計算機クラスタ環境における
プログラム並列化処理に関する研究

朝倉 宏一

概要

近年の計算機性能やネットワーク性能の発達にともない，ワークステーションや高性能 PC をネットワークで接続して並列計算機を構成する手法が提案されており，この手法で構成された計算機環境はワークステーション・クラスタ，計算機クラスタと呼ばれている．従来の並列処理環境である並列計算機やスーパーコンピュータと比べてコスト・パフォーマンスが高いため，最近は並列処理環境として広く利用されている．しかし，並列計算機やスーパーコンピュータに対して，計算機クラスタ環境はプロセッサの処理速度とネットワークの通信速度の速度差が相対的に大きく，並列処理の効率において重大な問題となる．並列計算機やスーパーコンピュータを対象とする従来の並列処理手法は，高速な通信網を有することを前提としている．したがって，従来の並列処理手法を計算機クラスタ環境へそのまま適用することは，並列処理の効率を考えると不適切であり，この問題に対処した新しいプログラム並列化手法や並列処理手法が必要となる．

本論文では，計算機クラスタ環境において効率よい並列処理を実現するためのプログラム並列化手法について述べる．ネットワークの通信速度が低速であるという計算機クラスタ環境の性質を考え，低速な通信速度に対処した並列処理手法を開発する必要がある．すなわち，異なる計算機で実行されるタスク間の通信処理を削減する並列処理手法の開発が重要となる．ここで，タスクとは計算機クラスタ環境上での実行においてスケジューリングの単位となる実行単位である．例えば，UNIX ワークステーションではプロセスがタスクに対応する．プログラムを並列化するこ

とにより複数のタスクが生成され，タスク間通信により相互にデータを交換することで処理が行われる．

一般に，並列処理の効率と通信処理の発生頻度は，並列化処理で生成されるタスクの粒度に左右される．ここでタスクの粒度とは，生成されたタスク内に含まれるソース・プログラムにおける文の量を規定するものである．本論文では，プログラム中の文やイタレーション単位のタスクを細粒度タスク，手続き単位のタスクを粗粒度タスクとそれぞれ定義する．細粒度タスクによる並列処理ではプログラムの並列性を最大限抽出可能である反面，タスク間の通信頻度が高くなる．逆に，粗粒度タスクによる並列処理ではタスク間の通信頻度は低く抑えられるが，プログラムの並列性を抽出して活用することが困難である．計算機クラスタ環境で効率よい並列処理を実現するためには，上に述べた通信頻度と並列実行性のトレードオフを考慮したプログラム並列化手法やタスクの並列実行手法など，新しい並列処理手法の開発が必要となる．

計算機クラスタ環境で効率よい並列処理を実現するため，上記の問題に対処した手法として，本論文では粗粒度タスクを用いた直接的アプローチと，細粒度タスクを用いた間接的アプローチを提案する．粗粒度タスクを用いた直接的アプローチではタスク間の通信頻度を低く抑えることを目的として，プログラム中の関数単位で並列処理を実現するための関数呼出し文並列化手法を提案する．関数をタスク生成の単位とすることにより，変数空間をタスクに局所化し，タスク間の余分な通信処理を削減することができる．また，細粒度タスクを用いた間接的アプローチではタスク間で発生する通信処理を隠蔽する手法として，計算と通信のオーバラップによりループ文を並列実行させるループ文漸進処理手法を提案する．ループ文漸進処理手法では，同一配列データを共有するループ文に対して，各イタレーションの実行と計算結果の通信をオーバラップさせることで，タスク間通信に起因するタスクの実行中断状態を回避し，ループ文を効率よく並列実行させることができる．本論文で提案するそれぞれのアプローチにより，計算機クラスタ環境上での並列処理で発生する問題を解決でき，計算機クラスタ環境で効率よい並列処理が可能となる．

目次

第 1 章

序論

1.1 研究の背景

近年，高性能な計算機の普及にともない，計算機で扱う問題も大規模になり，実行されるプログラムの規模，およびプログラム中で扱われるデータの量も増大の一途をたどっている．この傾向は特に，気象予測シミュレーションや有限要素法を用いた物性解析など，大規模計算と呼ばれる研究分野で著しい．大規模計算の研究分野では扱うデータ量や計算量の増加が計算結果の精度向上につながる．例えば，気象予測シミュレーションにおいて地面を区切るメッシュが細かくなれば，計算精度が向上したり従来観測できなかった現象なども観測できるが，計算量が膨大になる[1]．したがって，使用可能な計算機資源の量により計算精度を決定する必要があり，プログラムの計算精度とプログラムの計算時間にトレード・オフが存在する．このような大規模計算の研究分野において有意な計算結果を効率よく得るために，複数のプロセッサを同時に用いてプログラムを実行させる並列処理の概念が登場した．

1.1.1 計算機環境の変遷

初期の並列処理は並列計算機，スーパーコンピュータなどが主要プラットホームであった．単一のプロセッサの性能向上には物理的限界があり，計算機の性能向上の

ために複数プロセッサによる並列処理の概念が採り入れられた.

ベクトル・プロセッサは，ループ文における配列操作のように，多くのデータに対して単一の演算を施す処理を高速化する装置である [2,3]. プログラム中から同じ演算を適用する多くのデータを抽出し，それらのデータをベクトル・レジスタに割り当てて計算することで，処理の高速化を図る手法である. データの抽出によるベクトル長の確保がベクトル・プロセッサの処理効率を左右する. 現在，このベクトル・プロセッサの技術は Pentium や Athlon など，PC で使用されるプロセッサにおいてもマルチメディア命令として採り入れられている.

ベクトル・プロセッサは単一プロセッサに並列処理向けの機能を採用したハードウェアであるが，マルチ・プロセッサはプロセッサを複数用いることにより並列処理を実現する計算機構成である. マルチ・プロセッサ型並列計算機の場合，プロセッサとメモリの構成により以下の形式が存在する.

1. 共有メモリ型並列計算機

2. 分散メモリ型並列計算機

3. 分散共有メモリ型並列計算機

共有メモリ型並列計算機はすべてのプロセッサがバスを共有し，同じメモリ空間を有する計算機である [4,5]. プロセッサ間でのデータ授受はメモリ上のデータの読込み / 書込みで行われる. したがって，後述する分散メモリ型並列計算機と比較するとプロセッサ間通信が高速である. また，すべてのデータに対してすべてのプロセッサが平等にアクセス可能であるので，データの分散配置などを考える必要がなく，並列プログラム開発が簡単であるという利点もある. しかしながら，プロセッサ数が増加するとバスの競合などにより性能が低下し，スケーラビリティが低いという欠点がある. そのため，大規模な並列計算機の構築は難しく，性能が低下する. 現在では，バスの競合による性能低下を回避するためにクロスバ・スイッチなどを

用いた構成により性能向上を図った並列計算機も開発されているが，後述の計算機構成と比較するとスケーラビリティは低い．

分散メモリ型並列計算機は，すべてのプロセッサが独立のメモリ空間を有しており共有メモリは存在せず，プロセッサ間を高速結合網で接続した構成の並列計算機である [6–8]．プロセッサ間のデータの授受は結合網を介したメッセージ・パッシングで行われる．結合網の規模にもよるが，バスの競合などを考える必要がないのでスケーラビリティが高く，大規模な並列計算機システムを構成可能である [4]．しかし，プロセッサ間の通信はすべて結合網を通して行われるので，共有メモリ型計算機と比較するとプロセッサ間の通信速度は低い．また，メモリが分散されておりデータの配置方法などを考えなければならないので，共有メモリ型並列計算機と比較すると並列プログラム開発が困難である．

分散共有メモリ型並列計算機は，上記の計算機構成の中間的な構成である．すなわち，プロセッサ数の少ない共有メモリ型並列計算機を一つのプロセッサ・クラスタとし，プロセッサ・クラスタを分散メモリ型並列計算機で用いられる高速結合網で接続した構成である [9]．同一プロセッサ・クラスタ上のプロセッサとは共有メモリを介し，他のプロセッサとは結合網を介し，それぞれデータの授受を行うことができる．結合網上にすべてのプロセッサからアクセス可能な共有メモリを有する計算機構成もある．共有メモリ型，分散メモリ型の両方の計算機構成の利点をあわせ持った構成であり，近年の並列計算機の標準的な構成となっている．ただし，効率よい並列処理のためのプログラミングは困難であるという欠点がある．すなわち，プログラムが実行されるプロセッサの位置により，メモリを参照してデータを授受するか，結合網を介してデータを授受するかが異なり，またその処理速度も異なるので，プログラムの性質と計算機の構成を考えプログラムをプロセッサに適切に割り当てる処理が必要となる．

これら並列計算機，スーパーコンピュータに対して，近年の計算機性能やネットワーク性能の発達にともない，ワークステーションや高性能 PC をネットワークで接続して並列計算機を構成する手法が提案されており，ワークステーション・クラスタ，

計算機クラスタと呼ばれている [10–13]．上記の並列計算機やスーパーコンピュータに比べてコスト・パフォーマンスが高いという利点により，最近は並列計算機環境としてよく利用されている．

1.1.2 並列処理手法の推移

初期の並列プログラム開発においては，並列処理の対象となるプログラムの並列性をユーザが解析し，プログラムをユーザ自身が書き換え，並列処理を実現するのが一般的であった [2,14,15]．並列計算機にはプロセッサ間通信処理やタスク起動処理のための並列処理ライブラリが提供され，その並列処理ライブラリを用いてユーザが並列プログラムを開発し，並列処理を実現するのが一般的であった．並列処理ライブラリは並列計算機の特性を最大限に活用可能なように設計されているので，ライブラリの特性を活かして並列プログラムを作成することにより効率よい並列処理が可能であった．しかしその反面，通常のプログラム開発と比較すると，並列プログラム開発は非常に困難な作業であった．並列処理では複数のプロセッサを効率よく動作させるようにプログラムを開発しなければならない．そのため，通常の逐次プログラム開発では発生しない，以下のような問題が発生する．

負荷分散処理 複数のタスクが同時に実行されることで効率よい並列処理が可能となる．したがって，並列処理の対象となっているプログラムを分割して各プロセッサに割り当てるとき，各プロセッサの負荷がほぼ均等になるように割り当てなければならない．

同期タイミング調節 並列処理では，各プロセッサで実行されるタスクが相互通信により情報を交換して計算を進める．このとき，データ受信のためタスクの実行が一時的に停止して実行遅延状態が発生する状況を回避するため，タスク間通信における同期タイミングを適切に調節しなければならない．

デッドロック回避 同じ計算機資源を排他的に使用したり，他のタスクからのデータ

受信によりタスクの実行を開始，または再開するタスクが存在するとき，デッドロックが発生する可能性がある [16]．したがって，デッドロックが発生しないように，プログラムの分割方針やタスク間通信の処理方針を決定しなければならない．

このような問題を発生させないようにプログラムを開発する必要があるので，初期の並列プログラム開発は非常に困難であった．また，提供されている並列処理ライブラリはそれぞれの並列計算機の特徴を活用するため並列計算機によって仕様が異なっているのが普通であり，並列計算機間でのプログラムの相互運用性も低かった．

このような状況に対し，プログラミング言語の拡張や，新しいプログラミング言語によって並列プログラムの開発を支援する手法が提案された [17–21]．初期のプログラミング言語では，コメント文としてプログラムの並列化手法を記述することで，コンパイラに並列化戦略を指示し，プログラムを並列化するものが多かった [22]．コメント文に記述されたプログラム分割やデータ分散配置などの並列化戦略に基づいて，プログラムが自動的に分割されるので，ユーザは上で述べた複雑な作業から解放され，プログラムの並列性を抽出する作業に集中することができた．このプログラミング環境の変化は，プログラムの依存関係解析技術の発達に因るところが大きい [23]．プログラムの並列化処理においては，プログラム中のデータ依存関係の解析が重要である．データ依存関係はプログラム中の文におけるデータの参照・更新関係から導かれ，プログラム実行時の計算結果の正当性を保証する文の実行順序を規定する関係である [24,25]．すなわち，データ依存関係にある文の実行順序を変更しなければ，プログラムを逐次的に実行したときと同じ計算結果を得ることができる．データ依存関係の解析により，プログラムに内在する並列性が自動的に抽出され，コンパイラにより並列化処理が可能となった．また，if 文など条件分岐文においては，条件分岐文とその条件により実行されるか否かが決定される文との間に存在する関係として制御依存関係が定義された [26]．制御依存関係の解析技術により投機処理など柔軟な並列処理が可能となった [27]．これらのプログラム中の依存関係解析技

術の発達により，コンパイラによる並列化処理の自動化が可能となった [28,29].

さらに，プログラムの解析技術の発展により，ユーザからの指示をなくしてプログラムの並列性を抽出する完全な自動並列化コンパイラが出現した．プログラム全体にわたる並列性を自動的に抽出するのは非常に困難であり，多くの自動並列化コンパイラはプログラム中の do 文や for 文などのループ文を並列化の対象としている．つまり，プログラム中のループ文を対象としたプログラム再構築が自動並列化コンパイラにおける一般的な並列化手法である [30,31]．ループ文を並列処理するとき，イタレーション間の依存関係解析が重要となる．イタレーション間に依存関係が存在しない，すなわちそれぞれのイタレーションが独立に実行可能であることを解析できれば，doall 並列処理により全イタレーションを並列実行することができる [32,33]．また，ループ運搬依存と呼ばれるイタレーションに跨る依存関係が存在してもループ文を並列実行できる doacross 並列処理も提案されている [34,35]．その他，doalong 並列処理 [36] など様々なループ文並列化手法が提案され，実用に供されている [37–39].

ループ文のみにおける並列性抽出にとどまらず，プログラム全体から並列性を抽出し，より効率よい並列処理を目指す並列化手法も提案されている [40–42]．例えば，OSCAR コンパイラでは，ループ文以外にも基本ブロック間での並列性や文間の並列性を抽出し，対象とする並列計算機において効率よく実行させるマルチグレイン並列処理が行われている [43–46]．その他にも様々な並列化手法が提案されている [47–49].

また，従来は並列計算機ごとに独自であった通信ライブラリを共通化することで，並列プログラムの互換性，相互運用性を向上させる手法も一般化している．PVM は当初ワークステーション間での並列処理を容易にするためのランタイム・システムとして開発されたが，様々な並列計算機で動作可能となるよう改良され，PVM を用いて記述された同じ並列プログラムが様々な並列計算機環境で実行可能となった [50]．MPI は並列計算機のための通信処理ライブラリとして標準化された規格である [51,52]．1.1 節でも述べたように，従来はそれぞれの並列計算機の特徴を最大限

に利用可能なように開発された独自の通信ライブラリが提供されており，計算機の変更による並列プログラムの移植作業が困難であった．通信処理ライブラリとしてMPI が標準化されたことで，様々な並列計算機が相互利用可能となり，並列プログラム開発の容易さはもちろん，並列プログラムの蓄積なども可能となった．

このような様々な並列化手法や自動並列化コンパイラの開発，通信ライブラリの共通化などにより，並列プログラミング作業の効率は格段に向上した．

1.1.3 並列プログラミングのケース・スタディ

上記で述べた様々な並列化手法やコンパイル技術の開発，通信ライブラリの共通化などにより，初期と比較すると並列プログラミング作業の効率は向上した．計算機やプログラミングに関して高い知識を有する研究者などにとっては特に顕著であった．しかしながら，一般の利用者にとって並列プログラム開発環境の整備はまだまだ不十分であると考えられた．そこで，我々は情報工学科の学部 3 年生に対して並列プログラミング演習を実施し，演習の進捗状況やレポートなどから，一般のプログラム開発者の並列処理への適用度について調査した [53,54].

我々は，並列処理教育の一貫として 1995 年度から 1997 年度の 3 年間にわたり，名古屋大学工学部情報工学科の 3 年生に対して「プログラミング演習」の一部として並列プログラミング演習を行った．演習には富士通社製の分散メモリ型並列計算機 AP1000 が使用され，プログラミング言語 C を用いてコーディング課題を課し，プログラムを開発させた．演習の対象者は高い知識を有する研究者などより一般のプログラム開発者に近く，並列プログラミング演習の結果を検討することで，一般の利用者における並列処理の適用性について評価することが可能である．

本並列処理演習は，ある程度の基礎的なプログラミング能力を持った学生に対し，応用能力として並列処理における基本的なものの考え方を習得させ，応用的プログラミング能力の向上を図ることを目指した．すなわち，複数のプロセッサを効果的に動作させ，プログラムの実行効率を向上させることを最終的な目標とした．まず

表 1.1: 並列プログラミング演習の実施結果

年度	人数	演習は有効	難しい	難しいが有効	難しくない
1995 年度	57	37 (64.9%)	30 (52.6%)	19 (33.3%)	2 (3.5%)
1996 年度	61	52 (85.2%)	10 (16.4%)	8 (13.1%)	8 (13.1%)
1997 年度	49	33 (67.3%)	15 (30.1%)	13 (26.5%)	3 (6.1%)
合計	167	122 (73.1%)	55 (32.9%)	40 (24.0%)	13 (7.8%)

例題プログラムを示し，それを問題に従い各自で改良する過程において，並列プログラミング環境の学習実験を兼ねつつ，並列処理ライブラリの使用法，関数の機能の相違などを体得させることを目標とした．作成したプログラムを実行したときの挙動の差異と，使用した並列処理ライブラリとの関係を比較，検討させることで，並列処理ライブラリ中の各関数の機能，特徴などを理解させる．そして次のステップとして，問題の仕様のみを与え，プログラムを作成させた．ここでは，学生の問題理解力，プログラム設計能力，プログラム実装能力の向上を目指した．すなわち，与えられた問題をどのように並列化するかを検討し，問題に適切な並列化手法を分析・選択し，プログラムの実装において効率よく並列処理ライブラリを使用する能力，総合的なプログラミング能力の向上を目指した．

本並列処理演習のレポートにおいて演習の感想を尋ねたところ，表 1.1のような結果が得られた．この結果によると，70% 以上の学生が演習は有効であると回答しており，多くの学生が並列処理に対して好意的な印象を持ったことが分かった．しかし，それと並んで多くの学生が並列処理プログラミングを困難であると回答しており，難しくないと回答した学生は 10% に満たなかった．本並列処理演習では並列計算機として AP1000 を使用した．並列処理ライブラリは独自のものを使用しているが，ワークステーション上でのエミュレータや並列デバッガ，パフォーマンス・アナライザなどのツールが標準で装備されており，研究者等の利用者であればほとんど

困難を感じずに並列処理が可能なプログラミング環境である.

このケース・スタディより，多くの利用者は並列処理の有効性，必要性に対して理解を示しているが，並列処理に関する専門知識を有しない一般の利用者にとっては，プログラムの並列化処理は非常に困難な作業であることが確認できた．並列プログラムの開発労力を削減し並列処理を一般に普及させるためには，ソフトウェアによる並列プログラミング環境のより一層の研究が重要である．特に，利用者に対してプログラム並列化処理やタスク割当て処理など，並列処理のための特別な処理に対する労力を削減するため，プログラム自動並列化技術の発展が重要である．また，並列処理を多くの利用者に広めるため，近年コスト・パフォーマンスが高く注目されている計算機クラスタ環境など，従来の並列処理技術がそのままでは適用できない環境への対処が重要であると考えられる.

