の MD を計算する．これはデータフロー解析 (a) から (c) の結果を使ってなされる．Step 4 は，各基本ブロックと各変数に対して，その中に含まれるデータ再分散指示文によってある変数の MD が変更される場合に，その指示文直後で基本ブロックを分割して分割後の各ブロックをその変数に対する**サブブロック**と呼び，各サブブロックにおけるその変数の MD を計算する．ここで，各変数に対して，上記再分散指示文直後のプログラム点と各基本ブロックの入口点を合わせたものを，その変数に対する**再分散点**と呼ぶ．この処理は基本的に Step 3 の結果をデータフローグラフ中で伝播させることによってなされるが，解析 (d) の結果を使うことによってより精密な情報が得られる．

[4]Step 2 は Step 1 と同じ解析をデータ再分散指示文に適用するだけなので，本節では Step 2 に対するこれ以上の説明は省略する．

入力: データ再分散指示文を含むプログラム.
出力: 変数に対する任意の参照点における，その変数の分散記述子（MD）.

Step 1. 以下の 4 つのデータフロー解析を実行する:

(a) テンプレート解析,

(b) 到達アラインメント解析,

(c) 到達マッピング解析,

(d) 直接到達マッピング解析.

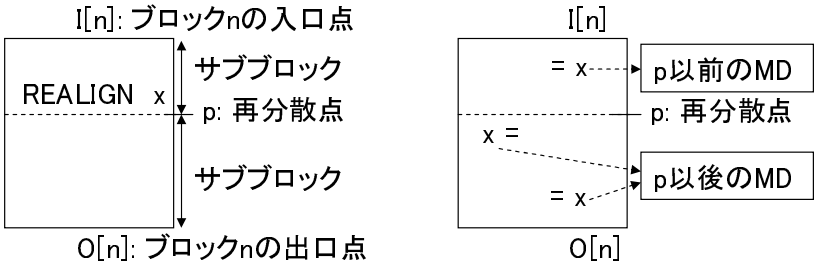
Step 2. Step 1 で得られたデータフロー集合を各データ再分散指示文直後の点に伝播させる

Step 3. データマッピング解析 (局所 MD 解析) 各データ再分散指示文に対してその指示文によって MD が変更されるような変数の集合を求め，それらの変数に対して変更後の MD を計算する.

Step 4. データマッピング解析 (大局 MD 解析) 各基本ブロックと各変数に対して，その中に含まれるデータ再分散指示文によってある変数の MD が変更される場合にその指示文直後で基本ブロックを分割して分割後の各ブロックをその変数に対するサブブロックとし，各サブブロックにおけるその変数の MD を計算する.

Step 5. (MD 結合) 任意の変数とその変数に対する各サブブロックにおいて，そのサブブロック内の任意の参照点にそのサブブロック内のその変数の MD を結びつける.

図 3.5: データマッピング解析の概要.



(a) 変数xに対する再分散点とサブブロック　(b)変数xに対するMD結合

図 3.6: 再分散点とサブブロック.

$$\textbf{E1:}\quad T_I(n) = \bigcup_{u\in pred(n)} (T_I(u)\cap T_P(u)) \cup T_D(u, T_I(u))$$

$$\textbf{E2:}\quad A_I(n) = \bigcup_{u\in pred(n)} (A_I(u)\cap A_P(u)) \cup A_D(u)$$

$$\textbf{E3:}\quad R_I(n) = \bigcup_{u\in pred(n)} (R_I(u)\cap R_P(u, T)) \cup R_D(u, T_I(u))$$

$$\textbf{E4:}\quad D_I(n) = \bigcup_{u\in pred(n)} (D_I(u)\cap D_P(u, T_I(u))) \cup D_D(u, T_I(u))$$

図 3.7: データフロー方程式.

最後に，Step 5 は任意の変数とその変数に対する各サブブロックにおいて，そのサブブロック内の任意の参照点にそのサブブロック内のその変数の MD を結びつける．この処理によって，コンパイラの最適化処理は各参照点からその変数の MD を得ることが出来る．

以降，Step 1 の詳細は 3.3.1 節で説明し， Steps 3 から Step 5 の詳細は 3.3.2 節と 3.3.3 節で説明する．

3.3.1 データフロー解析

本手法では図 3.7 に示された 4 つのデータフロー解析を行う．データフロー方程式 E1 から E4 は各々，テンプレート解析，到達アラインメント解析，到達マッピング解析，及び，直接到達マッピング解析に対するものである．

図 3.8 は各々のデータフロー方程式で用いるデータフロー集合を説明したものであり，図 3.9 はこれら 2 つの図及び本節で使われる記号の説明である．

テンプレート 解析:

任意の変数 x とブロック n に対して, テンプレート解析は $T_M(x, I[n])$,即ち，ブロック n の入口点 $I[n]$ において変数 x のテンプレートになる可能性のある変数又はテンプレートの集合を計算する．方程式 E1 は ‘`REALIGN x WITH y`’ を 変数 x の**定義**とみな

$T_I(n) = \{(x,t) \in VD | t \in T_{\mathrm{M}}(x, I[n])\}$

$T_D(n) = \{(x,t) \in VD | t \in T_{\mathrm{M}}(x, O[n])$ 中で変更される可能性がある $\}$

$T_P(n) = \{(x,*) \in VD | x$ のテンプレートは n 中で変更されない $\}$

$A_I(n) = \{(x,p) \in VP |$ 点 p 直前に '`(RE)ALIGN x`' があるが，p から $I[n]$ に至るパス上には点 p を除いて '`(RE)ALIGN x`' はない $\}$

$A_D(n) = \{(x,p) \in VP |$ ブロック n に含まれる点 p 直前に '`(RE)ALIGN x`' があるが，p と $O[n]$ の間のパス上には '`(RE)ALIGN x`' はない $\}$

$A_P(n) = \{(x,*) \in VP |$ ブロック n 内に '`(RE)ALIGN x`' はない $\}$

$R_I(n) = \{(x,p) \in VP |$ x の MD は p 直前のデータ（再）分散指示文によって変更される可能性があり，p 直後の MD は $I[n]$ に到達する $\}$

$R_D(n) = \{(x,p) \in VP |$ x の MD は n 内の p 直前のデータ（再）分散指示文によって変更される可能性があり，p 直後の MD は $O[n]$ に到達する $\}$

$R_P(n) = \{(x,p) \in VP |$ n の外側にある点 p における x の MD は n で無効化されない $\}$

$D_I(n) = \{(x,p) \in R_I(n) |$ p から $I[n]$ に至るパス上に,$(x,p') \in R_I(n)$ となるような p' は存在しない $\}$

$D_D(n) = \{(x,p) \in VP |$ n 内において p は x の MD を変更する可能性のある最後の再分散点である $\}$

$D_P(n) = \{(x,*) \in VP |$ x の MD は n で無効化されない $\}$

図 3.8: データフロー集合.

A	あるプログラム内で参照される alignee の集合
D	あるプログラム内で参照される distributee の集合
V	あるプログラム内で参照される変数の集合
P	あるプログラムに含まれる再分散点の集合
*	V（または P）に含まれる全ての要素からなる集合
VP	$V \times P$（V と P の直積）
VD	$V \times D$（V と D の直積）
$I[n]$	基本ブロック n の入口点
$O[n]$	基本ブロック n の出口点
$\mathcal{R}[n]$	ブロック n にある REALIGN の alignee，または， ある REDISTRIBUTE の distributee として現れる変数の集合
$l(x,n)$	ブロック n における，最後の 'REALIGN x'，または，最後の 'REDIST x' 直後の点
$l_{\mathcal{R}}(x,y,n)$	$(x, l(y,n))$
$L_{\mathcal{R}}(x,y,n)$	$\{l_{\mathcal{R}}(x,y,n)\}$
$M(x,p)$	点 p における変数 x の分散記述子の集合
T	全ての点における，変数とそのテンプレートからなる組の集合
$T_M(x,I[n])$	ブロック n の入口点 $I[n]$ において変数 x のテンプレートになる 可能性のある変数又はテンプレートの集合
$A_L(x,I[n])$	以下 2 つの性質を満たす点 p の集合. • p は x のアラインメントを変更する可能性のある REALIGN 指示文の直後の点 • x の点 p におけるアラインメントは $I[n]$ に到達する.
$R_C(x,I[n])$	以下 2 つの性質を満たす点 p の集合. • p は x のマッピングを変更する可能性のあるデータ再分 散指示文の直後の点, • x の点 p におけるマッピングは $I[n]$ に到達する.
$D_R(x,I[n])$	以下を満たす点 $p \in R_C(x,I[n])$ から構成される集合. • 点 p から $I[n]$ に至るあるパス上には，点 p 以外に $R_C(x,I[n])$ に含まれる点は存在しない.

図 3.9: 記号の説明 (1).

すことによって到達定義型の方程式となる．なぜなら，‘`REALIGN x WITH y`’はxのテンプレートを変更する唯一のデータ再分散指示文だからである．T_Dはnと$T_I(n)$に依存し，E1ではこのことを$T_D(n, T_I(n))$と記述している．また，方程式E1は**分散的**である．これら2つの事実は3.3.5節で証明される．

到達アラインメント解析:

任意のalignee xとブロックnに対して，到達アラインメント解析は$A_L(x, I[n])$,即ち，以下2つの性質を満たす点pの集合，を計算する．

- pはxのアラインメントを変更する可能性のあるREALIGN指示文の直後の点，
- xの点pにおけるアラインメントはブロックnの入口点$I[n]$に到達する．

方程式E2は‘REALIGN x’をxに対する定義と見なすと，到達定義型の方程式と同値になる．なぜなら，‘REALIGN x’はxのアラインメントを変更する唯一のデータ再分散指示文だからである．

到達マッピング解析:

任意の変数xとブロックnに対して，到達マッピング解析は$R_C(x, I[n])$,即ち，以下2つの性質を満たす点pの集合，を計算する．

- pはxのマッピングを変更する可能性のあるデータ再分散指示文の直後の点，
- xの点pにおけるマッピングはブロックnの入口点$I[n]$に到達する．

テンプレート解析が終了すれば，変数xのマッピングを変更する全てのデータ再分散指示文を見つけることができ，その時，方程式E3は到達定義型の方程式と同値になる．alignee xのマッピングを変更するのは2つのタイプの指示文である．1つは‘`REALIGN x WITH y`’のような，指示文のaligneeが変数xとなるREALIGNである．別の1つは変数xのテンプレート，tとする，が指示文のdistributeeとなるDISTである．指示文‘`DIST t`’の位置でtがxのテンプレートか否かはテンプレート解析からわかる．このようなテンプレート解析への依存性は方程式E3においてR_PがTに依存し「$R_P(u, T)$」，R_Dが$T_I[n]$に依存する「$R_P(u, T_I[u])$」と言う形で表現されている．これら2つの依存性は3.3.5節で証明される．

一般に，指示文があるプログラム点に到達するか否かを正確に解析するには多くのメモリを必要とする．例えば，変数xがあるプログラム点でm個のテンプレート$t_1, \cdots, t_m$を持つ可能性がある場合を考えよう．このような状況は複数の制御フローが合流する点で起こりえる．さらに，その点よりも後に$(m+1)$個の指示文‘`REALIGN y WITH x`’，‘`REDIST` t_1’, $\cdots$, ‘`REDIST` t_m’が続くと仮定しよう．この場合，1番目のREALIGN指示文直後におけるyのデータマッピングは最後のREDIST指示文直後には到達しない．yの（そしてxの）可能な全てのm個のデータマッピングは，m個のREDISTによって全て変更されてしまうからである．このことをコンパイラが知るには，コンパイラは各指示文，及び，1番目の指示文‘`REALIGN x`’から任意の点への各パス上において，変数xのm個のテンプレート各々へのアラインメントの状況を保持しなければならない．このようなことを実現するために，2種類の到達マッピングという概念を定義する．1つは正確な情報を保持する理想的なもの，他方は実際の利用に適した省メモリなもの，である．

定義 1 あるプログラム点 p における，ある変数 x の到達マッピングは以下の条件を満たす再分散点 p' の集合である．

(1) x が distributee の場合:
p' から p へ至るあるパスが存在して，p' 直前には ‘(RE)DIST x’があるが，そのパス上には ‘(RE)DIST x’はない．

(2) x が alignee の場合:
$T_M(x,p') = \{t_1, \cdots, t_m\}$ と仮定する．この時，以下の少なくとも一方の条件が成立する．

(2a) p' から p に至るあるパスとある $t_j (1 \leq j \leq m)$ が存在して，p' 直前には ‘(RE)DIST t_j’があるが，そのパス上には ‘REALIGN x’ も ‘(RE)DIST t_j’もない．

(2b) p' から p に至るあるパスとある $t_j (1 \leq j \leq m)$ が存在して，p' 直前には ‘(RE)ALIGN x’があるが，そのパス上には ‘REALIGN x’ も ‘(RE)DIST t_j’もない．

□

定義 2 あるプログラム点 p における，ある変数 x に対する弱到達マッピングは，定義 *1* における条件 *(1)* と *(2a)*，又は，条件 *(1)* 及び定数 K とある数 m に対する以下の条件 *(2b′)* を満たす再分散点 p' の集合である．ここで m は $T_M(x,p')$ の要素数を表す．

(2b′) p' から p に至るあるパスが存在して，*(1)*p' 直前には ‘(RE)ALIGN x’があるが，そのパス上にはそれがなく，かつ，*(2)* $m > K$，または，ある $t_j (1 \leq j \leq m)$ が存在して，‘REDIST t_j’がそのパス上にない．

□

到達マッピングの概念は弱到達マッピングの特殊な場合である．$K = \infty$ とすると両者は一致する．

コンパイラ実装では $K = 1$ に対する弱到達マッピングを採用した．なぜなら，その解析コストは低く，またその解析精度は多くのプログラムに対して十分であると考えたからである．

直接到達マッピング解析:

直接到達マッピング解析は，任意の変数 x と任意の基本ブロック n に対して，以下の性質を満たす点 $p \in R_C(x, I[n])$ から構成される集合 $D_R(x, I[n])$ を計算する．

- 点 p から点 $I[n]$ に至るあるパス上には，点 p 以外に $R_C(x, I[n])$ に含まれる点は存在しない．

この解析はデータマッピング解析の解析精度向上に役立つ．方程式 E4 もまたテンプレート解析の終了後には到達定義解析の方程式と同値になる．方程式 E4 の $T_I[n]$ への依存性（「$D_P(u, T_I[u])$」及び「$D_D(u, T_I[u])$」）もまた，3.3.5 節で証明される．

結局, 図 3.7 が示すように，上記 4 つのデータフロー解析の適用順序は以下となる：最初がテンプレート解析であり，次が到達マッピング解析または直接到達マッピング解析であり，到達アライン解析は他とは独立である．

表 3.1: align-target Y を持つ alignee X のマッピング記述子（MD）.

アラインメント記述子 (AD: $\mathcal{A}$)	
MP(i)	X の i 番目の次元に関連付けされた align-target の次元.
LB(i)	X の i 番目の次元の最初の要素は最終的に Y の MP(i) 番目の次元の $LB(i)$ 番目の要素にアラインされる.
ST(i)	X の i 番目の次元に沿ったアラインで使われるストライド.
RK	X のランク.
SP(i)	X の形状.
TP(i)	N: **もし** Y の次元 i にアラインする X の次元があるなら R: **もし** X が Y の次元 i に沿ってレプリケートされるなら S: **もし** X が Y の次元 i の一つの座標にアラインするなら
IF(i)	Y の次元 i にアラインする X の次元: **もし** TP(i) = N なら X がアラインする Y の次元 i の座標: **もし** TP(i) = S なら
TNM	Y の名前.
ディストリビューション記述子 (DD: $\mathcal{D}$)	
DTP(i)	Y の i 番目の次元の分散形状.
BS(i)	データ分散におけるブロックサイズ.
DRK	DISTRIBUTEE のランク.
DSP(i)	DISTRIBUTEE の配列形状.
PNM	プロセッサアレイの名前.
プロセッサ記述子 (PD: $\mathcal{P}$)	
PRK	プロセッサアレイのランク.
PSP(i)	プロセッサアレイの配列形状.

3.3.2 マッピング記述子とそれらの合成

マッピング記述子（MD) は変数のデータマッピングを表現するパラメータの集合であり，アラインメント記述子（AD)，データ分散記述子（DD)，及び，プロセッサ記述子（PD) から成る．表 3.1 は MD の内容をリストしたものである．この表で，‘(i)’ を含む行はその項目における i 番目の配列要素を表す[5]．MD のうち，AD は主として REALIGN 指示文によって変更される部分であり，DD は主として REDIST 指示文によって変更される部分である．PD はプロセッサアレイに関する情報を保持する．MD をこれら 3 つの部分に分割することにより，MD がデータ再分散指示文によってどのように変更されるかが明確になる．したがって，変数 x の MD を以下のように表現する．

$$\mathcal{M}_x =< \mathcal{A}(t), \mathcal{D}(pr), \mathcal{P} > . \tag{3.1}$$

ここで, t はテンプレートを， pr はプロセッサアレイを，$\mathcal{A}(t)$ は AD を, $\mathcal{D}(pr)$ は DD を, $\mathcal{P}$ は PD を表す．本章では，プログラム全体を通じて，プロセッサアレイを一つの

[5] もし，ある alignee の align-target が alignee の添字 i を含む次元を持たなければ，$MP(i)$, $LB(i)$, 及び $ST(i)$ は各々0 に設定される.

表 3.2: 2 つの相対アラインメント記述子の合成.

$\mathcal{A}_y(z)$	$\mathcal{A}_x(y)$	$\mathcal{A}_y(z) \circ \mathcal{A}_x(y)$
$MP_y(i)$	$MP_x(i)$	$MP_y(MP_x(i))$
$LB_y(i)$	$LB_x(i)$	$ST_y(MP_x(i)) * (LB_x(i) - 1) + LB_y(MP_x(i))$
$ST_y(i)$	$ST_x(i)$	$ST_y(MP_x(i)) * ST_x(i)$
RK_y	RK_x	RK_x
$SP_y(i)$	$SP_x(i)$	$SP_x(i)$
$TP_z(i)$	$TP_y(i)$	**if** $(TP_z(i) =$N$)$ **then** $TP_y(IF_z(i))$ **else** $TP_z(i)$
$IF_z(i)$	$IF_y(i)$	**if** $(TP_z(i) =$N$)$ **then** $IF_y(IF_z(i))$ **else** $IF_z(i)$
$TNM_z(= z)$	$TNM_y(= y)$	$TNM_z(= z)$

配列に固定した場合のみを考察する．よって，変数 x の MD を以下のように表現する．

$$\mathcal{M}_x =< \mathcal{A}(t), \mathcal{D} >=< \mathcal{A}, \mathcal{D} > . \tag{3.2}$$

相対的な AD と MD の合成：

指示文 '`REALIGN x WITH y`' について考えよう．もし y がテンプレートなら x の AD を計算するのにその指示文だけで十分である．しかしながら，もし y が配列で y をテンプレートにアラインする別の (RE)ALIGN があるとすると，x の AD を計算するのにその指示文だけでは不十分であり，さらに，y の AD と x の y に対する**相対的な** AD（以降，RAD と略す）を必要とする．RAD は y を x のテンプレートとみなす時の x の AD と定義する．即ち，x の (**絶対**) AD は x の y に対する RAD と y の AD を用いて計算される．この計算を**合成**と呼ぶ．'絶対'AD と '相対'AD の計算方法は同じであるため，用語 **合成**を，2 つの RAD を '合成' する時にも用いる．以下の指示文を考えよう:

```
!HPF$ ALIGN y(i) WITH z(2*i)          // A_y(z) = A(z(2*i))
!HPF$ REALIGN x(i) WITH y(i+1)        // A_x(y) = A(y(i+1))
```

ここで，$\mathcal{A}_x(y)$ は x の y に対する RAD を表す．もし z がテンプレートなら, x の AD は $\mathcal{A}_y(z)$ と $\mathcal{A}_x(y)$ の合成，$\mathcal{A}_y \circ \mathcal{A}_x = \mathcal{A}(z(2 * i + 2))$ となる．これは,$f(i) = 2 * i$ と $g(i) = i + 1$ を i に関する関数とみなすと，2 関数 f と g の合成と等しくなる．表 3.2 は 2 つの RAD の合成を示す．第 1 列と第 2 列の合成が第 3 列となる[6]. 例えば，第 2 行は，x の i 番目の次元が z の $MP_y(MP_x(i))$ 番目の次元にアラインすることを示す.

3.3.3 マッピング記述子解析

マッピング記述子解析は局所 MD 解析（図 3.11 に要約を示す），大局 MD 解析（図 3.12 に要約を示す），及び MD 結合から構成される．局所 MD 解析とは各データ再分散指示文の位置において，そこで MD が変更される変数に対して変更後の MD の集合を求めるものであり，REDIST に対する解析と REALIGN に対する解析からなる．

REDIST に対する解析:

本解析では，MD が指示文 '`REDIST t`' によって変更されるような変数の集合を求め，変更後の MD を計算する．あきらかにそのような変数はテンプレート t とその指示文

[6] $MP(0)$, $LB(0)$, 及び $ST(0)$ は 0 と設定される.

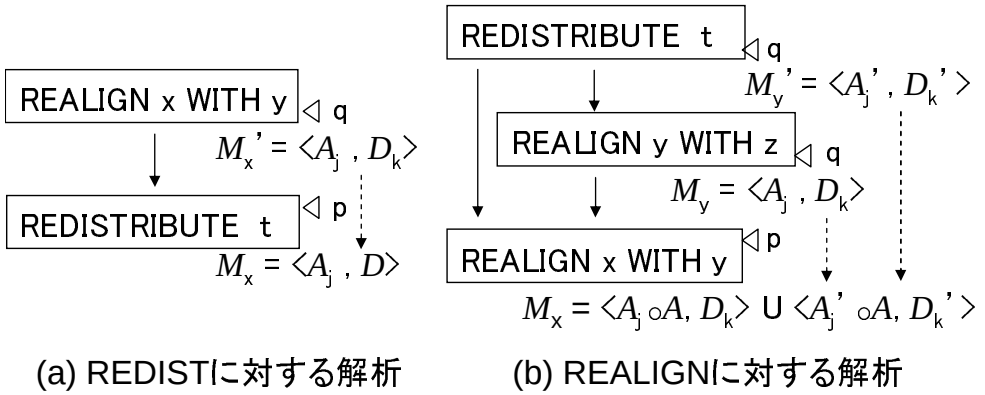


図 3.10: 局所 MD 解析におけるフローグラフの例.

の位置で t にアラインしている変数 x である．このような変数に対する MD，$\mathcal{M}_t$ と $\mathcal{M}_x$，は以下のように計算される．$\mathcal{D}$ を上記指示文によって定義されるテンプレート t の DD とし，$\mathcal{A}_{\mathrm{id}}$ を t から t 自身への恒等的なアラインメントに対する AD とする．この時，$\mathcal{M}_t$ は以下のように表せる．$\mathcal{M}_t =< \mathcal{A}_{\mathrm{id}}, \mathcal{D} >$. 一方，$\mathcal{M}_x$ は以下の方法によって，$< \mathcal{A}_j, \mathcal{D} >$ と計算される．(図 3.10 (a) を参照).

1. テンプレート解析の結果を使い，REDIST 指示文直後の点 p で t にアラインする全ての変数 x の集合を求める．
2. 到達アラインメント解析の結果から点 p に到達する全ての‘`REALIGN x`’を見つける．
3. 上記‘`REALIGN x`’直後の点 q において，そのテンプレートが t であるような x の MD， $\mathcal{M}'_x =< \mathcal{A}_j, \mathcal{D}_k >$，を求める．(これには以下に述べる REALIGN に対する解析を必要とする)

REALIGN に対する解析:

本解析はその MD が指示文‘`REALIGN x WITH y`’で変更されるような変数の集合を求め，変更後の MD を計算する．明らかにそのような変数は変数 x のみである．変数 x に対する MD，$\mathcal{M}_x$，は $< \mathcal{A}_j \circ \mathcal{A}, \mathcal{D}_k >$ となり，これは以下の方法で計算される (図 3.10 (b)).

1. 到達マッピング解析の結果を使い，y の MD を変更し，上記指示文直前の点 p に到達する全ての指示文を求める．
2. 上記で求めた指示文直後の点 q における y の MD，$< \mathcal{A}_j, \mathcal{D}_k >$, を全て見つける (これには上記 REDIST に対する解析を必要とする)．
3. 点 q における y の AD，$\mathcal{A}_j$, と REALIGN 指示文における x の RAD, $\mathcal{A}$ との合成を計算する．

今まで見てきたように，REDIST と REALIGN に対する解析は互いに他の解析結果を必要とする．そこで，一般繰返し法 (general iterative method) を用いる．即ち，MD 集合の計算が収束するまで，REDIST と REALIGN に対する解析を繰返し適用する．(図 3.11 を参照).

大域 MD 解析は各変数とその変数の各サブブロックに対して，そのサブブロックにおけるその変数の MD の集合を見つける解析である．

入力: 任意の再分散指示文におけるデータフロー集合.
出力: 任意の再分散指示文において MD が変化する変数に対する変化後の MD の集合.

1: **while** (MD の数が変化する間)
2:**for** (全ての再分散指示文とその直後の点 $p \in P$ に対して)
3: **if** ('REDISTRIBUTE t')
4: $M(t,p) = \{< \mathcal{A}_{\mathrm{id}}, \mathcal{D}(t,p) >\}$
5: **for** ($T_{\mathrm{M}}(x,p) \supseteq \{t\}$ を満足する全ての alignee x に対して)
6: $M(x,p) = \bigcup_{\substack{<\mathcal{A}_j, \mathcal{D}_k(t,p_l)> \in M(x,p_i) \\ p_i \in A_{\mathrm{L}}(x,p)}} \{< \mathcal{A}_j, \mathcal{D}(t,p) >\}$
7: **else if** ('REALIGN x WITH y')
8: $M(x,p) = \bigcup_{\substack{<\mathcal{A}_j, \mathcal{D}_k> \in M(y,p_i) \\ p_i \in R_{\mathrm{C}}(y,p)}} \{< \mathcal{A}_j \circ \mathcal{A}, \mathcal{D}_k >\}$
9: **end if**
10: **end for**; **end while**

図 3.11: 局所 MD 解析のアルゴリズム.

入力: 任意の再分散指示文において MD が変化する変数に対する変化後の MD の集合.
出力: 任意のサブブロックにおける任意の変数に対する MD の集合.

1:**for** ($\forall$ ブロック n, $\forall x \in V$, $\forall$ サブブロック b（n 中の）)
2:**if** (b が n における最初のサブブロック)
3: **for** ($\forall p \in D_{\mathrm{R}}(x, IN[n])$) MakeMappingDescriptor$(x,p,b)$
4:**else** /* p_n is the beginning point of b */
5: MakeMappingDescriptor(x, p_n, b)
6:**end for**

7:**MakeMappingDescriptor**(x,p,b)
8:**for** ($\forall p' \in R_{\mathrm{C}}(x,p)$)
9:**for** ($\forall < \mathcal{A}_j, \mathcal{D}_k(t_i,p_l) > \in M(x,p')$)
10: **if** (('REDISTRIBUTE t_i'が点 p 直前にある) $\wedge$ $(p \neq p')$)
11: ; // do nothing
12: **else**
11: $M(x,b) = M(x,b) \cup \{< \mathcal{A}_j, \mathcal{D}_k(t_i,p_l) >\}$

図 3.12: 大局 MD 解析のアルゴリズム.