1.2 本研究の目的

本研究では，近年注目を浴びている並列処理環境である計算機クラスタ環境に焦点をあてる．計算機クラスタ環境を従来の並列処理環境である並列計算機やスーパーコンピュータと比較すると，プロセッサの処理速度とネットワークの通信速度の速度差が相対的に大きいことが相違点として挙げられる．計算機クラスタ環境の計算機構成は，上記の分散メモリ型並列計算機や，分散共有メモリ型並列計算機と類似しており，それぞれの計算機環境で開発された並列処理手法と同様な手法で並列処理が可能であると推測される．しかし，計算機クラスタ環境はワークステーションや高性能 PC をイーサネットなどのコモディティ・ネットワークで接続したものであり，並列計算機やスーパーコンピュータで使用されている高速結合網と比較するとネットワークを介した通信処理が低速である．したがって，計算機間の通信処理が低速である点を考慮した並列処理が必要となり，高速な通信網を有することを前提とした並列計算機やスーパーコンピュータにおける従来の並列処理手法をそのまま適用することは困難である．計算機クラスタ環境において効率よい並列処理を実

現するためには，計算機クラスタ環境に適した並列処理手法，並列化手法を開発する必要がある．

上で述べたように，計算機クラスタ環境では，プロセッサの計算速度と比較するとネットワークの通信速度が低速である．したがって，低速な通信速度に対処した並列処理手法を開発する必要がある．一般に，並列処理における通信頻度と並列実行性はタスクの粒度に左右される [55,56]．プログラムの文単位やイタレーション単位の粒度の細かいタスクによる並列処理では，プログラムの並列実行性を最大限抽出可能であるが，通信頻度が高くなる．逆に，プログラムの手続き単位やプログラム単位[1] の粒度の粗いタスクによる並列処理では，通信頻度は低くなるがプログラム中の並列実行性を最大限に活用することは難しい．ここで，タスクとは計算機クラスタ環境上での実行においてスケジューリング処理の単位となる実行単位である．例えば，UNIX ワークステーションにおけるプロセスや，大型計算機におけるジョブなどがタスクに対応する．プログラムを並列化することにより複数のタスクが生成され，タスク間通信により互いにデータを交換することで処理が行われる．また，タスクの粒度であるが，一般に細粒度タスクとして文単位のタスクを，粗粒度タスクとしてループ文や手続き単位のタスクを，それぞれ想定している．本論文では，文単位やイタレーション単位のタスクを細粒度タスク，手続き単位のタスクを粗粒度タスクと，それぞれ定義する．

上記の通信頻度と並列実行性のトレードオフを考慮し，計算機クラスタ環境に合致した並列処理手法を考えるとき，我々は二種類の異なるアプローチを考えることができる．すなわち，

1. 通信頻度を低くするための，粗粒度タスクによる並列処理，
2. 通信処理を目立たなくするための，通信隠蔽を用いた細粒度タスクによる並列処理，

[1] プログラムは分割せず一つのタスクとして生成し，処理するデータを変更したり，複数の種類のタスクを実行することで並列処理を達成する．

である．前者は低速であるネットワークの使用頻度を抑えるという直接的なアプローチである．ネットワークの使用頻度を抑える並列処理手法として，我々は粗粒度タスクによる並列処理を実現する．後者は発生する通信処理をプロセッサにおける計算処理とオーバラップさせることで，通信処理のために発生するタスクの実行中断状態や実行遅延状態を減少させる間接的なアプローチである．細粒度タスクによりプログラムの並列性を抽出し，計算と通信のオーバラップを積極的に活用することで通信処理を隠蔽し，計算機クラスタ環境で細粒度並列処理を実現する．

本論文では，計算機クラスタ環境において効率よい並列処理を実現するための，プログラムの並列化手法について述べる．上でも述べたように，計算機クラスタ環境における並列処理では二種類の異なるアプローチを考えることができる．粗粒度タスクによる並列処理として，我々は，通信頻度と並列性抽出のトレードオフを考慮し，プログラム中の関数手続きをタスク生成の単位とする並列化手法を開発した [57]．関数手続きをタスク生成の単位とすることで，タスク間の通信は基本的にタスク起動時のパラメータ授受と，タスク終了時の戻り値授受に限られ，タスク実行中の頻繁なタスク間通信を回避することができ，通信速度が低速である計算機クラスタ環境に適した並列処理が可能となる．タスク生成処理ではプログラム中のそれぞれの関数手続き間の並列実行性を計算し，callee/caller 関係にある関数手続きをそれぞれ独立なタスクとして生成するか，並列実行せず一つのタスクに融合するかを決定する．この関数手続き間の並列実行性に基づき，プログラムから粗粒度レベルの並列性を抽出し，効率よい並列処理を可能とするタスク生成アルゴリズムを開発した．このタスク生成アルゴリズムにより，タスク間の通信頻度を抑え，粗粒度レベルの並列性を最大限に抽出することができ，効率よい並列処理を実現する．

また，通信隠蔽を用いた細粒度タスクによる並列処理として，我々はループ文をイタレーション単位でなくループ文単位で並列処理する手法を開発した [58]．同一の配列データを共有している複数のループ文を並列実行するため，ループ文をパイプライン的に並列実行させ，計算と通信のオーバラップを実現する [39]．既存の多くのループ文並列処理では，イタレーションを並列実行の単位としてそれぞれのイタレー

ションを同期的に並列実行させる．したがって，イタレーションの実行開始時や実行終了時にデータ分散とデータ集約のための通信処理が集中し，通信処理が低速な計算機クラスタ環境では実行遅延が発生し効率よく並列処理できないという問題点がある．そこで，イタレーションの実行と計算結果の通信処理を交互に実施し，計算と通信のオーバラップにより通信の集中を回避することで，イタレーション単位での並列処理が計算機クラスタ環境で可能となる．我々は，計算と通信のオーバラップにより計算機クラスタ環境で細粒度並列処理を実現するループ文漸進処理手法を開発した．ループ文漸進処理では計算と通信のオーバラップを最大限に活用するために，ループ文間のイタレーションの依存関係を解析し，パイプライン的に並列実行可能なイタレーション数をイタレーション依存比と呼ばれるパラメータにより解析する．そして，イタレーション依存比により，それぞれのループ文のイタレーションの最適な実行タイミングや同期点を計算することで，計算機クラスタ環境において細粒度並列処理を実現する．

ループ文漸進処理の適用には，ループ文中の配列データ操作文に対する解析が重要となる．ループ文実行中に配列データ操作文が配列データのどの領域を操作するかを解析することはもちろん，ループ文の実行にともない配列データの操作領域がどのように移動するかを解析することが重要である [59,60]．ループ文の実行にともなう配列データの操作領域の移動を解析することで，ループ文の漸進処理において効率よい同期処理を実現することができる．我々は，ループ文における配列データの操作を解析するためのモデルとして，アクセス・パターンを開発した [61,62]．アクセス・パターンはループ文が配列データのどのような領域をどのような順序で操作するかを表現するためのモデルである．配列データを操作する文に対するアクセス・パターンの解析により，ループ文に漸進処理を適用するためのイタレーション依存比の計算が可能となる．アクセス・パターンは任意の次元を持つ配列データを統一的に解析することができるので，ループ文の漸進処理の適用範囲を拡大することができる．

本研究の全体像を図 1.1に示す．本研究の目標は計算機クラスタ環境において効率

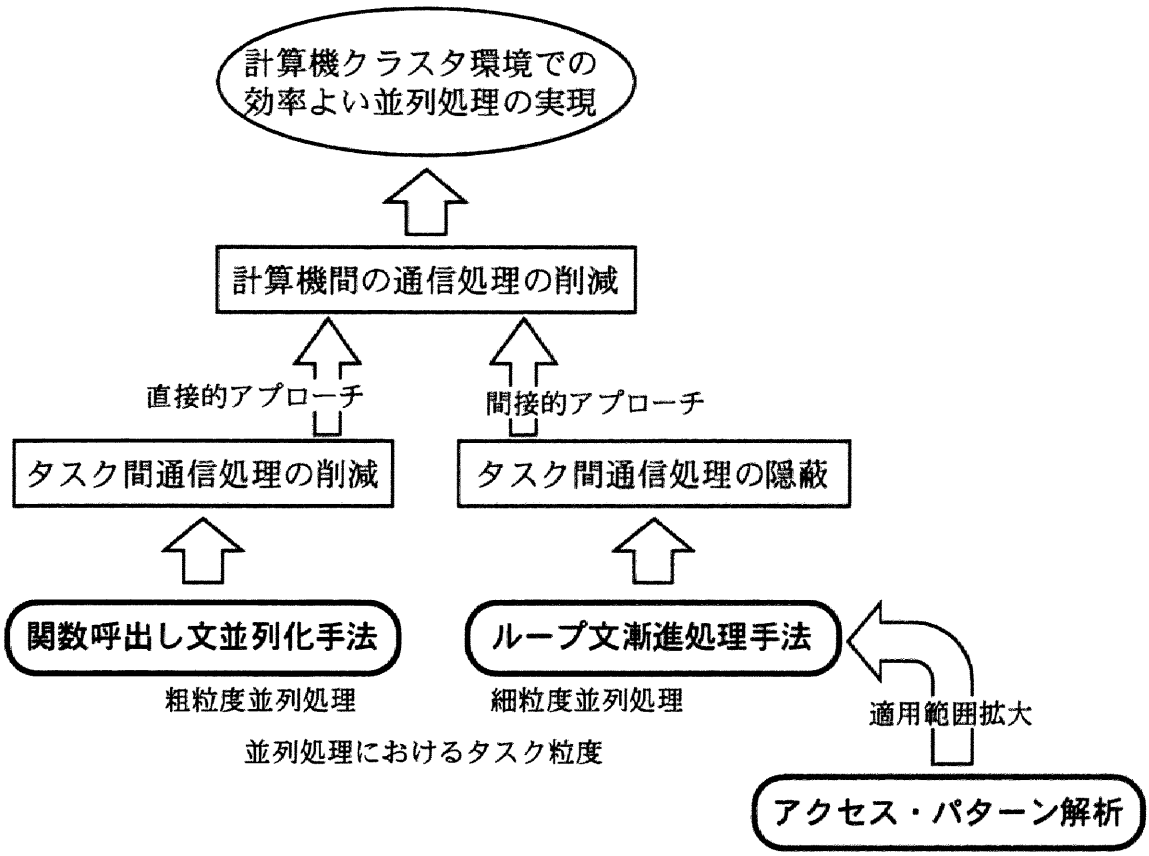

計算機クラスタ環境での
効率よい並列処理の実現
計算機間の通信処理の削減
直接的アプローチ
間接的アプローチ
タスク間通信処理の削減
タスク間通信処理の隠蔽
関数呼出し文並列化手法
ループ文漸進処理手法
粗粒度並列処理
細粒度並列処理
適用範囲拡大
並列処理におけるタスク粒度
アクセス・パターン解析

図 1.1: 本研究の全体像

よい並列処理を実現するための，ソフトウェア基盤の確立である．計算機クラスタ環境のハードウェア特性を考えると，上記の目標を達成するためにはネットワーク接続された計算機間で発生するタスク間通信処理の削減が重要となる．そのアプローチとして，直接的アプローチと間接的アプローチが存在する．直接的アプローチでは，タスク間通信の発生頻度の低下を考慮したプログラム並列化手法を実現することが課題となる．タスク間通信を削減するプログラム並列化手法として，粗粒度タスクを生成することができる関数呼出し文並列化手法を提案した．この並列化手法により，プログラム中の callee/caller 関係にある関数間の並列性を自動的に抽出し，計算機クラスタ環境で効率よく実行可能な粗粒度タスクを生成することができた．間接的アプローチでは，細粒度タスクを計算機クラスタ環境で実行可能とするため，通信処理を隠蔽して通信処理におけるオーバヘッドを削減するためのタスク生成手法，タスク実行制御手法が課題である．この通信処理の隠蔽のため，ループ文をタスクとしてイタレーションの実行とタスク間通信処理を交互に行うことで，計算と通信のオーバラップを図るループ文漸進処理手法を開発した．ループ文漸進処理手法では，ループ文間で同期タイミングを調節するために，ループ文間のイタレーション依存比により一度に実行するイタレーション数を決定する．そして，計算と通信のオーバラップにより，タスク間通信が頻繁に発生し計算機クラスタ環境では効率よい並列実行が困難であった細粒度タスクを用いた並列処理を可能とした．ループ文漸進処理を適用するためには，ループ文において配列データをどのように操作しているかの解析が必要となる．したがって，ループ文の配列操作を解析しモデル化することが課題となる．ループ文中の配列操作のモデル化としてループ文アクセス・パターンを提案した．ループ文アクセス・パターンはループ文中で配列データのどの部分がどのような順序で操作されるかを表現したモデルである．アクセス・パターン解析により，ループ文漸進処理の適用範囲を拡大することができた．

並列処理研究における本研究の位置付けを図 1.2 に示す．縦軸に計算機環境の種類を，横軸に並列処理における粒度を，それぞれ示した．また，従来の研究が対象としていた範囲を実線で，我々の研究で対象とした範囲を太線で，それぞれ示した．

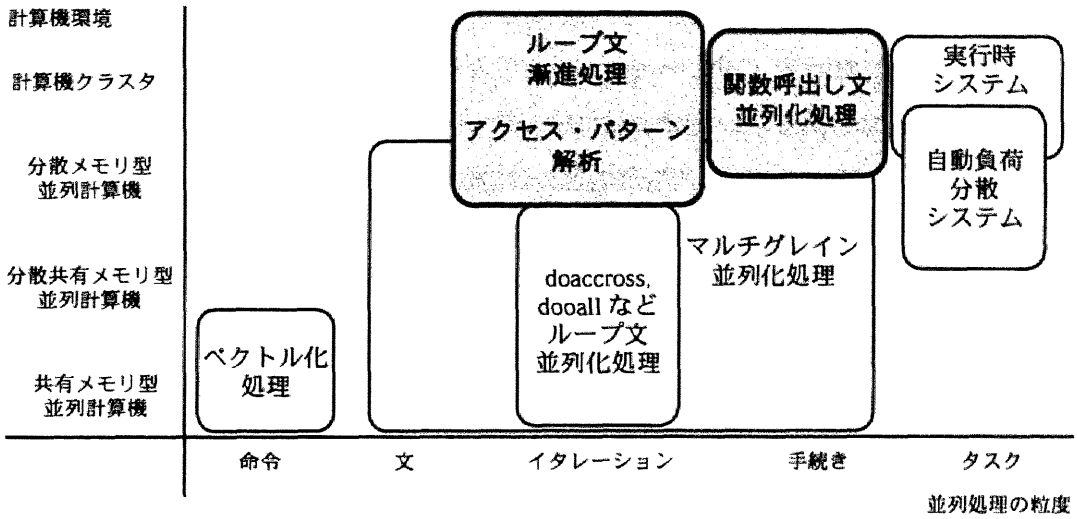


図 1.2: 本研究の位置付け

図を見ると分かるように，ほとんどの並列処理研究は一般的な並列計算機を対象としており，計算機クラスタ環境にそのまま適用可能は並列化手法は存在しない．前にも述べたように，これは計算機クラスタ環境でのネットワークの通信速度が計算速度と比較すると低速なことが原因である．計算機クラスタ環境を対象とした研究では，MPI や PVM などの実行時システム [63,64] や，自動負荷分散処理 [65,66] が行われている．我々は計算機クラスタ環境における手続きレベルの並列処理として，関数呼出し文並列化手法を開発した．マルチグレイン並列化処理でも手続きレベルの並列処理を対象としているが，計算機クラスタを対象としておらず，またプログラムからの並列性の自動的な抽出に関しては言及していない．すなわち，関数単位でタスクを生成することについて言及したのみであり，プログラム中のどの関数を独立なタスクとするかについての明確なアルゴリズムは提案されていなかった．それに対し，我々の提案する関数呼出し文並列化手法では，プログラム中の関数間の並列性を依存度で表現し，プログラムからタスクを自動的に生成可能である．また，並列処理の粒度の問題から，計算機クラスタ環境におけるイタレーションレベルの

並列処理は行われていなかった．それに対し，我々のループ文漸進処理手法は計算機クラスタ環境におけるイタレーションレベルの並列処理を実現する手法である．通信処理に必要な時間より，どれだけのイタレーションをまとめて計算と通信を交互に実施することで，効率よい並列処理が可能となるかをアルゴリズムとして示した．もちろん，我々の提案する手法は，計算機クラスタ環境だけでなく分散メモリ型並列計算機など従来の並列計算機環境においても適用可能であり，プログラム自動並列化技術に対して新たな手法を提案している．

1.3 本論文の構成

本論文は以下のような構成となっている．2章では粗粒度タスクによる並列処理の実現として，関数呼出し文並列化手法による手続き単位の並列処理を実現するためのプログラム並列化手法について述べる．プログラム中の関数呼出し文における並列実行性を計算し，計算機クラスタ環境で効率よく並列処理できるタスクを生成する手法について述べる．2章で提案するアルゴリズムでは，関数間の並列実行性を依存度というパラメータで評価している．従来の並列化手法と異なり，我々の提案するアルゴリズムは，依存度の解析によりプログラム中のどの関数を独立なタスクとして生成し，並列処理するかを明らかにしている．この並列化アルゴリズムにより，プログラム中の粗粒度並列性が適切に解析され，効率よく並列実行可能なタスクを生成することが可能となった．

3章では細粒度タスクによる並列処理を計算と通信のオーバラップにより実現する手法として，ループ文の漸進処理手法について述べる．通信処理が低速な計算機クラスタ環境において計算と通信のオーバラップを最大限活用するために，ループ文間の依存関係をイタレーション依存比により解析し，同期タイミングや一度に実行するイタレーション数などを決定するアルゴリズムを提案する．イタレーション依存比を解析することで計算と通信のオーバラップを最大限活用可能なイタレーション数を決定することができる．また同様に，ループ文漸進処理の適用によりループ

文の実行効率を向上させることができるか否かも解析することができるので，プログラム中から適切なループ文を抽出し，通信処理のコストから最適な同期タイミングを計算することで，イタレーション・レベルの細粒度並列処理を実現する．

4章では，ループ文中における配列データの操作状況を解析するためのアクセス・パターンについて述べる．3章で述べるループ文の漸進処理アルゴリズムでは，二次元配列データを操作する二重ループ文を対象としている．この制限をなくし漸進処理の対象を拡大するために，4章では多重ネスト・ループ文中の多次元配列データの操作に対処可能とする汎用的なアクセス・パターンを提案する．そして，ネスト・ループ文におけるアクセス・パターンを計算する手法として，アクセス・パターン間の畳込み演算を定義する．アクセス・パターンの畳込み演算により任意のネスト・ループ文におけるアクセス・パターンが解析可能となり，このアクセス・パターンにより任意のループ文においてイタレーション依存比を計算し，漸進処理を適用することができる．

5章では，本論文のまとめと今後の課題について述べる．

第 2 章

関数呼出し文並列化手法による手続き単位の並列処理の実現

本章では，タスク間通信を削減するという目的に対する直接的なアプローチとして，タスク間通信の頻度が低い粗粒度タスクを用いた並列処理のための，関数呼出し文並列化手法について述べる．本章で提案する並列化手法で生成される粗粒度タスクにより，通信速度が低速な計算機クラスタ環境においても効率よい並列処理が可能となる．

2.1 はじめに

近年，計算機環境として複数台のワークステーションや高性能 PC をイーサネットなどのローカル・エリア・ネットワークで接続した計算機クラスタ環境が普及してきた．拡張性の高さとその容易さ，保守性の容易さ，性能価格比の高さなどから，並列処理環境としても注目を浴びている [10]．計算機クラスタ環境において並列処理を達成するためのシステム・ソフトウェアとして PVM[50] や LAM[64] などが開発されているが，これらは並列処理に不可欠なタスク間通信や同期などの機能を実現するための手法を単に提供しているだけにすぎない．すなわち，PVM や LAM を

用いることで，計算機クラスタ環境で動作する並列プログラムを実装するときの労力は減少するが，並列プログラムを設計するときの労力はなんら減少せず，ユーザ自身がプログラムを並列化しなければならない．

並列プログラム設計の労力を削減し，計算機クラスタ環境で効率よく並列処理を達成するために，我々は プログラミング言語 C を対象とした自動並列化コンパイラを開発している [67]．自動並列化コンパイラとは，逐次実行を前提として記述された従来のプログラムから並列実行可能部分を自動的に抽出し，プログラムを再構成し，並列実行可能なタスク群を生成する言語処理系である．自動並列化コンパイラにより，ユーザは並列化処理における複雑な問題に対処する必要がなくなり，並列プログラミングにおける労力が削減され，効果的な並列処理が可能となる．現在までにも並列計算機環境を対象とした並列化コンパイラが多く開発されてきた．プログラム中の DO ループ文を並列化したり [33,34,38]，プログラムを基本ブロックに分割し並列化する [43,68,69] アルゴリズムが多く提案されている．しかし，これらの手法を計算機クラスタ環境に適用しても，効果的でないという問題がある．すなわち，これらの手法は一般的な並列計算機環境への適用を前提としているので，タスク間通信が頻繁に発生する．したがって，ネットワーク速度が低速な計算機クラスタ環境には合致しない．計算機クラスタ環境はネットワーク速度が低速であるため，プロセッサ間通信を最小限に抑えなければ並列処理の効果が得られない．

プロセッサ間通信を減少させるためには生成されるタスク間でのデータの共有を最小限にすればよい．つまり，各タスクが有する変数空間を局所化し，タスク間での変数空間の共有をなくすことにより，多くのプロセッサ間通信を削減することができ，通信処理の頻度を低く抑えることが可能となる．そこで我々は，タスクの変数空間を局所化し，プロセッサ間通信を減少させるために，プログラム中の関数を単位としたタスク生成手法を開発した．それぞれの関数は独立な変数空間を有しているので，関数を単位としてタスクを生成することで変数空間が局所化され，プロセッサ間通信の少ないタスクが生成可能となる．

本章は以下のような構成になっている．まず，2.2節では計算機クラスタ環境で並

列処理を達成するときの問題点を挙げ，その解決法について述べる．2.3節では，タスク生成アルゴリズムの動作に必要な情報を抽出するプログラム解析処理について述べる．2.4節では，計算機クラスタ環境に適したタスク生成アルゴリズムを提案する．2.5節において本章で提案するタスク生成アルゴリズムに対する評価実験について述べる．最後に，2.6節でまとめについて述べる．