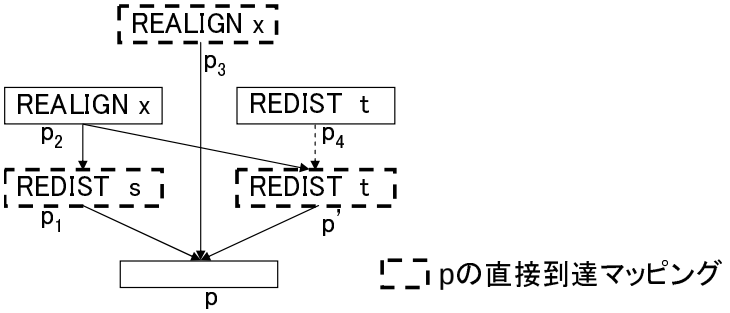


図 3.13: 大局 MD 解析で使われるフローグラフの例.

本解析については，基本アルゴリズムと2つの改良アルゴリズム Imp 1 と Imp 2 を使って説明する.

基本アルゴリズム

1. 到達マッピング解析と局所 MD 解析の結果を使い，任意の再分散点 p に対して，p に到達する全てのデータ（再）分散指示文とその直後の点におけるすべての MD の集合を求める.
2. 上記で得られた全ての MD の和集合を点 p における MD とする.

例: ブロックの入口点 p，p に到達するデータ再分散指示文直後の点，p_1 から p_4 及び p'，を図 3.13 に示す．点 p_2 における x のテンプレートの集合は t と s であると仮定する．即ち，$M(x,p_2)$ は p に到達し，$M(x,p) = \cup_{q\in\{p_1,p_2,p_3,p_4,p'\}} M(x,q)$ となる.

改良アルゴリズム：Imp 1
条件: ある指示文 D1: '`REDIST t`' が再分散点 p の直前にあり，別の指示文 D2: '`REDIST t`',が p に到達する場合.
アルゴリズム: 点 p で t にアラインする変数 x の MD の集合から，指示文 D2 直後の点で t にアラインする変数 x の MD の集合を取り除く.
理由: このような取り除かれた MD は指示文 D1 で無効化されるため.

例: 図 3.13 における $M(x,p_4)$ は $M(x,p')$ から取り除かれる.

$\pi(p', p)$ ：点 p' から点 p に至るあるパス.

“$<<$”：同じブロック中の 2 つの文の実行順序を定める記号; 即ち, もし S_2 が文 S_1 の後で実行されるなら， $S_1 << S_2$ と書く.

$\Delta(x, n) := \{t \in D | t \in \mathcal{R}[n] \cap T_M(x, I[n])\}$.

$T_1 := T_M(x, l(x, n))$. 即ち，$l(x, n)$ 直後の x のテンプレートの集合.

$\Lambda(x, n) := \{t \in T_1 | l(x, n) << l(t, n)\}$. ただし，$x$ は alignee.

図 3.14: 記号の説明 (2).

改良アルゴリズム：Imp 2
条件: 再分散点 p があるブロックの入口点である場合.
アルゴリズム: p に対する直接到達マッピング解析の結果を使い，点 p に直接到達する点を求める．次に，得られた各点に Imp 1 を適用し，得られた MD の和集合を求める.
理由: $M(x, p)$ は直接到達マッピング解析によって得られる指示文直後の点における MD の和集合に等しい．本アルゴリズムで得られた集合は基本アルゴリズムで得た集合よりも小さくできる可能性があるが，本集合が $M(x, p)$ に等しいことは以下の定理 1 によって保障される．(この定理は付録で証明される).

例: Imp 1 を p の直接到達マッピングである，図 3.13 の点 p_1, p_3, 及び p' に適用することによって，$M(x, p) = \cup_{q \in \{p_1, p_2, p_3, p_4, p'\}} M(x, q)$ となっていたのが，$M(x, p) = \cup_{q \in \{p_1, p_2, p_3, p'\}} M(x, q)$ となる.

定理 1 p をあるブロックの入口点とする．この時，以下が成立する.

$$M(x, p) = \bigcup_{p' \in D_R(x, p)} M(x, p'). \tag{3.3}$$

3.3.4 MD 結合

MD 結合は，各変数に対する各参照点からその変数の MD の集合を求めることを，コンパイラ最適化処理部に可能にさせる処理である．この処理は，変数 x の各参照点に，その参照点を含むサブブロック入口点における x の MD を対応させることで実現する (図 3.6 (b) を参照).

この結果，コンパイラ最適化処理部は変数 x の任意の参照点において x の MD を得ることが出来る.

3.3.5 データフロー方程式の特徴

本節では，3.3 節で述べた 4 つのデータフロー方程式の特徴を述べる．図 3.14 は本節で用いる記号の説明である．

テンプレート解析: 以下で，テンプレート解析に対するデータフロー方程式が **分散的** [25] であることを示す. 即ち，このデータフロー方程式に対して正確な解 T_I (図 3.7 と図 3.8 を参照) が得られる．[25]. これを証明するために，以下の定義が必要となる．

定義 3 $(L_{\mathrm{T}}(x,n))$ m 個の *REALIGN*, [7] '`REALIGN` $x_{\eta(1)}$ `WITH` $x_{\mu(1)}$', $\cdots$, '`REALIGN` $x_{\eta(m)}$ `WITH` $x_{\mu(m)}$' はブロック n においてこの順に並んでいると仮定する．この時，任意の $x_{\eta(i)}(1 \leq i \leq m)$ に対して，$x_{\mu(\alpha(i))}$ が存在して，

$$T_M(x_{\eta(i)}, O[n]) = T_M(x_{\mu(\alpha(i))}, I[n]). \quad (i = 1, \cdots, m;\ 1 \leq \alpha(i) \leq i)$$

となる．この $x_{\mu(\alpha(i))}$ をブロック n 内で $x_{\eta(i)}$ が最終的にアラインする *align-target* と呼び，$L_{\mathrm{T}}(x_{\eta(i)}, n)$ と書く． □

もしブロック n が以下に示す 2 つの指示文のみを含むなら, $L_{\mathrm{T}}(x,n) = \{z\}$ となる.

```
!HPF$ REALIGN y WITH z
!HPF$ REALIGN x WITH y
```

以下の定理は 図 3.7 における方程式 E1 の形

$$\textbf{E1:}\quad T_I(n) \quad = \bigcup_{u \in pred(n)} (T_I(u) \cap T_P(u)) \cup T_D(u, T_I(u))$$

と E1 が**分散的** [25] であることを証明する.

定理 2

(1) T_D は n と $T_I(n)$ に依存し，さらに以下のように表現できる．

$$T_D(n, T_I(n)) = \{(x,t) | x \in A \cap \mathcal{R}[n] \wedge (L_{\mathrm{T}}(x,n), t) \in T_I(n)\}. \tag{3.4}$$

(2) テンプレート解析に対するデータフロー方程式は分散的である.

証明

(1) x をブロック n に含まれる REALIGN の ailgnee とする．即ち，$x \in A \cap R[n]$．この時，定義 3 によって '$t \in T_M(x, O[n])$' は '$t \in T_M(L_{\mathrm{T}}(x,n), I[n])$' を意味する．よって，$T_D(n)$ は $T_M(*, I[n])$，即ち，$T_I(n)$ に依存することがわかる.

(2) X と Y を変数，$f(X)$ を $(X \cap T_P) \cup T_D(X)$ とする．このとき, $f(X) \cup f(Y) = ((X \cup Y) \cap T_P) \cup T_D(X) \cup T_D(Y)$.

$T_D(X)$ は $\{(x,t) | (L_{\mathrm{T}}(x), t) \in X\}$ に等しいので，以下を得る.

$$(x,t) \in T_D(X) \cup T_D(Y) \quad \Rightarrow \quad (x,t) \in T_D(X \cup Y).$$
$$f(X) \cup f(Y) = ((X \cup Y) \cap T_P) \cup T_D(X \cup Y) = f(X \cup Y).$$

□

[7] T_M の決定に影響を及ぼさないこと，及び，簡単のために，これらの REALIGN に対して align source と align index を略す.

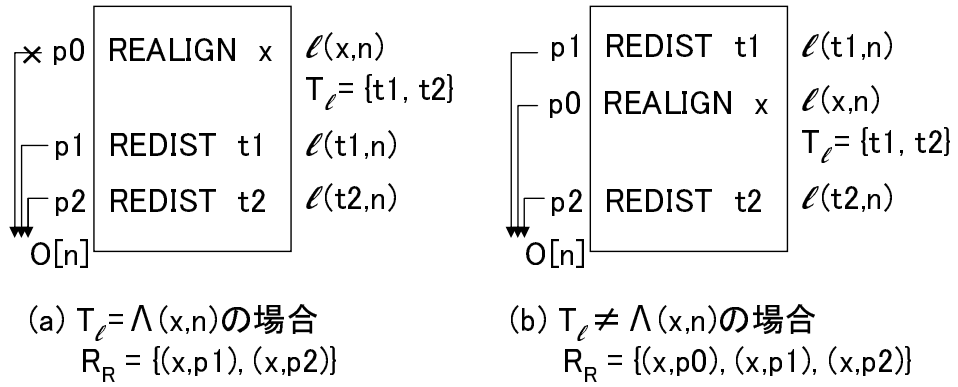


図 3.15: 定理 3 の説明.

到達マッピング解析:
以下の定理は図 3.7 における方程式 (E3)

$$\textbf{E3:}\quad R_I(n) \;=\; \bigcup_{u\in pred(n)} (R_I(u)\cap R_P(u,T))\cup R_D(u,T_I(u))$$

において R_D は n と $T_I(n)$ に依存し, R_P は n と T に依存することを証明する．ここで, $T_I(n)$ と T はテンプレート解析で得られるので，E3 はテンプレート解析の後では到達定義に対する方程式と同値になる．即ち，E3 は分散的である．

定理 3 $R_P \subseteq VP$ を第 *1* 要素が *distributee* であるもの $R_P^D \subseteq D\times P$ と第 *1* 要素が *aligneee* であるもの $R_P^A \subseteq A\times P$ に分割する．同様にして R_D を R_D^D と R_D^A に分割する．この時，$R_P = R_P^D \cup R_P^A$ は n と T の両方に依存し, $R_D = R_D^D \cup R_D^A$ は n と $T_I(n)$ の両方に依存する．ここで,

1a. $R_P^D(n) = \{(x,*)\in VP | x\in D\cap \overline{\mathcal{R}[n]}\}$,

1b. $R_P^A(n,T) = \{(x,p)\in VP | x\in A\cap \overline{\mathcal{R}[n]} \wedge ((\#T_M(x,p) > K)$
$\vee(\exists i\ s.t.\ T_M(x,p) - \mathcal{R}[\pi_i(p,O[n])] \neq \phi))\}$,

2a. $R_D^D(n) = \{l_{\mathcal{R}}(x,x,n)\in VP | x\in D\cap \mathcal{R}[n]\}$,

2b. $R_D^A(n,T_I(n)) = \bigcup_{x\in A\cap\overline{\mathcal{R}[n]}} R_N(x,T_I(n)) \cup \bigcup_{x\in A\cap\mathcal{R}[n]} R_R(x,T_I(n))$,

$$R_N(x) = \bigcup_{t\in\Delta(x,n)} L_{\mathcal{R}}(x,t,n),$$

$$R_R(x) = \begin{cases} L_{\mathcal{R}}(x,x,n) & \textbf{if } \Lambda(x,n)=\phi, \\ \bigcup_{t\in\Lambda(x,n)} L_{\mathcal{R}}(x,t,n) & \textbf{if } \#T_\mathrm{l}\le K\wedge T_\mathrm{l}=\Lambda(x,n), \\ \bigcup_{t\in\Lambda(x,n)\cup\{x\}} L_{\mathcal{R}}(x,t,n) & \textbf{if } \#T_\mathrm{l}\le K\wedge T_\mathrm{l}\neq\Lambda(x,n). \end{cases}$$

但し，$\overline{\mathcal{R}[n]}$ は $\mathcal{R}[n]$ の V における補集合 $V-\mathcal{R}[n]$ を表す.

証明
1a と 2a は R_P と R_D の定義より明らかである.
1b: $X = T_M(x,p)$ とする. $p \notin n$ となる p に対して, もし, '$\#X > K$' が成り立つなら, $x \in A$ なので $MD(x,p)$ は n 内の '`ALIGN x`' によってのみ無効化される. 即ち, 'x の MD は n において無効化されない' ことと '$x \in \overline{\mathcal{R}[n]}$' は同値である. 次に, $p \notin n$ となる p で '$\#X \leq K$' が成り立ち, $T_M(x,p) = \{t_1, \cdots, t_m\}$, $(m < K)$ と仮定する. この時, 全ての $j \in \pi_i(p, O[n])$ に対して, $M(x,p)$ は $\pi_i(p, O[n])$ 上の '`ALIGN x`' または '`DIST` t_j' $(j = 1, \cdots, m)$ によって無効化される. 即ち, 'x の p における MD は n 内で無効化されない' ことと '$x \in \overline{\mathcal{R}[n]}$, かつ, $\exists i$, s.t. $X - \mathcal{R}[\pi_i(p, O[n])] \neq \phi$' は同値である; これは定理の式 1b の右辺である. 但し, $\mathcal{R}[\pi_i(p, O[n])]$ は $\pi_i(p, O[n])$ 上に出現する全てのブロック $\mathcal{R}$ に対する $\bigcup_k \mathcal{R}[k]$ を表す. よって, R_P^A は任意の x と p に対する $X = T_M(x,p)$, 即ち, T に依存する.
2b: $R_D^A(n)$ を $\mathcal{R}[n]$ が alignee x を含むもの R_R と含まないもの R_N に分割する.

$[R_N]$: 'x の MD がブロック n で変更される' ことと '$\Delta(x,n) \neq \phi$' は同値である. この場合, 各 $t \in \Delta(x,n)$ に対して $M(x, l(t,n))$ は $O[n]$ に到達する. よって, R_N に対する式は証明された. $\Delta(x,n)$ は $T_I[n]$ に依存するので, R_N も $T_I[n]$ に依存する.

$[R_R]$: $O[n]$ に到達する x の MD は $M(x, l(x,n))$ または $t \in X = T_M(x, l(x,n))$ に対する $M(x, l(t,n))$ である. もし $\Lambda(x,n) = \phi$ なら, $O[n]$ に到達する MD は前者である. 一方, もし $\Lambda(x,n) \neq \phi$ なら, '$M(x, l(x,n))$ が $t \in \Lambda(x,n)$ によって n 内で無効化される' ことと '$\#X \leq K \wedge X = \Lambda(x,n)$' は同値である. このことから R_R に対する式が証明され, X の n と $T_I[n]$ に対する依存性が証明される. 図 3.15 は $[R_R]$ の証明の概略を図で説明したものである. 点 $p0, p1, p2$ を各々, $l(x,n), l(t1,n), l(t2,n)$ とし, $T_l = \{t1, t2\}$ とする. 図 3.15 (a) は '$X = \Lambda(x,n)$' の場合であり, この場合, 点 $p0$ におけるデータ分散は点 $p1, p2$ によって無効化され $O[n]$ に到達しないので, $R_R = \{(x,p1), (x,p2)\}$ となることがわかる. 一方, 図 3.15 (a) は '$X \neq \Lambda(x,n)$' の場合であり, この場合, 点 $p0$ におけるデータ分散は点 $p2$ によって無効化されず $O[n]$ に到達するので, $R_R = \{(x,p0), (x,p1), (x,p2)\}$ となることがわかる. □

直接到達マッピング解析:
以下の定理は図 3.7 における方程式 E4

$$\textbf{E4:}\quad D_I(n) \;=\; \bigcup_{u \in pred(n)} (D_I(u) \cap D_P(u, T_I(u))) \cup D_D(u, T_I(u))$$

において, D_P と D_D の両方が T_I に依存することを証明する. よって, E4 はテンプレート解析の後では到達定義に対する方程式と等しい. 即ち, E4 は分散的になる.

定理 4 $D_P \subseteq VP$ を第 *1* 要素が *distributee* であるもの $D_P^D \subseteq D \times P$ と第 *1* 要素が *alignee* であるもの $D_P^A \subseteq A \times P$ に分割する. 同様にして D_D を D_D^D と D_D^A に分割する. この時, $D_P = D_P^D \cup D_P^A$ と $D_D = D_D^D \cup D_D^A$ の両者は n と $T_I(n)$ に依存する. ここで,

1a. $D_P^D(n) = \{(x,*) \in VP | x \in D \cap \overline{\mathcal{R}[n]}\}$,

1b. $D_P^A(n, T_I(n)) = \{(x,*) \in VP | x \in A \cap \overline{\mathcal{R}[n]}, T_M(x, I[n]) \subseteq \overline{\mathcal{R}[n]}\}$,

2a. $D_D^D(n) = \{l_{\mathcal{R}}(x,x,n) \in VP | x \in D \cap \mathcal{R}[n]\}$,

```
!HPF$ DISTRIBUTE t1(block)
!HPF$ DISTRIBUTE t2(cyclic)
!HPF$ ALIGN x(i) WITH t1(i)
!HPF$ ALIGN y(i) WITH x(i+1)
      if (···) then
!HPF$ REALIGN x(i) WITH t2(i)
      endif
!HPF$ REALIGN y(i) WITH x(i+1)
!HPF$ REDISTRIBUTE t2(block)
      if (···) then
        ···
      endif
```

図 3.16: 例題プログラム.

2b. $D_D^A(n, T_I(n)) = \bigcup_{x \in A \cap \overline{\mathcal{R}[n]}} D_N(x, T_I(n)) \cup \bigcup_{x \in A \cap \mathcal{R}[n]} D_R(x, T_I(n))$,

$$D_N(x) = \{l_{\mathcal{R}}(x,t,n) | l(t,n) >> l(t',n) \quad for\ \exists t \in \Delta(x,n), \forall t' \in \Delta(x,n) - \{t\}\} \quad \textbf{if } \Delta(x,n) \neq \phi,$$

$$D_R(x) = \begin{cases} L_{\mathcal{R}}(x,x,n) & \textbf{if } \Lambda(x,n) = \phi, \\ \{l_{\mathcal{R}}(x,t,n) | l(t,n) >> l(t',n) \\ \quad for\ \exists t \in \Lambda(x,n), \forall t' \in \Lambda(x,n) - \{t\}\} & \textbf{if } \Lambda(x,n) \neq \phi. \end{cases}$$

証明

1a, 1b, および 2aは明らかである. 任意の $t \in \Delta(x,n)$ に対して，もし pが $l(t,n)$ 中で最も $O[n]$ に近いプログラム点なら，$(x,p) \in D_N$ となる. これによって D_N に対する式が証明された. 次に，$x \in A \cap \mathcal{R}[n]$ と仮定する. もし $\Lambda(x,n) = \phi$ なら，p を '`REALIGN x`' 直後の点とし，もし $\Lambda(x,n) \neq \phi$ なら，p を任意の $t \in \Lambda(x,n)$ に対する $l(t,n)$ において，$O[n]$ に最も近い点とする. この時, $(x,p) \in D_R$ となり，このことによって D_R に対する式が証明された. □

3.3.6 例

図 3.17 は例題プログラムのフローグラフ，及び，変数 x と y に対するデータフロー集合を示す. 図中の矩形は基本ブロックを表す. x, t_1, 及び t_2 に対する配列形状は同じとする. p_1 から p_5 までの各点における各変数の右側には，4 つのデータフロー集合が並んでいる. これらは左から順に，テンプレート解析 (T_M)，到達アラインメント解析 (A_L)，到達マッピング解析 (R_C)，及び直接到達マッピング解析 (D_R) に対するデータフロー集合である. 図 3.18 は図 3.17 におけるのと同じプログラムにおいて x と y の MD の集合を示したものである. p_2 と p_3 では, その点で変更されるような変数に対して変更後の MD のみが示されている. 以下において,$M(y,p_5)$ が唯一の要素からなることをデータマッピング解析を使って説明する.

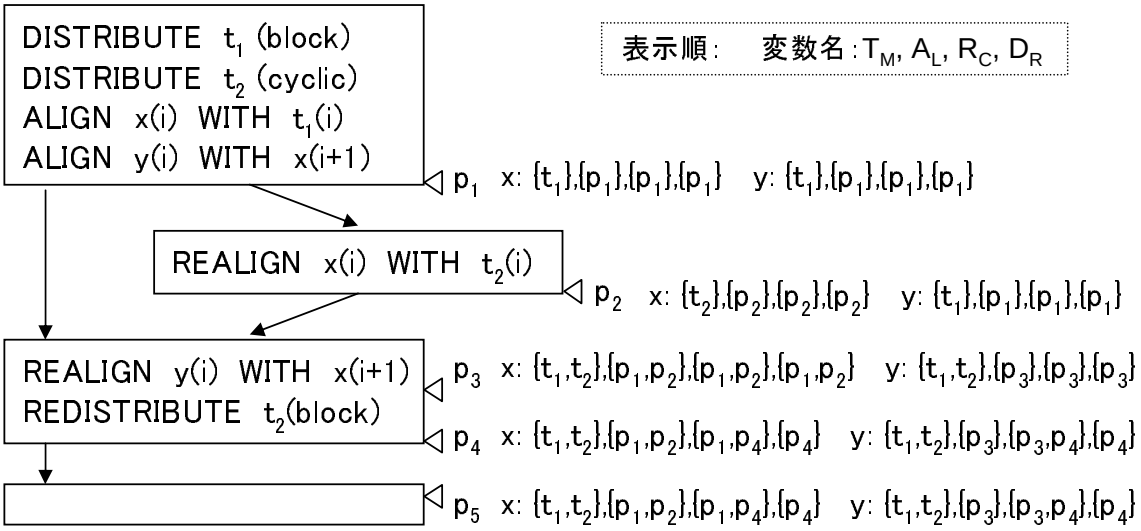


図 3.17: データフロー集合.

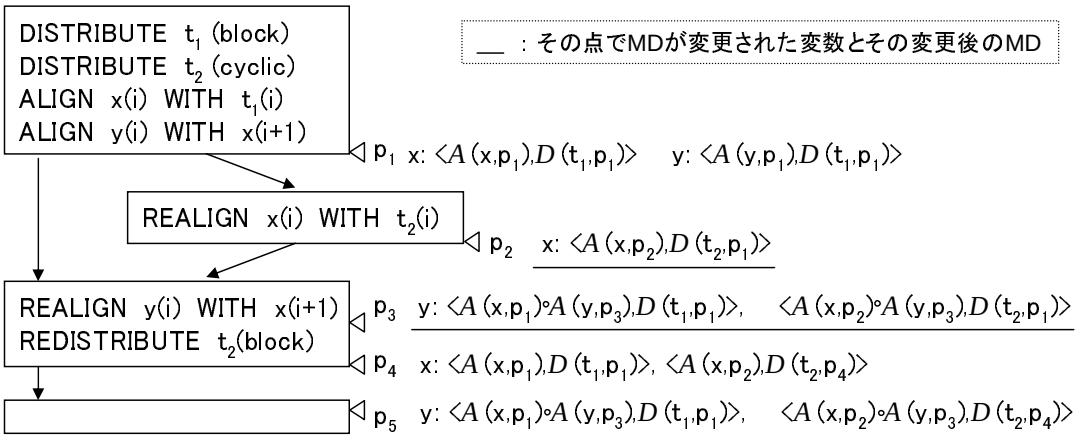


図 3.18: データ分散記述子.

1. 準備として, 図 3.11 のアルゴリズムを使って, $M(y,p_4)$ を決定する. p_4 は 'REDIST t_2' の直後にあるので, 3 行目の if 文が成立し, 5 行目の t は t_2 となる. $t_2 \in T_M(y,p_4)$ なので, 6 行目の処理に移ることができる. 図 3.17 が示すように $A_L(y,p_4)$ は $\{p_3\}$ となる. $M(y,p_3)$ のうち, template が t_2 なのは, 図 3.18 より $M(y,p_3) =< \mathcal{A}(x,p_2) \circ \mathcal{A}(y,p_3), \mathcal{D}(t_2,p_1) >$ なので, 6 行目より $M(y,p_4)$ は以下となる.

$$< \mathcal{A}(x,p_2) \circ \mathcal{A}(y,p_3), \mathcal{D}(t_2,p_4) >=< \mathcal{A}_{\text{ALIGN y(i) WITH x(i+1)}}, \mathcal{D}_{block} > . \qquad (3.5)$$

2. 次に図 3.12 のアルゴリズムを使って $M(y,p_5)$ を決定する. 図 3.17 からわかるように $D_R(y,p_5)$ は $\{p_4\}$ となる. よって, $M(y,p_5)$ は $M(y,p_4)$ に等しくなり, 3 行目において MakeMappingDescriptor (y,p_4,b) を呼ぶ. $R_C(y,p_4) = \{p_3,p_4\}$ となるので, 8 行目において $p' = \{p_3,p_4\}$ となる. $M(x,p_4)$ はこれから求めるものであり, 現在は空集合なので, 9 行目では $M(x,p_3)$ のみ考慮すればよい. $< \mathcal{A}(x,p_2) \circ \mathcal{A}(y,p_3), \mathcal{D}(t_2,p_1) > \in M(y,p_3)$ は p_4 直前の REDIST によって無効化されるので, p_4

表 3.3: FT, SP, 及び ADI に対するデータ再分散指示文.

	DIRECTIVE 1	DIRECTIVE 2
FTa	REDIST (*,BLOCK,*) :: t1	REDIST (BLOCK,*,*) :: t1
FTb	REALIGN WITH t2 :: x	REALIGN WITH t1 :: x
FTc	REDIST (*,BLOCK,*) :: x	REDIST (BLOCK,*,*) :: x
SPa	REALIGN WITH t2 :: e	REALIGN WITH t1 :: e
	REALIGN WITH e :: d	REALIGN WITH e :: d
	REALIGN WITH d :: c	REALIGN WITH d :: c
	REALIGN WITH c :: b	REALIGN WITH c :: b
	REALIGN WITH b :: a	REALIGN WITH b :: a
SPb	REALIGN WITH t2 :: X	REALIGN WITH t1 :: X
SPc	REDIST (*,BLOCK,*) :: X	REDIST (*,*,BLOCK) :: X
ADa	REALIGN WITH t1 :: z	REALIGN WITH t2 :: z
	REALIGN WITH z :: y	REALIGN WITH z :: y
	REALIGN WITH y :: x	REALIGN WITH y :: x
ADb	REDIST (BLOCK,*) :: x,y,z	REDIST (*,BLOCK) :: x,y,z

に到達する $M(y,p_3)$ は以下のように表される.

$$< \mathcal{A}(x,p_1) \circ \mathcal{A}(y,p_3), \mathcal{D}(t_1,p_1) >=< \mathcal{A}_{\text{ALIGN y(i) WITH x(i+1)}}, \mathcal{D}_{block} > . \qquad (3.6)$$

式 (3.5) と (3.6) における両方の MD は同じなので，$M(y,p_4)$ は唯一の要素からなる.

注意： $D_R(y,p_5) = \{p_4\}$ だが，$R_C(y,p_5)$ は p_4 の他に p_3 を含むことに注意する. 即ち, もし直接到達マッピングの結果を使わずに到達マッピング解析の結果のみを使って MD を計算すると，$M(y,p_5)$ が唯一の要素からなることがわからない.