2.2 計算機クラスタ環境での並列処理

本節では，計算機クラスタ環境で並列処理を行う場合の問題点について議論し，その解決法を述べる．計算機クラスタ環境において並列処理を行うためには，並列処理のために開発され，既に並列処理に関して多くの研究成果が得られている並列計算機環境との相違点を明確にする必要がある．そして，両環境の相違に起因する問題点を整理し，それを解決する手法について検討することが必要である．まず，2.2.1節では計算機クラスタ環境と並列計算機環境との相違点についてまとめる．次に，2.2.2節において，計算機クラスタ環境で効率よい並列処理を可能にするための手法について述べる．

2.2.1 並列計算機環境との相違点

我々が対象とする計算機クラスタ環境は，ネットワークで接続された複数台のワークステーションや高性能 PC により構成されている．それぞれの計算機は自律的に動作しており，主にイーサネットなどの外部ネットワークで相互接続されている．そのため，計算機内部の処理速度と比較すると相互接続されているネットワークの速度が低速である．つまり，異なる計算機上で動作しているタスク間での通信処理に非常に時間がかかることが並列処理において問題となる．また，計算機内部のバスの速度と，相互接続ネットワークの速度の間に非常に大きな差が存在するので，二つのタスクが同じ計算機上で実行されているか，異なる計算機上で実行されているかにより，タスク間通信処理に必要な時間に非常に大きな差が生じることも問題

である．つまり，計算機クラスタ環境と並列計算機環境のハードウェア面での相違は，ネットワーク速度と，それに起因した通信処理に必要となる処理時間である．

また，計算機クラスタ環境における各計算機は UNIX などの通常の OS の下で動作している．並列処理に特化した並列 OS や分散 OS の下では普通動作していない [70,71]．したがって，タスク間の通信や，新しいタスクの起動時に生じる OS の処理オーバヘッドに適切に対処しなければならない．つまり，計算機クラスタ環境と並列計算機環境のソフトウェア面での相違は，OS の各種処理オーバヘッドである．

2.2.2 計算機クラスタ環境に適した並列処理手法

上で述べたように，並列計算機環境と比較すると計算機クラスタ環境には，

- 相互接続ネットワークの速度が低速なため，計算機間での通信処理が低速である，
- OS の処理オーバヘッドが大きいため，新しいタスクの起動や，タスク間通信，タスク間同期の処理が低速である，

という問題が存在する．したがって，計算機間で発生する通信の頻度を如何に減少させるかが，計算機クラスタ環境で効率のよい並列処理を実現するための大きなポイントである．

タスク間での通信頻度を減少させるためには，タスク間で共有しなければならないデータを最小限に抑えることが必要である．すなわち，各タスクで使用するデータをそのタスク内だけに局所化し，最低限必要な通信だけが発生するように並列化処理によりタスクを構成しなければならない．そのためには各タスクの有する変数空間を局所化し，一つの変数空間を複数のタスクで共有するという状況を発生させなければよい．変数空間が共有されていると，あるタスクで変数の値が更新されたとき，その更新された変数の値を他のタスクにも通知しなければならず，これによりタスク間通信が頻繁に発生する．

各タスクの変数空間の共有を避けるために，我々はプログラム中の関数を単位としてタスクを生成する方針を採用した [67]．対象としているプログラミング言語 C での関数はそれぞれ独立な変数空間を有している．したがって，関数を分割することなく，関数をタスク生成の最小単位として採用することで，各タスクの変数空間を局所化することができ，そしてタスク間で発生する通信の頻度を最小限に抑えることが可能となり，効率よく並列に実行可能なタスクを得ることができる．このように，関数を単位としてタスクを生成することにより，各タスクが有する変数空間を局所化することができ，タスク間通信の頻度を低く抑えることができる．これで，相互接続ネットワークの通信速度が低速であるという計算機クラスタ環境での並列処理の問題を解決し，効率よい並列処理を達成することが可能となる．

次に，OS の処理オーバヘッドについて考慮する．前節でも述べたように，計算機クラスタ環境内の各計算機は UNIX などの独立な OS の下で自律的に動作している．したがって，高速な同期機構，ネットワーク・ワイドなスケジューリング機構，分散メモリ管理など，並列・分散 OS ではシステムで提供されている並列処理のための特別な機能は有していない [71,72]．そこで，計算機クラスタ環境での並列処理では新しいタスクの起動時やタスク間通信時，タスク間同期時に生じる OS の処理オーバヘッドが大きいという問題に対処しなければならない．上でも述べたように，プログラム中の各関数をタスク生成の最小単位とすることでタスク間の通信頻度を減少させることが可能となるが，そのままではタスク起動処理やタスク間同期処理における OS の処理オーバヘッドが大きく，並列実行の効果は得られない．計算機クラスタ環境で効果的な並列処理を達成するためには，これらの OS の処理オーバヘッドを最小限に抑えるための対処が必要となる．したがって，以下の点に注意してタスクを生成しなければならない．

- OS のタスク起動のオーバヘッドよりも処理時間の短いタスクを生成しない．
- 単に並列実行できればよいのではなく，タスクの並列実行により処理効率が向上するようなタスクを生成する．

- タスク間での同期が頻繁に生じないように，粒度に偏りのないタスクを生成する．

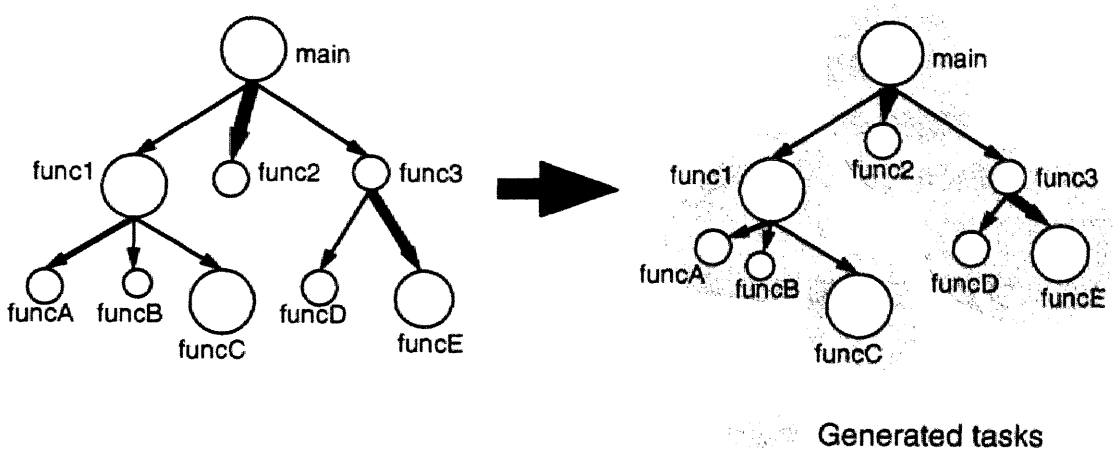
つまり，計算機クラスタ環境で並列処理を行うためには，計算機クラスタ環境に合致した粒度のタスクを生成し，各タスクの仕事量を平均化し，それらを各ワークステーションに分散することが非常に重要である．

生成されるタスクの粒度を調節するためには，プログラム中の関数をそれぞれ独立なタスクとして生成するのではなく，複数の関数をまとめて一つのタスクとして生成する，関数の融合処理が必要になる．これにより，複数の関数をまとめて，粒度がある程度平均化したタスクを生成することができる．我々は関数を融合するとき，それぞれの関数の並列実行可能性を考慮するタスク生成アルゴリズムを開発した．我々のアルゴリズムでは，関数間の並列実行可能性を依存度と呼ばれる評価尺度で表し，二つの関数の依存度が高い場合は二つの関数が融合され，依存度が低い場合はそれぞれ独立なタスクとして生成される．すなわち，依存度が高い二つの関数は並列実行可能性が低く，並列に実行されたとしても OS の処理オーバヘッドにより処理効率が向上しないと判断され，融合される．このように，関数を融合し生成されるタスクの粒度を調整する際に，関数間の並列実行可能性を考慮することで，効率よく並列実行可能なタスクが生成されることが期待できる．

関数の融合処理を模式的に図 2.1 に表した．図は関数呼出しグラフ [73] である．我々の対象とする関数呼出しグラフの定義は以下のようである．

関数呼出しグラフ

関数呼出しグラフは，プログラム中の関数を表す節の集合と，2 つの関数間の呼出し関係を表す有向辺の集合で表される，有向グラフである．ここで，関数 f_1 から関数 f_2 が複数回呼び出されるとき，関数 f_2 を表す節が複製され，それぞれの関数呼出し文に対応する有向辺を有する．また，プログラム中に再帰呼出し文，すなわち関数 f_1 内で関数 f_1 を呼び出す関数呼出し文が存在するとき，我々の関数呼出しグ



(a) 粗粒度タスク生成アルゴリズム適用前　　(b) アルゴリズム適用後

図 2.1: タスク生成例

ラフでは再帰呼出し文に対応する有向辺は削除される．また，コルーチンなど関数呼出し関係が環状になる関数呼出し文が存在するプログラムは本章では対象としない．したがって，本章で対象とする関数呼出しグラフは非循環有向グラフ (DAG) であり，main 関数に対応する節を根とする木構造グラフである． □

図では関数間の依存度の強弱を有向辺の太さで表している．有向辺が太い二つの関数は依存度が高いことを表している．また，節の大きさの大小がその関数の粒度，すなわち関数の実行ステップ数を表している．我々のタスク生成アルゴリズムでは生成されるタスクの粒度を平均化し，タスク間通信の頻度を低く抑えることを目標としている．依存度が高く，粒度が小さい関数は融合され，一つのタスクとして生成される．例えば，図 2.1(a) のような関数呼出しグラフを有するプログラムのとき，図 2.1(b) のように関数が融合され粒度が平均化したタスクが生成される．この融合処理により，OS の処理オーバヘッドによる処理効率の低下が抑えられる．

このように，我々の並列化コンパイラはプログラム中の関数を並列処理単位として採用し，関数間の並列実行可能性を依存度という評価尺度を用いて判断し，タス

クを生成する．したがって，計算機クラスタ環境における並列処理で発生する，外部ネットワークに起因する低速な通信処理の問題，および OS の処理オーバヘッドに起因する低速なタスク起動処理，タスク間同期処理の問題に対処しており，計算機クラスタ環境に適した並列処理が可能となっている．

2.3 プログラム解析処理

本節では，次節で述べるタスク生成アルゴリズムにおいて必要な，プログラムに関する各種情報を抽出するプログラム解析処理について述べる．我々のタスク生成アルゴリズムではプログラムの関数を単位としてタスクが生成されるので，プログラムの各関数，関数呼出し文についての情報の解析が中心となる．

2.3.1 関数の実行時間推定処理

関数単位でタスクを生成する場合，各関数の実行時間は非常に有用な情報である．しかし，一般にプログラムの実行時間は翻訳時には未知なので，何らかの推定手法により近似計算しなければならない．本手法では，コンパイラの字句解析器より得られる予約語，識別子などプログラムに現れる字句の個数で実行ステップ数を推定することにより，実行時間を近似する．実行ステップ数推定アルゴリズムを 図 2.2 に示す．各字句には処理の複雑さに従って重み w が与えられており，これを用いることで正確な実行ステップ数の推定を目指している．ここで，アルゴリズムの入力はプログラム中の文 S である．アルゴリズム中では，文は制御構造の有無により繰返し文，条件分岐文，そしてその他の単文に分類されている．繰返し文は for 文や，while 文など，繰返しの制御構造を有する文を表し，繰返し部分をすべて含んで文 S として扱われる．また，条件分岐文は if 文を対象としており，then 節，else 節すべてを含んで文 S として扱われる．その他の文はすべて単文として扱われ，その文に出現する字句の重みの総和として実行ステップ数が推定される．したがって，関数呼出し文の実行ステップ数には，呼び出される関数全体の実行ステップ数は含まれ

```
入力: S    … プログラム中の文
出力: step … 推定実行ステップ数

begin
  step = 0.
  p = 0.
  loop = 0.
  if ( S は繰返し文 ) then
    if ( 繰返し回数が計算可能 ) then
      loop = ( 繰返し回数 ).
    else
      loop = ( 推定繰返し回数 ).
    endif
    for (ループ内のすべての文 S )
      step = step + ESTIMATE(S) × loop.
    endfor
  else if (S は条件分岐文 ) then
    if (分岐条件が評価可能 ) then
      p = (分岐確率).
    else
      p = 0.5.
    endif
    for (then節のすべての文S )
      step = step + ESTIMATE(S) × p.
    endfor
    for (else 節のすべての文S )
      step = step + ESTIMATE(S) × (1-p).
    endfor
  else          (* S は単文 *)
    for (S 中のすべての字句 α)
      step = step + w(α).
    endfor
  endif
  return step.
end
```

図 2.2: 実行ステップ数推定アルゴリズム

ず，単に呼出し処理に必要な実行ステップ数のみが計算される．

例えば，文 “b = a + 1;” と “b = a++;” の実行ステップ数の推定を考える．前者の方が文中に現れる字句数が多いが，後者の文中の演算子 “++” の実行には変数 a への演算結果の代入処理が含まれている．すなわち，後者の文は “b = a; a = a + 1;” と同じ処理であり，後者の方が処理内容は多い．そこで，演算子 “+”, “++” の重みをそれぞれ 1, 4 とすることで後者の演算の方が前者の演算よりも計算に時間がかかることを表し，実行ステップ数を推定する．また，条件分岐文，繰返し文では，条件分岐確率，繰返し実行回数が解析可能な場合はその結果を用いて実行ステップ数を推定する．推定が不可能な場合，分岐確率は 0.5，繰返し回数は既知の繰返し文の繰返し回数の平均値として，それぞれ計算する．主な字句の重みと，具体的な実

行ステップ数の計算例を付録 A に示す.

2.3.2 関数間の依存度計算処理

依存度は二つの関数がどの程度並列実行可能かを表す尺度である. 依存度は呼出し関係にある二つの関数の間, つまり関数呼出しグラフ中の各有向辺の両端の節の間で計算される. 2.2.2 節で述べたように, 依存度により二つの関数間の並列実行可能性が評価され, 二つの関数が融合されるか, それぞれ独立のタスクとして生成されるかが決定される. この生成過程により, OS の処理オーバヘッドの影響を受けずに効率よく並列実行可能なタスクを生成することができる.

依存度は 0 から 1 の間の値で表される. 依存度が 0 である二つの関数は互いに独立に実行可能, すなわち完全に並列実行可能であることを表す. 逆に, 依存度が 1 である関数は同時には実行できない. すなわち, 逐次的に実行しなければならないことを表す. 我々は, 関数間の依存度は二つの関数が同時に実行可能なプログラム中の実行ステップ数から表現可能と考える. そこで, まず二つの関数が並列実行可能な実行ステップ数を表す並列実行可能領域を定める. 並列実行可能領域は, データ依存関係解析 [14,25] の結果を利用して計算される. ここで, 並列実行可能領域の計算に必要なデータ依存関係の定義を以下に示す.

データ依存関係

データ依存関係は, プログラム中の 2 つの文間の関係であり, 同じ変数に対する参照・更新関係で規定される. データ依存関係にある 2 つの文の実行順序を変更すると, プログラムの計算結果が変化する可能性がある. 逆に, データ依存関係にある文の実行順序を変更しなければプログラムの計算結果は変化しないので, 他の文の実行順序を変更し, 並列性を抽出することが可能となる. 変数の参照・更新関係により, データ依存関係には以下の 3 種類が存在する.

フロー依存関係 2 つの文 S_i, S_j において, S_i で更新される変数が S_j で参照される

とき，S_i と S_j はフロー依存の関係にあるという．

逆依存関係 2つの文 S_i，S_j において，S_i で参照される変数が S_j で更新されるとき，S_i と S_j は逆依存の関係にあるという．

出力依存関係 2つの文 S_i，S_j において，S_i で更新される変数が S_j でも更新されるとき，S_i と S_j は出力依存の関係にあるという．

2つの文の間に上の3種類の関係のいずれかが存在するとき，単に2つの文はデータ依存関係にあるという． □

このデータ依存関係の定義を用いて，関数呼出し文の並列実行可能領域は以下のように求められる．

並列実行可能領域

ある関数f_1 が文の列 $S_1, \cdots, S_N$ で構成されているとする．添字 1 ～ N はプログラム中の文の出現順序を表している．この関数内にある，関数f_2 を呼び出す関数呼出し文 S_i に対して，関数 f_2 と並列実行可能な関数 f_1 内の文の範囲 $PER(S_i) = [S_m, S_n]$ [1] は以下のように定義される．

1. $m < i$ かつ $i < n$.
2. 2つの文S_m と S_i がデータ依存関係にあり，$m < j$ かつ $j < i$ であるすべての文 S_j と S_i の間にはデータ依存関係が存在しない．
3. 2つの文 S_i と S_n がフロー依存関係あるいは出力依存関係にあり，$i < j$ かつ $j < n$ であるすべての S_j と S_i の間にはデータ依存関係が存在しない．

関数呼出し文 S_i において引数は「値渡し」により関数f_2 に渡されるので，関数を呼び出した後は変数の値を変更することができる．したがって，3. においては逆依存関係を無視することができる． □

[1] Parallel Executable Region

この並列実行可能領域を用いることで，関数間の依存度は以下のように定義される．

依存度

2 つの関数 f_1, f_2 が存在し，関数 f_1 内の関数呼出し文S_{f_2} によって関数 f_2 が呼び出される．また，関数f_1 は文の列$S_1, \cdots, S_N$ で構成されているとする．このとき，関数呼出し文S_{f_2} における二つの関数 f_1，f_2 間の依存度 $DV(f_1, f_2)$ は以下のように定義される．

$$DV(f_1, f_2) = 1 - \frac{ET(PER(S_{f_2}))}{ET(f_1)} = 1 - \frac{ET([S_m, S_n])}{ET([S_1, S_N])}$$

ここで，$ET(r)$ は r で表される文の範囲に含まれる文集合の推定実行ステップ数を表す．この値は，文の範囲 r 内のそれぞれの文の推定実行ステップ数を図 2.2に示したアルゴリズムにより計算し，その総和として表される． □

例えば，二つの関数 f_1，f_2 の間にデータ依存関係が存在せず，独立に並列実行可能な場合，$ET(PER(S_{f_2})) = ET(f_1)$ となり $DV(f_1, f_2) = 0$ と計算される．また，f_1, f_2 が同時に実行できない場合は $ET(PER(S_{f_2})) = 0$ であるので $DV(f_1, f_2) = 1$ と計算される．ここで，関数呼出しグラフの定義でも示したように，関数 f_1 内で関数 f_2 に対する複数の関数呼出し文が存在するとき，関数呼出し文はそれぞれ独立に扱われ，依存度もそれぞれ独立に計算される．

2.4 タスク生成アルゴリズム

本節では，前節で述べたプログラム解析処理により得られた情報により，計算機クラスタ環境で効率よく並列実行可能なタスクを生成するアルゴリズムを提案する．まず，タスク生成の方針について述べ，次に実際のアルゴリズムを述べる．

2.4.1 タスク生成の方針

2.2節でも述べたように，計算機クラスタ環境で効率よく並列実行可能なタスクを生成する場合，特に注意しなければならないのはタスクの粒度である．我々のタスク生成アルゴリズムはプログラム中の関数を単位としてタスクを生成する．しかし，プログラム中に記述されたすべての関数を独立なタスクとして生成すると，OS の処理オーバヘッドにより，効率よく並列実行できないという問題が生じる．すなわち，計算機クラスタ環境では，タスク起動処理，タスク間通信処理，タスク間同期処理などが OS の処理オーバヘッドにより遅くなるので，計算機クラスタ環境に合致した粒度のタスクを生成しなければ並列実行の効果が得られない．

我々のタスク生成アルゴリズムでは，プログラム解析処理で得られた依存度の情報により，その関数を独立なタスクとして生成するか否かを判断する．これにより並列実行の効果が得られる関数のみが独立なタスクとして生成されるので，OS の処理オーバヘッドによって実行時間が増大するという状況に対処可能となっている．

図 2.1にも示したように，我々のタスク生成アルゴリズムは関数呼出しグラフの変形処理として表現可能である．関数呼出しグラフの有向辺は，プログラム中の関数呼出し文に対応する．ある関数を並列実行することは，その関数呼出し文に対応する有向辺を切断することに相応する．すなわち，有向辺を切断することは，その関数を独立なタスクとして生成することを意味し，有向辺を切断しないことは，その関数を通常の関数呼出し文で逐次的に実行することを意味する．したがって，我々のタスク生成アルゴリズムでは，関数呼出しグラフのすべての有向辺に対して，その有向辺を切断するか否かを決定する．この有向辺切断の判断は以下の不等式により決定される．