3.4 評価

NAS benchmark suite [1] に含まれる FT と SP (共に Class A)，及び，ADI 法プログラム [17]，からなる 3 つのプログラムを使って，分散メモリ型スーパーコンピュータ日立 SR2201 上で提案手法を評価した. SR2201 は PA-RISC-ベースの CPU (150 MHz で 0.5 GFLOPS の処理性能), 16KB の命令キャッシュ，16KB のデータキャッシュ, 512-KB の共有 L2 キャッシュ, 及び 3 次元クロスバーネットワーク (300 MB/sec) を持つ. 評価には 8 つの CPU を用い，メッセージパッシングライブラリとして Parallelware を用いた. 3 つのプログラムに対して用いたプロセッサ数の最大値は各々，8, 6, 及び 8 であり, これらは各々のプログラムにおける主要配列のデータ分散次元の要素数の約数である. 用いた HPF コンパイラは日立の HPF コンパイラである Parallel FORTRAN をベースとしたものであり，3.3.1 節と 3.3.3 節の全ての解析をサポートしている.

FT:

図 3.19 はオリジナルの FT に修正を加えたプログラムである. FT において，配列 x の最適なデータ分散次元はループ L1 では 2 次元目，ループ L2 では 1 次元目，ループ L3 では 1 次元目または 2 次元目である. 一方，配列 $y1$, $y2$, および $y3$ に対する理

```
      real x(n1, n2, 2 * n3), y1(2 * n1), y2(2 * n1), y3(2 * n1)
!HPF$ TEMPLATE (n1, n2, 2 * n3) :: t1, t2
!HPF$ DISTRIBUTE t1(BLOCK,*,*) and t2(*,BLOCK,*)
!HPF$ ALIGN x(:,:,:) WITH *t1(:,:,:)
!HPF$ SEQUENCE :: y1, y2, y3
      < DIRECTIVE 1 >
      do k = 1, n3
L1:   do j = 1, n2
        y1(1 : n1) = x(1 : n1, j, k); y1(n1 + 1 : 2 * n1) = x(1 : n1, j, k + n3)
        call cfftz(y1)
        x(1 : n1, j, k) = y1(1 : n1); x(1 : n1, j, k + n3) = y1(n1 + 1 : 2 * n1)
      enddo; enddo
      < DIRECTIVE 2 >
      do k = 1, n3
L2:   do i = 1, n1
        y2(1 : n2) = x(i, 1 : n2, k); y2(n2 + 1 : 2 * n2) = x(i, 1 : n2, k + n3)
        call cfftz(y2)
        x(i, 1 : n2, k) = y2(1 : n2); x(i, 1 : n2, k + n3) = y2(n2 + 1 : 2 * n2)
      enddo; enddo
      do j = 1, n2
L3:   do i = 1, n1
        y3(1 : n3) = x(i, j, 1 : n3); y3(n3 + 1 : 2 * n3) = x(i, j, n3 + 1 : 2 * n3)
        call cfftz(y3)
        x(i, j, 1 : n3) = y3(1 : n3); x(i, j, n3 + 1 : 2 * n3) = y3(n3 + 1 : 2 * n3)
      enddo; enddo
```

図 3.19: FT プログラム ($n_1 = 64, n_2 = 64, n_3 = 64$).

c $X = \{a, b, c, d, e\}, U = \{u, r\}, T = \{a, b, c, d, e, u, r\}$
real $U(5, nx, ny, nz), X(nx, ny, nz)$
!HPF$ **TEMPLATE** (nx, ny, nz) :: t_1, t_2
!HPF$ **ALIGN** $U(*, :, :, :)$ and $X(:, :, :)$ **WITH** $*t_1(:, :, :)$
!HPF$ **DISTRIBUTE** t_1(*,*,BLOCK) and t_2(*,BLOCK,*)
do $istep = 1, itmax$
L1: $r(1:5, 1:nx, 1:ny, 1:nz) *= dt$
S1: **call** txinvr(T); **call** jacx $(3, T)$; **call** spentax3 (T); $\cdots$
S2: $\cdots$; **call** jacy $(5, T)$; **call** spentay $(5, T)$; **call** pinvr (T)
< DIRECTIVE 1 >
S3: **call** jacz $(3, T)$; **call** spentaz3 (T); **call** jacz $(4, T)$
S4: **call** spentaz $(4, T)$; **call** jacz $(5, T)$; **call** spentaz $(5, T)$
< DIRECTIVE 2 >
S5: **call** tzetar (T)
L2: $u(1:5, 2:nx-1, 2:ny-1, 2:nz-1) +=$
$r(1:5, 2:nx-1, 2:ny-1, 2:nz-1)$
S6: **call** l2norm (T); **call** rhs (T); **call** l2norm (T);
enddo

図 3.20: SP プログラム $(nx = 12, ny = 12, nz = 12)$.

表 3.4: FT プログラムの SR2201 上での台数効果.

procs	1	2	4	8
FTa	.59	1.50	2.68	**3.93**
FTb	.59	1.50	2.71	**3.77**
FTc	.59	1.49	2.70	**3.91**
FT1	**.87**	.20	.12	.10
FT2	**.84**	.20	.12	.09

想的なデータ分散は SEQUENCE である．これは各プロセッサが各々全配列のコピーを持つことを意味する．このようなデータ分散を指定した場合,FT に含まれる各ループはプロセッサ間データ通信を行うことなく並列化できる．データマッピング解析を評価するために，FT に対する 3 つのバージョン FTa, FTb, および FTc を作成した（表 3.3）．各々は異なるデータマッピング指示文を使って，上記の理想的なデータ分散を実現している．

表中，‘DIRECTIVE 1’ または ‘DIRECTIVE 2’ と書かれた列中に含まれる指示文は，図 3.19 に示されたプログラム中の <DIRECTIVE 1> または <DIRECTIVE 2> と書かれた個所に挿入されるべき指示文である．

データ再分散するプログラムとしないプログラムとの処理性能を比較するために，2 つのバージョン， FT1 と FT2 を作った．ここで，FT1 は配列 x を第 1 次元でデータ分散し，FT2 は第 2 次元でデータ分散する．FT1 中のループ L2 と L3 における配列 x の

表 3.5: SPプログラムのSR2201上での台数効果.

procs	1	2	4	6
SPa	.57	1.03	1.55	**1.94**
SPb	.57	1.03	1.57	**1.96**
SPc	.59	1.03	1.57	**1.95**
SP2	**1.00**	.05	.04	.03
SP3	**1.00**	.05	.04	.03

表 3.6: FTプログラムのマッピング方法.

バージョン名	変数名	L1	L2	L3
FTa-FTc	x	2	1	←
	y1,y2,y3	S	←	←
FT1	x	1	←	←
	y1	*	←	←
	y2,y3	S	←	←
FT2	x	2	←	←
	y1,y3	S	←	←
	y2	*	←	←

数字：分割次元，S: Sequence，*: Distribute(*)

データ分散（あるいはFT2中のループL1とL3における配列xのデータ分散）はFTに対する最適なデータ分散と一致しているので，配列$y2$と$y3$ (またはFT2に対しては$y1$と$y3$) のデータ分散をSEQUENCEとした．一方，FT1中のループL1において$y1$のデータ分散を‘`DIST (*)`’（またはFT2中のループL2において$y2$のデータ分散を‘`DIST (*)`’），即ち，配列全体を0番プロセッサに配置する，ことにした．これによって，全てのプロセッサと0番プロセッサの間でプロセッサ間データ通信が発生する．しかしながら，SEQUENCEとするともっと通信時間のかかる全対全のデータ通信が発生する．表3.6はFTに対する全バージョンのデータマッピングを示す．表中，数字はデータ分割する分割次元，Sは Sequence，*は‘`DIST (*)`’を適用したことを表す．

FTa, FTb, およびFTc中の全てのループはHPFコンパイラで並列化できた．表3.4は全てのバージョンの処理性能を比較したものである．表3.4より以下を得る．

- FT1とFT2に対する処理性能の向上度は低い．これはループL1とL2においてプロセッサ間データ通信が発生したためと考えられる．

- FTaからFTcまでの3つのプログラムに対する処理性能の向上度はほとんと同じである．よって，データマッピング解析は解析精度を減ずることなく，全てのデータマッピング指示文に効果的に適用された．台数効果は8プロセッサで約4倍である．台数効果がプロセッサ数程度に達しないのはデータ再分散オーバーヘッドのためと考えられる．

SP:

表 3.7: SP プログラムのマッピング方法.

バージョン名	変数名	S1-S2	S3-S4	S5-S6
SPa-SPc	X	3	2	3
	U	4	3	4
SP2	X	2	←	←
	U	3	←	←
SP3	X	3	←	←
	U	4	←	←

数字：分割次元

図 3.20 は SP プログラムの概要を示す．この図において，配列 $a, \cdots, e$ に共通な文はこれらのかわりに X を使って表現している．例えば，2 行目の '$X(nx, ny, nz)$' は '$a(nx, ny, nz), \cdots, e(nx, ny, nz)$' を表す．同様にして，$u, r$ に対しては U を使い，$a, \cdots, e, u, r$ に対しては T を使う．

ここで，配列 $X = \{a, \cdots, e\}$ 及び $U = \{u, r\}$ に対する理想的なデータ分散次元は S1, S2, S5, および S6 に対して各々，3 番目と 4 番目であり，S3 から S4 に対しては各々，2 番目と 3 番目である．この場合, 各ループはプロセッサ間データ通信を発生させること無く並列化される．3 つのバージョン SPa, SPb, および SPc を作った．これらは異なるデータ分散指示文を使って上記の理想的なデータ分散を実現している (表 3.3 を参照)．SPa は align-target がテンプレートでない，4 つの REALIGN を使っていることに注意する．SPa はデータマッピング解析の主たるターゲットである．

元々，最適なデータマッピング指示文は各関数の入口に置かれていた．しかし，これらは < DIRECTIVE 1 > および < DIRECTIVE 2 > に挿入されたデータ再分散指示文を無効化するので，挿入された指示文に対する提案手法の解析を評価することができない．この問題を解決するために，SPa と SPb においていくつかのサブルーチンをインライン展開した．Spentax3 と Spentaz3 をインライン展開対象として選択した．これらの関数の処理時間の合計は全体の処理時間の 9%であった．一方，インライン展開の効果を比較するために，SPc ではインライン展開を適用しなかった．

また 2 つのバージョン SP2 と SP3 を作った．SP2 で配列 $T = \{a, \cdots, r\}$ は最後から 2 番目の次元で固定的にデータ分散され，SP3 でこれら T の配列は最後の次元で固定的にデータ分散された．表 3.7 は SP に対する全バージョンのデータマッピングを示す．

SPa, SPb, および SPc における全てのループは HPF コンパイラで並列化された．各配列の各分散次元の要素数は 12 だったので，それらのバージョンは 12 個のプロセッサ上で実行した (表 3.5 を参照)．表 3.5 から以下が得られる．

- SPa から SPc までのプログラムに対する処理性能の向上度はほとんど同じである．よって，データマッピング解析は align-target が配列であるようなデータ再分散指示文に対しても効果的に適用された．台数効果は 6 プロセッサで約 2 倍である．これは FT に対する効果よりも低い．この理由として，データ分散次元の要素数が FT の場合よりも少ないため，データ通信オーバーヘッドが総実行時間において比較的大きな時間を占めたからと考えられる．

- SP2 と SP3 に対する処理性能の向上度は低い．これはプロセッサ間データ通信

```
      real, dimension (M,M) :: x,y,z
!HPF$ TEMPLATE (M,M) :: t1,t2
!HPF$ DISTRIBUTE t1(BLOCK,*) and t2(*,BLOCK)
!HPF$ ALIGN (:,:) WITH t1(:,:) :: x,y,z
      do iter = 1,IMAX
        < DIRECTIVE 1 >
L1:     do j = 2,M; do i = 1,M
            x(i,j)-=x(i,j-1)*y(i,j)/z(i,j-1)
            z(i,j)-=y(i,j)*y(i,j)/z(i,j-1)
        enddo; enddo
L2:     x(1:M,M) = x(1:M,M)/z(1:M,M)
L3:     do j = M-1,1,-1; do i = 1,M
            x(i,j) = (x(i,j)-y(i,j+1)*x(i,j+1))/z(i,j)
        enddo; enddo
        < DIRECTIVE 2 >
L4:     do j = 1,M; do i = 2,M
            x(i,j)-=x(i-1,j)*y(i,j)/z(i-1,j)
            z(i,j)-=y(i,j)*y(i,j)/z(i-1,j)
        enddo; enddo
L5:     x(M,1:M) = x(M,1:M)/z(M,1:M)
L6:     do j = 1,M; do i = M-1,1,-1
            x(i,j) = (x(i,j)-y(i+1,j)*x(i+1,j))/z(i,j)
        enddo; enddo
      enddo
```

図 3.21: ADIプログラム $(M = 512, IMAX = 10)$.

が多くのサブルーチン中の多くのループ内で発生したため，プロセッサ間データ通信にかかる時間が総実行時間の中で大きな時間を占めたためと考えられる．

ADI:

図 3.21 は論文 [17] から引用した ADI 法プログラムである．配列 x, y, および z に対する最適なデータ分散次元は L1 から L3 までのループでは 1 番目で，L4 から L6 までのループでは 2 番目である．この場合，各ループはプロセッサ間データ通信を発生することなく並列化できる．2 つのバージョン ADa と ADb を作った．これらは異なるデータ分散指示文で上記の最適なデータ分散を実現する．(表 3.3 を参照)．ADa は align-target がテンプレートでないような 2 つの REALIGN を用いることに注意する．また，配列 x, y, および z が 1 次元目で固定的にデータ分散されるバージョン AD1 と配列 x, y, および z が 2 次元目で固定的にデータ分散されるバージョン AD2 を作った．表 3.9 は ADI に対する全バージョンのデータマッピングを示す．ADa と ADb における全てのループは HPF コンパイラで並列化された．表 3.8 から以下を得る．

- AD2 に対する最大性能は 8 プロセッサで 1.75 倍である．この値は FT, SP, および ADI に対する固定データ分散バージョン中最大である．この理由はループ L1

表 3.8: ADIプログラムの SR2201 上での台数効果.

[procs]	1	2	4	8
ADa	0.41	1.11	2.01	**2.24**
ADb	0.41	1.07	1.67	**1.97**
AD1	1.03	1.43	**1.15**	0.85
AD2	1.03	1.56	**1.75**	1.67

表 3.9: ADIプログラムのマッピング方法.

バージョン名	変数名	L1-L3	L4-L6
ADa-ADb	x,y,z	1	2
AD1	x,y,z	1	1
AD2	x,y,z	2	2

数字：分割次元

と L3 の外側で少ないデータ量のプロセッサ間通信が少ない回数だけ発生したためと考えられる.

- ADa と ADb に対する処理性能の向上度はほとんど同じである．よって，データマッピング解析は align-target が配列であるようなデータ再分散指示文に対しても効果的に適用された．台数効果は 8 プロセッサで約 2 倍である．これは FT に対する効果よりも低い．この理由として，計算時間に対する通信時間の割合が FT の場合よりも大きかったためと考えられる.

3.5 関連研究

関連研究:

Chapman [4] は Vienna Fortran で書かれたプログラムにおいて，テンプレートに対するデータマッピング解析を行った．同様にして Hall [11] は Fortran D で書かれたプログラムにおいて，テンプレートに対するデータマッピング解析を行った [10]. しかしながら，彼らの論文は alignee に対する解析については記述していない.

Palermo [19] はテンプレートと alignee の両方に対するデータマッピング解析を行った．しかしながら，彼らのコンパイラは align-target であるテンプレートがデータ再分散されたとき，その alignee に対するデータ分散を正しく解析することはできない.

Bozkus [2] は逆アライン関数を使って alignee のデータマッピングを計算した．Coelho [6, 7] はテンプレートと alignee の両方に対するデータマッピング解析を行い，それらを多面体を使って表現した.

これら従来研究におけるコンパイラは「REALIGN 指示文における align-target はテンプレートである」という制限の下でデータマッピング解析を行っている．このような制限は 3.2 節で述べたように，プログラムを記述しにくくしたり，プログラムの保守を困難にする．そこで，上記のような制限を取り外し，REALIGN 指示文における

align-targetが一般の配列である場合に対しても適用可能なHPFプログラム向けデータマッピング解析を提案した．本解析はまた，高度な最適化に必要な非常に正確なデータマッピング情報をコンパイラに与えることが出来る．

他言語への適用性:

以下，データ分散解析の，最近のHPFライクな他言語への適用性を議論する．

Unified Parallel C (UPC) [9] は分散共有メモリモデルに対するglobal address space言語と呼ばれる言語の一つである．これは共有データに対してDISTRIBUTEのような'block-cyclic' 型のデータ分散指示文を持つが，REALIGN指示文は持たない．しかし，UPCポインタをデータ分散対象配列の先頭以外の配列要素にポイントさせることでREALIGNのalign添字中の定数項に相当する機能が実現でき，UPCポインタのターゲットが別のUPCポインタにすることで，REALIGNのアラインターゲットが配列でその配列が別のテンプレートにアラインすることに相当する機能が実現できる．よって，REDISTRIBUTEやREALIGNに対する解析を，UPCのデータ分散とポインタに対する解析に置き換えれば，提案解析はUPCプログラムに適用可能である．

ZPL [8, 22] は暗黙的に並列な配列言語であるが，データ分散指示文を持たず，データ分散はコンパイラが行う．ZPLはリージョンと呼ばれるインデックス集合を持つ．それは配列宣言における添字範囲や配列添字等に使われる．リージョンはHPFのテンプレートと同様なデータ分散の対象である．そこで，ZPLにおいて配列添字とリージョンとの差を与える構文はREALIGNに相当する．例えば，dynamic regionsやstrided regionsにより配列添字とリージョン中の添字との対応付けを動的にシフトさせたり，リージョン中の定数倍の添字へ動的に変更させることができる．これらはREALIGNのalign-subscriptにおける定数や係数の変更に対応する．以上により，テンプレートをリージョンに置き換えることにより提案解析はZPLプログラムに適用可能である．

Chapel [3, 5] はHPFとZPLに基づく言語であり，Cascade projectで策定中である．これもドメインと呼ばれるインデックス集合を持つ．これはZPLのリージョンと同様な機能を持つ．ChapelではREALIGN機能は部分配列参照とストライド・ドメインで実現することができる．上記は各々，REALIGNのalign-subscriptにおける定数と係数の変更に対応する．それゆえ，データ分散解析はテンプレートのかわりにドメインを解析することによりChapelプログラムに適用可能である．

以上述べたように，上記各言語はHPFライクなデータ再分散及びデータ再アライン機能を持つ．よって，提案解析はこれらの言語に対するコンパイラに適用可能である．

3.6 まとめ

align-targetが配列となるREALIGNはコンパイラの解析を困難にするが，align-targetがテンプレートである時よりもプログラミングの生産性を向上できる．そこで，前者のREALIGNを含むプログラムに適用可能な新しいデータマッピング解析を提案し，HPFコンパイラに実装した．提案手法を評価するために3種類のHPFプログラムの各々に対して以下の2つのバージョンを作った：一方は高生産性だが複雑なREALIGN指示文を含み，他方は単純だが生産性の低い指示文を含む．これらにHPFコンパイラに適用し，分散メモリ型並列プロセッサである日立SR2201上で評価した結果，各バージョンは同様なコードに変換され，同様な処理性能で実行された．

参考文献

[1] D. Bailey, J. Barton, and T. Lasinski, eds., The NAS Parallel Benchmarks, RNR Technical Report, RNR-94-007, NASA Ames Research Center, 1994.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Y. Wu, Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers, *J. Parallel and Distributed Computing* 21 (1994) 15-26.

[3] D. Callahan, B. L. Chamberlain, H. P. Zima, The Cascade High Productivity Language, *Proc. HIPS '04* (2004) 52-60.

[4] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima, Dynamic Data Distribution In Vienna Fortran, *Proc. Supercomputing '93* (1993) 284-293.

[5] Cray Inc., Chapel Specification 0.4, http://chapel.cs.washington.edu/, 2005.

[6] F. Coelho and C. Ancourt, Optimal Compilation of HPF Remappings, *J. Parallel and Distributed Computing* 38 (1996) 229-236.

[7] F. Coelho, Compiling Dynamic Mappings with Array Copies, *Proc. PPoPP '97* (1997) 168-179.

[8] S. J. Deitz, B. L. Chamberlain, S.-E. Choi, and L. Snyder, The design and implementation of a parallel array operator for the arbitrary remapping of data, *Proc. PPoPP '03* (2003).

[9] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper, UPC Language Specifications Ver. 1.1.1, http://upc.gwu.edu/, 2003.

[10] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, Fortran D language specification, Technical Report CRPC-TR90079, CRPC, Rice Univ., 1990.

[11] M.W. Hall, S. Hiranandani, K. Kennedy, and C. W. Tseng, Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines, *Proc. Supercomputing '92* (1992) 522-534.

[12] High Performance Fortran Forum, High Performance Fortran Language Specification Ver. 2.0.α.2, CRPC, Rice Univ., 1996.

[13] S. Hiranandani, K. Kennedy, and C. W. Tseng, Evaluating of Compiler Optimizations for Fortran D, *J. Parallel and Distributed Computing* 21 (1994) 27-45.

[14] HPF/JA Language Specifcations Ver. 1.0, http://www.hpfpc.org/jahpf/, 1999.

[15] E. Kalns and L. Ni, Processor Mapping Techniques Toward Efficient Data Redistribution, *Proc. IPPS'94* (1994) 469-476.

[16] T. Kamachi, K. Kusano, K. Suehiro, Y. Seo, M. Tamura, and S. Sakon, Generating realignment-based communication for HPF programs, *Proc. IPPS'96* (1996) 364-371.

[17] U. Kremer, J. M. Crummey, K. Kennedy, and A. Carle, Automatic Data layout for Distributed-Memory Machines in D Programming Environment, in: C. W. Kessler, ed., *Automatic parallelization - new approaches to code generation, data distribution, and performance prediction*, (Vieweg, Wiesbaden, 1994) 136-152.

[18] D. J. Palermo and P. Banerjee, Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers, *Proc. LCPC'95* (LNCS 1033, Springer, 1995) 392-406.

[19] D. J. Palermo, E. W. Hodges, and P. Banerjee, Interprocedural Array Redistribution Data-Flow Analysis, *Proc. LCPC'96* (1996) aa.1-aa.15.

[20] S. Ramaswamy, B. Simons, and P. Banerjee, Optimizations for Efficient Array Redistribution on Distributed Memory Multicomputers, *J. Parallel and Distributed Computing* 38 (1996) 217-228.

[21] H. Sakagami, H. Murai, Y. Seo, and M. Yokokawa. 14.9 TFLOPS three-dimensional fluid simulation for fusion science with HPF on the earth simulator, *Proc. Supercomputing '02* (2002) 1-14.

[22] L. Snyder, A Programmers Guide to ZPL Version 6.3., http://www.cs.washington.edu/research/zpl/home/index.html, 1999.

[23] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges, and P. Banerjee, Advanced Compilation Techniques in the PARADIGM Compiler for Distributed-Memory Multicomputers, *Proc. ICS '95* (1995) 424-433.

[24] A. Wakatani and M. Wolfe, A New Approach to Array Redistribution: Strip Mining Redistribution, *Proc. PALE '94* (1994) 323-335.

[25] H. Zima and B. Chapmann, *Supercompilers for Parallel and Vector Computers* (Addison-Wesley, 1991).

第4章　HPF言語に対する計算分散の解析手法とコード生成手法

4.1　HPF言語の図式表現と効果的計算分散手法

4.1.1　概要

今日，スーパーコンピュータは，多くの計算ノードをネットワークで接続した分散メモリ型計算機の構成をとることが多い．このような構成に対するプログラミング手法として，プログラマがMPIなどのメッセージパッシング・ライブラリを用いて並列プログラムを書く並列プログラミングが現在の主流である．並列プログラミングにおいてプログラマはデータ分散，計算分散，及びプロセッサ間通信という並列処理に関わる全ての処理を記述する．このため，非常に効率の良いプログラムの作成が可能になる一方で，プログラマの負担は重い．応用プログラムは益々複雑化・大規模化しているため，今後，このような並列プログラミング手法では，プログラマに過大な負担がかかると予想される．

この課題を解決するために，プログラマ，言語，及びコンパイラが協調して並列化を行なうことが重要であると考え，そのようなアプローチの一つとしてデータ並列言語High Performance Fortran (HPF)[5]に対するコンパイル技術の研究を進めている．HPFを使う場合，プログラマはデータ分散指示文等を逐次プログラムに挿入し，コンパイラはその指示文を解析して計算分散やプロセッサ間通信の生成を行なう，という風にプログラマとコンパイラが役割を分担する．

HPF言語には規則的データ分散と不規則データ分散がある．分子動力学で使われ，たんぱく質の構造解析など創薬分野で役立っている固有値計算は行列要素がランダムに分布しており，規則的データ分散で処理される．一方，不規則データ分散は疎行列に対して用いられることが多い．HPFに対するコンパイル技術は規則的データ分散に対してもまだ不十分なため，本節ではデータ分散を規則的なものに限定し，これに対してHPFコンパイラの処理の一つである計算分散を研究対象とした．

HPFの規則的データ分散には2つの特徴がある．その1つはblock-cyclic分散である．block-cyclic分散とは，配列をブロックと呼ばれるある連続な要素の塊に分割し，各ブロックをラウンドロビン方式で順にプロセッサに割付けるデータ分散方法である．プログラマはこの分散方法をループ繰り返し毎の処理負荷が不均一なループ中に出現する配列に指定してブロックサイズを変更することにより，ブロックという配列の連続要素参照によるメリットと負荷分散によるメリットとのトレードオフを調整できる．HPFのHPFの規則的データ分散の2つ目の特徴としてALIGN指示文を用いた2レベルマッピングがある．これは配列を一旦テンプレート（領域を確保しない仮想配列）に1次式写像等によってアラインし，このテンプレートをデータ分散することにより，

表 4.1: 記号.

A	データ分散される配列
A_l, A_u	配列 A の下限値，上限値
P	論理プロセッサの配置形状を表わす配列
T	配列 A がアラインするテンプレート
T_l	テンプレート T のある次元の下限値
b	データ分散におけるブロックサイズ
c	cyclic 分散におけるデータ割付けの繰返し回数
e	テンプレート T のある次元の寸法
$f_a i + f_b$	アライン添字．f_a, f_b：整数，変数，又は式
g	T から標準テンプレートへの埋め込み写像
h	標準ループ空間 L^0 からループ空間 L への写像
$i_a i + i_b$	配列添字．i_a, i_b：整数，変数，又は式
k	ループ制御変数に配列添字を対応させる写像
l, u, m	ループ下限値，上限値，ストライド
p	プロセッサ数
q	あるプロセッサの番号
n, r, s, v	AXIS_TYPE が NORMAL,REPLICATED, SINGLE，VANISHED となる次元数
$\mathcal{L}$	グローバル添字をローカル添字に変換する配列
ρ	標準テンプレートのローカルアドレス表現
π	$= f_a i_a m$
$\psi(i)$	$A(i)$ がマッピングされるプロセッサ番号

元の配列を間接的にデータ分散する手法である．この手法により，より柔軟で一般的なデータ分散が可能になる．

一方，このような 2 種類の複雑なデータ分散指示文が指定されたプログラムに対してコンパイラが効率の良いコードを出力するのは非常に困難である．なぜなら一般の場合には，配列（又はループ）が均等に分散されなくなり，元の配列要素とデータ分散後の配列要素との対応付けが複雑になるためである．

上記の 2 種類のデータ分散指示文が同時に指定されたプログラムに適用可能な計算分散方法としてテーブル参照法 [3] がある．この方法は，線形添字を持つ配列参照を含むループに対して，データ分散後の配列要素の参照パターンは不均等であるがある周期ごとに同じパターンを繰り返すことを利用し，ある小さいループ繰返し範囲における参照パターンを 2 種類のテーブルに格納し，次にそれらのテーブルを用いて全ループ繰返し範囲における配列参照コードを生成する．本方法は上記複雑なプログラムに対して従来，最も高速であったと思われるが，配列参照時にテーブルを介して参照を行うという間接参照オーバーヘッドが常に伴うという問題がある．

本節は，上記 2 つのデータ分散指示文が同時に指定された 1 次元配列を左辺に持つ文を含む 1 重ループに対して一般的枠組みによる計算分散法を提案する．この枠組みはループ制御変数と配列次元が 1 対 1 に対応するような d 次元配列・d 重ループに容易

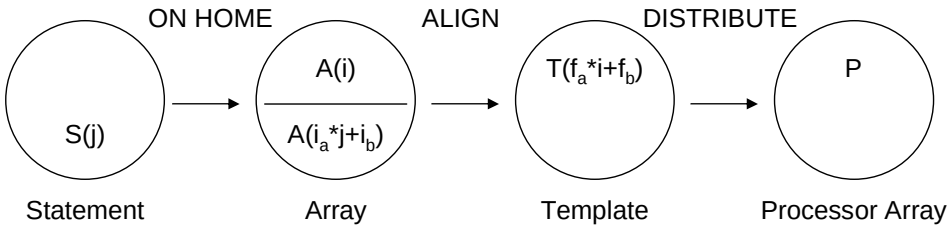


図 4.1: HPF によるデータ分散と計算分散.

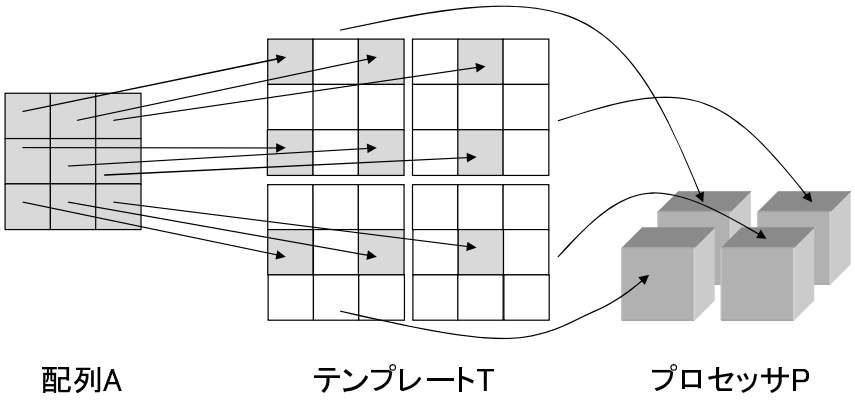


図 4.2: 2 レベルデータ分散の例.

に拡張できる．この枠組みを用いることにより，従来より高速なテーブル参照コードを導く．さらに，ある特別な条件が成り立つ場合に最適なコードが出力できることも示す.

本節の残りは以下となる．第 4.1.2 節では HPF 指示文を説明し，本節が扱うプログラムパターンを定義し，第 4.1.3 節ではこのプログラムパターンを見通し良く表現するのに用いる図式を説明する．第 4.1.4 節では配列添字式—ループ制御式—データ分散の間の関係を図式で表現し，第 4.1.5 節ではこれを用いて計算分散公式を導き，第 4.1.6 節では特別な場合に対して最適コードを導く．第 4.1.7 節では本公式における添字参照の擬周期性を示してテーブル参照コードを与える．第 4.1.8 節では本節で提案したコードを評価し，第 4.1.9 節では関連研究について述べ，第 4.1.10 節で本研究をまとめる.

4.1.2 HPF 指示文と RDLS(1, 1) パターン

本節では，HPF 指示文 [5] を説明し，本節において考察の対象とするプログラムパターンを定義する.

HPF によるデータ分散と計算分散

図 4.1 は HPF によるデータ分散と計算分散を説明した図である．HPF ではデータ分散は以下に述べる 2 段階で行なわれる（2 レベルマッピング）.

第 1 段階では配列 A をテンプレートと呼ばれる実メモリを確保しない仮想配列 T に

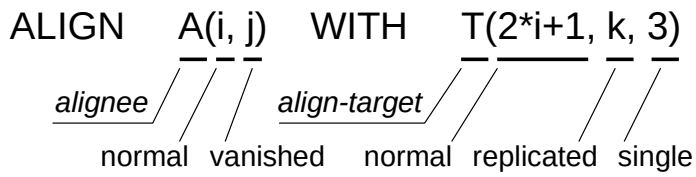


図 4.3: AXIS_TYPE の例.

アラインさせる[1]．$A(i)$ を $T(f_a i + f_b)$ [2]にアラインさせることをプログラマは ALIGN 指示文で記述する．ここで，アライン指示とは，配列要素 $A(i)$ を $T(f_a i + f_b)$ と同じプロセッサにマッピングすることを指示することである．第 2 段階ではテンプレート T をプロセッサ形状を表わす配列 P にマッピングさせる．プログラマはこれをテンプレート T に対して BLOCK(b), CYCLIC(b) 等の規則的データ分散または GEN_BLOCK 等の不規則データ分散を DISTRIBUTE 指示文を使って指示することで実現する．以上の 2 段階により，配列 A はプロセッサ P にマッピングされる．尚，本節では規則的データ分散のみを扱う．図 4.2 は 2 レベルデータマッピングの例である．この図では，配列 $A(3,3)$ をテンプレート $T(6,6)$ に 'ALIGN $A(i,j)$ WITH $T(2*i, 2*j)$' によってアラインし，テンプレート T をプロセッサ配列 $P(2,2)$ へ 'DISTRIBUTE $T(block(3), block(3))$ ONTO P' によって分散したときの配列 A の要素のマッピングの様子を表す.

次に，計算を実行するプロセッサの決定（計算分散）は owner-computes rule (OCR) または ON HOME 指示文で行なう．OCR では代入文 $S(j)$ はその左辺の配列要素 $A(i_a j + i_b)$ がマッピングされるプロセッサで実行される（図 4.1）．ON HOME 指示文では上記のような配列要素をプログラマが任意に指定できる.

AXIS_TYPE

図 4.3 は ALIGN 指示文とその AXIS_TYPE の例である．ALIGN 指示文における ALIGN 節直後の配列は alignee，WITH 節直後のテンプレート（または配列）は align-target と呼ばれる．align-target の各次元は以下の 3 つの AXIS_TYPE に分類される：

- NORMAL:その次元に含まれる変数は，alignee のある次元の添字と一致する.
- REPLICATED:その次元の添字は alignee のどの次元にも現れない.
- SINGLE:その次元の添字は定数である.

さらに，第 4.1.4 節で用いるため，alignee の各次元に対しても以下の AXIS_TYPE を定義する：

- NORMAL:その次元の添字は align-target のある次元の添字に現れる.
- VANISHED:その次元の添字は align-target のどの次元にも現れない.

[1]HPF では A をテンプレートでない配列 B にアラインさせることもできる．しかし，結局は（アラインの連鎖により）あるテンプレート T にアラインされることになる.

[2]以降，特別な場合以外は $f_a i$ のように f_a と i の間の乗算記号を略す.

```
1: !HPF$ PROCESSORS P(p)
2: !HPF$ TEMPLATE T(T_l : T_l + e - 1)
3: !HPF$ DISTRIBUTE T(CYCLIC(b)) ONTO P
4: !HPF$ ALIGN A(i) WITH T(f_a i + f_b)
5:       REAL A(A_l, A_u)
6:       do i = l, u, m
7:          A(i_a i + i_b) = ···
8:       end do
```

図 4.4: RDLS(1,1) パターンの例.

RDLS(1,1) パターン

本節では，本節において考察の対象とする RDLS(1,1) パターンの定義を述べる．

定義 4 RDLS(1,1) パターン
以下の配列 A, ループ L, 及び代入文 S から構成されるプログラムパターンを *RDLS(1, 1)* パターン（*Regular data Distribution of 1-dimensional array and 1-tuple nested loop with Linear Subscript*）と呼ぶ：

1. 一般の規則的 *HPF* データ分散指示文が指定された *1* 次元配列 A,
2. 一般のループ制御式を持つ一重ループ L,
3. ループ L の制御変数の線型式を添字とする A の要素を左辺に持つループ L 中の代入文 S.

□

図 4.4 は RDLS(1,1) パターンの例である．条件 (1) は文 1 から 5 で，条件 (2) は文 6 で，条件 (3) は文 7 で指定される．尚，図 4.4 に現れる変数はコンパイル時に定数である必要はない．

RDLS(1, 1) パターンは多くの科学技術計算プログラムの基本となる．なぜなら，d 重ループに含まれる代入文の左辺に現れる d 次元配列において，各次元が異なるループ制御変数を 1 つだけ含むというよく出現するパターンは次元ごとに RDLS(1, 1) パターンになっているからである．

4.1.3 図式

図式とは集合及び集合間の写像を見通しよく表現する 1 つの方法 [8] である．以下は最も簡単な例である．

$$X \xrightarrow{\phi} Y \xrightarrow{\sigma} Z \quad (\text{exact}) \qquad (*)$$

図式 (*) において，X，Y，Z は集合を，ϕ，σ は写像を表わす．本節では集合として整数全体（$\mathbf{Z}$）の部分集合，または，ある整数 a の剰余集合 $\mathbf{Z}_a = \{\overline{0}, \overline{1}, \cdots, \overline{a-1}\}$ を考える．

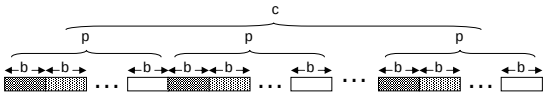


図 4.5: Block-cyclic 分散.

図式の特別なものに完全系列がある．図式 (*) 中の ϕ を線形写像，$\mathrm{Im}\,\phi = \phi(X)$, $\mathrm{Ker}\,\phi = \{x \in X | \phi(x) = 0\}$ と定義する．但し，Y は 0 を含むとする．このとき，もし，$\mathrm{Im}\,\phi = \mathrm{Ker}\,\sigma$ が成立するならば，図式 (*) は完全系列と呼ばれ，図式の右に (exact) と書く．

X，Y，Z を群，ϕ 等を準同型写像とするとき，完全系列の重要な例として以下がある：

$$0 \xrightarrow{i} X \xrightarrow{\phi} Y \xrightarrow{\sigma} Z \xrightarrow{j} 0 \quad (\mathrm{exact}) \tag{**}$$

ここで，完全系列の定義より ϕ は単射，σ は全射，$Y/\mathrm{Im}\,\phi \cong Z$ となる．また，$\sigma \circ \eta = id$ となる線形写像 $\eta : Z \mapsto Y$ があれば Y は直和分解される [8]：

$$Y \cong Y_1 \oplus Y_2. \tag{***}$$

4.1.4　データ分散の図式表現

本節では ALIGN や規則的データ分散に対する DISTRIBUTE を完全系列を含む図式 [8] で表現する．

テンプレートのデータ分散の図式表現

本節では，図 4.4 における，下限値 T_l と要素数 e を持つテンプレート T のデータ分散を図式で表現する．以降，テンプレートや配列と，それらの添字から成る集合とを同一視する．

標準テンプレートへの埋めこみ　テンプレート T の要素数 e が図 4.4 中のプロセッサ数 p やブロックサイズ b で割りきれない場合，データ分散の取り扱いが煩雑になる．そこで，以降では T のかわりにその要素数がこれらで割り切れるようなより大きな添字区間を扱い，T に特有な性質は別に取り扱う．

定義 5 標準テンプレート
$c = \lceil e/pb \rceil$ とする．この時，$I_{pbc} = [0 : pbc - 1]$ を T に対する標準テンプレートと呼ぶ．

□

次に，T のデータ分散は，T と I_{pbc} との関係，及び，I_{pbc} のデータ分散の 2 つに分解できることを示す．まず，$pbc \geq e$ が成り立つので，T は標準テンプレート I_{pbc} の部分集合とみなせ，T から I_{pbc} の中へ写像

$$g : T \ni x \mapsto x - T_l \in I_{pbc} \tag{4.1}$$

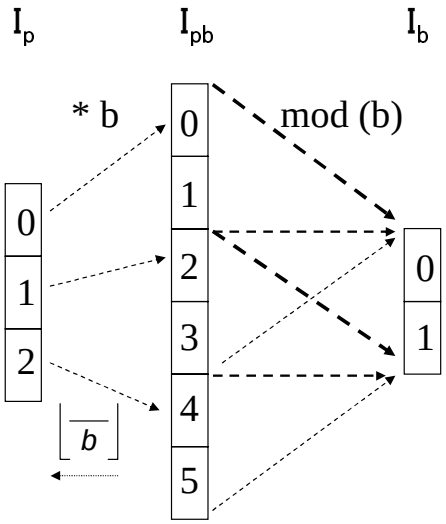


図 4.6: I_p, I_{pb}, 及び I_b の間の関係（$p = 3, b = 2$）.

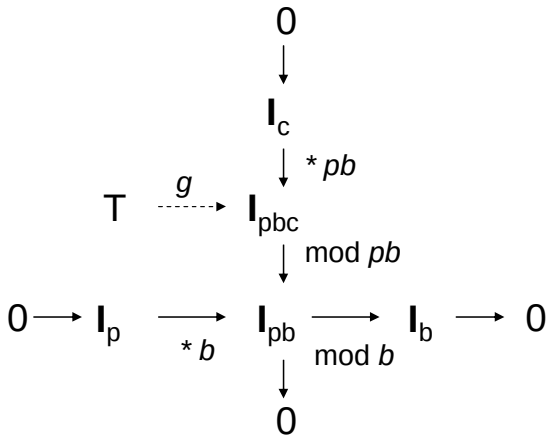


図 4.7: テンプレートに対するデータ分散の図式表現.

が定義できる．次に，I_{pbc} に対して T と同じ指示文

$$\text{DISTRIBUTE } I_{pbc} \text{ (CYCLIC}(b)\text{) ONTO } P \tag{4.2}$$

を指示すると，図 4.4 による $x \in T$ のマッピングは式 (4.1) と (4.2) の合成による x のマッピングと等しくなることが容易にわかる．よって，以降では，テンプレート T のデータ分散のかわりに式 (4.1) と (4.2) のペアを扱う.

標準テンプレートの図式表現と区間表現　本節では標準テンプレートの図式表現を考察する．指示文 (4.2) によって分散されるデータを，同じプロセッサにマッピングされるもの別に標準テンプレート I_{pbc} 上で表現すると，b 個の連続要素から成る区間（**ブロック区間**と呼ぶ）が p 個だけ連続的に並び，これら bp 個の連続要素列が c 個並ぶことになる（図 4.5）．そこで，ブロック区間を $I_b = [0 : b-1]$，bp 個の連続要素列を $I_{pb} = [0 : bp-1]$，p 個のプロセッサ列を $I_p = [0 : p-1]$ 等の区間でモデル化する．以下，これらの区間とそれらの間の関係を図式で表現する．表現方法は色々考えられる

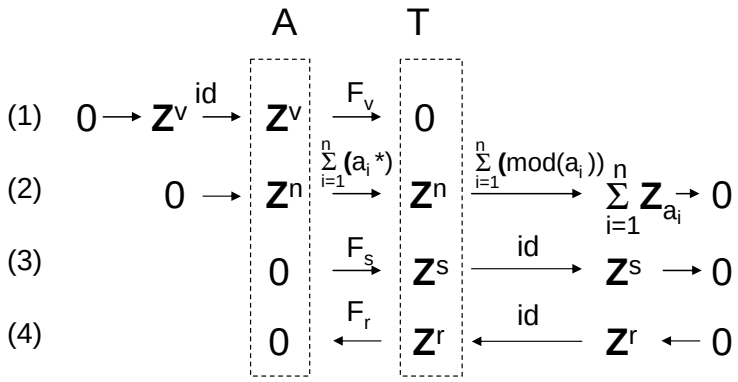


図 4.8: 各 AXIS_TYPE に対する図式表現.

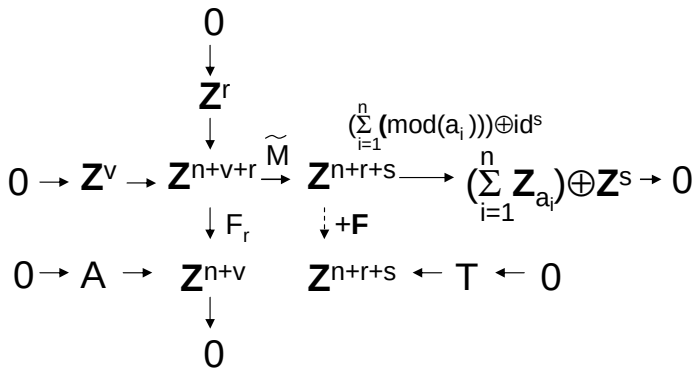


図 4.9: アラインメントの図式表現.

が，ブロック区間 I_b 中の添字が I_{pb} において連続と解釈される自然なモデルを選んだ．これは，データの連続性による通信最適化等への応用を考えたためである．

$\mathbf{Z}_b$，$\mathbf{Z}_p$，及び $\mathbf{Z}_{pb}$ の間には写 '$\bmod b$' と 'b 倍' で結ばれた完全系列がある [8]．これのアナロジーとして，区間 I_b 等に対して以下の完全系列を考える：

$$0 \longrightarrow I_p \xrightarrow{*b} I_{pb} \xrightarrow{\bmod b} I_b \longrightarrow 0 \quad (\text{exact}) \tag{4.3}$$

図式 (4.3) では，'$\bmod b$' で $I_b = [0 : b-1]$ に写されるブロック区間 $[0 : b-1], [b : 2b-1], \cdots$ は連続要素から構成されるので添字の連続性をモデル化できている．次に，'b 倍' によって I_p の要素 $0, 1, \cdots, p-1$ には I_{pb} の各ブロック区間の先頭要素 $0, b, 2b, \cdots$ が対応する．このことは，逆に，これら先頭要素が I_p の各要素，即ち，プロセッサ $0, 1, \cdots, p-1$ にマッピングされると解釈できる．さらに，ブロック区間内の他の要素も同じプロセッサに写されると解釈する．即ち，I_{pb} の元 x は写像 $x \mapsto \lfloor x/b \rfloor$ で計算できるプロセッサ番号にマッピングされる．以上のことは I_{pb} にブロックサイズ b のブロック分散が適用されたことを意味する（図 4.6）．さらに，図式 (4.3)，$h : I_b \ni x \mapsto x \in I_{pb}$，及び，図式 (***) より以下の直和表現が得られる：

$$I_{pb} = \bigcup_{q=0}^{p-1} (bq + I_b). \tag{4.4}$$

ここで，$\bigcup$ は disjoint な和集合を表わし，$x + I_b$ を $[x : x + b - 1]$ と定義する．同様にして，以下の図式 (4.5) および直和表現が得られる．図式 (4.5) は I_{pbc} において I_{pb} が連続的に並ぶこと，即ち，図式 (4.3) と合わせると I_{pbc} が $\text{cyclic}(b)$ でデータ分散されることを表わす．

$$0 \longrightarrow I_c \xrightarrow{*pb} I_{pbc} \xrightarrow{\bmod pb} I_{pb} \longrightarrow 0 \quad (\text{exact}) \tag{4.5}$$

$$I_{pbc} = \bigcup_{j=0}^{c-1} \bigcup_{q=0}^{p-1} (pbj + bq + I_b). \tag{4.6}$$

以降，式 (4.6) の j の値を**分散周期**と呼ぶ．式 (4.1) 及び図式 (4.3) と (4.5) を I_{pb} で交差させることによって図 4.7 の図式を得る．図中，点線矢印は線形でない写像を表わす．また，図 4.7 及び以降では '(exact)' は省略する．

アラインメントの図式表現

本節では D_A 次元配列 A と D_T 次元テンプレート T から成る以下の ALIGN を図式で表現する．

$$\text{ALIGN } A(\mathbf{I}) \text{ WITH } T(M\mathbf{I} + \mathbf{F} + \mathbf{J}) \tag{4.7}$$

ここで，$\mathbf{I}$ は D_A 次元ベクトルで各次元は異なるインデックス変数から成る．$\mathbf{J}$ と $\mathbf{F}$ は D_T 次元ベクトルで，$\mathbf{J}$ は REPLICATED な次元にインデックスを持ち，他の次元は 0；$\mathbf{F}$ は NORMAL 及び SINGLE な次元における定数項を対応する次元に含み，他の

次元は0；$\mathbf{M}$はアライン添字を表現する(D_T, D_A)型整数行列とする．これらは図4.3に対して以下となる：

$$\mathbf{I} = \begin{pmatrix} i \\ j \end{pmatrix}, \mathbf{J} = \begin{pmatrix} 0 \\ k \\ 0 \end{pmatrix}, \mathbf{F} = \begin{pmatrix} 1 \\ 0 \\ 3 \end{pmatrix}, \mathbf{M} = \begin{pmatrix} 2 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

一方，AXIS_TYPEがNORMAL，REPLICATED，SINGLE，VANISHEDとなる次元数を，各々，n，r，s，vとすると，以下が成立する：

$$n + v = D_A, \quad n + r + s = D_T.$$

よって，配列添字集合を$\mathbf{Z}$の部分集合と見なすと，

$$A \subset \mathbf{Z}^{n+v}, \qquad T \subset \mathbf{Z}^{n+r+s}. \tag{4.8}$$

図4.8は配列AおよびテンプレートTの次元をAXIS_TYPE別にまとめて得た4行の完全系列である．（1）から（4）は各々，VANISHED，NORMAL，SINGLE，及びREPLICATEDを表現する．式(4.8)より，A及びTは，図4.8中の左及び右の点線枠内の$\mathbf{Z}$加群の直和の部分集合として表現される[3]．ここで，idは恒等写像，F_v(又はF_r)は全ての元を0に写す全射を表わす．以下，図4.8における(1)から(4)の各図式を説明する．

1. テンプレートTの任意の要素に対して$\mathbf{Z}^v$の全ての元がアラインすることを表現する．
2. 行列$\mathbf{M}$はHPFの規定より各行の非零要素は高々1つ．即ち，NORMALな次元に対し$\mathbf{M}$は定数倍写像である．それをa_i倍と表わすと，剰余集合$\mathbf{Z}_{a_i}$を導入して上記完全系列を得る．
3. 配列AからテンプレートTへの写像F_sが0に一定値0を対応させることから，テンプレートTにおいて1つの定数ベクトルが対応することを表現する．定数は図4.9で表現される．
4. 配列Aの任意の要素に対して$\mathbf{Z}^r$の全ての元が対応することを表現する．逆に考えると，配列Aの任意の要素が$\mathbf{Z}^r$の全ての要素にコピーされることを表わす．

図4.9の図式は図4.8の完全系列をつないで得られる．即ち，1行目は（1），(2)，と(3)から，1列目は（1）と（4），及び，$id : \mathbf{Z}^n \mapsto \mathbf{Z}^n$から得られる．ここで，$\mathbf{F}$は指示文(4.7)の$\mathbf{F}$を，$\tilde{\mathbf{M}}$は$\mathbf{Z}^r$に対応する次元に恒等写像が適用されるように$\mathbf{M}$に列を追加した行列を表わす．図4.3に対して$\tilde{\mathbf{M}}$は以下となる：

$$\tilde{\mathbf{M}} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

[3]図4.8は式(4.7)の項$\mathbf{F}$を含んでない．これは図4.9で含める．

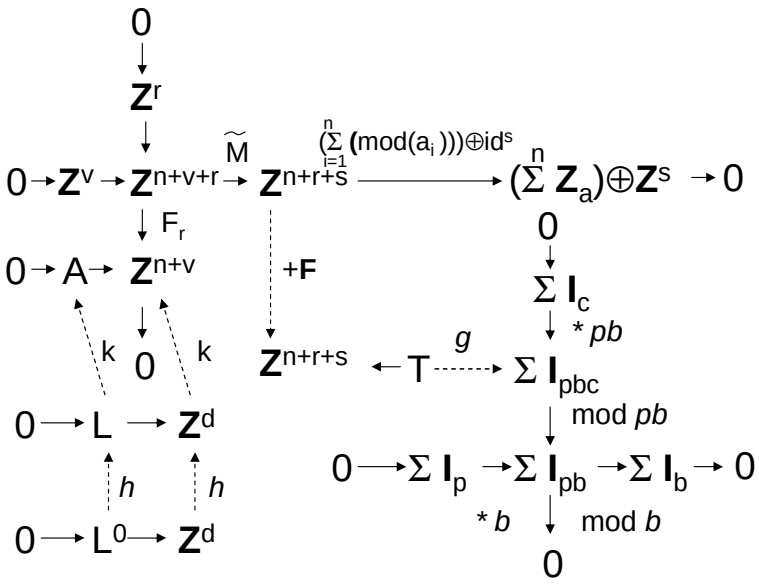


図 4.10: 一般のデータ分散の図式表現.

一般の場合の図式表現

図 4.10 は多次元配列 A に対する HPF の規則的データ分散の図式表現である．この図式の左下以外の部分は図 4.7 中の各区間を次元毎に直和を取ったものと図 4.9 の図式をつないで得られる．左下部分において，L は d 重ループによるループ繰り返し空間 $[l:u:m]^d \subset \mathbf{Z}^d$，$L^0$ は標準ループによるループ繰り返し空間 $[0:U:1]^d \subset \mathbf{Z}^d$，$h$ は L^0 から L への写像 $h : x \mapsto m * x + l$，k はループ制御変数 i に配列 A の添字を対応させる写像 $k : i \mapsto i_a i + i_b$ を表わす．ここで，記号 $[x:y:z]$ は，左からループ下限値，ループ上限値，ストライドを表わす．

尚，以降では 1 つのループ制御変数は 1 つの配列次元に現われ，1 つの配列次元には 1 つのループ制御変数が現れる場合のみ扱う．一般の場合は別に報告する．

4.1.5 計算分散公式

本節では計算分散の対象となる AXIS_TYPE は NORMAL と SINGLE のみであることを示し，これらの AXIS_TYPE を持つ RDLS(1,1) パターンに対する計算分散を与える．

配列 A とテンプレート T においてループの計算分散に関連する次元は，ループ制御変数が現われる配列 A の次元，及び，その次元がアラインするテンプレート T の次元である．このことから以下の次元は計算分散に無関係なことがわかる．

- AXIS_TYPE が REPLICATED な T の次元：
  全ての配列 A の要素に対応するため．したがって，全てのループ範囲にも対応するので，ループは分散されず，元のままである．

- AXIS_TYPE が VANISHED な A の次元：
  対応するテンプレート T の次元を持たないため．この次元に制御変数を含むルー

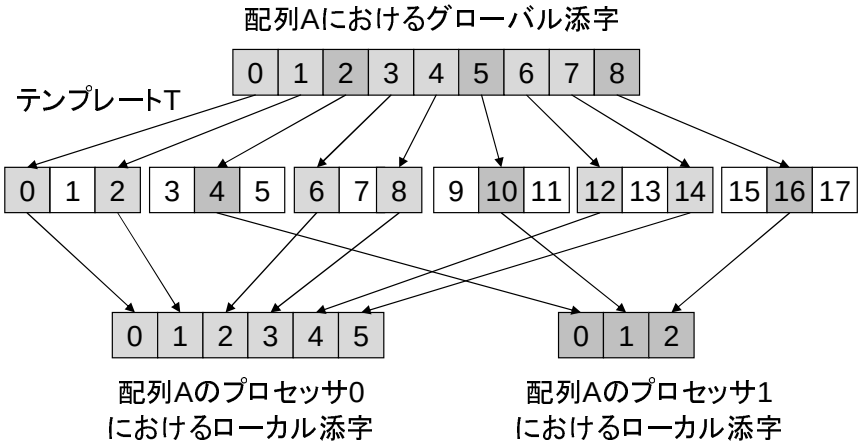


図 4.11: グローバル添字とローカル添字の対応の例.