関数の並列実行によって 短縮可能となる実行時間	>	新しくタスクを起動することによって 生じる OS の処理オーバヘッド時間

この不等式が成立する関数呼出し文は，並列実行により実行時間が短縮するので，それぞれを独立なタスクとして生成した方がよい．逆に，不等式が成立しない有向

辺は通常の関数呼出し文として実行され，余分なタスク起動，タスク間通信のオーバヘッドを発生させず，並列実行により逆に実行時間が増加する状況に対処する．

この不等式の計算には 2.3節で述べた依存度を利用する．依存度は関数呼出し文のパラメータと戻り値変数のデータ依存関係解析により計算される，並列実行可能領域により求められる．したがって，我々のタスク生成アルゴリズムは関数呼出し文のパラメータと戻り値のみを考慮している．大域変数，ポインタ変数を用いたプログラムはそれぞれの変数を局所変数に書き換え，関数の引数に追加することで，本タスク生成アルゴリズムの対象としている．

2.4.2 タスク生成アルゴリズム

図 2.3 に我々のタスク生成アルゴリズムを示す．このアルゴリズムへの入力は次に挙げる情報である．

1. 関数呼出しグラフ，
2. 各関数の推定実行ステップ数 $ET(f_i)$，
3. 関数間の依存度 $DV(f_i, f_j)$．

また，本アルゴリズムでは，対象とする計算機環境に依存するパラメータとして，OH があらかじめ与えられているものとする．OH は対象とする計算機環境において新しくタスクを起動し，必要なデータを転送するときのオーバヘッドを表している．アルゴリズム適用前に上記のオーバヘッドを測定し，実行ステップ数に換算して OH として表す．

本アルゴリズムでは $T(f_i)$ で関数呼出しグラフ中の関数 f_i 以下を実行した際の推定実行ステップ数，すなわちプログラムの実行中に関数 f_i が起動され，実行が終了するまでの推定実行ステップ数を表している．つまり，$ET(f_i)$ は関数 f_i のみ，すなわち関数 f_i から呼び出される関数の実行は含めずに推定された実行ステップ数で

アルゴリズム: PARTITION
入力: 関数呼出しグラフ
　　$ET(f_i)$　　… 関数 f_i の推定実行処理ステップ数
　　$DV(f_i, f_j)$ … 関数 f_i と f_j の間の依存度
出力: 関数呼出しグラフ中の切断される有向辺

初期化ルーチン

```
begin
  for (すべての関数 fi)
    if ( 関数 fi は関数呼出し
         グラフで終端節である) then
      T(fi) = ET(fi).
      finish(fi) = TRUE.
    else
      T(fi) = 0.
      finish(fi) = FALSE.
    endif
  endfor
  CS = 0.
  for (すべての関数 fi)
    if ( 関数 fi から起動される
         すべての関数の finish
         フラグが真である) then
      fi を CS に加える.
    endif
  endfor
end
```

メイン・ルーチン

```
begin
  while (CS ≠ 0) do
    CS から適当な関数 fi を選択する.
    T(fi) = ET(fi).
    for ( 関数 fi から呼び出され
          ているすべての関数 fj )
      短縮可能な実行ステップ数 =
        min( T(fj), T(fi)×(1- DV(fi, fj)) ).        … 式(1)
      if ( 短縮可能な実行ステップ数 > OH ) then
        「fi と fj の間の有向辺を切断」と出力.
        処理ステップ数 =
          max( 0, T(fj) + OH - T(fi)×(1- DV(fi, fj)) ).   … 式(2)
        T(fi) = T(fi) + 処理ステップ数.
      else
        T(fi) = T(fi) + T(fj).                      … 式(2)'
      endif
    endfor
    finish(fi) = TRUE.
    for (すべての関数 fi)
      if ( 関数 fi から起動されるすべての関数の
           finish フラグが真である ) then
        fi を CS に加える.
      endif
    endfor
  done
end
```

図 2.3: 粗粒度タスク生成アルゴリズム

あり，$T(f_i)$ は関数 f_i から呼び出される関数の実行も含めて推定された実行ステップ数である．この $T(f_i)$ は，関数 f_i から呼び出される関数の並列実行を考慮して，すなわち関数呼出し文が並列実行されるか否かを判別しながら，計算される．つまり，$T(f_i)$ を計算するとき，ある関数が通常の関数呼出し文として実行される場合はその関数の推定実行ステップ数が用いられ，独立なタスクとして生成される場合はタスク起動のオーバヘッドと並列実行による処理時間の短縮を加味して計算された実行ステップ数が用いられる．また，フラグ $finish(f_i)$ は関数 f_i 以下の関数に対して本アルゴリズムが適用されたことを表す．$finish(f_i)$ が真であれば，関数 f_i を呼び出す関数呼出し文，すなわち関数呼出しグラフ上で関数 f_i へ向かう有向辺に対して本アルゴリズムが適用可能である．

本アルゴリズムの処理の詳細について説明する．まず，初期化ルーチンにおいてすべての関数に対する $T(f_i)$，および $finish(f_i)$ が計算される．関数呼出しグラフ中で終端節である関数では $T(f_i) = ET(f_i)$ と計算され，$finish(f_i)$ も真となる．すなわち，本アルゴリズムを適用可能である．その他の節では，この時点ではまだ $T(f_i)$ を計算することができないので，$T(f_i) = 0$ として計算を保留し，$finish(f_i)$ を偽とする．本タスク生成アルゴリズムは，関数呼出しグラフの終端節から順にボトム・アップに適用される．次に，本アルゴリズムが適用可能な節を候補集合 CS に加える．その関数内のすべての関数呼出し文で呼び出される関数に対する $finish$ フラグが真である節，すなわち対象となる節を根とした関数呼出しグラフの部分グラフ内のすべての有向辺へのアルゴリズムの適用が終了した節がタスク生成の対象となる．

メイン・ルーチンにおいて，その関数を独立なタスクとして生成するか否か，すなわち有向辺を切断するか否かが判断される．まずその関数を独立なタスクとして並列実行することにより短縮可能な実行ステップ数が計算される．関数f_i から関数 f_j が呼び出されているとき，短縮可能な実行ステップ数は以下の計算式で評価される．

$$T(f_i) \times (1 - DV(f_i, f_j))$$

この計算式を用いて短縮可能な実行ステップ数を評価しているのがアルゴリズム中の式 (1) である．実際に短縮可能な実行ステップ数は最大でも $T(f_i)$ なので，式 (1) では min 関数が用いられている．式 (1) で計算された値と，並列実行によって生じるオーバヘッド OH を比較することでその関数を独立なタスクとするか否かが決定される．

次に，$T(f_i)$ が計算される．関数 f_j を独立なタスクとして生成する場合は，式 (2) において関数 f_j の実行ステップ数が計算され，$T(f_i)$ に加えられる．ここでは，計算結果が負にならないように max 関数が用いられている．関数 f_j が通常の関数呼出し文として実行される場合は，式 (2)' のように単に $T(f_j)$ を加えるだけでよい．

関数 f_i に対する計算がすべて終了したら $finish(f_i)$ フラグを真にし，候補集合 CS を再計算する．

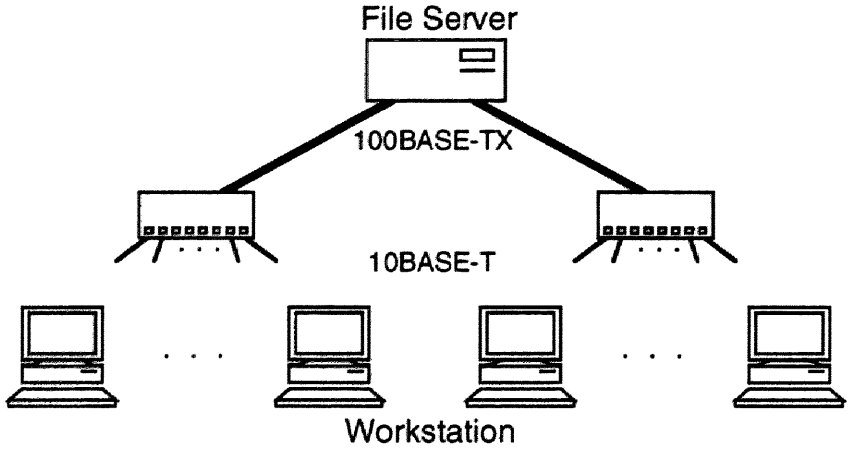
以上の処理を候補集合 CS が空になるまで続けることにより，関数呼出しグラフのすべての有向辺に対して本アルゴリズムが適用される．その結果，計算機クラスタ環境において効率よく並列実行可能なタスク群が生成される．

2.5 評価

本章では，提案したタスク生成アルゴリズムの有効性を確認するために行った評価実験について述べる．

2.5.1 実験環境

本実験は SUN SPARCstation で構成された計算機クラスタ環境上で行われた．図 2.4 に実験環境の構成を示す．ネットワークは二つのセグメントに分割されており，スイッチング・ハブとファイル・サーバを介して接続されている．ワークステーショ



Type	Name	CPU	Memory
Workstation	SPARCstation-5	MicroSPARC/1CPU/70MHz	32MB
File Server	SPARCserver-1000E	SuperSPARC/4CPU/50MHz	128MB

図 2.4: ワークステーション・クラスタ環境

ンとハブ間は 10BASE-T で，ハブとファイル・サーバ間は 100BASE-TX で，それぞれ接続されている．実験では本計算機クラスタ環境を占有的に使用した．すなわち，本環境中のワークステーションには実験で生成される以外のユーザ・タスクは実行されておらず，ネットワーク上にも実験に無関係なパケットはほとんど存在しない．また，実行時に必要な実行ファイルなどは，すべてファイル・サーバ上のディスクに存在し，NFS を介して参照される．各ワークステーションには並列処理システム・ソフトウェア PVM[50] が搭載され，すべてのタスクは PVM 上で起動，並列実行される．PVM では，新しく起動されたタスクは，ワークステーションの負荷に関係なく順に割り当てられ，実行時の動的負荷分散処理などは行われていない．

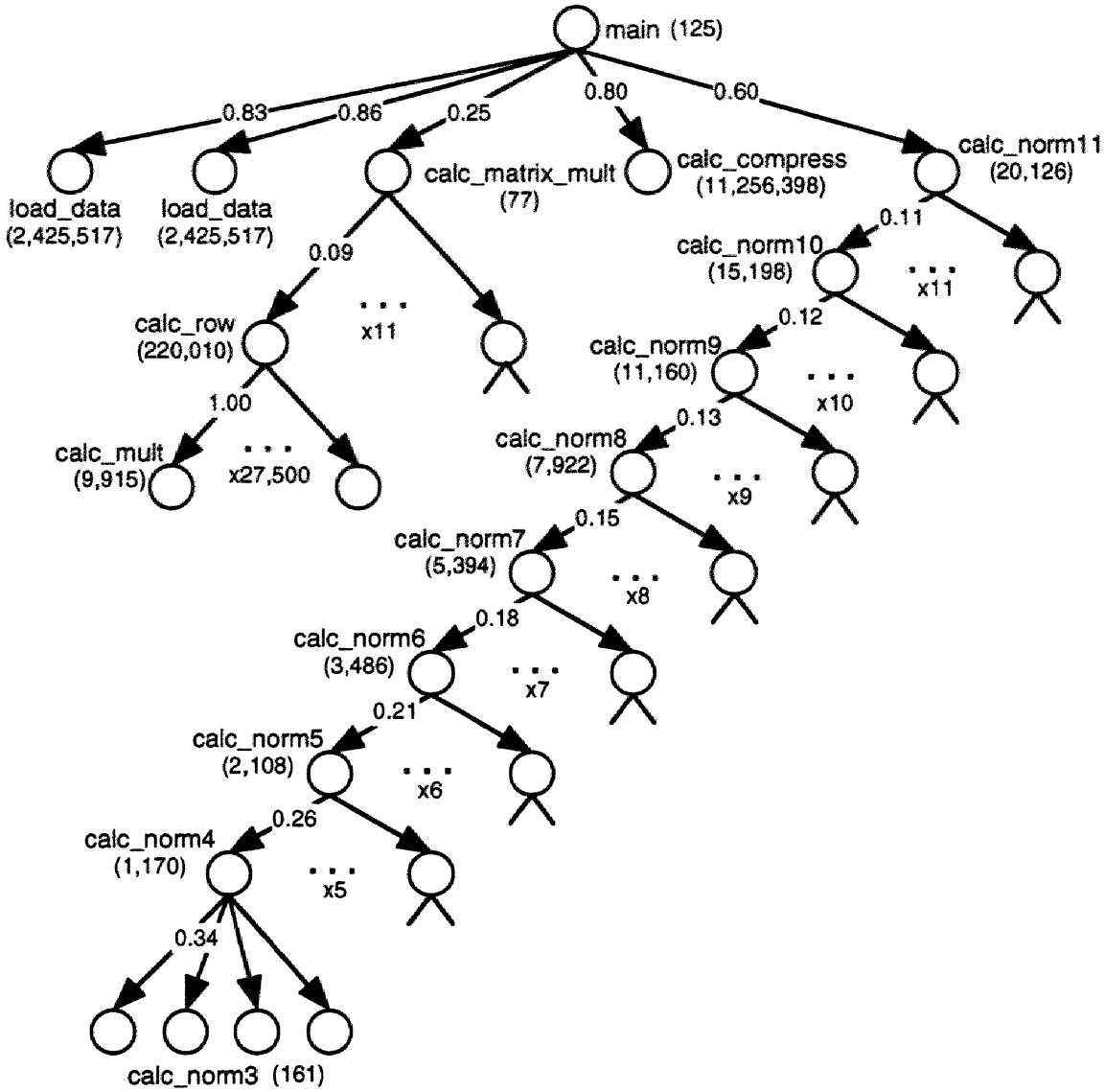


図 2.5: 例題プログラムの関数呼出しグラフ

2.5.2 アルゴリズムの動作実験

まず，アルゴリズムの動作を確かめるために実施した実験について述べる．例題プログラムとして，付録 B に挙げた行列の乗算とノルムを求めるプログラムを用いた[2]．例題プログラムの関数呼出しグラフを 図 2.5 に示す．グラフ中の各節の名前はそれぞれの関数名を表している．また，括弧内の数字はその関数の推定実行ステップ数を表している．さらに，各有向辺の数字は両端の関数間の依存度を表している．この関数呼出しグラフに対して，我々の提案した粗粒度タスク生成アルゴリズムを適用する．ここで，並列実行により生じるオーバヘッド OH は，実験により 1e+8 と定めた．この値は，新しいタスクが生成されるときに空ループが何回実行されたかを測定するプログラムを事前に実行させて求めた．

本タスク生成アルゴリズムでは，関数呼出しグラフの終端節（葉ノード）から処理が開始されるので，最初は関数 load_data, calc_mult, calc_compress, calc_norm3 の各終端節にアルゴリズムが適用される．処理の途中結果を 図 2.6(a) に示す．ハッチで囲まれた部分は処理が終了し，粗粒度タスクが生成されるか否かが確定したことを表している．四角で囲まれた数字はその節より下の部分のすべての実行ステップ数，すなわち $T(f_i)$ を表している．ここで，関数 calc_matrix_mult と関数 calc_row の間の有向辺について考える．関数calc_matrix_mult からは，関数 calc_row 以下が 11 回呼び出されている．最初は $T(\texttt{calc\_matrix\_mult}) = 77$ なので，

$$\text{短縮可能な実行ステップ数} = 77 \times (1 - 0.09) = 70$$

となり，有向辺は切断されない．したがって，図 2.6(b) が示すように，対象となる関数呼出し文においてタスクは生成されない．

次の 関数 calc_matrix_mult と 関数 calc_row の間の有向辺を考える．今回は $T(\texttt{calc\_matrix\_mult}) = 272,882,577$ なので，

[2]本例題プログラムは大域変数，ポインタ変数を用いないように作成した．

$$\begin{aligned} \text{短縮可能な実行ステップ数} &= 272,882,577 \times (1 - 0.09) \\ &= 248,323,145 \end{aligned}$$

となり，これは OH よりも大きな値である．したがって，対象となる関数呼出し文を表す有向辺が切断され，関数 calc_row 以下が独立なタスクとして生成される．関数 calc_matrix_mult からの他の有向辺も同様に切断されて，図 2.6(c) に示すように独立な粗粒度タスクが 10 個生成される．関数 calc_norm11 と 関数 calc_norm10 の間も同様に，

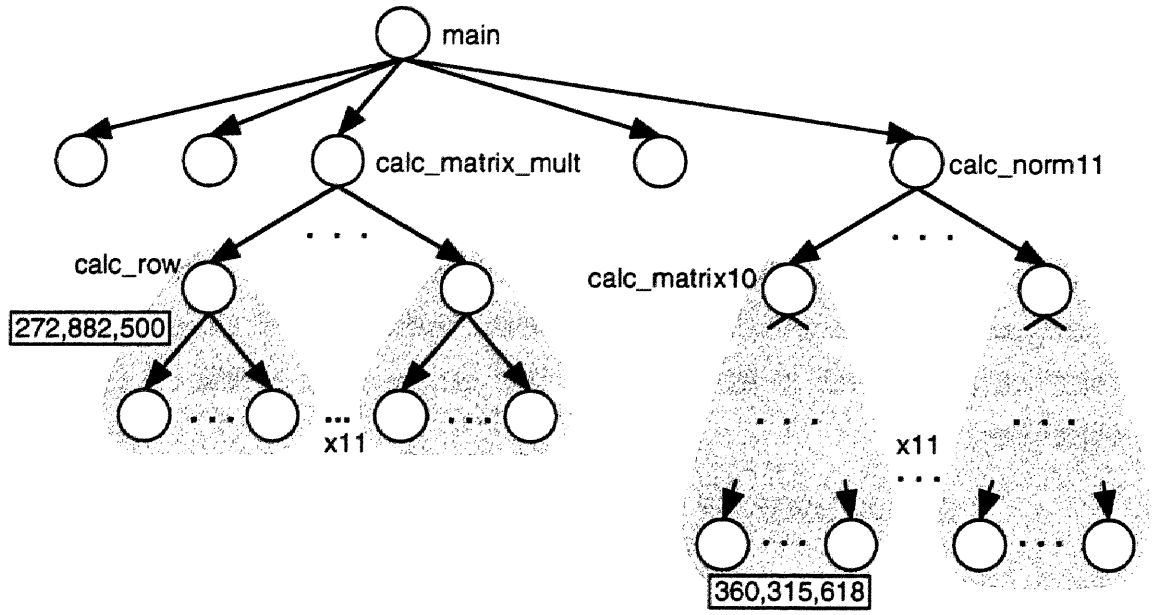
$$\begin{aligned} \text{短縮可能な実行ステップ数} &= 360,335,744 \times (1 - 0.11) \\ &= 320,698,812 \end{aligned}$$

となり，関数 calc_norm11 以下の有向辺が切断され 10 個の粗粒度タスクが生成される．

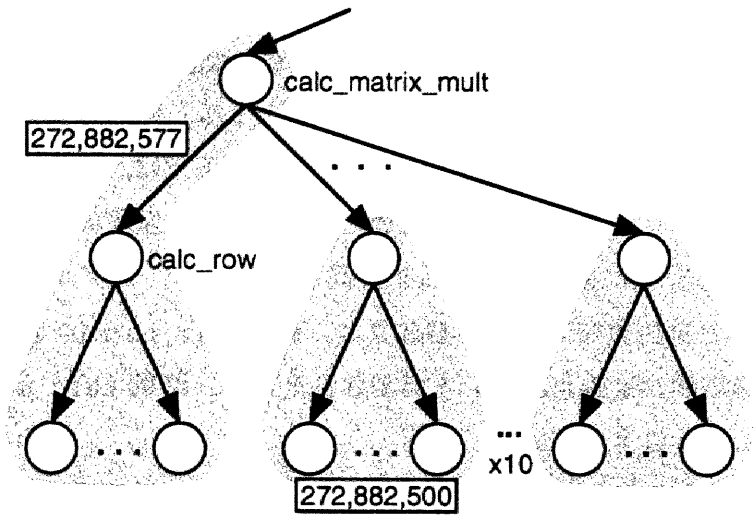
最終結果を 図 2.6(d) に示す．合計で 22 個のタスクが生成された．実際，本例題プログラムにおいて主要な計算部分は，calc_matrix_mult，および calc_norm11 以下の部分である．関数 calc_matrix_mult 以下では，行列を列方向に分割して行列の積を計算している．また，関数 calc_norm11 以下では，行列を部分行列に分割してノルムを計算している．ここで，関数 calc_norm11 以下のすべての関数は与えられた行列を部分行列に分解しノルムを計算しているが，本アルゴリズムにより各関数での計算量と並列実行可能性が評価され，関数calc_norm10 以下の関数は並列実行しても処理時間が短縮されないと判断されたため，関数 calc_norm10 以下が一つのタスクとして生成されている．

本アルゴリズムにより生成されたタスク群を実行したときの処理時間を図 2.7 に示す．比較のため，人手により最適に並列実行可能なように生成されたタスク[3] の処理時間，およびプログラム中の関数呼出し文をランダムに並列化して生成されたタス