プは分散されず，元のままである.

よって，計算分散の対象となる AXIS_TYPE は NORMAL と SINGLE のみであり，図式を用いて計算分散を考えるには，図 4.10 を単純化，即ち，図 4.10 において $r = v = 0$ とすればよい．この時，F_r =id となるので，L^0 と I_{pbc} の間を結ぶ図式は以下となる[4]：

$$L^0 \xrightarrow{h} L \xrightarrow{k} A \xrightarrow{f} T \xrightarrow{g} I_{pbc}$$

ここで，$f : i \mapsto f_a i + f_b$ は $\tilde{\mathbf{M}}$ と $\mathbf{F}$ を合成したもので，アライン添字に対応する.

また，図 4.4 のプログラムは NORMAL と SINGLE の AXIS_TYPE を表現することを注意しておく．よって，以下では，図 4.4 のプログラムの計算分散を与える．その前に，以降で用いる記号を定義する.

定義 6 記号

$\psi(i)$: $A(i)$ がマッピングされるプロセッサ番号.

$\pi :=$ $f_a i_a m$.

$t(j,q,x) :=$ $(pbj + bq + x - gfk(l))/\pi$.

$\mathcal{L}(i)$: グローバル添字 i に対するローカル添字. □

ここでローカル配列とは，データ分散によりあるプロセッサに割り付けられる配列要素群を，そのローカルメモリ上に連続して並ぶように圧縮配置させた配列を指し，ローカル添字とはローカル配列における添字を指す．また，以降，ローカル配列の最小添字は元のグローバル配列の最小添字と同じとする．図 4.11 はグローバル添字とローカル添字の対応の例である．この図では，配列 $A(9)$ をテンプレート $T(18)$ に 'ALIGN $A(i)$ WITH $T(2*i)$' によってアラインし，テンプレート T をプロセッサ配列 $P(2)$ へ 'DISTRIBUTE $T(block(3))$ ONTO P' によって分散したときの配列 A のグローバル添字とプロセッサ 0 及び 1 における配列 A のローカル添字との対応を表す.

[4] この図式は完全系列ではない.

定理 5 図 *4.4* のループ L に対する計算分散コードは以下となる.
(A) $i_a f_a = 0$ の時

$$
\begin{array}{l}
\quad if\ (\psi(i_b) = mype) \\
\quad\quad do\ i = l, u, m \\
\quad\quad\quad A(i_a i + i_b) = \cdots \\
\quad\quad enddo \\
\quad endif
\end{array}
$$

(B) $i_a f_a \neq 0$ の時

$$
\begin{array}{ll}
 & j = A_l; \quad q = mype \\
L1: & do\ i = A_l, A_u, 1 \qquad // \ Get\ \mathcal{L} \\
 & \quad if\ (\psi(i) = q) \quad \mathcal{L}(i) = j{+}{+} \\
 & enddo \\
 & C_l = \lfloor gfk(l)/pb \rfloor; C_u = \lfloor gfk(u)/pb \rfloor \\
L2: & do\ j = C_l, C_u, sign(\pi) * 1 \\
 & \quad S(j, q) \\
L3: & \quad do\ i = L(j, q), U(j, q), m \\
 & \quad\quad A(\mathcal{L}(i_a i + i_b)) = \cdots \\
 & \quad enddo;\ enddo
\end{array}
$$

但し，$S(j, q)$ は以下のコードである.

$$
\begin{array}{ll}
if\ (\pi > 0) & L(j, q) = \max(m\lceil t(j, q, 0) \rceil + l, l) \\
 & U(j, q) = \min(m\lfloor t(j, q, b-1) \rfloor + l, u) \\
else & L(j, q) = \min(m\lceil t(j, q, b-1) \rceil + l, l) \\
 & U(j, q) = \max(m\lfloor t(j, q, 0) \rfloor + l, u)
\end{array}
$$

証明 付録参照. □

4.1.6 最適計算分散コード

本節では，ある配列 A に ALIGN 指示文が指定され，かつ，その align-target が block-cyclic 分散されるという条件の下でも，$\pi | b$，即ち，ブロックサイズ b が $\pi = f_a i_a m$ で割り切れる場合には最適なコードが生成できることを示す．これを示す準備として，以下を順に説明する.

- プロセッサ q にマッピングされる配列 A の最小要素,
- 配列 A のデータ分散前の添字（グローバル添字）をデータ分散後の添字（ローカル添字）に写す写像,
- データ分散前の配列・ループとデータ分散後の配列・ループとの関係（計算分散の図式表現）.

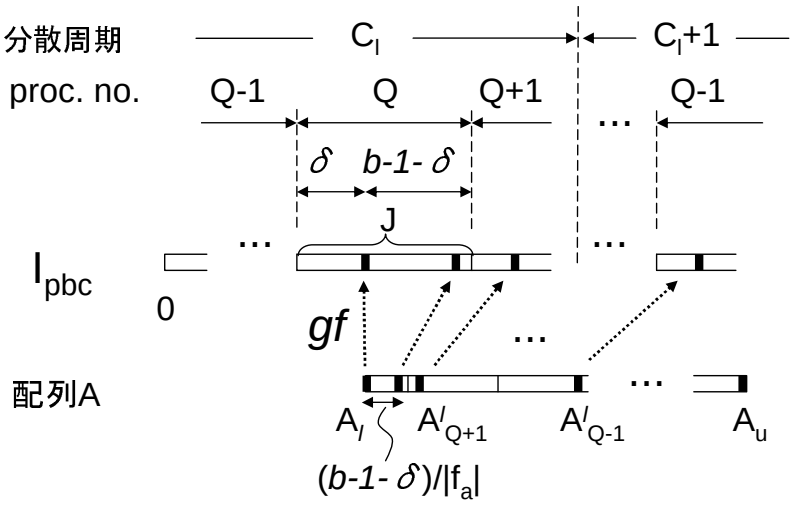


図 4.12: 配列添字の I_{pbc} における像.

integer function GetMin(ϕ, x, y, z)
$Q = \lfloor (\phi(x) \bmod pb)/b \rfloor$
if $(q = Q)$ return (x)
else
  $\delta = \phi(x) \bmod b$
  if $(y > 0)$ $\delta' = b - 1 - \delta$; $\Delta_q = q - Q$
  else $\delta' = \delta$; $\Delta_q = Q - q$
  $\Delta_i = \lfloor \delta'/|y| \rfloor + 1 + \{(\Delta_q \bmod p) - 1\} b/|y|$
  return$(x + \Delta_i z)$
endif

図 4.13: 最小値取得関数 GetMin.

各プロセッサでの配列の下限添字

データ分散前の配列 A においてプロセッサ q にマッピングされる部分集合を A_q，データ分散後，プロセッサ q にマッピングされる配列を $\tilde{A}$，A_q の下限添字を A_q^l とする．この時，A_q^l は以下のように計算できる．

補題 1 *$f_a | b$ の時，A_q^l は図 4.13 の関数 GetMin を用いて以下のように計算できる：*

$$A_q^l = GetMin(gf, A_l, f_a, 1).$$

証明 図 4.12 を用いて説明する．$gf(x) = f_a x + f_b - T_l \in I_{pbc}$ による配列 A の下限値 A_l の像を $A'_l = gf(A_l)$ とする．A'_l がマッピングされるプロセッサ番号を Q とすると，$Q = \lfloor A'_l \bmod pb/b \rfloor$ となる（図 4.7）．A'_l が含まれるブロック区間 J の先頭から A'_l までの要素数を δ とすると，$\delta = A'_l \bmod b$ であり，A'_l からブロック区間 J の最後の要素までの要素数は $b - 1 - \delta$ となる．

まず，プロセッサ番号 $q = Q$ なら明らかに $A_q^l = A_l$.

次に，アファイン写像 gf の 1 次係数 $f_a > 0$ とする．この時，図 4.12 において，配列添字の増加方向（右）と配列添字の gf による像の標準テンプレート I_{pbc} における増加方向（右）は一致する．したがって，その像がブロック区間 J の最後の要素になるような配列添字は $X = A_l + \lfloor (b-1-\delta)/|f_a| \rfloor$ となる．$f_a | b$ より $f_a < b$．よって，$gf(X+1) = gf(X) + f_a$ は区間 J を越えて，隣接するブロック区間に含まれ（但し，$f_a < b$ よりこの区間を越えることは無い），$Q+1$ にマッピングされる最初の A の要素となる．したがって，$q = Q+1$ に対して，$A^l_q = X+1$ となる．また，$f_a | b$ より $Q+2$ 以降のプロセッサに対する A^l_q は $X+1$ に $b/|f_a| \in \mathbf{Z}$ を順次加えることにより得られる．一方，$q < Q$ となるプロセッサ番号 q に対しては，図 4.12 より q が属する分散周期の次の分散周期に最小値を持つので，$q + p\,(> Q)$ をプロセッサ番号と見なすことにより上記と同様の計算が適用できる．

また，$f_a < 0$ の場合は，図 4.12 における配列添字の増加方向（右）と配列添字の gf による像の増加方向（左）は逆転する．この時，その像がブロック区間 J の最後の要素になるような配列添字は $X = A_l + \lfloor \delta/|f_a| \rfloor$ となる．後は上記と同様にして A^l_q の値が得られる．

以上の結果を整理すると，図 4.13 の関数 GetMin を用いて $A^l_q = \text{GetMin}\ (gf,\ A_l,\ f_a,\ 1)$ が得られる． □

ローカルアドレス表現

本節では，配列 A のグローバル添字をローカル添字に写す写像を定式化し，次にこれを求める．

定義 7 X（と Y）をグローバルな（ローカルな）インデックスで表現される $\mathbf{Z}$ の部分集合とする．この時，以下を満たす写像 $\phi : X \ni x \mapsto y \in Y$ を X のローカルアドレス表現と呼ぶ：

1. ϕ は狭義単調増加: $x < y \Rightarrow \phi(x) < \phi(y)$,
2. $\phi(X)$ は連続な区間.

□

補題 2 ρ を標準テンプレート I_{pbc} にローカルな添字 I_{bc} を対応させる写像，$\Delta_A = (gf)^{-1}\rho(gf)(A^l_q) - A_l$，$\tilde{f}(x) = f(x + \Delta_A)$，$\tilde{g} = g$ とする．$f_a | b$ の時，$\alpha = (\tilde{g}\tilde{f})^{-1}\rho(gf)$ は配列 A のローカルアドレス表現であり，$\alpha(A^l_q) = A_l$ となる．

証明 $\omega : x \mapsto x + \Delta_A$ とすると $\tilde{f} = f \circ \omega$ より $\tilde{f}^{-1} = \omega^{-1} \circ f^{-1}$．よって，$\alpha(x) = (gf)^{-1}\rho(gf)(x) - \Delta_A$．したがって，$\alpha(A^l_q) = A_l$．次に，$\Delta_A$ は定数なので，$\epsilon = gf$ として，$E(x) = \epsilon^{-1}\rho\epsilon(x)$ がローカル・アドレス表現であることを示せばよい．式 (4.4) と (4.6) より，

$$\rho : x = pbj + bq + y \mapsto bj + y$$

となるので，

$$j = \lfloor x/pb \rfloor, \quad y = x \bmod b, \quad \rho(x) = \lfloor x/pb \rfloor b + x \bmod b.$$

また，$\epsilon(x)$ を b で割った商は以下のようにも書ける．

$$\lfloor \epsilon(x)/b \rfloor = (\epsilon(x) - \epsilon(x) \bmod b)/b.$$

以上の 2 式より，

$$\begin{aligned} E(x) &= \epsilon^{-1}(\lfloor \epsilon(x)/pb \rfloor b + \epsilon(x) \bmod b) \\ &= \{\lfloor \epsilon(x)/pb \rfloor - \lfloor \epsilon(x)/b \rfloor\} b/f_a + x \in \mathbf{Z}. \end{aligned}$$

ϵ と ϵ^{-1} は 1 次式なので単調．$\epsilon(A_q)$ は I_{pbc} の中でプロセッサ q にマッピングされる部分 $I_q = pbJ + bq + I_b$（但し，J はある区間）に含まれる．ρ をこの I_q に制限したもの

$$\rho|_{I_q} : pbJ + bq + I_b \mapsto bJ + I_b$$

は単調増加．よって，α は単調増加．次に，A_q は以下の形で表現できる：

$$[A_q^l \ : \ B] \cup \bigcup_{i=1}^{N-1} [C + pbi/|f_a| \ : \ B + pbi/|f_a|] \cup [C + pbN/|f_a| \ : \ D].$$

ここで，図 4.12 より $f_a > 0$ の時

$$\begin{gathered} C = A_q^l - \lfloor \delta/f_a \rfloor, \quad B = C + b/f_a - 1, \\ D : A_q\text{に含まれる最大の添字}, \quad \delta : \text{補題 1 の}\delta. \end{gathered}$$

α が狭義単調増加なので，$\alpha(A_q)$ の連続性は，より大きな以下の区間に対して証明すればよい：

$$A'_q = \bigcup_{i=0}^{N} [C + pbi/|f_a| : B + pbi/|f_a|].$$

A'_q は

$$A'_q = [0 : N : 1] * pb/f_a + C + [0 : b/f_a - 1 : 1]$$

とも書けるので

$$gf(A'_q) = [0 : N : 1] * pb + gf(C) + [0 : b - f_a : f_a].$$

よって，

$$E(A'_q) = [0 : (N+1)b/f_a - 1 : 1] + E(C).$$

したがって，$\alpha(A_q)$ はストライド 1 の単一区間，即ち，連続領域となる．$f_a < 0$ でも同様．

□

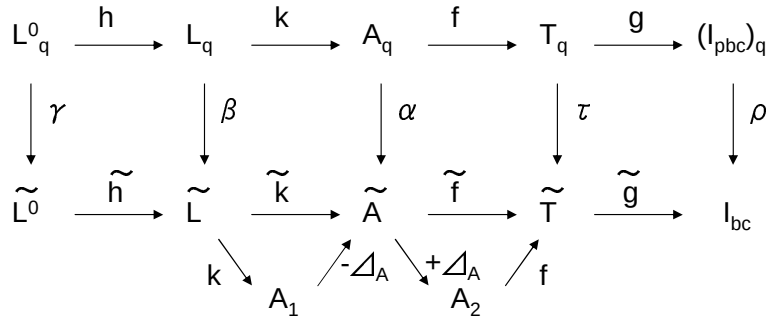


図 4.14: 計算分散の図式表現.

計算分散の図式表現と最適計算分散コード

本節では計算分散の図式表現を示し，最適計算分散コードを導く.

補題 3 （グローバル・ローカルアドレス間の関係）
$\tilde{h} = h$, $\tilde{k} = k - \Delta_A$, $L_q = \{i \in L | k(i) \in A_q\}$, $L^0_q = \{i \in L^0 | h(i) \in L_q\}$, $\beta = \tilde{k^{-1}} \alpha k$, $\gamma = \tilde{h^{-1}} \beta h$ とする．また，A_q 等の α 等による像を $\tilde{A}_q$ 等と記す．$\pi | b$ の時，図 *4.14* の図式における α，β，γ は各々，A_q，L_q，L^0_q に対するローカルアドレス表現を与える.

証明 $\tilde{f}\tilde{k} = fk$, $\tilde{f}\tilde{k}\tilde{h} = fkh$ を用いれば，補題 2 と同様に証明される. □
以上より最適計算分散コードは次の形で述べられる.

定理 6 $\pi | b$ の時，図 *4.4* のループ L を計算分散した結果は以下の 1 重ループで与えられる.

$$
\begin{aligned}
&A^l_q = GetMin(gf, A_l, f_a, 1)\\
&\Delta_A = \alpha(A^l_q) - A_l\\
&l_1 = GetMin(gfk, l, \pi, m)\\
&L_c = \{\lfloor gfk(l_1)/pb \rfloor - \lfloor gfk(l_1)/b \rfloor\} b/f_a i_a + l_1\\
&u' = l + \lfloor |u - l|/|m| \rfloor * m\\
&u_1 = GetMin(gfk, u', -\pi, -m)\\
&U_c = \{\lfloor gfk(u_1)/pb \rfloor - \lfloor gfk(u_1)/b \rfloor\} b/f_a i_a + u_1\\
&do\ i = L_c, U_c, m\\
&\quad A(i_a i + i_b - \Delta_A) = \cdots\\
&enddo
\end{aligned}
$$

証明 補題 3 より $\tilde{L^0_q} = \gamma(L^0_q)$ はローカルなループ空間 $\tilde{L^0}$ における連続区間；即ち，ストライド 1 の 1 重ループのインデックス空間を表現する．よって，ストライド m の 1 重ループ $L = h(L^0_q)$ の β による像 $\beta(h(L^0_q)) = \tilde{h}(\tilde{L^0_q})$ もローカルなループ空間 $\tilde{L}$ におけるストライド m の 1 重ループになる.
次に，$\beta(i) = \tilde{i}$ は $\tilde{L_q}$ のループ制御変数を表わす．よって，配列添字 $i_a i + i_b$ の α による像 $\alpha(i_a i + i_b)$ は $\alpha(k(i)) = \tilde{k}(\beta(i)) = i_a \tilde{i} + i_b - \Delta_A$ となる.

最後に，ループ下限値 L_c に対しては補題 1 の証明と同様にして

$$l_1 = \mathrm{GetMin}(gfk, l, \pi, m)$$

が証明でき，$L_c = \beta(l_1)$ を得る．上限値 U_c に対してもループ制御変数の最終値 u' を用いて L_c と同様に求められる． □

尚，$\pi \not| b$ だが $\pi | 2^n b$ かつ $2^n | p$ の時，2^n-way の SMP ノードから成るクラスタ向けに最適な階層並列コードを生成することができる．即ち，p プロセッサ上の cyclic(b) 分散を $p/2^n$ プロセッサ上の cyclic$(2^n b)$ 分散に変更して定理 6 のコードを生成し，その生成コードに 2^n 個のスレッドから成る SMP 向けスレッド並列化を適用すれば良い．

4.1.7 擬周期性とテーブル参照法コード

本節では，block-cyclic 分散に対する計算分散コードにおける添字参照が，従来 [3] よりも小さい周期を持つことを示し，do ループを用いた計算分散コードを導く．以下，$f_a i_a \neq 0$ とする．

参照添字の擬周期性

本節では図 4.4 のループ L の添字参照の擬周期性とローカル添字が持つ性質を述べる．

補題 4

$$\lambda = LCM(pb, \pi), \quad \kappa = \lambda/\pi \tag{4.9}$$

とすると，図 *4.4* のループ L の添字参照において以下の周期性が成立する；即ち，ループインスタンス $L(i)$ と $L(i + m\kappa)$ は同じプロセッサで実行される：

$$\psi(i_a(i + m\kappa) + i_b) = \psi(i_a i + i_b). \tag{4.10}$$

証明 $f_a i_a m\kappa \bmod pb = \pi\kappa \bmod pb = \lambda \bmod pb = 0$ より明らか． □

よって，$[i : i + m\kappa - 1 : m]$ の範囲でプロセッサ q で実行されるループ制御変数を見つければ，プロセッサ q で実行される任意のループ制御変数はその値と $m\kappa$ の定数倍との和で表わされる．

補題 4 の周期は従来，示されていた周期 [3] よりも短い．従来，文献 [3] では各プロセッサに対して b 個の参照ごとに周期を持つとしていた．この時，全プロセッサに対する周期の合計は pb となる．一方，補題 4 は，全プロセッサに対して第 1 回目の周期はストライド m を持つループ範囲 $[i : i + m\kappa - 1 : m]$ に含まれることを主張する．この時，全プロセッサに対する周期の合計は $\kappa = LCM(pb, \pi)/\pi \leq pb$ となる．したがって，本周期は従来の周期よりも短い．

また，ローカル添字を現す配列 $\mathcal{L}$ は以下を満たす．

補題 5 I をループ区間 $[l : u : m]$，$l_q \in I$ をループインスタンス $L(l_q)$ がプロセッサ q で実行されるような最初の値，G_q^l を $\mathcal{L}(i_a(l_q + m\kappa) + i_b) - \mathcal{L}(i_a l_q + i_b)$ とする．この時，ループインスタンス $L(i_q)$ がプロセッサ q で実行されるような任意の i_q に対して，

$$G_q^i = G_q^l.$$

証明 式 (4.10) より以下が容易に得られる：

$$\begin{aligned} G_q^i &= \mathcal{L}(i_a(i_q + m\kappa) + i_b) - \mathcal{L}(i_a(l_q + m\kappa) + i_b) \\ &\quad + \mathcal{L}(i_a(l_q + m\kappa) + i_b) - \mathcal{L}(i_a i_q + i_b) \\ &= \mathcal{L}(i_a i_q + i_b) - \mathcal{L}(i_a l_q + i_b) \\ &\quad + \mathcal{L}(i_a(l_q + m\kappa) + i_b) - \mathcal{L}(i_a i_q + i_b) = G_q^l. \end{aligned}$$

□

よって，任意の i に対してある定数 G_q が存在して，

$$\mathcal{L}(i_a(i + m\kappa) + i_b) = \mathcal{L}(i_a i + i_b) + G_q.$$

となる．この式を参照添字の**擬周期性**と呼ぶ.

テーブル参照法コード

前節の結果より以下の定理が成り立つ.

定理 7 図 *4.4* のループ L の計算分散コードは以下となる.

$N_e = 0$
$do\ i = l, l + m\kappa - 1, m$ // *inspector*
  $if\ (\psi(i_a i + i_b) = q) i_x(N_e{+}{+}) = \mathcal{L}(i_a i + i_b)$
$enddo$
$do\ i = l + m\kappa, l + 2m\kappa - 1, m$ // *get* G_q
  $if\ (\psi(i_a i + i_b) = q) g = \mathcal{L}(i_a i + i_b);\ exit$
$enddo$
$G_q = g - i_x(0); \quad N_c = \lfloor (u - l + 1)/m\kappa \rfloor$
$do\ i = 0, N_c - 1$ // *executor*
  $do\ j = 0, N_e - 1$
    $A(i_x(j) + iG_q) = \cdots$
$enddo;\ enddo$
j=0
$do\ i = l + N_c m\kappa, u, m$ // *residue loop*
  $if\ (\psi(i_a i + i_b) = q) A(i_x(j{+}{+}) + N_c G_q) = \cdots$
$enddo$

証明 擬周期性と補題 5 より明らか. □

また，$N_e^- = N_e - 1$ とし，配列 Γ と Δ を

$$\Gamma(i_x(i)) = \begin{cases} i_x(i+1) & \text{if} \quad 0 \le i < N_e^- \\ i_x(0) & \text{if} \quad i = N_e^- \end{cases}$$

$$\Delta(i_x(i)) = \begin{cases} i_x(i+1) - i_x(i) & \text{if} \quad 0 \le i < N_e^- \\ i_x(0) - i_x(i) + G_q & \text{if} \quad i = N_e^- \end{cases}$$

と定義すると，定理 7 の結果は従来タイプのテーブル参照法コード [3],[10] になることが容易にわかる.

表 4.2: 実測用パラメータ.

$p = 8, \quad i_a = -2, \quad i_b = 7,$	$f_a = -3, \quad f_b = 1,$
$l = 100,000 * (-m) * p,$	$u = -m,$
$A_l = i_a l + i_b,$	$A_u = i_a u + i_b,$
$T_l = f_a A_u + f_b,$	$T_u = f_a A_l + f_b.$

表 4.3: 適用された最適化.

rr	if-conversion, pseudo-vectorization
th	(to inner loop) pseudo-vectorization
t-w	if-conversion
t-d	(to inner loop) 2× loop unrolling
t-2d	(to inner loop) 2× loop unrolling, pseudo-vec.
opt	4× loop unrolling

系 1 定理 7の *executor* 及び *residue* ループは以下の従来タイプのテーブル参照法コードになる：

$N_r = 0$
do $j = l + N_c m\kappa, u, m$ *// count residues*
  if $(\psi(i_a j + i_b) = q) N_r{+}{+}$
enddo
$i = i_x(0); \quad \delta = i_x(0); \quad j = 0$
while $(j < N_c N_e + N_r)$ *// executor*
  $A(i) = \cdots; \quad j{+}{+}$
  $i{+}{=} \Delta(\delta); \quad \delta = \Gamma(\delta)$
endwhile

4.1.8 評価

種々の計算分散法と性能を比較した．対象プログラムは表 4.2 に示すパラメータを持つ図 4.4 のプログラムである．但し，ブロックサイズ b とループストライド m には定理 6 の前提条件を満たす値（$m = -1, -3, -25$，b は $b/\pi = -1, -5, -20, -80$ となる値）を組み合わせた 12 種類の値を用いた．ここで，表 4.2 より $\pi = 6m$ となる．

適用した計算分散法は，定理 6 の手法 (opt)，実行時解決法 (rr)，定理 5 の手法 (th)，系 1 による従来の while ループを用いたテーブル参照法 (t-w)，t-w における while ループを do ループに変更したもの (t-d)，及び，定理 7 で示した 2 重 do ループによるテーブル参照法 (t-2d) の 6 種類である．

測定マシンは分散メモリ型並列計算機である日立 SR8000/E0 である．SR8000/E0 の処理性能は 9.6 GFLOPS/node である．OS は HI-UX/MPP 03-07，Fortran90 コンパイラは OFORT90 V01-06-/A であり，指定オプションは最速オプション o(ss) と mp(p(0))

表 4.4: SR8000 上での実測結果 [ms].

$(m, b/\pi)$	rr	th	t-w	t-d	t-2d	opt
(-1,-1)	1100	53.7	5.64	3.41	4.82	.47
(-1,-5)	1103	22.4	5.64	3.43	4.55	.47
(-1,-20)	1102	8.7	5.64	3.43	1.82	.47
(-1,-80)	1107	4.5	5.65	3.43	1.25	.47
(-3,-1)	1103	53.8	5.64	3.43	4.86	.47
(-3,-5)	1108	28.5	5.64	3.43	4.64	.48
(-3,-20)	1106	14.3	5.64	3.42	2.16	.47
(-3,-80)	1110	8.8	5.66	3.43	2.53	.47
(-25,-1)	1107	53.7	5.64	3.43	4.87	.52
(-25,-5)	1110	55.0	5.64	3.42	4.65	.52
(-25,-20)	1107	46.1	5.65	3.44	2.14	.52
(-25,-80)	1118	36.5	5.69	3.46	2.53	.54

である．mp(p(0)) は，SMP クラスタ構成の SR8000 において自動 SMP 並列化の適用を避け，ノード内では 1 プロセッサのみ実行させるために指定した．尚，エラーを避けるため，プログラム th にはループ展開しないオプション loopexpand(0) を追加した．

本評価は計算分散の評価であり，プロセッサ間通信は発生しないため，実測は SR8000 の 1 ノード中の 1 プロセッサを用いて 8 プロセッサ分の計算を行なった．具体的には，プロセッサ番号を変数とした SPMD 型の計算部分を作成し，その計算部分をプロセッサ番号が 0 から 7 まで変化するループで囲んだ．但し，測定間におけるキャッシュ再利用を避けるため，各プロセッサ番号及び各パラメータによる実測直前に巨大配列への代入文を実行し，キャッシュをクリアした．キャッシュは L1（サイズは 128KB）のみで，store データは必ずキャッシュに書き込まれるので上記代入文でキャッシュはクリアされる．

測定区間は，テーブル参照法では executor ループのみ，その他のプログラムでは計算分散したループ以外にループの上下限値を求める計算も含めた．但し，配列 $\mathcal{L}$ を求めるループは全プログラムで除外した．

表 4.3 に各プログラムに適用された最適化を示す．擬似ベクトル化は rr，並びに，th と t-2d の内側ループに適用された．また，t-d 及び t-2d の内側ループは 2 倍展開され，opt の内側ループは 4 倍展開された．

表 4.4 及び図 4.15 は SR8000 上での実測結果である．表中の値は 8 プロセッサ分の実行時間の平均である．尚，全てのプログラムにおいて各プロセッサで参照された配列要素数は 100,000 個だったので，各プロセッサの負荷は均一であり，各プロセッサの実行時間もほぼ同一であった．表 4.4 及び図 4.15 より以下がわかる：

1. opt は他のコードと比べて 2 倍以上高速である．特に従来法 t-w に比べて 10 倍以上高速である．これは 1 重ループかつ配列添字が単純であるためと考えられる．
2. th や t-2d はパラメータによって性能が変化する．これは内側ループをほぼ b/π 回実行するため，その回数が増えるに従い，ループ最適化の効果が現れたためと考

表 4.5: SR8000上での実測結果（同程度の最適化）[ms].