[3]プログラムを逐次的に実行させ，プロファイラを用いてそれぞれの関数の実行時間を測定した．これに基づいてプログラムを並列化し，タスクを生成した．

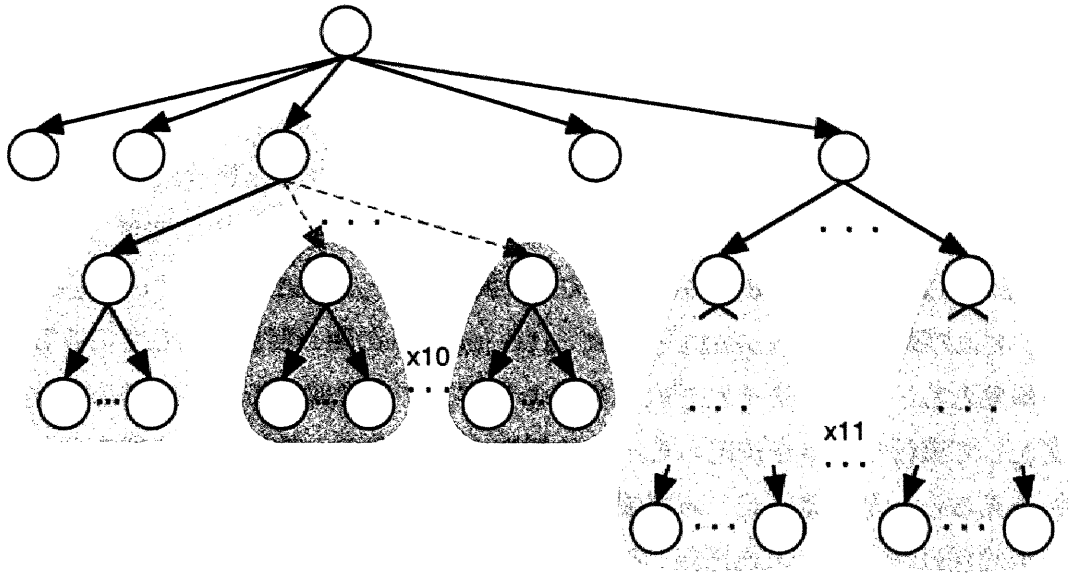


(a) 途中経過

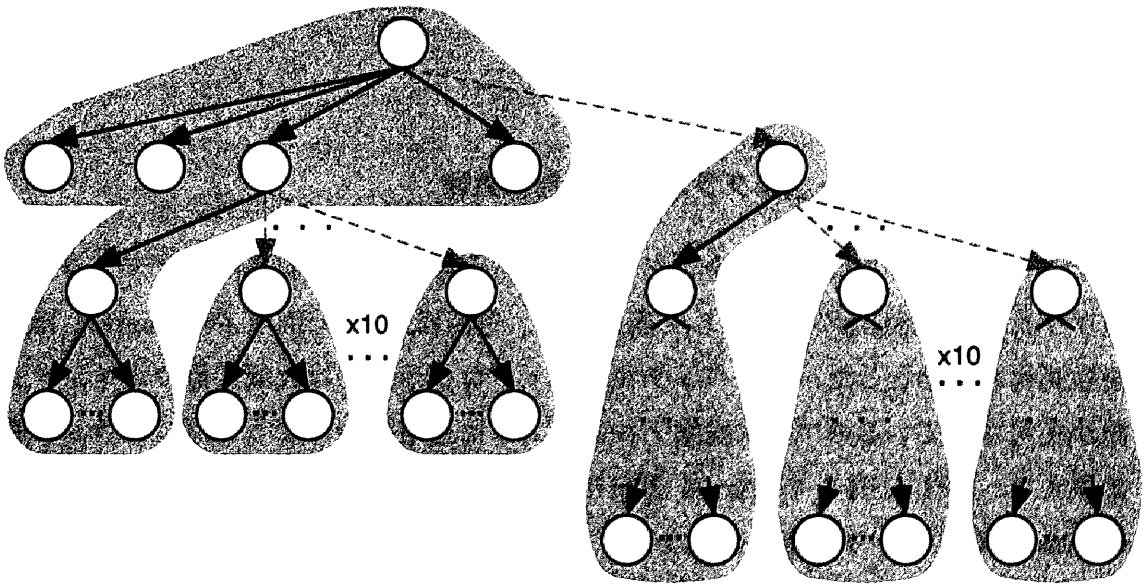


(b) 関数 calc_matrix_mult における処理

図 2.6: タスク生成アルゴリズム実行過程



(b) 関数 calc_matrix_mult へのアルゴリズム適用終了



(d) 最終結果

図 2.6: タスク生成アルゴリズム実行過程 (つづき)

クの処理時間も同様に示す．人手による分割では，関数calc_matrix_mult と関数 calc_row の間の関数呼出し文，および関数 calc_norm11 と関数 calc_norm10 の間の関数呼出し文が並列化され，合計 23 個のタスクが生成された．また，ランダムな並列化では，すべての関数呼出し文を並列化の対象とすると膨大な数のタスクが生成され，並列処理の効果が得られないのは自明である．そのため，関数 calc_matrix_mult と関数 calc_row の間，関数calc_norm11 と関数 calc_norm10 の間，および関数calc_norm11 と関数 calc_norm10 の間の関数呼出し文のみを並列化の対象とし，実行時にそれぞれ乱数によりタスクを生成するか，通常の関数呼出し文として実行するかを決定した．それぞれ 5 回実行し，平均処理時間を示した．これを見ると分かるように，本提案アルゴリズムにより生成されたタスクは人手により生成されたタスクとほぼ同じ処理効率が得られている．ランダムに並列化した場合は，処理時間の短い関数が独立なタスクとして生成され，OS の処理オーバヘッドにより処理時間が増大していることが分かる．

以上により，我々の提案したタスク生成アルゴリズムは，人手による最適な分割と同程度の性能を持つことが分かった．

2.5.3 ベンチマーク・プログラム による評価実験

次に，本提案アルゴリズムをスタンフォード大学で開発されたベンチマーク・プログラム SPLASH-2[74] に対して適用した．今回は SPLASH-2 内の kernel 部分を並列化の対象とした．このベンチマーク・プログラムでは大域変数やポインタ変数が使用されているので，本提案アルゴリズムに適用可能なようにプログラムを書き換えた．SPLASH-2 の kernel 部分は，疎行列をコレスキー分解するプログラム cholesky，高速フーリエ変換を行うプログラム fft，行列を LU 分解するプログラム lu，整数ラディックス・ソートを行うプログラム radix の 4 種類のプログラムから構成されている．このうち，cholesky と lu では，本章で提案したタスク生成アルゴリズムを適用しても並列実行可能部分が検出されず，タスクを生成することができな

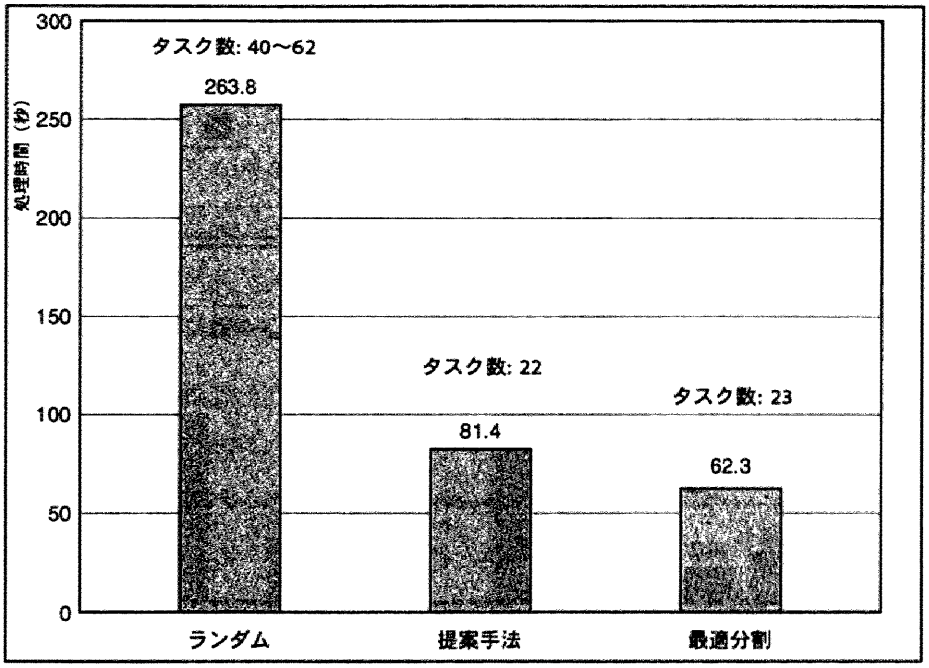


図 2.7: 実行時間の比較

かった．これらのプログラムでは，1つの関数でほとんどの処理が行われていたりするため，関数単位では並列実行可能性が検出されなかったためである．したがって，本評価実験では fft と radix を用いた．

ベンチマーク・プログラムを逐次的に実行させたときの処理時間と，本アルゴリズムを適用して得られたタスクを並列実行させたときの処理時間を 図 2.8 に示す．我々のタスク生成アルゴリズムを施すことにより，fft では 7個のタスクが，radix では 3個のタスクが，それぞれ生成された．fft1 と fft2，および radix1 と radix2 はそれぞれ同じプログラムで入力データ数を変更させたものである．これを見ると分かるように，すべての場合で本アルゴリズムを適用して得られたタスクを並列実行させたときの処理時間が少ないことが分かる．特に，fft2 を逐次実行させたときは，メモリ不足のためスラッシングが発生し非常に多くの処理時間が必要となった．

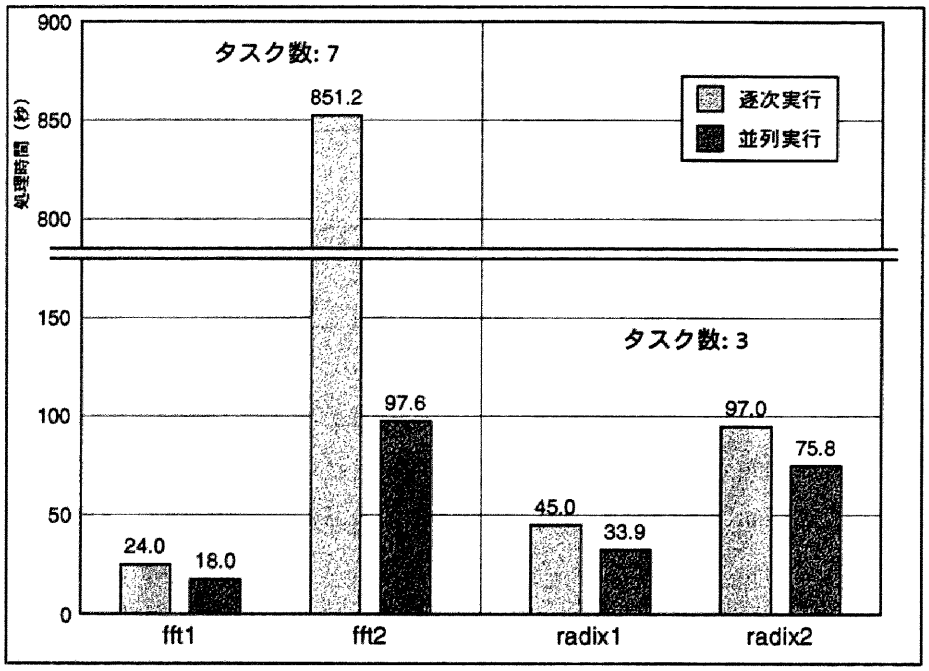


図 2.8: 実験結果

それに対し，fft2 を並列実行させたときは，メモリが各ワークステーションで分散して確保され，スラッシングが発生せず実行された．また，radix はプログラム自体にあまり並列性が見られず，データを初期化する部分と，計算を行う部分でタスクが生成されたのみであった．そのため，あまり並列実行の効果は得られていないが，逐次的に実行するよりは処理時間が少なくなっていることが分かる．

以上の実験結果により，我々の提案したタスク生成アルゴリズムでは依存度によって並列処理可能な関数が検出され，それらが独立なタスクとして生成され，計算機クラスタ環境上で効率よく動作することが分かった．これにより，我々のタスク生成手法は，計算機クラスタ環境に適していることが確認された．

2.6 おわりに

本章では，計算機クラスタ環境において並列処理を実現するための，粗粒度タスク自動生成手法について述べ，それに基づいたタスク生成アルゴリズムを提案した．計算機クラスタ環境における並列処理で重要な点は，タスク間通信処理の削減である．これに対して本章で述べた手法は，粗粒度タスクを生成することによりタスク間通信の発生を減少させるという直接的アプローチに基づいている．本手法ではプログラムに記述された関数を単位としてタスクが生成される．これにより，変数空間が各タスクで局所化され，タスク間通信の少ないタスクが生成される．また，我々の提案したタスク生成アルゴリズムでは，タスク間の並列実行可能性を表す，依存度と呼ばれるパラメータを用いてタスクを生成している．依存度を導入することによってタスクの並列実行可能性を評価し，より効率よく並列実行可能なタスクを生成することが可能になった．評価実験で得られた結果も，計算機クラスタ環境で適切に動作するタスクが生成されたことを示している．したがって，計算機クラスタ環境において粗粒度タスクによる並列処理を行う場合，本章で提案したタスク生成手法により，最適なタスクが生成可能である．

第 3 章

ループ文をパイプライン実行させるループ文漸進処理の実現

本章では，タスク間通信処理を削減するという目的に対する間接的なアプローチとして，計算と通信のオーバラップによりループ文を並列処理するループ文漸進処理手法について述べる．細粒度タスクによる並列処理ではタスク間通信処理が頻発するので，そのままでは計算機クラスタ環境に適用することができない．ループ文漸進処理では，計算と通信のオーバラップによりタスク間通信を隠匿することができ，結果的にタスク間通信のみにかかるタスクがアイドル状態となる処理時間を削減し，効率よい並列処理を実現できる．

3.1 はじめに

近年，様々なプログラム並列化手法が提案されているが，その多くは Fortran の DO ループ文や C 言語の for ループ文などの繰返し構文に対する並列化手法である．イタレーション間に依存関係が存在しないループ文を並列実行する doall 並列処理 [75]，ループ伝搬依存が存在するループ文を並列実行する doacross 並列処理 [32,75]，doalong 並列処理 [36]，波頭 (wave front) 計算により並列処理を可能とするための

ループ傾斜変換 (loop skewing)[31] など，様々な手法が提案されている．また，ループ交換 (loop interchange) やループタイル化 (loop tiling) により，イタレーションをパイプライン的に制御し，計算と通信のオーバラップにより効率よく並列実行する手法も提案されている [39].

しかし，複雑なループ運搬依存 (loop-carried dependency) が存在し，並列化できないループ文も依然として存在する．また，上記のようなループ文並列化手法では，イタレーション単位の並列実行が基本となっている．このようなイタレーション単位の並列実行は，通常の並列計算機環境では問題ないが，近年普及している計算機クラスタ環境に対しては並列実行の粒度が細かく，タスク起動オーバヘッドにより効率よく実行できないという問題がある．特に，イーサネットなどの一般的なネットワークにより接続されたワークステーションや PC で構成された NOW (Network of Workstation) システム [13] や，Class I の Beowulf システム [11] などの計算機クラスタ環境にとって，イタレーション単位の並列実行は通信オーバヘッドが大きく，効率的ではない．

複雑な依存関係により並列実行できないループ文や，イタレーション単位の並列実行が適さない計算機環境においても効率よくループ文を並列実行するために，我々はイタレーション単位ではなく，ループ文単位で並列実行を実現する漸進処理を提案する．漸進処理とは，同一配列データを操作する複数のループ文を独立のタスクとして生成し，パイプライン的に並列実行させる手法である．単一ループ文におけるイタレーション間のソフトウェア・パイプライン並列処理 [31] の考えを複数のループ文に拡張して適用した並列処理手法である．ループ文が 1 つのタスクとなり，タスクの実行中に定期的にタスク間通信を行うことで，データの供給者・消費者の依存関係にあるループ文間の並列処理を実現する．タスク中のイタレーションの実行と通信が交互に行われるので，イタレーションが部分的，漸進的に実行されることから，漸進処理と呼ばれる．ループ文はそれぞれ並列実行されるが，各ループ文内のイタレーションは逐次的に実行されるので，複雑なループ運搬依存が存在し並列処理が困難なループ文にも適用可能である．また，イタレーションでなく，ループ文

全体がタスクとして生成されるので，タスク起動オーバヘッドの削減と，計算と通信のオーバラップによる通信オーバヘッドの削減が可能となり，複数のループ文を適切に並列実行させることができる．

ループ文の漸進処理で計算と通信のオーバラップによる並列処理を実現するためには，各ループ文が配列データのどの要素を参照，更新するか，またその要素をどのような順序で参照，更新するかを解析しなければならない．そして，その参照，更新順序の類似性により，ループ文の漸進処理に基づいた並列実行が有効か否かを判定しなければならない．現在までにも，ループ文が配列データのどの要素を操作するかを解析するアルゴリズムが多く提案されている [59,76,77]．しかし，これらのアルゴリズムでは，各ループ文が操作する配列データの要素に重なりがあるか否かを解析するのみである[1]．漸進処理の適用可否を判断するためには，操作する要素に重なりがあるか否かだけでなく，ループ文の実行経過に従って，要素をどのような順序で操作するかの解析が必要になる．本章では，ループ文における配列データの要素の操作順序をアクセス・パターンとして定義する．アクセス・パターンは，ループ文のループ変数で構成される式で定義され，ループ文で操作される配列データの要素が，時間の経過に従いどのように遷移するかを表現する．前述のアルゴリズムにより，二つのループ文が同じ配列データの要素を操作するか否かを解析し，同じ要素を操作する場合は，アクセス・パターンを解析し，漸進処理を適用する．これにより，従来並列実行されなかったループ文を並列実行させることが可能となる．

3.2 アプローチ

3.2.1 配列データを共有するループ文の漸進処理

漸進処理では，ループ文のイタレーション間ではなく，プログラムのループ文間で並列性を抽出する．ループ文をタスク生成の単位とし，タスク間で複数回データを

[1]要素に重なりがなければ依存関係が存在しないので，ループ文は完全に並列実行可能であるが，そうでなければ並列実行されないとして解析している．

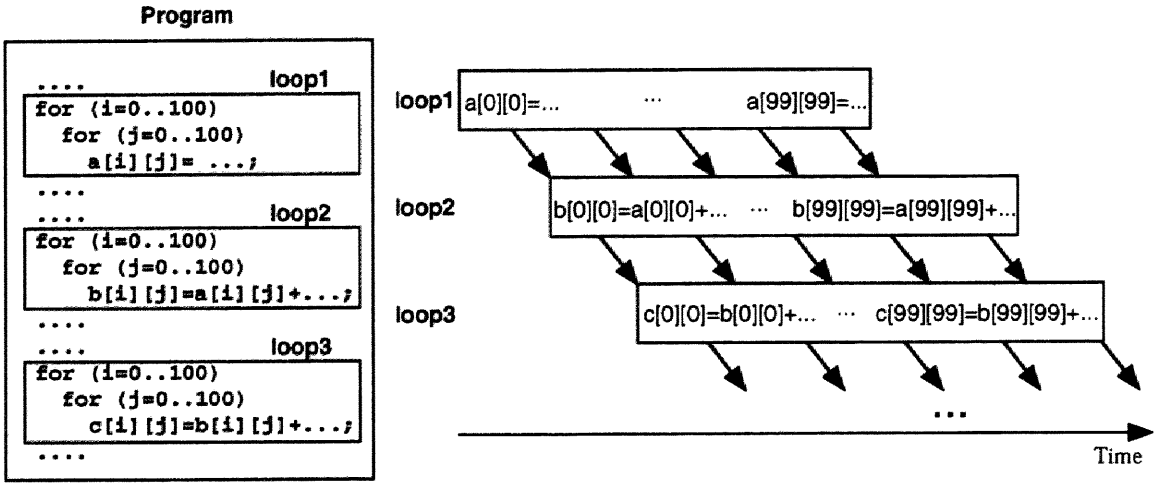


図 3.1: 漸進処理

授受しながら計算と通信をオーバラップさせる．ここで，細粒度なイタレーション単位の並列処理を計算機クラスタ環境で実現する漸進処理の概念を 図 3.1 に示す．漸進処理では，ループ文全体がタスクとして実行される．ループ文が実行されるとき，実行の経過に従ってタスク間で必要なデータが通信され，それぞれのタスクがパイプライン的に並列実行される．このとき，実行させるイタレーション回数を調整し，計算と通信を最大限オーバラップさせ，効率的な並列実行を実現する [78,79]．

漸進処理はイタレーション間に複雑な依存関係が存在するループ文に有効である．例えば，図 3.2(a) に示すような漸化関係が存在するループを考える．イタレーション間に存在するループ運搬依存によりすべてのイタレーションは通常逐次的に実行されなければならず，並列実行できない．また，図 3.2(b) のように配列データのインデックスに配列データが使用されているループ文の場合も，依存関係が複雑でイタレーション間の並列性を抽出できず，イタレーション単位の並列実行が不可能になる．

ループ運搬依存が複雑でなく，イタレーション単位で並列実行が可能なループ文であっても，対象とする計算機環境によっては効率よく実行できない場合がある．

```
for (i=2; i<10000; i++)
  a[i] = a[i-1] + a[i-2];
```

(a) 漸化関係にあるループ文

```
for (i=0; i<100; i++)
  for (j=0; j<100; j++)
    a[i][j] = a[i][j] + a[b[i]][c[j]];
```

(b) インデックスに配列が使用されているループ文

図 3.2: 複雑な依存関係を有するループ文

イタレーション単位の並列実行では，通常バリア同期などを用いて，すべてのイタレーションの実行終了までプロセッサの実行を停止させなければならない．すなわち，すべてのプロセッサの実行が一番処理時間の長いプロセッサの実行に影響され，プロセッサがアイドル状態に陥る．多くのプロセッサが使用可能で，高速な通信機構，同期機構を備えた計算機環境であれば，各プロセッサの処理のばらつきも小さくなり同期処理も高速に行われるので，アイドル状態の発生は少ない．しかし，我々の対象としている計算機クラスタ環境の場合，並列実行の粒度とバリア同期のオーバヘッドの問題のため，イタレーション単位の並列実行は効率的ではない．複数のイタレーションをまとめて一つのタスクとすることで粒度の問題を回避可能な場合もあるが，イタレーションをまとめることによりプロセッサ間の処理のばらつきが大きくなる可能性があり，一番処理の遅いプロセッサに引きづられる形で，他のプロセッサのアイドル状態がより多く発生し，効率的な並列実行を実現できない．

これに対し，我々の提案する漸進処理は，イタレーション間の並列実行ではなく，同じ配列データを操作するループ文間の並列実行を実現する．各ループ文のイタレーションは通常の逐次処理と同じ順序で実行されるので，ループ運搬依存を考慮する必要がない．また，ループ文全体が一つのタスクとして生成されるので，計算機ク

ラスタ環境における粒度の問題も回避できる．もちろん，漸進処理を適用すると，タスク間で複数回データ通信が発生するが，同期タイミングを計算し，イタレーションの実行回数とデータ通信を調整することで，計算と通信のオーバラップを最大限活用し，並列実行を実現することができる．

以上で述べたように，漸進処理は次の状況に適した並列実行手法である．

- ループ運搬依存が存在し，イタレーション間の並列実行が不可能なループ文を効率よく制御したいとき．
- 命令，演算，文単位の細粒度並列処理が適さない計算機クラスタ環境においてループ文を効率よく制御したいとき．

3.2.2 漸進処理適用のアプローチ

漸進処理では，ループ文で共有される配列データの各要素に対する操作が，その操作順序に従い部分的，漸進的に実行される．したがって，漸進処理の効果は，それぞれのループ文における配列データの操作順序の類似性に大きく依存する．例として，図 3.3(a) のように配列データの操作順序が一致している二つのループ文を考える．ここで，図 3.3(a) の正方形は二次元配列データを表し，その内部の矢印が各要素の操作順序を表している．`loop1` のループ文が更新した配列データを，`loop2` のループ文が参照しているとする．以下，`loop1` を前ループ文，`loop2` を後ループ文と呼ぶ．図 3.3(a) のように二つのループ文における配列データの操作領域と操作順序が一致しているとき，ループ文のイタレーションの実行とループ文間の通信がうまくオーバラップされ，効率よく並列実行できる (図 3.3(b) 参照)．しかし，図 3.3(c) のようにループ文間で配列操作順序が著しく異なるとき，効率のよい漸進処理を行うことができない (図 3.3(d) 参照)．したがって，効果的な漸進処理を実現するためには，ループ文における配列データの操作順序を解析し，その類似性を評価しなければならない．

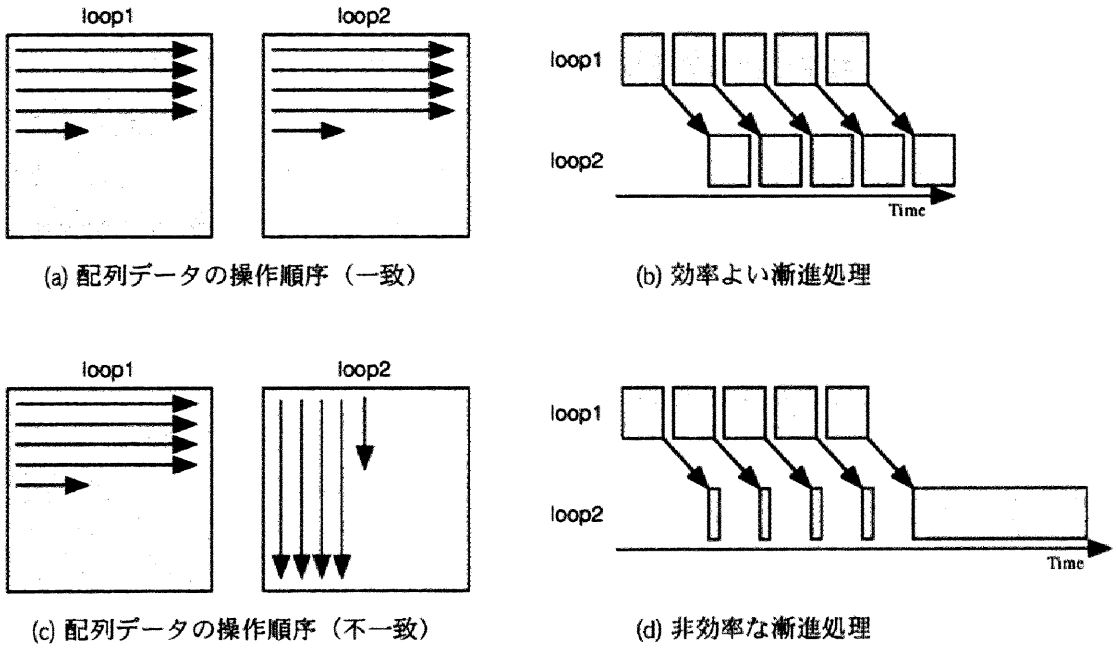


図 3.3: 配列操作順序の違いによる漸進処理の効果

我々はループ文の配列データに対する操作順序をアクセス・パターンとして定義し，ループ文間のアクセス・パターンの類似性により，漸進処理を適用したときの並列実行性を評価する．ループ文における配列データの操作順序は，一般にループ変数の値の変化に従って変化する．したがって，我々はループ文のループ変数に着目し，ループ変数の値の変化によりアクセス・パターンを定義する．ループ文のアクセス・パターンの解析方法，類似性の評価方法については 3.3節で述べる．

効率的な漸進処理を実現するためには，ループ文間の通信頻度が重要となる．すなわち，何イタレーション毎にループ文間で通信するかを決定する必要がある．通信頻度を高くするとオーバヘッドの増大により処理効率が低下する．逆に，通信頻度を低くすると，並列実行されるイタレーションが少なくなり，並列実行の効果が得られない．したがって，適切なタイミングで通信処理を発生させ，計算と通信の

オーバラップを実現することが重要となる．この漸進処理による実行効率の向上と，通信オーバヘッドの増加による実行効率の低下のトレードオフを解決するため，我々はイタレーションの実行時間より，漸進処理を適用したときのループ文全体の実行時間をモデル化し，最適な通信頻度を与えるアルゴリズムを提案する．このアルゴリズムでは，アクセス・パターンを評価し，ループ文に対して漸進処理を適用する場合の通信タイミングが計算される．この計算結果により，漸進処理の適用可能性，適用する場合の最適な通信頻度，すなわち各通信処理間のイタレーションの実行回数が評価される．このアルゴリズムについては 3.4節で述べる．

3.3 アクセス・パターン

本節ではループ文における配列データの操作順序を規定するアクセス・パターンの解析方法について述べる．二つのループ文間で操作する配列データの要素に重なりがあるか否かを解析し [59,76,77]，重なりが存在する場合にアクセス・パターンを解析し，漸進処理適用の可否を判断する[2]．

3.3.1 対象プログラム

並列処理の対象となる数値計算プログラムには，二次元配列を基本とした配列操作が多く見られる．これは，流体力学や熱力学などに対する数値解析法では，問題の対象領域をメッシュに分割し，メッシュ上の格子点間での計算により全体の解析処理を達成するアルゴリズムが多いからである．配列データはループ文中で操作され，配列添字式はループ文の実行とともに規則的に変化するプログラムが多い．本章では，図 3.4(a) に示すような二次元配列データに対する操作を記述した二重のパーフェクト・ループ文を漸進処理の対象とする．パーフェクト・ループ文とは，ループ文中で実行される文がすべて内側ループ文内に存在するものである．また，ループ

[2]操作する配列データの要素に重なりがなければ，二つのループ文間には依存関係が存在しないので，漸進処理を適用しなくても並列実行が可能である．

```
for(i= lb_i; i<= ub_i; i=i+ st_i) {
  for(j= lb_j; j<= ub_j; j=j+ st_j) {
    ... a[c_0 + c_1*i + c_2*j][d_0 + d_1*i + d_2*j] ...
  }
}
```

(a) 対象プログラム

```
for(i'=0; i'<= ub_1; i'=i'+1) {
  for(j'= 0; j'<= ub_2; j'=j'+1) {
    ... a[off_1 + mov_i1*i' + mov_j1*j'][off_2 + mov_i2*i' + mov_j2*j'] ...
  }
}
```

(b) 正規化後のプログラム

図 3.4: 対象とするプログラム例

変数の初期値，最終値，増分値を表す $lb_{i,j}$, $ub_{i,j}$, $st_{i,j}$ は共に定数であるとする．さらに，配列添字式はループ変数の線形結合で表現されており，図中の c_i, d_i $(i = 0, 1, 2)$ も定数であるとする．このような二重ループ文に対して，それぞれのループ変数の初期値が 0，増分値が 1 となるように正規化処理を施し，図 3.4(b) のようなプログラムに変換する[3]．ここで，定数 off_1, off_2 は配列添字式の初期値，すなわちループ文で最初に操作される配列データの要素を表す．また，定数 mov_{i1}, mov_{j1}, mov_{i2}, mov_{j2} はループ文の実行が進行し，ループ変数の値が変化することにより，配列添字式がどのように変化するかを表す．

以下のアクセス・パターンの議論では配列データを操作する文がループ文内に一箇所のみ存在する正規化後のループ文を対象にしている．配列データを操作する文が複数存在するループ文の扱いについては 3.6節で述べる．また，前ループ文では配列データに対する更新操作に対して，後ループ文では配列データに対する参照操作に対して，それぞれアクセス・パターンが解析される[4]．

[3]ループ文の正規化処理に関しては様々な手法が提案されている [32,75,80].

[4]図 3.2(b) のようなループ文の場合，前ループ文であればアクセス・パターンを解析できるが，後

3.3.2 アクセス・パターン表現

上記のように正規化されたループ文を構成する定数 ub_1, ub_2, off_1, off_2, mov_{i1}, mov_{j1}, mov_{i2}, mov_{j2} により，我々は以下のようにアクセス・パターンを定義する．

アクセス・パターン

配列データの操作順序を定義するアクセス・パターン $\mathbf{A}$ は，以下の三つ組で表現される．

$$\mathbf{A} = (\mathbf{S},\ \mathbf{R},\ \mathbf{V})$$

ここで，開始点 $\mathbf{S}$ はループ中で最初に操作される配列要素に対応する添字式値であり，

$$\mathbf{S} = (s_1,\ s_2) = (off_1,\ off_2)$$

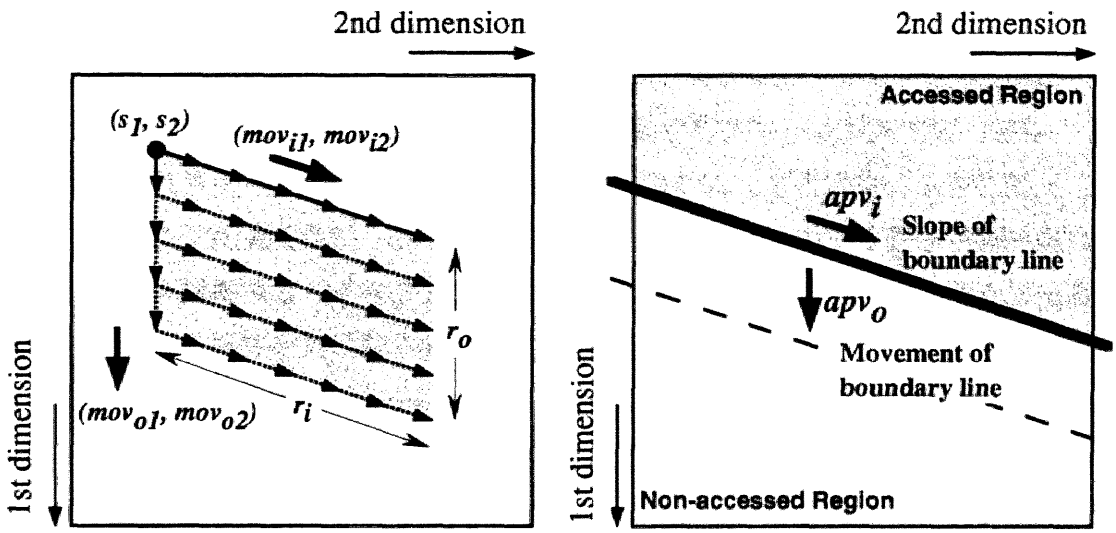
である．範囲 $\mathbf{R}$ は二重ループにおける内側ループ (inner loop)，外側ループ (outer loop) それぞれのループの繰返し回数であり，

$$\mathbf{R} = (r_i,\ r_o) = (ub_1,\ ub_2)$$

である．アクセス・パターン・ベクトル対 $\mathbf{V}$ は内側，外側ループ文のイタレーション毎に発生する配列添字式値の変化であり，

$$\begin{aligned}\mathbf{V} &= \{\overrightarrow{apv_i},\ \overrightarrow{apv_o}\} \\ &= \{(mov_{j1},\ mov_{j2}),\ (mov_{i1},\ mov_{i2})\}\end{aligned}$$

である．ベクトル $\overrightarrow{apv_i}$, $\overrightarrow{apv_o}$ をそれぞれ，内側，外側ループに対するアクセス・パターン・ベクトルと呼ぶ． □



(a) 操作される要素領域　　(b) 要素の操作順序の規則性

図 3.5: アクセス・パターンの捉え方

図 3.5(a) は二次元配列空間上でループ文により操作される配列要素の集合を模式的に表現している．図中の黒丸は，ループ文の実行で最初に操作される配列要素を表す．ループ文の実行中に操作される要素の集合は，二つのベクトル $\overrightarrow{apv_i}$, $\overrightarrow{apv_o}$ で囲まれる平行四辺形領域で表現され，図中の網掛け部分に対応する．平行四辺形の各辺の長さは，各ループの繰返し回数 r_i, r_o に基づいて決定される．

図 3.5(a) で表されるループ文では，まず最初に開始点 $\mathtt{A}[s_1][s_2]$ から内側ループに対するアクセス・パターン・ベクトル $\overrightarrow{apv_i}$ の方向へ操作箇所が遷移する．内側ループの実行が終了すると，外側ループのループ変数の値の変化により，開始点が $\overrightarrow{apv_o}$ だけ移動した要素から，$\overrightarrow{apv_i}$ の方向に操作箇所が遷移する．ここで，内側ループのイタレーションで操作される要素，つまり $\overrightarrow{apv_i}$ の傾きを持つ直線上に配置された要素を集合として考えると，この要素集合は外側ループの実行に従って $\overrightarrow{apv_o}$ の方向に

ループ文であればアクセス・パターンを解析できないので，漸進処理の対象にはならない．

遷移すると捉えることができる．したがって，配列データの傾き $\overrightarrow{apv_i}$ の直線に対して，$\overrightarrow{apv_o}$ の方向はループ文の実行に伴い，これから操作する可能性のある領域を表し，$-\overrightarrow{apv_o}$ の方向はループ文で今後操作する可能性がない領域を表していると捉えることができる．すなわち，二つのアクセス・パターン・ベクトルを用いて配列要素の操作順序を図 3.5(b) のように捉えることができる．二次元配列上で網掛けされた部分は今後のループ文の実行で操作されない領域であり，この領域を既操作領域 (accessed region) と呼ぶ．また，配列空間内の白塗り部分は今後操作される可能性のある領域であり，この領域を未操作領域 (non-accessed region) と呼ぶ．これら二つの領域は，図 3.5(b) の太線で表される傾き $\overrightarrow{apv_i}$ の境界線により分割される．ループ文の実行による操作領域の変化は，外側ループのイタレーション毎に発生する境界線の移動方向 $\overrightarrow{apv_o}$ で表現できる．この外側ループの実行に伴う境界線の移動を境界線遷移と呼ぶ．

このように，ループ文で操作される配列データの要素集合と操作順序をアクセス・パターンで表現することで，配列操作順序の規則性を解析する．

3.3.3 アクセス・パターンの類似性

本節では，ループ文間でのアクセス・パターンの違いが漸進処理に与える影響として，後ループ文の実行遅延と，ループ文間での配列操作の進行具合について考察する．以下，外側ループ，内側ループのイタレーションを，それぞれ外側イタレーション，内側イタレーションと表記する．また，議論の簡単化のため，アクセス・パターンの類似性解析では，開始点が等しいループ文を対象にする．開始点の違いについては 3.6節で議論する．さらに，コード生成時の複雑さの回避のため，漸進処理は前，後ループ文の外側イタレーション単位で行われるとする．

図 3.6 のようなアクセス・パターン・ベクトルの異なるループ文間での漸進処理を考える．このとき，後ループ文の各外側イタレーションは，各イタレーションで操作する配列データの要素すべてが前ループで操作済みでないと実行することができな

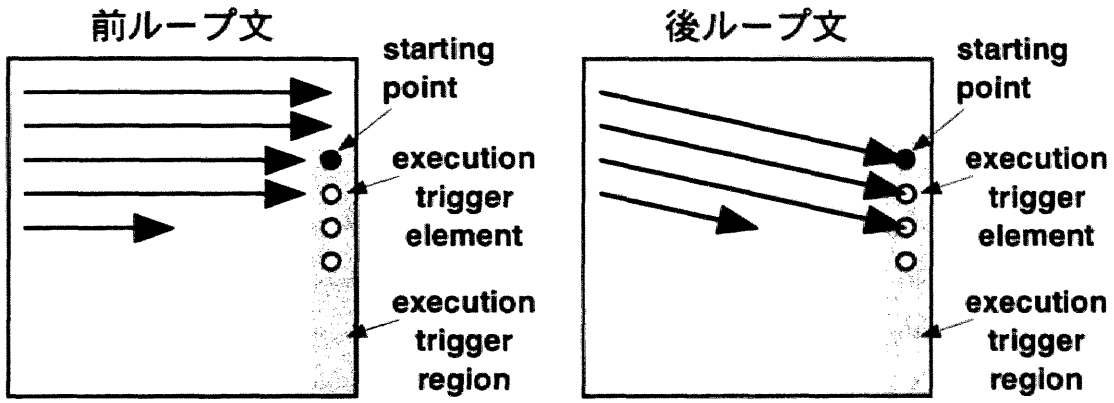


図 3.6: アクセス・パターンが異なる例

い．すなわち，後ループ文の外側イタレーションが操作する配列データの要素のうち，前ループ文で最後に操作される要素が存在し，その要素が前ループ文で操作された後，後ループ文の外側イタレーションが実行可能となる．図 3.6 における黒丸，白丸がそのような配列データの要素を表している．これらの要素は後ループ文の各外側イタレーションの実行を規定する．これらを実行トリガ要素と呼ぶ．特に，黒丸の要素は後ループ文の実行開始を規定する要素であり，これを漸進処理開始点と呼ぶ．また，これらの要素が存在する図中のハッチ部の領域を実行トリガ領域と呼ぶ．

後ループ文の実行遅延

後ループ文は，前ループ文で漸進処理開始点が操作されるまで実行することができない．したがって，漸進処理開始点を操作する前ループ文の外側イタレーションを求めることで，漸進処理における後ループ文の実行遅延の程度を表すことができる．すなわち，前ループ文の i 番目の外側イタレーションが漸進処理開始点を操作するとすると，後ループ文の実行前に，前ループ文は $i-1$ 番目までの外側イタレーションを実行しなければならない．この前ループ文の外側イタレーションの実行回

数 $i-1$ を i_{wait} として表すと，この i_{wait} は以下のように計算することができる．

前，後ループ文のアクセス・パターン $\mathbf{A}_p$, $\mathbf{A}_s$ が，それぞれ以下のようであるとする．

$$\mathbf{A}_p=(\mathbf{S_p},\ \mathbf{R_p},\ \mathbf{V_p}) \qquad \mathbf{A}_s=(\mathbf{S_s},\ \mathbf{R_s},\ \mathbf{V_s})$$
$$\mathbf{S_p}=(s_{p1}, s_{p2}), \qquad \mathbf{S_s}=(s_{s1}, s_{s2}),$$
$$\mathbf{R_p}=(r_{pi}, r_{po}), \qquad \mathbf{R_s}=(r_{si}, r_{so}),$$
$$\mathbf{V_p}=(\overrightarrow{apv_{pi}}, \overrightarrow{apv_{po}}). \qquad \mathbf{V_s}=(\overrightarrow{apv_{si}}, \overrightarrow{apv_{so}}).$$

このとき，後ループ文の各外側イタレーションで操作される要素は，傾き $\overrightarrow{apv_{si}}$ の直線上に配置され，漸進処理開始点は境界線の端に配置される．つまり，最初の外側イタレーションで操作される要素の中で最初か最後に操作される要素，すなわち

$$(s_{s1}, s_{s2}) \qquad \text{または,}$$
$$(e_{s1}, e_{s2}) \ = \ (s_{s1}, s_{s2}) + (r_{si}-1)\cdot\overrightarrow{apv_{si}}$$

のいずれかが漸進処理開始点となる[5]．(s_{s1}, s_{s2})，(e_{s1}, e_{s2}) が参照可能となるまでの前ループ文の外側イタレーションの実行回数を求めるために，前ループ文のアクセス・パターン・ベクトルを用いてそれぞれの点を表現すると，以下のようになる．

$$(s_{s1}, s_{s2}) \ = \ (s_{p1}, s_{p2}) + j'\cdot\overrightarrow{apv_{pi}} + i'\cdot\overrightarrow{apv_{po}}$$
$$(e_{s1}, e_{s2}) \ = \ (s_{p1}, s_{p2}) + j''\cdot\overrightarrow{apv_{pi}} + i''\cdot\overrightarrow{apv_{po}}$$

[5]ここでは，後ループ文の外側イタレーションで参照する要素は，前ループ文のそれと等しいか少なく，漸進処理開始点が後ループ文の境界線に存在するとしたときの計算法を示している．そうでない場合は，前ループ文のアクセス・パターンを用いて，
$(e_{s1}, e_{s2}) = (s_{s1}, s_{s2}) + \frac{s_{p2}+(r_{pi}-1)\cdot mov_{pj2}-s_{s2}}{mov_{sj2}}\cdot\overrightarrow{apv_{si}}$
と表すことができる．ただし，$\overrightarrow{apv_{pi}} = (mov_{pj1}, mov_{pj2})$, $\overrightarrow{apv_{si}} = (mov_{sj1}, mov_{sj2})$.