$(m, b/\pi)$	rr	th	t-w	t-d	t-2d	opt
(-1,-1)	1119	68.6	5.63	4.04	4.05	.52
(-1,-5)	1137	24.1	5.64	4.03	4.20	.52
(-1,-20)	1136	8.5	5.64	4.03	1.80	.52
(-1,-80)	1140	4.9	5.68	4.05	1.50	.52
(-3,-1)	1159	81.5	5.62	4.03	4.06	.52
(-3,-5)	1179	27.7	5.65	4.03	4.30	.52
(-3,-20)	1176	13.8	5.64	4.04	2.40	.52
(-3,-80)	1181	9.7	5.69	4.05	2.79	.52
(-25,-1)	1415	81.5	5.63	4.02	4.06	.55
(-25,-5)	1428	53.1	5.64	4.03	4.28	.55
(-25,-20)	1424	44.5	5.66	4.04	2.42	.54
(-25,-80)	1430	42.3	5.70	4.07	2.77	.55

えられる．一方，これらを除いた他のプログラムは1重ループでループ繰り返し回数が不変のため，ストライドやブロックサイズの影響を受けない．

3. thはb/πが大きな値の場合，従来法t-wよりも高速になる場合がある．
4. テーブル参照法ではwhileループよりもdoループの方が高速である．これはdoループの方がコンパイラで最適化され易いためと考えられる．

上記実測では，高速になったプログラムにはループ展開等の最適化が適用されたため，純粋にコード生成法の比較になっていなかった．そこで，適用される最適化がほぼ同じになるようにして比較を行なった．即ち，コンパイルオプションをo(ss), swpl(0), loopexpand(0), nopvec, mp(p(0))とし，これ以外は上記と同じ測定環境を用いて実測した．この結果，どのプログラムに対しても，ソフトウエアパイプライニング，ループ展開，擬似ベクトル化は適用されなかった．

表4.5及び図4.16はSR8000上での実測結果である．これにより，適用される最適化を同一にしてもほぼ同様な結果が得られることがわかる．

4.1.9 関連研究

テーブル参照法は論文[3]において導入された．この方法は，線形添字を持つ配列参照を含むループに対して，データ分散後の配列要素の参照パターンは不均等であるがある周期ごとに同じパターンを繰り返すことを利用し，ある小さいループ繰返し範囲における参照パターンをDiophantine方程式をコンパイル時に解くことによりテーブルに格納し，そのテーブルを用いて全ループ繰返し範囲における配列参照コードを生成する．本方法は配列参照時にテーブルを介して参照を行うという間接参照オーバーヘッドが常に伴うという問題がある．

ある配列がテンプレートにALIGNされてblock-cyclic分散される場合，テーブル参照法以外に，その配列参照を含むループの計算分散方法がいくつかある．

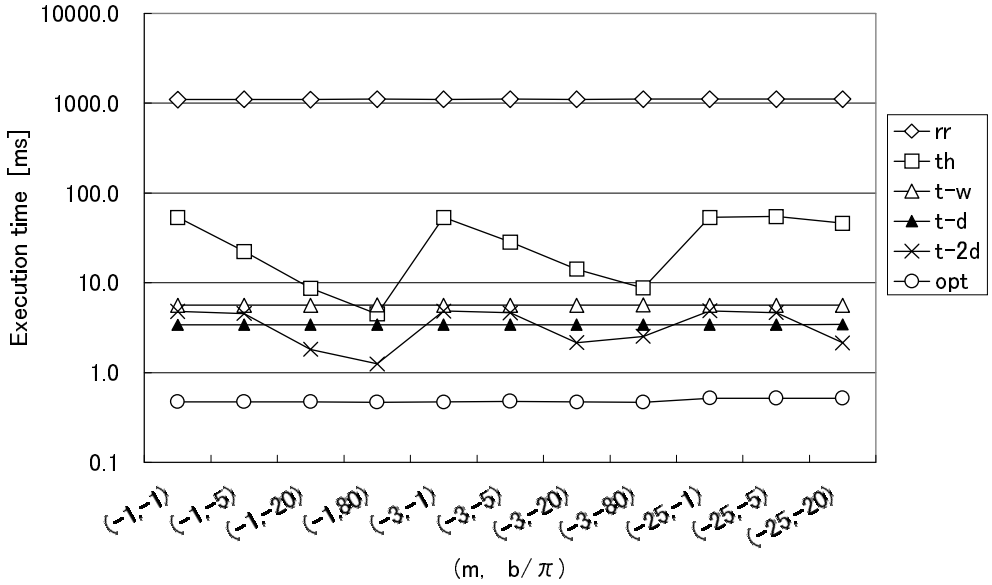


図 4.15: SR8000 上での実測結果.

実行時解決法は任意のプログラムに適用可能であるが，処理性能は非常に低い [7].

D system[1] における計算分散法は定数ストライドを持つループに適用可能である．D system はプロセッサ数やブロックサイズがコンパイル時に定数となる場合，オメガテスト [11] を用いてコードを生成する．生成コードの質はオメガテストの解析精度によるため，その実行性能は一般に不明確である．また，block-cyclic 分散に対してオメガテストを使ったコード生成方法は記述されてない．一方，プロセッサ数やブロックサイズがコンパイル時に定数でない場合は Virtual Processor 法（VP 法）[4] によるコードを生成する．これは block-cyclic 分散を block 分散と cyclic 分散の 2 つに分割して行なうコード生成法である．配列は 2 次元化されループは 2 重ループ化される．block-cyclic 分散に対して VP 法コードはテーブル参照法コードより遅い [10].

文献 [2] は行列を解いて計算分散する．block-cyclic 分散に対しては配列を 2 次元化した後にデータ分散し，データ分散後の配列中に生じた．元の配列に存在しない要素（hole[9]）を除去するために各次元を圧縮するが，hole は完全には除去されない．圧縮によって配列要素の参照順は元の参照順と異なるため，DOALL ループにしか適用できない．生成ループは 2 次元圧縮配列の各次元をたどるように 2 重ループ化される．評価はないが，2 重ループ化されていることから性能は VP 法程度であると予想される．

文献 [9] は ALIGN に対応できるように VP 法を拡張した．配列を 2 次元化し，ループを 2 重ループ化する．

文献 [12] は配列をポインタで参照するが，実質，配列は 2 次元化され，ループも 2 重ループ化される．よって，これも VP 法と同等の性能であると予想される．

以上のいずれの研究においても，効率的な 1 重ループに変換した例はない．本節はある特別な場合に，効率的な 1 重ループが出力できることを示した．尚，本出力コードではデータ分散後の配列において要素は連続に配置されるため，hole は生じない．逆に言うと，hole が生じない条件を求め，その時のコードを明示的に示したといえる．

Block-cyclic 分散に対応するが ALIGN に対応しない方法として以下がある．文献 [6] はループのストライドが正の場合のみ扱う．文献 [10] の方法は参照添字が作る格子に対

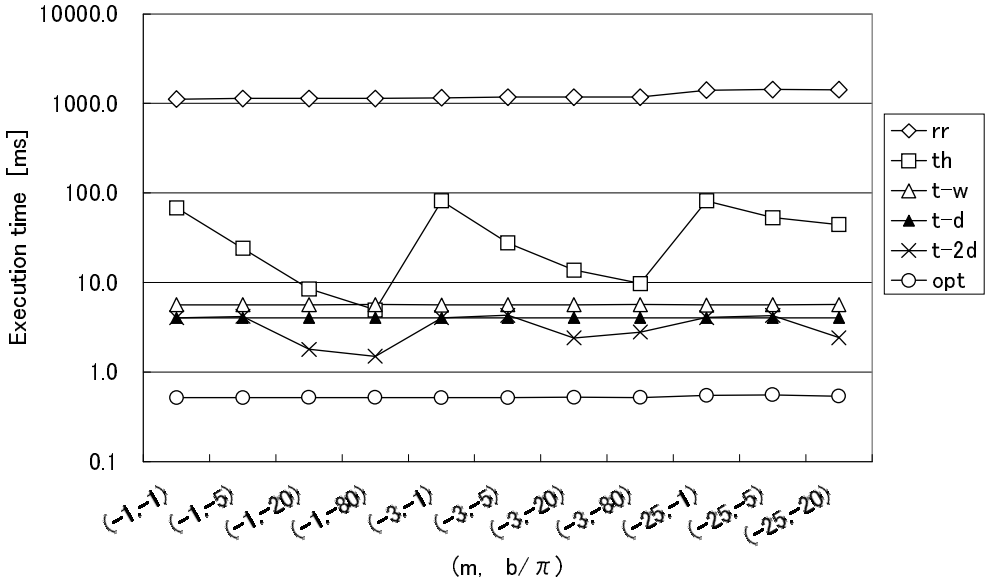


図 4.16: SR8000 上での実測結果（同程度の最適化）.

する 1 つの基底ベクトルを求め，それを使って参照添字を計算する．文献中の実測[5]ではテーブル参照法よりも最高で 1.5 倍高速になるが，その時の条件は不明確である.

4.1.10　まとめ

HPF の一般の規則的データ分散が指示された配列への参照を含む一般の 1 重ループを，図式による一般的枠組みの中で取り扱うことにより以下に示す結果を得た.

- 上記パターンに対する計算分散コードを与えた.
- block-cyclic 分散と ALIGN 指示文が指定された配列への参照を含むループに対して，ある 3 つのパラメータの積がブロックサイズの約数となる場合に最適な計算分散コードを与えた.
- 一般的計算分散コードにおける添字参照において従来より短い周期があることを示し，これより 2 重 do ループ，1 重 do ループ，又は 1 重 while ループからなるテーブル参照法コードを導いた．特に，ストライドが負の場合に対してもテーブル参照法コードを与えた.
- 提案手法を複雑な block-cyclic 分散を持つプログラムに適用して日立 SR8000 上で実測した．その結果，最適計算分散コードは従来，block-cyclic 分散に対して最速だったテーブル参照法コードよりも 10 倍以上高速であり，do ループによるテーブル参照法コードは上記従来法コードよりも高速であることがわかった.

本節の結果は，一般のループ制御式を持つ多重ループ，一般の規則的データ分散指示が与えられた多次元配列，互いに異なるループ制御変数による 1 次式を各次元の添字に持つ多重ループ中の配列参照，を含むプログラムに拡張可能である.

[5] テーブルや基底ベクトルの計算時間は含まない.

参考文献

[1] Adve, V. and Mellor-Crummey, J.: Using Integer Sets for Data-Parallel Program Analysis and Optimization, *Proc. of PLDI '98*, pp. 186–198 (1998).

[2] Ancourt, C., Coelho, F., Irigoin, F. and Keryell, R.: A linear algebra framework for static HPF code distribution, *Proc. of CPC '93*, pp. 161–172 (1993).

[3] Chatterjee, S., Gilbert, J., long, F., Schreiber, R. and Teng, S.-H.: Generating local addresses and communication sets for data- parallel programs, *Proc. of PPoPP '93*, pp. 149–158 (1993).

[4] Gupta, S., Kaushik, S., Huang, C.-H. and Sadayappan, P.: Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines, *J. of Parallel and Distributed Computing*, Vol. 32, pp. 155–172 (1996).

[5] High Performance Fortran Forum: High Performance Fortran 2.0　公式マニュアル (1999).

[6] Hiranandani, S., Kennedy, K., Crummey, J. M. and Sethi, A.: Compilation Techniques for Block-Cyclic Distributions, *Proc. of ICS '94*, pp. 392–403 (1994).

[7] Hiranandani, S., Kennedy, K. and Tseng, C.-W.: Compiling Fortran D for MIMD Distributed-Memory Machine, *CACM*, Vol. 35, pp. 66–80 (1992).

[8] 岩井斉良: ホモロジー代数入門, サイエンス社 (1978).

[9] Kaushik, S. D., Huang, C.-H. and Sadayappan, P.: Efficient Index Set Generation for Compiling HPF Array Statemets on Distributed-Memory Machines, *J. of Parallel and Distributed Computing*, Vol. 38, pp. 237–247 (1996).

[10] Kennedy, K., Nedelijkovic, N. and Sethi, A.: Efficient Address Generation for Block-Cyclic Distribution, *Proc. of ICS '95*, pp. 180–184 (1995).

[11] Pugh, W.: A practical algorithm for exact array dependence analysis, *CACM*, Vol. 35, No. 8, pp. 102–114 (1992).

[12] van Reewijk, K., Denissen, W., Sips, H. J. and Paalvast, E.: An Implementation Framework for HPF Distributed Arrays on Message-Passing Parallel Computer Systems, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 9, pp. 897–914 (1996).

4.2 HPF コンパイラにおける計算分散最適化手法

4.2.1 概要

高性能計算に対してユーザは強い要求を持っており，並列プロセッサ，特に，MIMD 型分散メモリ計算機に対する期待は高まっている．なぜなら，共有メモリ型ベクトルスーパーコンピュータでは性能に限界があるためである．分散メモリ型計算機に対してプログラマがやらなければならないことは，大量のデータをローカルメモリに分散すること，あるデータがローカルに参照できるのかリモートに参照するのかを区別すること，及び，リモート参照に対してプロセッサ間データ通信コードを生成することである．このようなプログラマの負担は並列プログラミングにおける深刻な問題である．

近年，従来の Fortran 言語に指示文が追加された拡張 Fortran 言語 High Performance Fortran (HPF)[5] が提案され，業界標準となりつつある．HPF でプログラムを書くことはプログラマにとってたやすいことである．なぜなら，プログラマは複雑でエラーを引き起こしやすい通信や同期を書くかわりに逐次プロセッサ向けプログラムにデータ分散指示文を加えるだけでよいからである．HPF 言語は通信指示文も同期指示文も含まない．これらのレベルの最適化は HPF コンパイラの仕事である．

多くの商用 HPF コンパイラがある [2, 9]．HPF や他の類似言語に対するコンパイラ変換アルゴリズムや最適化は様々に研究されてきた．[1, 3, 6, 7, 10, 11]．計算分散は HPF コンパイラにおける 1 つの変換フェーズである．それは元の計算を各プロセッサに分散させる処理である．ループ中に単一の代入文のみを含み，その左辺の配列要素の添字が単純である場合には計算分散は簡単に，かつ，効果的に行われる [7]．しかしながら，以下の場合には計算分散を効果的に適用できなかった．

- ループ中の代入文の左辺に現れる配列要素に関連する以下の 2 つのケース．
  - **(a)** 同じループ制御変数が異なる次元に現れる場合．例えば，$a(i, i+1)$
  - **(b)** 複数のループ制御変数が一つの次元に現れる場合．例えば，$a(i+j)$
- ループ中の文の計算分散後のループ繰返し範囲に関する以下の 2 つのケース．
  - **(c)** 1 つのプロセッサに対して複数の文の計算分散後のループ範囲が異なる可能性がある場合，
  - **(d)** 1 つの文の計算分散後のループ範囲が複数のプロセッサで異なる可能性がある場合．

(a) と (b) は従来の研究では言及されなかった．本節における計算分散法，拡張配列添字関数法，は文の実行プロセッサをみつけ，各次元の制約に関するディオファントス方程式を解くことによってその文の計算分散後のループ繰返し範囲を求め，多重ループを内側ループから外側ループに向かって変換する．

(c) に対しては従来手法は実行時解決コードを生成していた．即ち，計算分散後のループ範囲はループ中の各文の計算分散後のループ範囲の和集合であり，各文はループ制御変数の値がそのループの計算分散後のループ範囲に含まれているときにその文を実行するような if 文（＝ガード）で囲まれる [7]．このようなコードの実行性能は低

い．なぜなら，プロセッサは十分に分散されないループ範囲を実行し，しかも，毎回，全ての文を囲むガードを実行しなければならないからである．

このようなガードを減らす2つの方法が提案されている．Hiranandani [7] らはもし，複数の文が計算分散後に同じループ範囲を持つならばそれらに対して共通の単一の if 文を生成する．それは if 文の数を減らすが，全ての if 文をなくしたり，ループ範囲を減らしたりすることはない．

Miyoshi [10] らはガードの条件式が常に真か偽になるような範囲にループ範囲を分割することによって if 文を削除する．この方法はガードを減らし，計算分散後のループ範囲を減らすが，(d) の場合に対応できない．

本節は上記に挙げた (a) から (d) までの全ての場合を扱う．これを行うためガード削除法と拡張配列添字関数法を提案する．

本研究は元のループ繰返し空間から新しいループ繰返し空間へアファイン写像を使ってマッピングするループ・リオーダリング変換に関係する．

Kelly [8] らはループ中の各文に対して異なる可能性のある 1 対 1 写像が使われるとき，1プロセッサ向けのコードを生成する．これに対して，本研究における方法はループ中の各文に対して異なる可能性のある非 1 対 1 写像が使われるとき，複数プロセッサ向けの SPMD コードを生成する．但し，上記非 1 対 1 写像はプロセッサ配置空間と結び付けられるときには 1 対 1 となるが，各プロセッサに対して異なる可能性がある．

第 4.2.2 節は試作コンパイラの特徴と構成を述べる．第 4.2.3 節は多次元データ分散における LIXS 解析を説明する．第 4.2.4 節は拡張配列添字関数法を説明する．第 4.2.5 節はガード削除法を述べる．第 4.2.6 節は性能評価について述べ，第 4.2.7 節でまとめを述べる．

4.2.2 試作コンパイラ

特徴

試作コンパイラは HPF データ分散指示文を含む Fortran プログラムをメッセージパッシングライブラリ呼出しを含む Fortran プログラムに変換する．特に，本コンパイラは複数次元ブロック分散をサポートする．現在の実装では以下の最適化をサポートする．

- メッセージ・ベクトライゼーション
- メッセージ・コアレッシング
- バッファ無しメッセージ
- イディオム認識
- ガード削除法

ループ上下限値削減法とガード削除法は第 4.2.5 節で説明する．

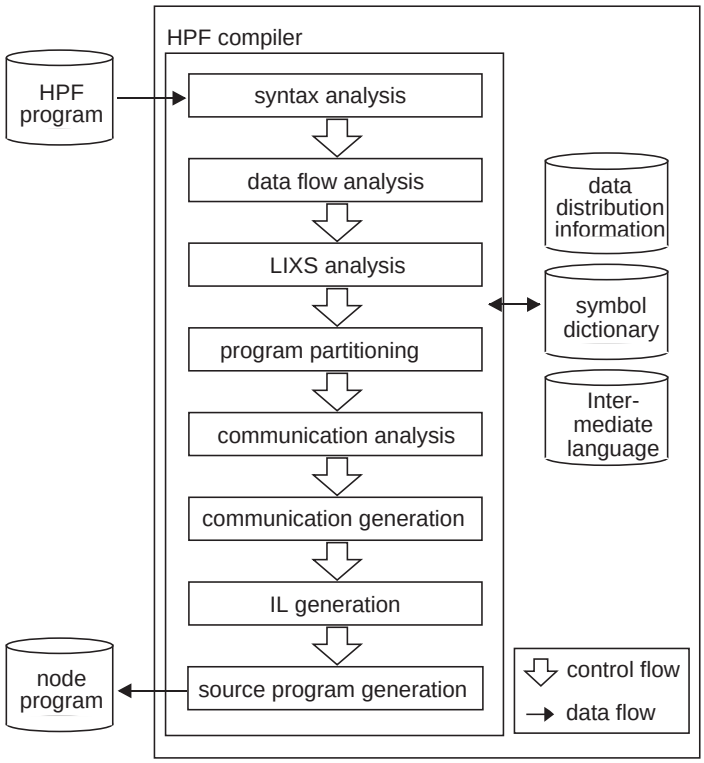


図 4.17: HPF コンパイラの構成.

構成

図 4.17 はコンパイラの構成を示す.

LIXS 解析はプロセッサに分散される部分配列を決定し，配列添字をプロセッサ番号にマッピングする関数のようないくつかの関数を生成する．このとき，各配列要素は上記で得られたプロセッサ番号を持つプロセッサのローカルメモリに格納される．このことはまた，このプロセッサがこの要素を所有する，とも呼ぶ.

計算分散はループ中の各文と各プロセッサに対してループ繰返し範囲を決定する．即ち，プロセッサはループ制御変数が上記繰返し範囲内にある時にのみその文を実行する．本研究では”owner computes rule”[6]，即ち，代入文の左辺にあるデータの所有プロセッサがその文を実行する，という規則を用いる.

通信解析はプロセッサ間通信が必要か否かを代入文の右辺のデータを解析することによって決定し，そのデータの所有プロセッサとプログラム中で通信を行う最も良い個所をを見つける.

通信生成は通信ライブラリルーチンに対する中間語と引数を生成し，メッセージ・コアレッシングのような通信最適化を適用する.

IL 生成は計算分散情報を使ってガード等の中間語を生成する．本研究の実装ではガード削除をこの処理フェーズで行う．この処理フェーズで行うことにより，一連のコピーされたループ列に対して，正確な通信解析と一対のプレループ・ポストループ通信が生成できる．もし，ガード削除が計算分散フェーズで行われたら，ループ上下限値は複雑になり，通信解析を正確に行うのを妨げ，一連のコピーされたループ列に対して，一対のプレループ・ポストループ通信を生成するのを妨げる.

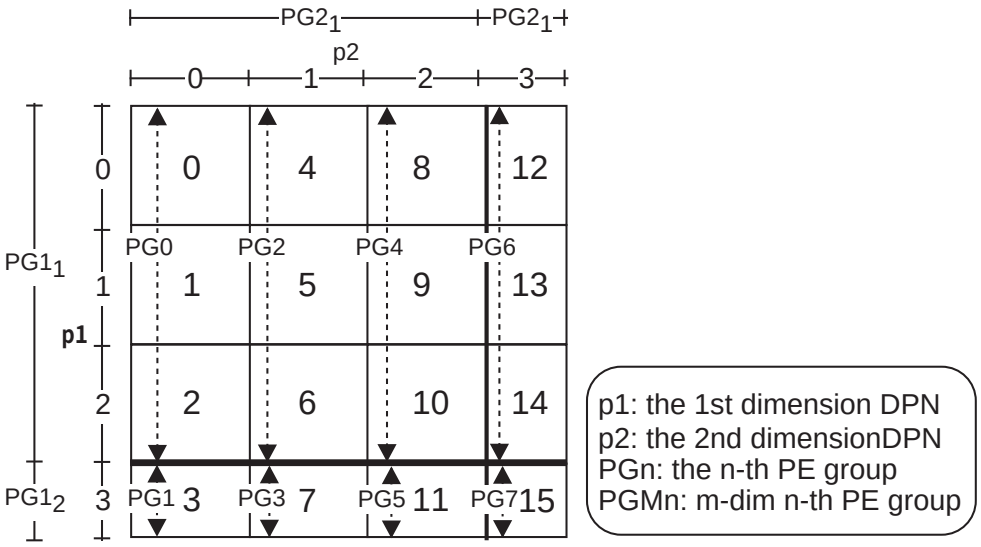


図 4.18: DPN と PE グループ.

ソースコード生成は中間語を通信ライブラリ呼出しを含む Fortran ソースプログラムに変換する.

4.2.3 LIXS 解析

出力情報

LIXS 解析は全てのプロセッサをグループ（PE グループ）に分割する．各グループはプロセッサ番号が連続で，全ての分散配列の各次元において同じ添字範囲を持つようなプロセッサから構成される．これらのグループは計算分散，通信解析，及び，通信生成の単位となる．LIXS 解析は各 PE グループに対して，分散配列の添字範囲のグローバル・ローカル表現や配列添字をその所有プロセッサ番号に対応させる関数，等の情報を生成する．これらの情報は後続するコンパイラフェーズにおいて基本的な情報となる.

多次元データ分散

上記で述べた情報を多次元データ分散に拡張するのは容易である．しかしながら，1 次元のプロセッサ番号は多くの PE グループを生み出し，多量の出力コードを生成することになる．例えば，もし，2 次元配列の両方の次元が分散され，各次元のサイズが対応する次元のプロセッサ数で割り切れないなら，プロセッサは多くの PE グループに分割される．図 4.18 はプロセッサ番号が 1 次元的に連続に並んでいるために 8 つの PE グループ，$P0$ から $P7$ が生成されることを示す.

PE グループの数を減らすために次元毎プロセッサ番号（DPN），即ち，多次元プロセッサ配置におけるプロセッサ位置座標，が導入される.