ここで，i'，i'' はそれぞれの要素を操作するのに必要な外側ループのイタレーション回数を表している．$\overrightarrow{apv_{pi}}$ と $\overrightarrow{apv_{po}}$ が一次独立であれば，i' や i'' が一意に定まり，i_{wait} は，

$$i_{wait} = \max(i', i'') - 1$$

で表される．したがって，前ループ文の外側イタレーションが i_{wait} 回実行されると後ループ文が実行可能となる．

ループ文間での配列操作の進行具合

前ループ文の外側イタレーションが i_{wait} 回実行された後，前，後ループ文それぞれの外側イタレーションをある一定の割合で漸進的に実行させることができる．しかし，この並列実行性はアクセス・パターンの類似性に強く依存する．同一のアクセス・パターンを持つループ文の場合，前，後ループ文の外側イタレーションで操作される配列要素が完全に一致するので，前ループ文の外側イタレーションが一回実行されると，後ループ文の外側イタレーションを一回実行することができ，理想的な漸進処理を実現できる．しかし，アクセス・パターンに差異がある場合は，前，後ループ文の外側イタレーションで操作される要素が異なるので，前ループ文の一定回数の外側イタレーションの実行が終了するまで，後ループ文の外側イタレーションが実行できない状況が生じる．このような配列操作の依存関係を表現するために，前ループ文，後ループ文それぞれの外側イタレーションの実行可能な回数の比をイタレーション依存比として定義する．イタレーション依存比は，漸進処理により得られる並列性を特徴付ける重要な指標である．

後ループ文の各外側イタレーションは，それぞれの実行トリガ要素を操作する前ループ文の外側イタレーションが実行された後に，実行可能となる．したがって，各実行トリガ要素を操作する前ループ文の外側イタレーションの実行間隔を求めれば，それがイタレーション依存比となる．

イタレーション依存比は，実行トリガ領域上における境界線遷移，つまり外側ループのアクセス・パターン・ベクトルにより特徴付けられる．前，後ループ文の外側ループのアクセス・パターン・ベクトルの実行トリガ領域方向成分の大きさを，それぞれ a, b とすると，前ループ文の外側イタレーションを b 回実行することで，後ループ文の外側イタレーションが a 回実行可能となる．したがって，イタレーション依存比は

$$\text{前ループ文: 後ループ文} = b : a$$

と表される．このイタレーション依存比は，二つの実行トリガ要素を操作する間の外側イタレーションの実行回数を計算することで得られる．すなわち，実行トリガ要素を (t_1, t_2), (t'_1, t'_2) として，

$$\begin{aligned}
(t_1, t_2) &= (s_{p1}, s_{p2}) + j_p \cdot \overrightarrow{apv_{pi}} + i_p \cdot \overrightarrow{apv_{po}} \\
(t'_1, t'_2) &= (s_{p1}, s_{p2}) + j'_p \cdot \overrightarrow{apv_{pi}} + i'_p \cdot \overrightarrow{apv_{po}} \\
(t_1, t_2) &= (s_{s1}, s_{s2}) + j_s \cdot \overrightarrow{apv_{si}} + i_s \cdot \overrightarrow{apv_{so}} \\
(t'_1, t'_2) &= (s_{s1}, s_{s2}) + j'_s \cdot \overrightarrow{apv_{si}} + i'_s \cdot \overrightarrow{apv_{so}}
\end{aligned}$$

で得られる i_p, i'_p, i_s, i'_s を用いて，

$$\begin{aligned}
a &= i'_p - i_p \\
b &= i'_s - i_s
\end{aligned}$$

と計算できる．

3.4 アクセス・パターン類似性と同期タイミング

配列データを共有するループ文は，上で述べたイタレーション依存比で表される依存関係に基づき漸進処理可能である．例えば，イタレーション依存比が

$$前ループ文 : 後ループ文 = dep_p : dep_s$$

のとき，前ループ文の外側イタレーションが dep_p 回実行されると，後ループ文の外側イタレーションが dep_s 回実行可能であることを示している．ここで，前ループ文の dep_p 回，後ループ文の dep_s 回の外側イタレーションを漸進処理適用時に同期処理を挿入する最小単位と見なすことができ，これを漸進処理単位と呼ぶ．つまり，前ループ文の漸進処理単位が一つ実行されれば，後ループ文の漸進処理単位も一つ実行可能となる．

イタレーション依存比とともに，ループ文間での並列実行性を特徴付ける要素として漸進処理単位の実行時間が挙げられる．前，後ループ文の外側イタレーションの実行時間をそれぞれ $iter_p$, $iter_s$ とすると[6]，漸進処理単位の実行時間比は

$$t_p : t_s = iter_p \times dep_p : iter_s \times dep_s$$

で求められる．ここで，t_p, t_s はそれぞれ前，後ループ文の漸進処理単位の実行時間に対応する．

漸進処理で同期処理のタイミングを決定するとき，各漸進処理単位間で実行遅延状態が発生しないこと，及び余分な同期処理を削減することが重要となる．このための同期処理を挿入するタイミングの計算方法を以下に述べる．

3.4.1 同期タイミング計算方法

後ループ文の最初の漸進処理単位を実行する時点において，既に前ループ文で x 個の漸進処理単位が実行されているとする．また，前，後ループ文の漸進処理単位の実行時間をそれぞれ t_p, t_s とする．前ループ文で x 個の漸進処理単位が実行される

[6]コンパイラにおけるプログラムの実行時間推定は非常に重要であるが，困難な問題である．ここでは，実行ステップ数などから推定できるとしている．

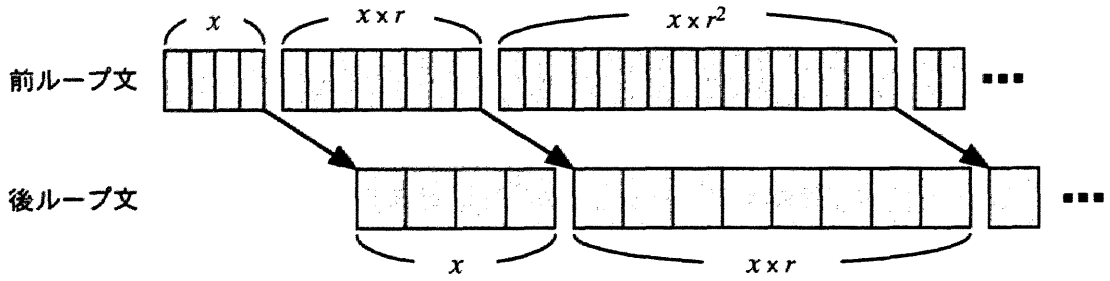


図 3.7: 同期処理を挿入するタイミング

と，後ループ文でも同様に x 個の漸進処理単位が実行可能となる．漸進処理単位の実行時間比を考慮すると，後ループ文で x 個の漸進処理単位を実行するのに $x \times t_s$ 時間必要となり，その間に前ループ文では $\lfloor x \times \frac{t_s}{t_p} \rfloor$ 個の漸進処理単位が実行可能である．ここで，

$$r = \frac{t_s}{t_p}$$

とすれば，その後同様に，後ループ文が $\lfloor x \times r \rfloor$ 個の漸進処理単位を実行可能であるのに対して，前ループ文では $\lfloor x \times r^2 \rfloor$ 個の漸進処理単位が実行可能となる．したがって，図 3.7 に示すように，実行遅延状態が発生しないように漸進処理を適用するためには，前ループ文の $(x \times r^i)$ 個の漸進処理単位毎に同期処理を挿入すればよい．ここで，図中の長方形は漸進処理単位に対応している．このように計算される同期タイミング及び同期回数に基づき，ループ文間の配列操作時間をモデル化し，ループ文全体の実行時間を最小にする x を求めればよい．

3.4.2 漸進処理適用判定アルゴリズム

図 3.8 に示すように，前，後ループ文の漸進処理単位の実行時間をそれぞれ t_p, t_s とし，前ループ文に含まれる漸進処理単位数を p_{all} とする．また，一回の同期処理で発生する通信オーバヘッドを T_o とする．これらのパラメータを用いて，漸進処理適

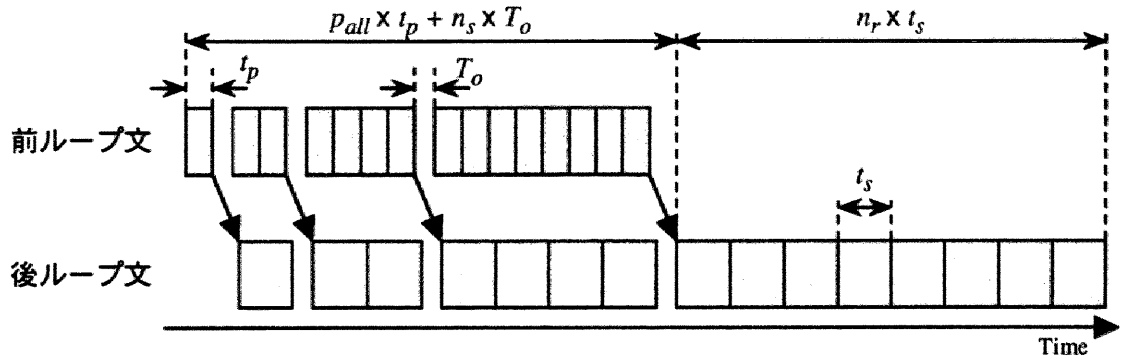


図 3.8: 配列操作時間のモデル化

用後のループ文全体の実行時間 t_{inc} は以下のように表現できる.

$$\begin{aligned} t_{inc} &= \text{前ループ文の操作時間} + \text{同期処理時間} + \\ &\quad \text{前ループ文実行終了後の後ループ文の操作時間} \\ &= p_{all} \cdot t_p + n_s \cdot T_o + n_r \cdot t_s \end{aligned}$$

ここで，式中の p_{all}, n_s, n_r は以下のように計算できる.

まず最初に，前ループ文の全漸進処理単位数 p_{all} は,

$$p_{all} = ub_1 / dep_p$$

である．前ループ文の漸進処理開始点までに実行される漸進実行単位数を p_{pre} とすると，実際に漸進的に実行される前ループ文の漸進処理単位数は

$$p'_{all} = p_{all} - p_{pre}$$

となる．図 3.7 に示したように漸進処理適用時には $x \times r^i$ 個の漸進処理単位毎に同期処理が発生するので，$r \neq 1$ のとき

$$p'_{all} = \sum_{i=0}^{n_s} x \times r^i = \frac{x(r^{n_s} - 1)}{r - 1}$$

が成り立ち，漸進処理適用時に挿入される同期処理回数 n_s は，

$$n_s(x) = \log_r\{\frac{p'_{all}(r-1)}{x} + 1\} \quad (r \neq 1)$$

と求められる．$r = 1$ のときは，漸進処理単位 x 個毎に同期処理が発生する．したがって，まとめると，

$$n_s(x) = \begin{cases} \log_r\{\frac{p'_{all}(r-1)}{x} + 1\} & (r \neq 1) \\ \frac{p'_{all}}{x} & (r = 1) \end{cases}$$

となり，n_s は x の関数である．最後に n_r であるが，前ループ文で最後に同期処理が発生するまでに，後ループ文で実行された漸進処理単位数は

$$\sum_{i=0}^{n_s-1} x \times r^i = \frac{x(r^{n_s-1} - 1)}{r - 1}$$

と計算することができる．前ループ文の配列操作が終了した時点において，後ループ文で操作されていない漸進処理単位数 n_r は，

$$n_r(x) = \begin{cases} ub_1/dep_s - \frac{x(r^{n_s-1}-1)}{r-1} & (r \neq 1) \\ ub_1/dep_s - x \cdot (n_s - 1) & (r = 1) \end{cases}$$

となり，n_r も x の関数として表される．以上より，ループ文全体を漸進処理したときの実行時間 t_{inc} は，

$$t_{inc}(x) = p_{all} \times t_p + n_s(x) \times T_o + n_r(x) \times t_s$$

のように x の関数としてモデル化することができる．この t_{inc} を最小にする x により，タスク間で同期処理を行えばよい．この式を解析的に解くのは困難であるが，x は $1 \leq x \leq p'_{all}$ の範囲の整数値であるので，$t_{inc}(x)$ を最小にする x を計算すること

は可能である．ここで，$x = p'_{all}$ はループ文の実行に同期処理を一度も挿入しないとき，実行時間が最短となることを表す．すなわち，対象とするループ文は漸進処理を適用しても効果が得られないことを表している．したがって，この式により漸進処理適用の可否も判断可能となる．

漸進処理適用判定アルゴリズムを 図 3.9 に示す．アルゴリズムの入力は，前，後ループ文の外側イタレーション実行時間 $iter_p$, $iter_s$，イタレーション依存比 $dep_p : dep_s$，外側ループ文のループ変数の値の最大値 ub_1，及び漸進処理開始点までの外側イタレーションの実行回数 i_{wait} である．これにより，ループ文に漸進処理を適用するか否か，適用する場合の同期タイミングを出力する．

3.5 評価実験

本節では，アクセス・パターン表現の有効性，同期タイミング計算法の妥当性を確認するために行った評価実験について述べる．アクセス・パターンの類似性が並列実行性を表現しているか否かを，様々な配列添字式のアクセス・パターンの比較により調査した．

3.5.1 対象とする配列添字式

実験対象とする配列添字式として，図 3.10 を使用した．これらはすべて正規化された後の配列添字式である．図 3.10(a), (b) は配列データを順序よく操作する単純な配列添字式であり，それぞれ操作方向が異なっている．図 3.10(c) ～ (f) は i と j の一次式になっている配列添字式を表している[7]．図中の網掛け領域がループ文で操作する要素の集合を表している．また，領域内の矢印は要素の操作順序を表している．そして，領域外の矢印が外側ループのアクセス・パターン・ベクトルを表している．

[7]各イタレーションの繰返し回数はすべて一定とし，操作される配列 a は十分な大きさを持っているとする．

ALGORITHM: INCREMENTAL_PROCESSING
Input:
- dep_p:dep_s イタレーション依存比
- $iter_p$, $iter_s$ 前，後ループ文のイタレーション実行時間
- ub_1 前ループ文の外側ループのループ変数の最終値
- i_{wait} 漸進処理開始点までのイタレーション実行回数

Output:
漸進処理を適用しない場合は，「漸進処理適用せず」と出力
漸進処理を適用する場合は，同期タイミングを規定する x を出力

```
begin
  t_p := iter_p * dep_p;
  t_s := iter_s * dep_s;
  p_all := ub_1 / dep_p;
  p'_all := p_all - i_wait / dep_p;
  t_min := MAXVALUE;
  foreach x = 1 .. p'_all
    t := tinc(x, t_p, t_s, p_all, p'_all);
    if t < t_min then
      t_min := t;
      x_min := x;
    endif
  endfor
  if x_min = p'_all then
    output("漸進処理を適用せず");
  else
    output(x_min);
  endif
end
```

```
FUNCTION tinc(x, t_p, t_s, p_all, p'_all)
begin
  r := t_s / t_p;
  if r ≠ 1 then
    ns := log(r, p'_all * (r - 1) / x + 1);
    nr := ub_1 / dep_s - x * (r**(ns - 1) - 1)/(r - 1);
  else
    ns := p'_all / x;
    nr := ub_1/dep_s - x * (ns - 1);
  endif
  tinc := p_all * t_p + ns * T_0 + nr * t_s;
end
```

図 3.9: 漸進処理適用判定アルゴリズム

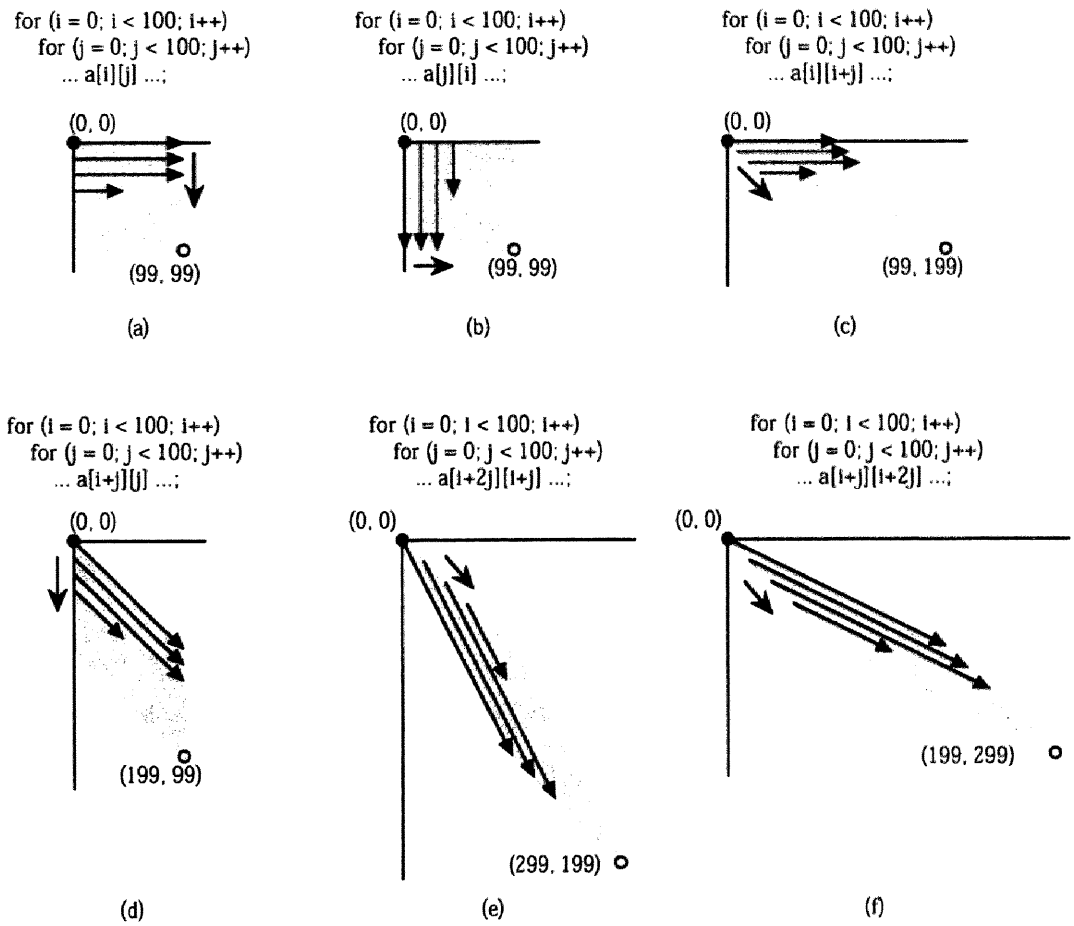


図 3.10: 例題配列添字式

これらのループ文をそれぞれ前ループ文，後ループ文として，漸進処理適用判定アルゴリズムを適用した結果を 表 3.1 に示す．イタレーション依存比，漸進処理開始点までのイタレーション回数 i_{wait}，漸進処理適用判定アルゴリズムの出力をそれぞれ示している．ここでは，イタレーションの実行時間を 10 ステップ，同期オーバヘッドを 100 ステップとして計算した．表を見ると，漸進処理の適用可否の判断が適切であることがわかる．例えば，漸進処理開始点までの外側イタレーションの実行回数 i_{wait} の値が大きいループ文間には漸進処理の適用は指摘されていない．これは，i_{wait} の大きいループ文間では依存関係により並列実行できないことを表している．