DPN は PE グループ数を減らし，次元毎にデータ分散を指定する，というＨＰＦのデータ分散戦略に合致しており，しかも，配列添字関数の逆関数を使う計算分散方法

[6] を簡略にする．図 4.18 は次元毎に 2 つの PE グループ，1 次元目は $PG1_1$ と $PG1_2$ で 2 次元目は $PG2_1$ と $PG2_2$，が存在する様子を示す．DPN はプロセッサ配置と元の 1 次元版プロセッサ番号から得られる．

DPN は PE グループ数を減らし，次元毎にデータ分散を指定する，というＨＰＦのデータ分散戦略に合致しており，しかも，配列添字関数の逆関数を使う計算分散方法 [6] を簡略にする．図 4.18 は次元毎に 2 つの PE グループ，1 次元目は $PG1_1$ と $PG1_2$ で 2 次元目は $PG2_1$ と $PG2_2$，が存在する様子を示す．DPN はプロセッサ配置と元の 1 次元版プロセッサ番号から得られる．

たとえば，プロセッサが 4×4 の状態に配置されており，この 2 次元配列は両方の次元が図 4.18 のようにブロック分散されていると仮定する．$p, p1, p2$ を各々，元の 1 次元版プロセッサ番号，1 次元目及び 2 次元目に対する次元毎プロセッサ番号 DPN，とする．このとき，以下の関係式が成り立つ．

$$p = p1 + 4 \times p2 \tag{4.11}$$

$p1, p2$ の値は上式を使って p から計算できる．

4.2.4 計算分散

計算分散は”owner computes rule”，即ち，代入文の左辺データの所有プロセッサがその文を実行するというルール，を使い，ループ繰返し範囲を配列添字関数の逆関数を使って削減し，ガードを生成する．

本節ではループ中の分の左辺上の配列要素添字に関する以下の 2 つの場合を扱うために配列添字関数法を拡張する．

(a) 同じループ制御変数が異なる次元に現れる場合．例えば，$a(i, i+1)$

(b) 複数のループ制御変数が一つの次元に現れる場合．例えば，$a(i+j)$

(a) の場合, 同じ lcv を持つ全ての次元に対してループ繰返し集合 (LITS)[6] が DPN を使って得られる．そのような配列要素にアクセスするループ繰返しが存在することはそれらの LITS が交わりを持つことを意味し，そのことから DPN 間の関係が導かれる．最後に，その関係を満たす PE グループに対する LITS が得られる．

(b) の場合，プログラム変換は最内側ループから始まる．もし，ループ j がループ i の外側ループなら，配列要素 $a(i+j)$ における変数 j はループ i の内側では不変である．即ち，j はループ i の内側では定数と見なされるので，従来の配列添字関数法がループ i に適用できる．その後，変換は外側ループへ向かって続けられる．内側ループに対する処理ステップで変換された配列要素はもう外側ループに対する処理ステップは適用されないことに注意する．なぜなら，前者のステップで配列要素はすでにローカル化されているからである．配列添字関数法は内側 lcv を含まない配列添字にのみ適用されるので，Fourier 消去法 [4] を使う必要はない．

図 4.19 は上記のアルゴリズムを示す．ここで，配列要素は両方の lcv とループ不変変数の 1 次結合であり，ループ中の文の左辺は配列であると仮定する．

(1) ループ中の文の左辺にある配列のグローバル添字範囲を得る

(2) 配列添字から，その文を含む最内側ループのループ制御変数 lcv を見つける

(3) 逆添字関数をその添字範囲に適用することによって上記 lcv に対する（複数の）ループ繰返し範囲を得る．

(4) もし 2 つ以上のループ繰返し範囲が存在するなら，それらの集合の交わりが空集合にならない条件を求める．

(5) 上記条件の下で集合の交わりを求める．

(6) 元のループ繰返し範囲と (5) で得た結果との交わりを計算する

(7) ループ中の全ての文に対して (6) の結果の和集合をループ範囲として設定する

(8) ループ中の各文に対して (6) の結果のループ繰返し範囲だけその文を実行するようなガードを生成する．

図 4.19: 計算分散アルゴリズム．

```
      real a(9,9)
!hpf$ processors p(3,3)
!hpf$ distribute a(block,block) onto p

      do j=1,8
S1:     a(j,j+1)=···
      enddo
```

図 4.20: HPF プログラム．

定義 1 The loop iteration range of the L (LIR=LIR(S)=LIR(S,L)) に対する文 S のループ繰返しとは 図 4.19(6) から得られる範囲である．

例 図 4.19 におけるアルゴリズムを図 4.20 におけるプログラムに適用する．表 4.6 は図 4.19 におけるステップ (1) から (3) を図 4.20 における S1 の左辺に適用した結果を示す．ここで，$GIXS$, $inv(x)$, 及び $UGITS$ は，各々，グローバルインデックス集合，添字関数の逆関数，及び，無制限グローバル繰返し集合を意味する [6].

ステップ (4) における条件 "$I_1 \cap I_2 \neq \emptyset$" は以下の Diophantine 不等式に等しい:

$$\begin{cases} 3*p1+1 & \leq 3*(p2+1)-1 \\ 3*p2 & \leq 3*(p1+1) \end{cases}$$

```
real a(3,3)
if (p2.eq.p1) then
  low=1; upp=2
else if (p2.eq.p1+1 .and. p1.le.1) then
  low = 3; upp = 3
else ; loop is not executed
  low=1; upp=0
endif
do j=low,upp
  a(j,j+1)=···
enddo
```

図 4.21: HPF コンパイラ生成コード .

表 4.6: 2 次元配列に対するインデックス情報 (p1,p2=0,1,2).

	First dimension	Second dimension
$GIXS$	$[3*p1+1:3*(p1+1)]$	$[3*p2+1:3*(p2+1)]$
$Inv(x)$	x	$x-1$
$UGITS$	$I_1=[3*p1+1:3*(p1+1)]$	$I_2=[3*p2:3*(p2+1)-1]$

以下はこの不等式から得られる.

$$p2 = p1 \quad or \quad p2 = p1+1$$

IS は上記の 2 つの場合に対する I_1 と I_2 の交わりとして得られ，一方，$[1:8]$ はステップ (5) と (6) から得られる:

$$IS = [3*p1+1:3*p1+2] \quad for\ p2 = p1$$
$$IS = [3*p1+3:3*p1+3] \quad for\ p2 = p1+1$$

インデックスをローカル化することにより以下が得られる：

$$IS = [1:2] \quad for\ p2 = p1 \qquad (p1 = 0,1,2)$$
$$IS = [3:3] \quad for\ p2 = p1+1 \quad (p1 = 0,1)$$

図 4.21 はステップ (7) と (8) を適用した結果を示す.

4.2.5 ガード削除法

ガード削除法はループ中の文につく if 文をループ中の文のループ繰返し範囲に関する以下の場合において削除する.

(a) 1 つのプロセッサに対して複数の文の計算分散後のループ範囲が異なる可能性がある場合,

```
      real a(30),b(30)
!hpf$ processors p(3)
!hpf$ distribute a(block) onto p
!hpf$ distribute b(block) onto p
      do j=1,29
S1:    a(j)=···
S2:    b(j+1)=···
      enddo
```

(a) HPF プログラム

```
      real a(10),b(10)
      do j=0,10
       if(1.le.j.le.10) a(j)=···
       if(0.le.j.le.9) b(j+1)=···
      enddo
```

(b) 従来手法による結果

```
      real a(10),b(10)
      do j=1,10
       a(j)=···
      enddo
      do j=0,9
       b(j+1)=···
      enddo
```

(c) ループ分割を適用した結果

```
      real a(10),b(10)
D1:   do j=0,0
       b(j+1)=···
      enddo
D2:   do j=1,9
       a(j)=···
       b(j+1)=···
      enddo
D3:   do j=10,10
       a(j)=···
      enddo
```

(d) ガード削除法の適用結果

図 4.22: ガード削除法の例.

(b) 1つの文の計算分散後のループ範囲が複数のプロセッサで異なる可能性がある場合.

(a) と (b) は各々，4.2.5 節と 4.2.5 節で説明される.

1プロセッサに対するコード生成

図 4.22 はガード削除法を，(b)(c)(d) はプロセッサ番号 1 に対するコードを示す. 図 4.22(a) は左辺の添字が異なる 2 つの文を持つループを示す．2 つの配列 a と b は同じブロック分散方法で分散され，ループはループ分割可能であると仮定する．以下で，このようなループに対する 2 つのループ並列化手法を説明する.

最初の方法は 4.2.4 節で説明したのと同じである．LIR(S1) と LIR(S2) は 図 4.19 におけるステップ (1) から (6) を使って，各々，[1:10] と [0:9] になる．ステップ (7) は 2 つの LIR [0:10] の和集合を作る．図 4.22(b) は 2 つの文に対して [1:10] と [0:9] に対応したガードを生成した後の結果を示す.

第 2 の手法は最初の手法とループ分割を組み合わせた手法である．ループ分割が適用された後の 2 つのループに対して最初の手法を適用するのは容易である．なぜなら，

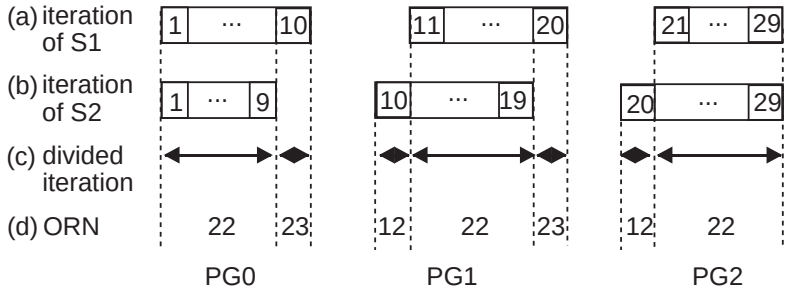


図 4.23: ループ繰返し範囲分割と ORN.

上記 2 つのループでは文の数が 1 個なので，これらの文に対してガードは不要となるからである．図 4.22(c) は変換後のプログラムを示す．

最初の方法はどのループにも適用可能である．しかしながら，適用後に得られるプログラムの実行性能は低い．なぜなら，ループ中の 2 つの文にガードが存在するからである．2 番目の方法はループ分割可能なループにのみ適用可能である．適用後に得られるプログラムは高い実行性能を発揮するが，いくつかの欠点を持つ．

(1) ループ出口における分岐コードの総数が増加する（今の場合，約 2 倍）

(2) 前方のループから後方のループへ値を引き継ぐためにロード・ストア命令が必要になる．

図 4.22(d) はガード削除法の結果を示す．この方法を説明するためにまず，図 4.22(b) におけるループを考察する．

最初に，ループ繰返し範囲は 3 つの部分，[0:0], [1:9], [10:10]，に分割される．これらは 2 つの LIR，[1:10] と [0:9]，の disjoint な分割を与える．もし，ループ繰返し範囲が [0:0] なら，文 S2 のみが実行され，ガードは削除される (D1)．もし，ループ繰返し範囲が [1:9] なら，両方の文が実行される (D2)．もし，ループ繰返し範囲が [10:10] なら，文 S1 のみが実行される (D3)．最後に，ガードのない 3 つの文が得られる．

もし，ループ中の全ての文 S の $LIR(S)$ が各プロセッサに対して定数なら，もしループがループ分割不可能であっても，そのループはガード削除法によって効果的に並列化される．さらに，最初の方法は 2 番目の方法とは異なり，その結果はどんなオーバーヘッドも持たない．

全プロセッサグループに対するコード生成

第 4.2.5 節のアルゴリズムでは, ある PE グループに対する分割されたループ数は他のグループに対するそれと異なる可能性があった．たとえば, 図 4.22(d) はプロセッサ番号が 1 でループ数が 3 の場合を示す．しかし，ループ数はプロセッサ番号 0 や 2 では 2 になる．このことを図 4.23 を用い説明する．

図 4.23 の (a) と (b) は全ての PE グループに対して，図 4.22(a) における LIR(S1) と LIR(S2) を示したものである．図 4.23(c) は LIR(S1) と LIR(S2) によるループ実行範囲を disjoint に分割を行った結果である．PG0 では [1:9] と [10:10] の 2 つの範囲に，PG1

```
    pg = get_pe_group_number (pe)
    if (pg .eq.  0) then
      l1=1; u1=0; l2=1; u2=9; l3=10; u3=10;
    endif
    if (pg .eq.  1) then
      l1=10; u1=10; l2=11; u2=19; l3=20; u3=20;
    endif
    if (pg .eq.  2) then
      l1=20; u1=20; l2=21; u2=29; l3=1; u3=0;
    endif
L1: do j = l1, u1
      B(j+1) = ...
    enddo
L2: do j = l2, u2
      A(j) = ...
      B(j+1) = ...
    enddo
L3: do j = l3, u3
      A(j) = ...
    enddo
```

図 4.24: ガード削除適用後のプログラム.

では [10:10], [11:19] と [20:20] の 3 つの範囲に，PG1 では [20:20] と [21:29] の 2 つの範囲に，ループ実行範囲が disjoint 分割される.

このように PE によって細分されるループ数が異なる場合に，SPMD プログラムを少ないコード量で生成するため，以下の戦略を使った.

- もし，ある PE グループの分割後ループが別の PE グループと同じ実行文を持つならば，両者に対して一つのループを生成する.

分割後ループを実行文の集合と実行順序に対応させるため，複数区間を基準にした順序表現数 (ORN) を導入する.

定義 2 ある数 x に対するある基準区間 $I = [low : upp]$ を基準にした順序表現数 $ord_I(x)$ とは以下である：

$$ord_I(x) = \begin{cases} 1 & \text{if } x < low \\ 2 & \text{if } low \leq x \leq upp \\ 3 & \text{if } upp < x \end{cases}$$

表 4.7: 実行時間の比較.

	(a) 従来手法	(b) ループ分割	(c) ガード削除法
(a) に対する比	1	0.68	0.53
(b) に対する比	-	1	0.78

定義 3 ある数 x に対する複数の基準区間 $\tilde{I} = \{I_j\}_{j=1,\cdots,n}$ を基準にした順序表現数 $ord_{\tilde{I}}(x)$ とは各 I_j に対する $ord_{I_j}(x)$ を並べてできる数である：

$$ord_{\tilde{I}}(x) = \sum_{j=1}^{n} 10^{n-j} * ord_{I_j}(x)$$

本手法では区間集合 $\tilde{I} = \{I_j\}_{j=1,\cdots,n}$ (n: ループ内の文の数) をそのループにおける LIR の集合と見なし，与えられた数 x を各 PE グループに対する分割ループの下限値と見なし，各分割ループと各 PE グループに対して ORN を計算する．図 4.23(d) は全ての PE グループに対する ORN を示す.

例 $\tilde{I} = \{I_1, I_2\}$, $I_1 = [11:20]$, $I_2 = [10:19]$ を各々，LIR(S1) と LIR(S2) と見なし，$x = 10$, とし，PG1 に対してのみ計算する．$10 < 11$ で，$10 \in I_2$, $ord_{I_1}(10) = 1$ かつ $ord_{I_2}(10) = 2$, なので，$ord_{\tilde{I}}(10) = 12$ となる.

順序表現数に対して，次の 2 点がいえる.

(a) 同じ順序表現数を持つ 2 つのループの繰返し範囲は，実行される文がまったく同じである.

(b) 順序表現数の大小関係は，プログラムの実行順序と同じである.

コード生成ステップを 図 4.23(d) と上記 2 つの性質を用いて説明する.

図 4.23(d) における最小の ORN は 12 である．これは S2 のみが実行され，PG0 に対する繰返し範囲が空集合であり，(なぜなら図 4.23(d) により ORN は PG0 に対して 12 でないから) PG1 に対して [10:10] であり，PG2 に対して [20:20] であることを示す. このとき，各 PE グループに対してループ繰返し範囲が [1:0] (即ち，不実行), [10:10], 及び [20:20] となる新しいループが作成され，S2 がその中に挿入される．これでコード生成の最初のステップが終了する．ORN が 22 の時と 23 の時も同様に処理される.

図 4.24 は上記処理ステップで生成されたプログラムである．L1, L2 及び L3 に対する ORN は各々， 12, 22, 及び 23 となる.

表 4.7 は 図 4.22 における 3 つの方法 (b), (c), (d) の実行結果を比較したものである．ガード削除法は，従来法 1 の約半分，従来法 2 の約 8 割の実行時間となる.

図 4.2.5 と図 4.26 はガード削除のアルゴリズムを示す．このアルゴリズムの実行時間は n をプログラム行数としたとき，$O(n^2)$ である．本手法は全ループ範囲を 3 つに分割することによって一般の場合に容易に拡張できる．即ち，(1) 定数インデックスのアクセス範囲，(2) ブロック分散範囲，(3) サイクリック分散範囲．最後のものは，ある周

/* 分割ループの上下限値を生成する */
for (全ての PE グループ pg)
  for (ループ中の全ての文 k)
    $S \leftarrow \cup_k$ {($LIR(k)$ の下限値 [上限値],1[2])
  endfor
  $\{(b_j, f_j)\}_{j=1,\ldots,m} \leftarrow S$ を辞書式順序でソートし，
  同じ要素を削除
  for $(j = 1, m-1)$ $I_j^{pg} \leftarrow$
    $[low_j^{pg} : upp_j^{pg}] = [b_j + f_j - 1 : b_{j+1} + f_{j+1} - 2]$
endfor
/* ORN を作成 */
for (全ての PE グループ pg)
  for $(j = 1, m-1)$
    for (ループ中の全ての文 k)
      S_k=LIR(k) とする
      $S_k^-[S_k^+]$ を S_k よりも小さな [大きな] 範囲と設定
      if $(low_j^{pg} \subseteq S_k^-[S_k,\ S_k^+])$
        $ord_{S_k}(low_j^{pg}) \leftarrow 1[2,3]$
      endif
    endfor
    for (ループ中の全ての文 k)
      $ord_{S_k}(low_j^{pg})$ を並べ替えて $ord_{\tilde{S}}(low_j^{pg})$ を作成
  endfor
endfor

図 4.25: ORN 作成アルゴリズム.

```
while (未選択の I_j^pg が存在する間)
 for (全ての PE グループ pg)
  M^pg ← min_{all unselected I_j^pg}(ord_S̃(low_j^pg))
 endfor
 M ← min_{all PE groups}(M^pg)
 新規ループ "do i=low,upp" を作成し、
  ORN=2 に対応する分を生成する
 /* ループ直前に変数 pg に関する以下の文を生成する */
 for (全ての PE グループ pg)
  if (M = M^pg)
   "if(pg=pg) low=low_j^pg[upp=upp_j^pg]"
   この I_j^pg を選択済みとマークする
  else
   "if(pg=pg) low=1[upp=0]"
  endif
 endfor
endwhile
```

図 4.26: コード生成アルゴリズム.

期的な範囲（それらの長さは cyclic(m) に対する全ての m の最小公約数となる），とその範囲を繰り返し実行する範囲，すなわち，2 重ループに変換される．さらに，ORN の値の範囲は 2 つ以上の disjoint な範囲を扱うために全ての正の整数の集合に拡張される．

尚，本試作では簡易化した以下のアルゴリズムを用いた．

- ループ実行範囲を，順序表現数が $2\cdots2$ となるもの，これ以下，これ以上，の 3 つに分割する．
- $2\cdots2$ となる範囲に対してのみループ内の if 文を削除する

4.2.6 性能評価

超並列機向けベンチマークプログラムとしてよく使われる米国国立大気研究所の Shallow-Water に対して，試作コンパイラによる自動並列化と人手並列化との比較結果を行った．測定マシンはの nCUBE 社の超並列機 nCUBE2（128 台構成）である．言語は nCUBE Fortran であり，PE 間通信関数などの並列ライブラリも nCUBE 社のものを使用した．Fortran に対する最適化オプションは $-O$ を用い，時間測定はマイクロ秒単位で測定可能な amicclk 関数を用いた．配列に対するデータ分散は 2 次元目 BLOCK 分散を指定した．

図 4.27 に人手による並列化結果と試作コンパイラによる並列化結果を PE 台数を 2 台から 128 台まで変えて実行したときのプログラム実行時間を示す．両者の実行時間はほぼ同じになった．

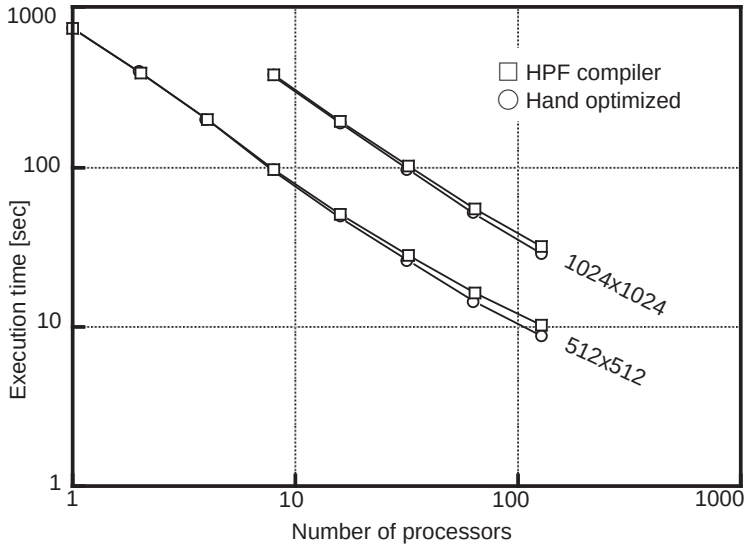


図 4.27: Shallow Water プログラムに対する評価結果.

表 4.8: 並列化実行結果の比較.

ループ	人手並列化	コンパイラ出力
do i = 1, 99 S1: a(i) = S2: a(i+1) = enddo ループ 1	do i = 1, 10 S1 enddo do i = 0, 9 S2 enddo (a) loop distribution	do i = 0, 0 S2 enddo do i = 1, 9 S1 S2 enddo do i = 10, 10 S1 enddo (b) guard removal

do i = 1, n
a(i,1) = a(i,n)
enddo
 	
ループ 2
	
	if (p_{m-1}) send(a(1:n,n), p_0)
	if (p_0) recv (a(1:n,1), p_{m-1})
	
	(c)
		
		if (p_{m-1}) u(1:n)=a(1:n,n)
		if (p_{m-1}) send(u(1:n), p_0)
		if (p_0) recv(u(1:n), p_{m-1})
		if (p_0) a(1:n,1)=u(1:n)
		
		(d)

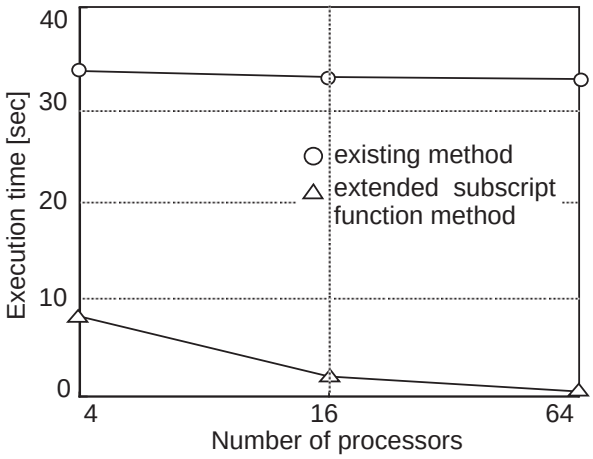


図 4.28: BEM プログラムに対する評価結果.

表 4.8 に人手による並列化結果と試作コンパイラによる並列化結果の違いを示す. ループ 1 に対して，人手ではループ分割し，分割ループ毎に並列化した．一方，試作コンパイラではガード削除法で並列化した．人手による並列化結果よりも，コンパイラによる並列化結果の方が性能が良い.

ループ 2 に対して，人手では (C) のようにループを解消し，1 回の通信に変換した. 一方，試作コンパイラでも (D) のように 1 回の通信に変換したが，隣接 PE 間通信以外では一時配列を用いて通信するため，1 行目および 4 行目に余分にループを生成した．明らかに (C) の方が (D) より実行性能が良い.

(D) のおける余分な配列代入は，代入文右辺が一般に多項式であることから生成される．したがって，ループ 2 の代入文右辺が単項式であることを認識して，(C) のプログラムを生成しなければならない.

上記 2 つのループにおいて，試作コンパイラは，ループ 1 に対しては人手より良く，ループ 2 に対しては人手より悪く，結局，両者の効果が相殺されて人手とほぼ同じ実行時間になったと考えられる.

図 4.28 は拡張配列添字関数法と従来の配列添字関数法を境界要素法（BEM）における線形方程式の係数行列を作成するプログラムに適用し，nCUBE2 上で実行した結果を比較したものである．2 次元配列に対して両方の次元をブロックデータ分散した.

4.2.7 まとめ

2 つの計算分割最適化，拡張配列添字関数法とガード削除法，を提案した.

最初の方法はループ中の代入文の左辺にある配列要素に関する 2 つの場合を扱うために，配列添字関数を使う従来の計算分散手法を拡張するものである.

2 番目の手法は 2 つの場合において計算分散によってループ中に生成されるガードを削除する手法である．特に，複数プロセッサに対する単一の文のループ繰返し範囲が異なる可能性のある場合におけるガード削除法は新規に提案されたものである．本手法は簡略な方法で HPF コンパイラに実装した．これらは Shallow Water プログラムと BEM プログラムに対して効果的であることがわかった.

参考文献

[1] J. M. Anderson, S. P. Amarasinghe, M. S. Lam, Data and Computation Transformations for Multiprocessors, *PPoPP '95*, pp.166-178, July 1995.

[2] R. Babb, A. Choudhary, L. Meadows, S. Nakamoto, V. J. Schuster, Retargetable High Performance Fortran Compiler Challenges, *COMPCON '93*, pp.137-146, February 1993.

[3] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, M-Y. Wu, Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers, *J. of Parallel and Distributed Computing* 21, pp.15-26, 1994.

[4] J. Fourier, Analyse de travaux de l'Academie Royale des Sciences, pendant l'annee 1824, partie mathematique, *Histoire de l'Academie Royale des Sciences de l'Institut de France*, 1827.

[5] High Performance Fortran Forum, *High Performance Fortran Language Specification ver.1.0*, Rice Univ., Jan. 1993.

[6] S. Hiranandani, K. Kennedy, C. W. Tseng, Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines, *ICS '92*, pp.1-14, July 1992.

[7] S. Hiranandani, K. Kennedy, C. W. Tseng, Preliminary Experiences with the Fortran D Compiler, *Supercomputing '93*, pp.338-350, 1993.

[8] W. Kelly, W. Pugh, E. Rosser, Code Generation for Multiple Mappings, *Frontiers '95*, pp.332-341, 1995.

[9] J. Levesque, Using High Performance Fortran on the Intel Paragon, Intel Users Group Meeting, May, 1994

[10] I. Miyoshi, K. Maeyama, S. Goto, S. Mori, H. Nakashima, S. Tomita, TINPAR: A Parallelizing Compiler for Message-Passing Multiprocessors (in Japanese), *JSPP '95*, pp.51-58, May 1995.

[11] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges, P. Banerjee, Advanced Compilation Techniques in the PARADIGM Compiler for Distributed-Memory Multicomputers, *ICS '95*, pp.424-433, July 1995.

第5章　結論

5.1　本研究により明らかにされたこと

安全安心な社会の実現や産業競争力の強化には現在行われているよりもより精密な計算機シミュレーションが必要であり，このために，高水準なプログラミング言語で書かれたプログラムを高効率に並列処理することが重要である．これを実現するために並列化コンパイル技術に着目し，高レベルなプログラミング言語が引き起こす課題をまとめ，その課題を解決する技術の検討を実施した．

第2章では，共有メモリ型並列プロセッサを対象として，Fortran 言語という高水準なプログラミング言語で書かれた逐次プログラムから高並列なコードを生成する上での課題である手続き呼出しを含むループの自動並列化に着目し，高並列だけでなく実行効率及びメモリ消費量の観点から並列化コンパイル技術の検討を行った．プログラムを並列化する時にはメモリ消費量やコード量の増加を伴うプログラム変換が必要になる場合が多い．プライベート化と呼ばれる変換もその一つである．従来，コモン変数をプライベートする場合，単一のコモン変数だけでなく，それを含むコモンブロックという変数群全体をプライベート化する必要があり，メモリ消費量を著しく増加させるという課題があった．そこで，メモリ消費量を従来より低減しながら並列度を向上させる技術としてプライベート化が必要なコモン変数のみを選択的にプライベート化する技術を提案しコンパイラに実装した．SPECfp92 ベンチマークプログラム集に含まれる 14 本中の 3 本のプログラムに本技術を適用した結果，そのうちの 2 本に対してプライベート化で発生するメモリ消費量を従来技術によるメモリ消費量の 1.3%及び 10%に抑えながら並列化率を各々5%向上させられることを明らかにした．一方，並列度を向上させるがコード量が増加する恐れがあるため実行効率を向上させるかについては不確定な技術として手続きクローニングと手続き間定数伝播の両方を適用させるコード生成技術を評価した．SPECfp95 中の turb3d プログラムに両方を適用した結果，どちらも適用しない場合とで並列化できるループは同じだが，1 から 16 プロセッサ上で 14 %から 33 %高速化した．高速化の原因は上記 2 つの処理によってプログラム中の式が簡単化された結果，コンパイラによるレジスタ利用効率が向上したためと判明した．

第3章では，分散メモリ型並列プロセッサを対象として，現在主流のメッセージパッシング型言語より高水準なデータ並列言語の一種である High Performance Fortran (HPF) 言語で書かれたプログラムから高効率なコードを生成する上での課題であるデータ分散解析技術に着目し，高水準な 2 レベルデータ分散指示文を精密に解析するコンパイル技術の検討を行った．2 レベルデータ分散指示文には ALIGN と言う指示文があり，これによりある配列の要素とテンプレートと呼ばれる仮想配列の要素を同一のプロセッサのメモリ上に置くことを指定できる．一方の配列要素の添字として他方の配列要素

の添字の1次式を指定できるため，プログラム中の配列参照の仕方に合わせる事によりプロセッサ間通信の少ない指定が可能となる．また，テンプレートのかわりに一般の配列を指定することが出来，さらに，データ分散指示文は関数の先頭に置けるだけでなく関数の途中にも置くことができる．これらの高水準な指示文によってプログラミングの生産性が非常に高まることを示したが，一方，コンパイラによる解析が非常に複雑になり，得られる解析結果もあいまいになってしまう懸念があった．そこで，精密に解析できるように通常1次元であるビット集合を2次元とした4つのデータフロー解析と複雑なALIGN指定を表現可能なデータ分散形状構造体を用いたデータマッピング解析を提案し実装した．うち1つのデータフロー解析は解析結果のあいまい性を低減するために導入した．また，これらデータフロー解析の収束性や分散性，およびデータマッピング解析の収束性も証明した．NPBベンチマークプログラム集に含まれるFTとSP, およびADI法プログラムに対して，プログラマにとって高生産だがコンパイラにとって解析が困難な指示文を含むバージョンと，プログラマにとって生産性が低いがコンパイラにとって解析が容易な指示文を含むバージョンを作成して分散メモリ型スーパーコンピュータSR2201上で評価した．その結果，両者とも同様なコードに変換され，同様な処理性能で実行された．これにより，本技術によって高生産な指示文が精密に解析されたことがわかった．

第4章では，分散メモリ型並列プロセッサを対象として，現在主流のメッセージパッシング型言語より高水準なデータ並列言語の一種であるHPF言語で書かれたプログラムから高効率なコードを生成する上での課題である計算分散技術に着目し，元のプログラム中の計算を複数のプロセッサに効率的に分散させるための条件とその時のコードを生成するコンパイル技術の検討を行った．

4.1節では一般な規則的データ分散が指示された配列参照を含む一般の1重ループをに対して，ループ中の文が同一のプロセッサで処理される場合に最適な計算分散を行う条件とその時のコード生成手法を検討した．HPFはblock-cyclic分散と2レベルマッピングという高生産な指示文を持つが，従来のコンパイル技術では1つの配列要素に対して2つの間接参照を含むコードが生成されるため，1つの配列要素をループ中で参照するたびに合計3回のロードが発生し，実行効率が低いという課題があった．そこで，ループ繰返しインデックス空間，配列添字空間，テンプレート空間，これらをデータ分散や計算分散によってローカル化した各空間，及び，これらの空間の間の対応関係を図式を用いて表現し，グローバル空間とローカル空間の間にある関係式がプロセッサ番号によらず常に成り立つ条件を求め，その時の計算分散コードを導いた．その条件とはループのストライド，配列添字式中の1次係数，及びALIGN添字式の1次係数の3つの積がblock-cyclic分散におけるブロックサイズの約数となることであり，この条件が成立した時に計算分散コードとしては最適な1重ループコードを生成した．また，本条件によらず，計算分散コードにおける添字式の規則性に関して従来より短い周期があることを示し，これよりdoループによるテーブル参照法コードを導いた．提案方法を複雑なblock-cyclic分散を持つプログラムに適用してSR8000上で実測した結果，最適計算分散コードが従来，block-cyclic分散に対して最速であるとされたテーブル参照法コードよりも10倍以上高速であること，並びに，doループによるテーブル参照法コードが上記従来法コードよりも高速であることが明らかになった．本結果は，一般のループ制御式を持つ多重ループ，一般の規則的なHPFデータ分散指示文が与えられた多次元配列，及び，互いに異なるループ制御変数による一般の1次式を各

次元の添字に持つ多重ループ中の配列参照，を含むプログラムに拡張可能である．

4.2 節ではブロックデータ分散を例に取り，ループ中の複数の文が必ずしも同一のプロセッサで処理されない場合に最適な計算分散を行う条件とその時のコード生成手法を検討した．従来，以下の 2 つの場合に効率的な計算分散を行うことは困難であるという課題があった．(a) 同じループ制御変数が異なる次元に現れる場合，(b) 複数のループ制御変数が一つの次元に現れる場合．そこで，ループ中の代入文の左辺にある配列要素に関して配列添字関数を使う従来の手法と各配列次元の制約に関する不定方程式を解くこと組み合わせた拡張配列添字関数法を提案した．境界要素法における係数行列作成プログラムに適用して nCUBE2 上で評価した結果，4 から 64 プロセッサ上で 3 から 30 倍程度高速であることが明らかになった．また，従来，以下の 2 つの場合に計算分散によって生成されたループ中の if 文を削除することが困難であり，実行効率が低いという課題があった．(c) 1 つのプロセッサに対して複数の文の計算分散後のループ範囲が異なる可能性がある場合，(d) 1 つの文の計算分散後のループ範囲が複数のプロセッサで異なる可能性がある場合．そこで，計算分散後に同じ条件式を持つ if 文が同じループの属するように元のループを分割することでガード削除手法を提案し実装した．米国国立大気研究所の Shallow Water ベンチマークプログラムに適用し nCUBE2 上で評価した結果，2 から 128 プロセッサ上で別方式による人手最適化コードとほぼ同様な処理性能であることがわかった．

5.2 今後の課題

共有メモリ型並列プロセッサを対象とした場合に Fortran77 レベルの言語で書かれた逐次プログラムから高並列なコードを生成するコンパイル技術や，分散メモリ型並列プロセッサを対象とした場合に HPF 言語の規則的データ分散指示文で書かれたプログラムから最適な計算分散コードを生成するコンパイル技術に関しては本研究以前の研究および本研究に示した方法によりかなり良いコードが生成できると考えられる．しかしながら，上記で示した範囲を超えたプログラムに対しては大きな課題が存在する．以下に主要なものを挙げる．

1. 共有メモリ型並列プロセッサを対象とした Fortran77 以外の言語に対する自動並列化
2. 分散メモリ型並列プロセッサを対象とした自動データ分散またはプログラマとの協調によるデータ分散
3. 分散メモリ型並列プロセッサを対象とした高水準で高効率なコード生成が可能な並列言語の提案とそのコンパイラの開発

第一の課題に関しては様々な種類の言語要素がコンパイラの解析にとって障害となる．Fortran 言語に限定しても Fortran90 以降，構造体，制限付きポインタ，複雑な引数渡しなどが導入され，コンパイラの解析を困難なものにしている．しかしながら，Fortran 言語は元々処理性能重視の言語であるので，言語に制限が多く，制限付きポインタがあっても並列化できる場合が存在する．一方，C 言語における制限のないポインタはコンパイラの解析にとって非常に大きな障害となっており，並列化解析に限らず様々な解析を困難にしている．制約付き C 言語という C 言語の機能を Fortran レベ

ルの機能に制限する言語も提案されており，それに対して自動並列化は可能であることが示されているが，通常の C 言語を制約付き C 言語に変換するのは容易でないため，自動並列化は依然問題である．

第二の課題のうち，自動データ分散に関しては 10 数年前に一時研究されていたが，その後はほとんど見られなくなった．研究を困難にする課題がいくつか考えられる．一つ目はデータ分散の組合せが多いことである．プログラムに存在する配列の次元毎に様々なデータ分散を指定できる他，プログラムの実行途中でデータ分散を変更しても良いため，組合せは膨大である．二つ目は各データ分散毎に最適なコードを生成することが困難なことである．本論文で示したように規則的データ分散に対する計算分散まではかなり可能となったが，あらゆるデータ分散やあらゆるプログラムに対して効率的なプロセッサ間データ通信を生成できるコンパイラは存在しない．プロセッサ間データ通信の起動には 1000 サイクル以上の時間がかかるため，少しでも間違った通信を出力するとコード処理性能は大幅に低下する．三つ目は二つ目とも関係するが，データ分散の善し悪しの判断が困難な点である．理想的にはプロセッサ間データ通信を含むコードを生成してコンパイラ内で処理サイクル数を計算し，最もサイクル数の少ないデータ分散を選択すれば良いが，二つ目の個所で説明したようにコード生成自体が困難であり，また，仮にコード生成ができたとしてもプログラムの実行サイクルは入力データに依存するためコンパイル中に評価することは困難なためである．そこで，プログラマと協調し，小さな入力データで実行して大きな入力データによる実行時間の傾向を把握しながら，様々なデータ分散を試行するようなシステムが考えられるが，まだそのようなものは提案されていないようである．

第三の課題に関しては今も様々な言語が提案され，コンパイラが開発されている．米国のスーパーコンピュータ用に UPC，ZPL，Chapel，X10，Fortress 等の並列言語が提案され，日本でも HPF の拡張である JHPF の他，XcalableMP といった並列言語とそのコンパイラが提案されている．並列プログラミングは共有メモリ型並列プロセッサでは比較的容易で，分散メモリ型並列プロセッサでは困難になる．結局，並列プロセッサシステムのうちメモリシステムをプログラマにどのように見せるかを，ハードウエア，言語，及び，コンパイラで協調して設計することが成功の鍵と考えられる．

付 録A　定理 1の証明

定理 1を証明する準備として，以下に 2つの用語を定義する．

定義 8 再分散点列
πを点p'からpに至るパスとし，$p_1, \cdots, p_s$をパスπにそって実行順に並んだ再分散点とする．この時，以下の順序付きの点列をパスπに沿った再分散点列と呼ぶ．

$$f = f(p', p) = \{p'; p_1; \cdots; p_s; p\}$$

また，$\Sigma(p', p)$をそのような点列すべてからなる集合とする．さらに，$\tilde{f}$をfと同じ点列からなるが，それらの順序を無視した集合とする．即ち，

$$\tilde{f} = \{p', p_1, \cdots, p_s, p\}.$$

定義 9 再分散点列のMDへの適用
$M(x, p')$を点p'における変数xのMDの集合とし，$d \in M(x, p')$，$f \in \Sigma(p', p)$とせよ．この時，$F_f(d)$を以下の指示文をdに順に適用することによって得られる，点pにおけるMDの集合と定義する．

- 点列$p_1, \cdots, p_s$直前のデータ再分散指示文
- 点p直前に存在するかもしれない上記とは別のデータ再分散指示文 □

この時，R_CとD_Rの定義より，$M(x, p)$と$D_R(x, p)$は$f \in \Sigma(p', p)$を使って以下のように表現できることがわかる:

$$M(x, p) = \bigcup_{p' \in R_C(x,p)} \bigcup_{f \in \Sigma(p', p)} F_f(M(x, p')) \tag{A.1}$$

$$D_R(x, p) = \{p' \in R_C(x, p) | \exists f \in \Sigma(p', p) \text{ s.t. } \tilde{f} \cap R_C(x, p) - \{p\} = \{p'\}\}. \tag{A.2}$$

定理 1の証明

最初に，3.3.3 節における式 (3.3) の右辺$\bigcup M(x, p')$が左辺$M(x, p)$に含まれることを証明する．式 (A.2) を満たす任意の$p' \in D_R(x, p)$と$f \in \Sigma(p', p)$に対して，$\tilde{f}$の中に$M(x, p')$を変化させる再分散点はない．これは以下の理由からわかる．

- あるブロックの入口点pはブロックの最初の点なのでその直前にはどんな再分散点もない．
- p'直前の指示文で変化した後のMDが$M(x, p')$なので，p'は$M(x, p')$を変化させない．

- 式 (A.2) は p と p' のみが $\tilde{f} \cap R_C(x,p)$ の要素の候補であることを示す.

よって，以下を得る.

$$M(x,p') = M(x,p) = F_f(M(x,p')) = \bigcup_{f \in \Sigma(p',p) \quad \mathrm{s.t.} \quad \tilde{f} \cap R_C(x,p) = \{p',p\}} F_f(M(x,p')). \tag{A.3}$$

式 (A.1), (A.2), (A.3) より以下を得る.

$$\begin{aligned} \bigcup_{p' \in D_R(x,p)} M(x,p') &= \bigcup_{p' \in D_R(x,p)} \bigcup_{f \in \Sigma(p',p) \ \mathrm{s.t.} \ \tilde{f} \cap R_C(x,p) = \{p',p\}} F_f(M(x,p'))) \\ &\subseteq \bigcup_{p' \in R_C(x,p)} \bigcup_{f \in \Sigma(p',p)} F_f(M(x,p')) = M(x,p). \end{aligned}$$

次に，式 (3.3) の左辺 $M(x,p)$ が右辺 $\bigcup M(x,p')$ を含むことを証明する．任意の $p_1 \in R_C(x,p)$ と任意の $f_1 = \{p_1; \cdots ; p_s; p\} \in \Sigma(p_1,p)$ に対して，以下を得る.

$$\tilde{f_1} \ \cap \ R_C(x,p) \quad = \quad \{p_{\sigma(1)}, \cdots, p_{\sigma(m)}, p\} \quad \subset \quad \{p_1, \cdots, p_s, p\} \ \ (m \quad \geq \quad 1) \tag{A.4}$$

ここで，σ は包含写像: $\{1, \cdots, m\} \mapsto \{1, \cdots, s\}$ を表し，もし，$i < j$ なら $\sigma(i) < \sigma(j)$ となり，さらに以下を満たす.

$$p_{\sigma(m)} \in D_R(x,p). \tag{A.5}$$

式 (A.5) は式 (A.2) に含まれる p' と f を各々，$p_{\sigma(m)}$ と $\{p_{\sigma(m)}; \cdots ; p\}$ で置き換えることにより証明される．また，$p_1 \in R_C(x,p)$，即ち，$M(x,p_1)$ は p に到達するので当然，途中の点 $p_{\sigma(m)}$ にも到達する．即ち，

$$p_1 \in R_C(x, p_{\sigma(m)}). \tag{A.6}$$

この関係と 2つの式 (A.1) と (A.5) とによって以下を得る.

$$\begin{aligned} \bigcup_{p_1 \in R_C(x,p_{\sigma(m)})} \bigcup_{f \in \Sigma(p_1,p_{\sigma(m)})} F_f(M(x,p_1)) = M(x,p_{\sigma(m)}) \\ \subseteq \bigcup_{p' \in D_R(x,p)} M(x,p'). \end{aligned} \tag{A.7}$$

式 (A.7) は任意の $p_1 \in R_C(x,p)$ に対して成り立つ．p はデータ再分散指示文直後の点ではないので，x のデータ分散は $p_{\sigma(m)}$ と p の間で一定である．このことを使うと，$f \in \Sigma(p_1, p_{\sigma(m)})$ と $f' \in \Sigma(p_1,p)$ に対して以下を得る.

$$F_f(M(x,p_1)) = F_{f'}(M(x,p_1)). \tag{A.8}$$

よって

$$\bigcup_{f \in \Sigma(p_1,p_{\sigma(m)})} F_f(M(x,p_1)) = \bigcup_{f' \in \Sigma(p_1,p_{\sigma(m)})} F_{f'}(M(x,p_1)) = \bigcup_{f' \in \Sigma(p_1,p)} F_{f'}(M(x,p_1)). \tag{A.9}$$

式 (A.1), (A.7), (A.6), および (A.9) により以下を得る

$$M(x,p) = \bigcup_{p_1 \in R_C(x,p)} \bigcup_{f' \in \Sigma(p_1,p)} F_{f'}(M(x,p_1))$$
$$\subseteq \bigcup_{p_1 \in R_C(x,p_{\sigma(m)})} \bigcup_{f \in \Sigma(p_1,p_{\sigma(m)})} F_f(M(x,p_1)) \subseteq \bigcup_{p' \in D_R(x,p)} M(x,p'). \quad \text{(A.10)}$$

□

付 録B　定理 5の証明

まず，$f_a i_a \neq 0$ の場合を証明する．ループ L1 はグローバル添字にプロセッサ q におけるローカル添字を対応させる配列 $\mathcal{L}$ を計算するコードである．後の証明でわかるように，ループ L3 のループ制御変数 i は，プロセッサ q にマッピングされる A の部分配列のみを参照するような範囲を動く．よって，配列 A の添字は単にローカル添字 $\mathcal{L}(i_a i + i_b)$ を使って表現できる．

以下，ループ L2 と L3 に対するループ範囲（上限値，下限値，及びストライドの組）について証明する．プロセッサ q にマッピングされる標準テンプレート I_{pbc} の添字集合は 3．1 節より，

$$\bigcup_{j\in[0:c-1:1]} [pbj + bq : pbj + bq + b - 1 : 1] \tag{B.1}$$

となる．ここで，$c = \lceil e/pb \rceil$ である．また，

$$\bigcup_{j\in[l:u:m]} X_j \tag{B.2}$$

は，区間 $X_l, X_{l+m}, \cdots, X_{l+m\lfloor (u-l)/m \rfloor}$ がこの順序に並んでできる区間集合を表す．順序も考慮するのはループ範囲を求めるためである．

次に以下の方針で計算分散後のループ範囲を求める．

1. 式 (B.1) より計算分散後のループ範囲を計算．
   (a) 式 (B.1) に $(gfkh)^{-1}$ を作用させ，その結果と $\mathbf{Z}$ との交わりをとる．
   (b) 上記結果に h を作用させ，その結果と $[l:u:1]$ との交わりをとる．
2. 式 (B.1) の j の範囲からループ L2 の範囲を計算．

この方針に関して以下にいくつかの注意をする．

注意 1： 方針 (1)(a) は本来は g^{-1} 等を作用させるたびにその結果と A 等との交わりをとるべきと考えられる．しかし，上記の各関数が 1 対 1 写像なので，以下の補題 6 により，$(gfkh)^{-1}$ を一度に作用させた後に $(L^0 \subset)\mathbf{Z}$ との交わりをとっても同じ結果を得る．

補題 6 *$f : X \mapsto Y$，$g : Y \mapsto Z$ を 1 対 1 写像とする．この時，以下が成り立つ:*

$$f^{-1}(g^{-1}(Z) \cap Y) \cap X = (gf)^{-1}(Z) \cap X. \tag{B.3}$$

証明 f を 1 対 1 写像とする時，以下が成り立つ：

$$f(V \cap W) = f(V) \cap f(W).$$

式 (B.3) の両辺の各々に gf を作用させ，この等式を用いることで補題は証明される．

□

注意 2： 方針 (1) は，本来，$(gfk)^{-1}$ を作用させ，その結果と $[l:u:m]$ との交わりをとるべきである．しかし，交わりの計算が困難なため，$[l:u:m]$ をストライドと上下限値の 2 つに分解して交わりを計算した．即ち，(a) で $(gfkh)^{-1}$ を作用させ，$\mathbf{Z}$ との交わりを取った後で h を作用させることでストライド m を，(b) で $[l:u:1]$ との交わりをとることで上下限値との交わりをとった．

以下，上記方針に従って証明する．まず，

$$(gfkh)^{-1}(i) = (i + T_l - f_b - f_a i_b - f_a i_a l)/\pi$$

より，方針 (1)(a) に従って式 (B.1) の区間は以下となる：

$$\bigcup_{j \in [L_c:U_c:sign(\pi)*1]} [L(j):U(j):1]. \tag{B.4}$$

ここで，$\pi < 0$ なら各区間の上下限は逆になり，区間列も逆順になることから以下を得る．

$$L(j) = \lceil l_j \rceil, U(j) = \lfloor u_j \rfloor,$$

$$\begin{aligned} &\pi > 0 \text{ なら} l_j = t(j,q,0), u_j = t(j,q,b-1), \\ &\pi < 0 \text{ なら} l_j = t(j,q,b-1), u_j = t(j,q,0). \end{aligned} \tag{B.5}$$

但し，

$$t(j,q,x) = (pbj + bq + x - gfk(l))/\pi.$$

これと，方針 (1)(b) より，定理 5 のループ L3 を得る．

次に，方針 (2) に従って j のループ範囲を計算する．

$$w(x) = gfk(x) = f_a(i_a x + i_b) + f_b - T_l \tag{B.6}$$

とおくと，

$$m > 0 \text{ なら } l \le u, m < 0 \text{ なら } l \ge u$$

であり，$w(x)$ の x の係数は $f_a i_a$ なので

$$\begin{aligned} &\pi = f_a i_a m > 0 \text{なら } w(l) \le w(u), \\ &\pi = f_a i_a m < 0 \text{なら } w(l) \ge w(u). \end{aligned} \tag{B.7}$$

また，l は標準テンプレート I_{pbc} において $C_l = \lfloor w(l)/pb \rfloor$ 番目の分散周期に，u は $C_u = \lfloor w(u)/pb \rfloor$ 番目の分散周期に含まれる．

ここで，l や u がマッピングされないプロセッサ q に対して，その j の範囲は上記の範囲より狭くなる可能性がある．しかし，範囲の厳密な計算は方針 (1) (b) で行なわれ

ているので，ここでは少し広い範囲であっても間違いではない．よって，定理 5 のループ L2 を得る．以上により，$f_a i_a \neq 0$ の場合に定理が証明された．

次に，$f_a i_a = 0$ の場合に対して証明する．この時，$A(i_a i + i_b)$ は $T(f_a(i_a i + i_b) + f_b) = T(f_a i_b + f_b)$ と同じプロセッサにマッピングされるので，図 5 よりそのプロセッサ番号は以下となる．

$$\lfloor (f_a i_b + f_b - T_l) \bmod pb/b \rfloor .$$

よって，$f_a i_a = 0$ の場合に定理が証明された． □

謝辞

本論文を執筆するにあたって，東京工業大学大学院総合理工学研究科物理情報システム専攻 前島英雄教授には，大学院社会人博士課程コースに入学以前より懇切丁寧に御指導を頂き，本論文の執筆に際しても更なる御指導並びに御鞭撻を賜りましたことを深く感謝いたします．また，本論文の作成に当たり，懇切な御指導と御支援を頂きました，東京工業大学大学院総合理工学研究科物理情報システム専攻 羽鳥好律教授，小林隆夫教授，黒澤実准教授，杉野暢彦准教授にここに心からの感謝の意を表します．

本論文の研究は筆者の勤務する株式会社 日立製作所 システム開発研究所において行ったものであり，本研究の遂行に当たっては，筆者がシステム開発研究所に移って以来の歴代の所長でおられた，春名公一，片岡雅憲，和歌森文男（現日立インフォメーションアカデミー経営管理部），小坂満隆（現北陸先端技術大学教授），前田章（現日立製作所情報制御システム社ＣＴＯ），堀田多加志（現日立製作所システム開発研究所所長）の各氏をはじめとして，筆者の所属センターの歴代センター長でおられた，山本彰（現日立製作所研究開発本部 技師長），松並　直人（現日立製作所システム開発研究所 情報プラットフォーム研究センタ長）の各氏，ならびに，筆者の所属部の歴代部長でおられた，磯辺寛，前澤裕行（現日立ソフトウェアエンジニアリング株式会社執行役常務），増位庄一（現日立製作所研究開発本部 主管技師長），菊池純男（現日立製作所 情報・通信システム社 事業部員），小島啓二（現日立製作所中央研究所所長），小泉忍（日立製作所モノづくり技術事業部 組込みシステム改革戦略センタ），佐川暢俊（現日立製作所システム開発研究所 情報サービス研究センタ長），石崎健史（日立製作所 情報・通信システム社 経営戦略室事業戦略本部　HC 統括部部長），樋口達雄（現日立製作所システム開発研究所 第 2 部部長）の各氏においては，多大なる御支援を頂きました．厚く感謝の意を表します．

また，筆者がシステム開発研究所に移って以来の筆者の所属ユニットの歴代ユニットリーダーでおられた菊池純男，十山圭介（現日立製作所中央研究所主任研究員），飯塚孝好（現日立製作所情報通信システム社プラットフォームソリューション事業部主任技師），久島伊知郎（現日立製作所システム開発研究所主任研究員），西山博康（現日立製作所システム開発研究所ユニットリーダ）の各氏においては，研究全般にわたり暖かい御指導を頂きました．ここに深く感謝の意を表します．筆者が本論文をまとめることができたのも，これらの方々の的確な御指導と励ましのおかげであります．

また，根岸清（日立製作所情報・通信システム社 ソフトウエア事業部技師）及び人見洋一（日立製作所情報・通信システム社 ソフトウエア事業部技師）の各氏には SR8000 の使用に際して御協力いただきました．會田一弘氏（日立製作所情報・通信システム社 ソフトウエア事業部基ソ本管理センタ長）には SR2201 の使用を許可して頂きました．太田寛氏（日立製作所情報・通信システム社 経営戦略室主任技師）には我々の評価のベースとなった，FT 及び SP プログラムの HPF 版を提供して頂きました．日立東京

センターには nCUBE2 を使わせて頂きました．小林篤，黒澤隆（日立東日本ソリューションズ主任技師）の各氏には HPF プロトタイプコンパイラの構文解析部，ソースコード生成部，及びメッセージコアレッシング部を開発して頂きました．ここに深く感謝の意を表します．筆者が本論文における開発・評価を進めることができましたのも，これらの方々のおかげであります．

最後に，西谷康仁（日立製作所情報・通信システム社 ソフトウエア事業部主任技師），中島恵（日立製作所情報・通信システム社 ソフトウエア事業部担当部長），吉川聡（日立製作所情報・通信システム社 ソフトウエア事業部主任技師），を初めとする多くの方々には本論文の研究内容に関して多くの御議論・御助言を頂きました．ここに，厚く感謝の意を表します．

本研究の一部はリアルワールド・コンピューティング・パートナーシップによって支援されました．

本論文に関する発表論文

学術論文

1. 佐藤真琴. HPF におけるデータ分散の図式表現と効果的計算分散法, 情報処理学会論文誌, Vol. 46, No. 9, pp. 2347-2360, Sep. 2005.
2. Makoto Satoh, Kiyoshi Negishi, and Atsushi Kobayashi. Analysis of two-level data mapping in an HPF compiler for distributed-memory machines, Parallel Computing 32, pp. 280-300, 2006.

国際会議

1. Makoto Satoh, Yuichiro Aoki, Kiyomi Wada, Takayoshi Iitsuka, Sumio Kikuchi. Interprocedural Parallelizing Compiler WPP and Analysis Information Visualization tool Aivi, Proceedings of The Second European Workshop on OpenMP (EWOMP2000), pp. 38-47, 2000.
2. Makoto Satoh, Takashi Hirooka, Kiyomi Wada, Fujio Yamamoto. Program Partitioning Optimizations in an HPF prototype Compiler, in Proc. of Twentieth Annual International Computer Software and Applications Conference (Compsac '96), pp. 124-131. 1996.

国内学会・研究会

1. 佐藤真琴:データ分散の図式表現と計算分散公式の提案及び評価, 情処ハイパフォーマンスコンピューティング研究会研究報告, Vol.2001, No.77(20010725) pp. 81-86, 2001-HPC-87-15
2. 佐藤真琴、青木雄一郎、和田清美、太田寛、飯塚孝好:手続き間自動並列化コンパイラ WPP の評価, 計算機アーキテクチャ研究会研究報告, Vol.2001, No.10(20010126) pp. 17-22, 2001-ARC-141-4
3. 佐藤真琴, 根岸清, 小林篤 :HPF 処理系における再分散解析機能の開発, 情処ハイパフォーマンスコンピューティング研究会研究報告, Vol.96, No.81(19960828) pp. 63-68
4. 佐藤真琴, 青木雄一郎, 菊池純男:手続き間並列化コンパイラ WPP の試作:変数プライベート化技術, 情処第 56 回平成 10 年前期全国大会, No.1(19980317) pp. 284-285, (1998).

5. 佐藤真琴，広岡孝志，和田清美，山本富士男：HPF 処理系における最適化機能：実行時判定の削除，情処第 51 回平成 7 年後期全国大会, No.6(19950920) pp. 91-92, (1995).