また，漸進処理を適用すると判定されたときの同期タイミングの計算結果であるが，イタレーション依存比 1:1，$i_{wait} = 0$ のときの分割数について考察する．最適分割数が 37 という計算結果に基づき，分割数を 36, 37, 38 としたときの実行経過を図 3.11 に示す．図を見るとわかるように，分割数 36 の場合では同期回数の増加が，分割数 38 の場合では分割数の増加が発生し，これにより漸進処理の効率低下が表れている．したがって，分割数を 37 とするときが効率的な漸進処理を実現可能なことがわかる．

また，前ループ文が (d) で，後ループ文が (c) あるいは (f) のとき，前ループ文の最初のイタレーションが実行された後は依存関係が存在せず，並列実行が可能である．実験結果を見ると，分割数が 1 となっており，前ループ文の漸進実行単位が 1 個実行終了した後，後ループ文の漸進実行単位 100 個が実行可能となることを表している．すなわち，前ループ文の 外側イタレーションが 1 回実行終了後，後ループ文がすべて実行可能であることを表しており，ループ文間の並列性が適切に抽出されている．

以上の結果より，我々の漸進処理適用判定アルゴリズムにより，漸進処理の適用可否，および漸進処理適用時の最適分割数が適切に計算されていることがわかった．

表 3.1: 実験結果

後＼前	(a)	(b)	(c)	(d)	(e)	(f)
(a)		依存比 1:100 i_{wait} 99 適用せず	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:1 i_{wait} 0 分割数 37
(b)	依存比 1:100 i_{wait} 99 適用せず		依存比 1:1 i_{wait} 0 分割数 37	依存比 1:100 i_{wait} 99 適用せず	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:1 i_{wait} 0 分割数 37
(c)	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:100 i_{wait} 99 適用せず		依存比 1:100 i_{wait} 0 分割数 1	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:1 i_{wait} 0 分割数 37
(d)	依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 99 適用せず		依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 99 適用せず
(e)	依存比 1:100 i_{wait} 99 適用せず	依存比 1:1 i_{wait} 49 分割数 20	依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 99 適用せず		依存比 1:1 i_{wait} 0 分割数 37
(f)	依存比 1:1 i_{wait} 49 分割数 20	依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 0 分割数 1	依存比 1:1 i_{wait} 0 分割数 37	

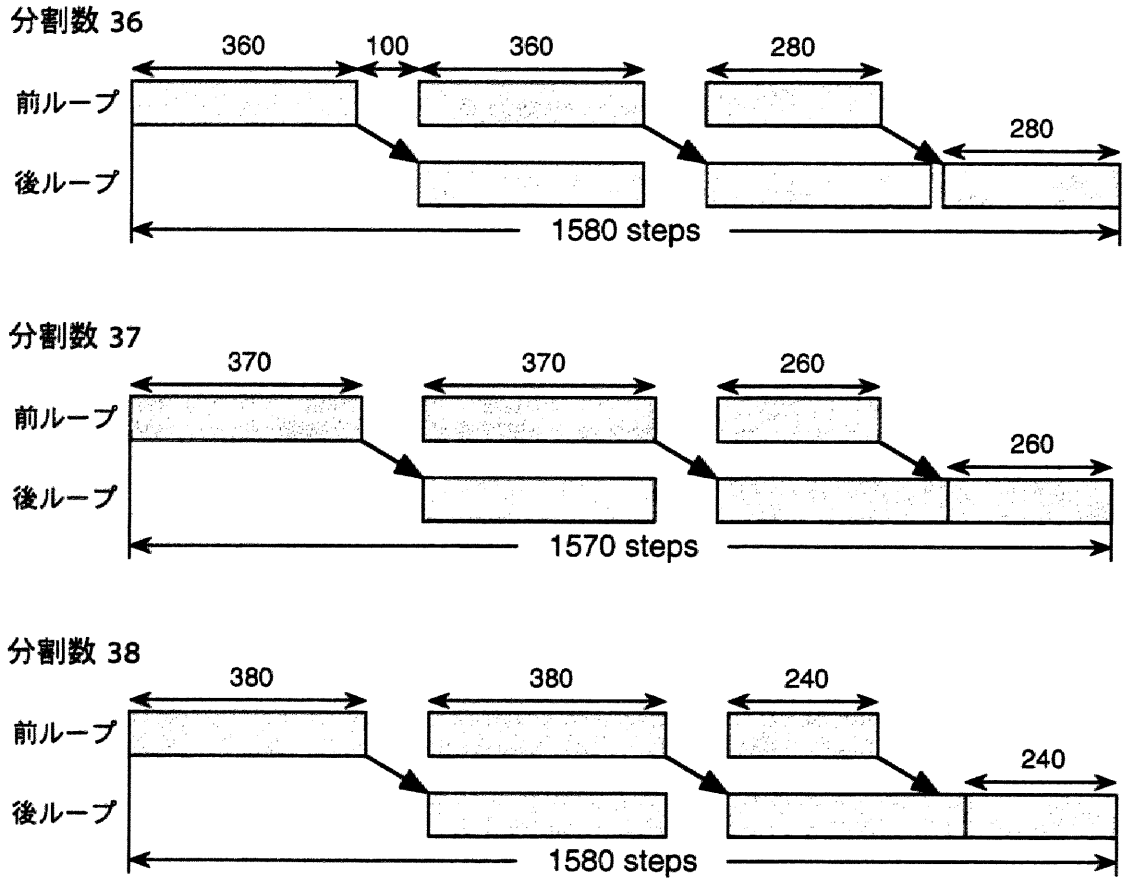


図 3.11: 実行時間の比較

3.6 様々なループ文への対応

本節では，アクセス・パターンの議論中で導入したループ文に対する制限について検討し，本章で提案したアクセス・パターンと漸進処理の適用可能性について述べる．3.6.1 節では，配列データを操作する文がループ文中に複数存在する場合のアクセス・パターンの計算法について述べる．3.6.2 節では開始点の異なる前，後ループ文の扱いについて述べる．

3.6.1 複数の配列操作文の扱い

3.3節でのアクセス・パターン解析では，配列データを操作する文が 1 文であるとしていた．しかし，多くのループ文では，内側ループ内に複数の文が存在し，配列データの異なる要素を操作している．このとき，内側イタレーションの一回の実行で操作される配列データの要素は複数であり，領域として表される．そして，その領域が内側ループの実行に従って，内側ループのアクセス・パターン・ベクトル方向に移動する．例として，図 3.12(a) のようなループ文を考える．このループ文の場合，内側イタレーションの一回の実行では三点からなる領域を操作し，この領域が内側ループの実行に従い $\overrightarrow{apv_i}$ 方向に遷移する．その結果，外側イタレーションの一回の実行では，網掛け領域が操作される．そしてこの領域が外側ループの実行に従い，$\overrightarrow{apv_o}$ の方向へ遷移する (図 3.12(b) 参照)．

このような複雑な操作領域を持つループ文の場合，以下のようにアクセス・パターンを近似して対処することができる．

- 前ループ文のとき，既操作領域に操作していない配列データの要素が含まれない箇所を境界線として，アクセス・パターンを計算する．

- 後ループ文のとき，未操作領域に操作した配列データの要素が含まれない箇所を境界線として，アクセス・パターンを計算する．

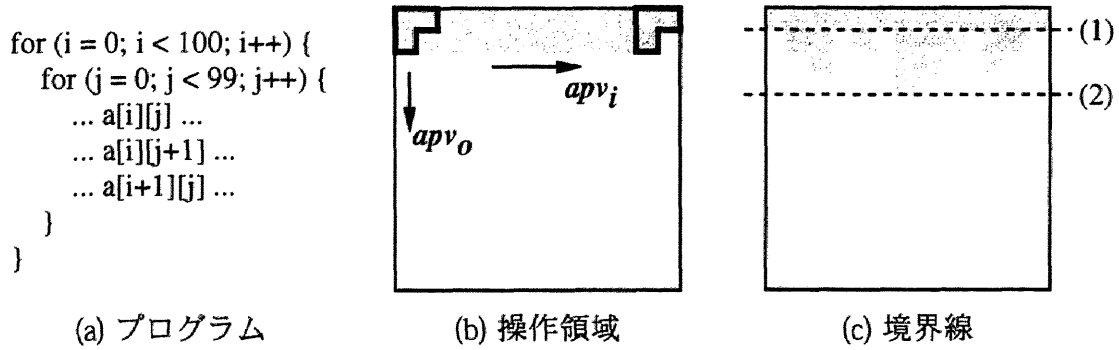


図 3.12: 複数の配列操作文

境界線の設定を 図 3.12(c) に示す．外側イタレーションの実行により操作される領域が図のハッチ部で表されるとき，前ループ文の場合は (1) を，後ループ文の場合は (2) を，それぞれ境界線としてアクセス・パターンを解析する．このように対処すると，実際は操作していない配列データの領域も操作したものとして既操作領域に含めたり，実際は操作した領域を未操作領域に含めて扱うので並列実行性は減少するが，漸進処理を適用することができる．

3.6.2 開始点が異なるループ文の扱い

前，後ループ文で開始点が異なることで問題となるのは，i_{wait} の計算と，漸進処理の適用可否の判断である．後ループ文の開始点が，前ループ文の開始点から見てどの方向に存在するかにより，図 3.13(a) ～ (d) の四つの場合に分けることができる．図中のハッチ部が実行トリガ領域を，黒丸が漸進処理開始点を，それぞれ表している．このとき，図 3.13(a), (b) の場合の漸進処理開始点は，後ループ文の最初の外側イタレーションで操作する要素に含まれるので，図 3.3 章で述べたアクセス・パターンの解析法が適用可能である．しかし，図 3.13(c), (d) のように，前ループ文の実行に先立ち実行可能な外側イタレーションが後ループ文に存在する場合，漸進処理開始点は後ループ文の最初の外側イタレーションが操作する要素に含まれない．した

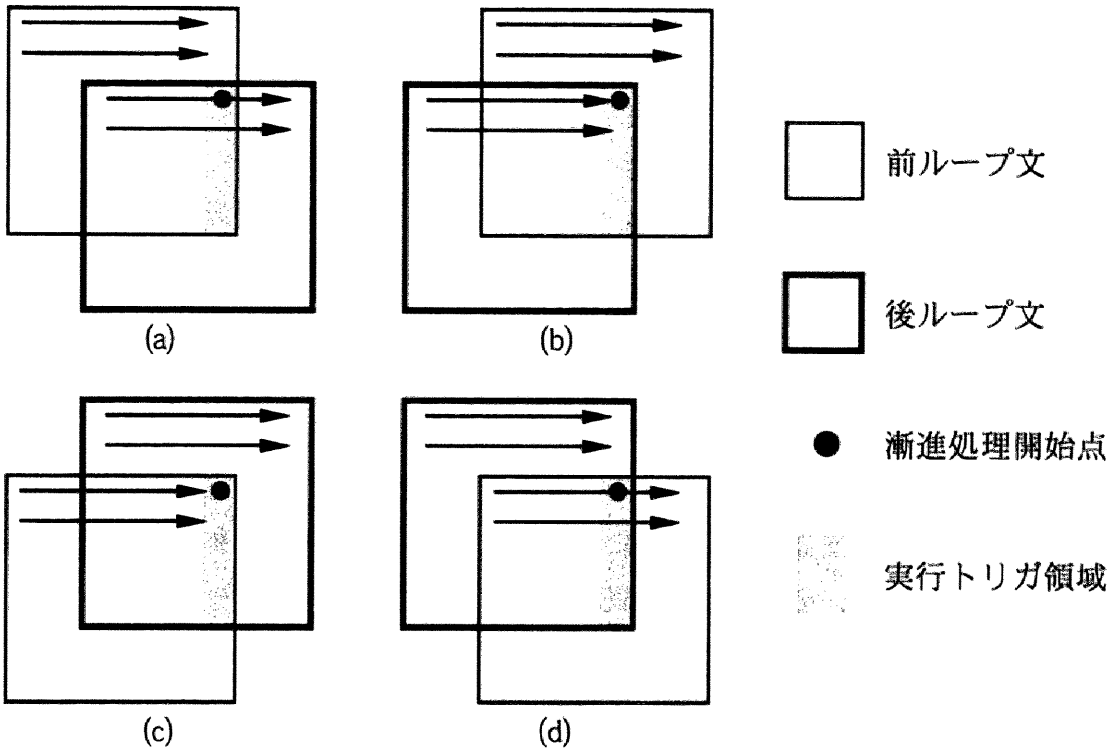


図 3.13: 開始点の異なるループ文

がって，i_{wait} の計算法をそのまま適用することができない．

しかし，このような後ループ文の外側イタレーションを，前ループ文に先立ち実行可能か否かで分割し，漸進処理を適用することが可能である．したがって，前処理により後ループ文を適切に変形できれば，図 3.13(c), (d) のような場合も，本章の議論をそのまま適用することができる．

また，漸進処理開始点が端点に表れないようなループ文も存在する．このようなループ文に対する i_{wait}，イタレーション依存比の計算方法は今後の課題である．

3.7 おわりに

本章では計算機クラスタ環境におけるループ文の並列実行手法として漸進処理を提案した．漸進処理では，イタレーション単位の並列実行ではなく，ループ文単位の並列実行が行われる．それぞれのループ文は逐次実行時と同じ実行順序でイタレーションが実行されるので，複雑なループ運搬依存によりイタレーション間の並列性が抽出できないループ文に対しても適用可能である．また，イタレーション単位の細粒度タスクによる並列処理が適さない計算機環境においても，ループ文全体をタスクとすることによりタスク起動時のオーバヘッドを削減し，計算と通信のオーバラップによりタスク間通信を隠匿することでループ文を効率よく並列実行できる．ループ文に漸進処理を適用するためには，ループ文間の並列データの操作順序に類似性が必要となるが，その類似性を表現するためアクセス・パターンを定義した．アクセス・パターンはループ文の実行経過に従って配列データがどのような順序で操作されるかをモデル化している．アクセス・パターンはループ文が操作する配列データの領域を表現するだけでなく，時間軸方向における操作領域の遷移を表現可能としている．また，ループ文間での配列操作の開始時期のずれやアクセス・パターンの違いを考慮し，漸進処理適用時の同期処理を適切に挿入する手法を提案した．このアルゴリズムにより，漸進処理の適用可否と，適用するときの最適な同期タイミングが計算される．この同期タイミングに従いループ文を実行させることで，計算と通信のオーバラップにより効率のよい並列実行を実現することが可能となった．

第 4 章

ループ文における配列操作解析のためのアクセス・パターン

本章では，前章で述べたループ文漸進処理手法を任意のループ文に適用可能とするための，ループ文アクセス・パターン解析手法について述べる．前章では，二次元配列を操作する二重ループ文のみを対象としてアクセス・パターンを定義した．本章では，ループ文の 1イタレーションで操作される配列領域とループ文の実行に伴う操作領域の移動をモデル化することを目的とした，汎用的なアクセス・パターン表現を提案する．このアクセス・パターンを計算することにより，任意のループ文間でのイタレーション依存比を評価可能となる．本章では，特にネスト・ループ文におけるアクセス・パターンの計算手法として，畳込み演算に焦点をあてる．

4.1 はじめに

プログラム並列化処理では，プログラム中のループ文を対象にする手法が一般的である．これは，プログラム並列化処理の対象となるプログラムにおいては，プログラムの実行に時間を要する部分はループ文であることが多く，ループ文を並列化することで効率よい並列処理が期待できるからである．ループ文を効率よく並列実

行させるため，現在までに様々なループ文並列化手法が提案されている [31,32,75]. 我々は前章で，ループ文並列化手法としてループ文の漸進処理手法を提案した [58,79]. ループ文の漸進処理は，同じ配列データを共有している複数のループ文をパイプライン的に並列実行させる手法である．複雑なループ運搬依存を有するループ文など，従来の手法では並列化が困難であったループ文を並列実行させることができる．

ループ文の漸進処理では，ネストしたループ文において，最外ループ文のイタレーションを並列処理の最小単位としている．そして，ループ文に対して漸進処理を適用するためには，ループ文における配列データの操作領域の解析だけでなく，ループ文がその配列データをどのような順序で操作しているかの解析が必要となる．配列データの操作順序を解析することによって，同じ配列データを類似した操作順序で操作する複数のループ文を，漸進処理の適用によりパイプライン的に並列実行可能となる．現在までにも，ループ文で操作される配列データに関する解析として様々な手法やモデルが提案されている．伝統的な手法としては，ループ文全体を実行したときに操作される配列データの要素を不等式で表す手法がある [30]．不等式表現では，操作される配列要素が存在する領域を，空間を囲む不等式の集合で表現することで，ループ文における配列データの操作の解析を幾何問題に帰着している．また，複雑な配列添字式を持つ配列操作を解析可能な Access Region[59] や LMAD (Linear Memory Access Descriptor)[60] なども提案されている．しかし，これらの手法はすべて配列データの操作領域のみの解析を目指したものであり，ループ文の実行に伴う操作領域の移動，変形を解析の対処としてはいない．もちろん，配列データのプライベート化やループ文間依存解析などに対しては，ループ文全体の実行における配列データの操作領域の解析のみで十分である．しかし，我々が提案したループ文の漸進処理では，ループ文における配列データの操作領域の解析はもちろんのこと，最外ループ文のイタレーションの実行において配列データの要素をどのような順序で操作しているかの解析が必要である．つまり，従来の手法で得られる解析情報は，ループ文の漸進処理に対して十分でない．

我々は，ループ文における配列データの操作領域や操作順序を解析しモデル化す

るために，アクセス・パターン表現を提案している [81]．アクセス・パターンについては次の 2 点を考慮しなければならない．1 つは，アクセス・パターンで表現される情報を用いることにより，漸進処理をどのように適用できるかである．もう 1 つは，アクセス・パターンをどのように計算するか，またその計算手法のプログラムに対する適用範囲である．前章では，漸進処理の適用可能性解析に必要なイタレーション依存比などの情報を，アクセス・パターンより計算する手法を示した [58]．前章で示したアクセス・パターンは二次元配列データを操作する二重ループ文のみを対象としていた．そのため，アクセス・パターンを二次元平面上の直線として幾何的に表現することができ，イタレーション依存比の計算なども直線の交点の計算で実現可能であった．しかし，この手法は多次元配列データにはそのまま適用できず，汎用性が低いという問題があった．

本章では，より汎用的なループ文のアクセス・パターンを計算する手法について述べる．特に，漸進処理において重要となるネストしたループ文のアクセス・パターンを解析するための，アクセス・パターンの畳込み演算について述べる．ネストしたループ文に対してアクセス・パターンを解析するためには，ネストした個々のループ文のアクセス・パターンを計算し，それらを融合する処理が必要となる．本章では，ネストした 2 つのループ文のアクセス・パターンが畳込み可能な条件，および畳込み後のアクセス・パターンの計算アルゴリズムについて述べる．

4.2 アクセス・パターン解析

アクセス・パターンは，ループ文における配列データの操作状況をモデル化したものである．ループ文の実行中に配列データのどのような領域をどのような順序で操作しているかを解析することを指向している．本節では，アクセス・パターンを解析するために必要な畳込み演算について述べる．まず，アクセス・パターンにおける配列操作のモデル化方法について述べる．そして，アクセス・パターンを計算するための，アクセス・パターン間の演算について述べる．

4.2.1 配列操作モデル

本章で提案するアクセス・パターンでは，Access Region や LMAD での手法と同様に，ループ文における配列データの操作位置を表す記法として，配列データの先頭要素からのオフセット値を用いる．通常，計算機のメモリ構成は線形であり，多次元配列データは column-major か row-major の方式によりメモリ空間に線形に配置される．したがって，配列データの要素の位置を先頭要素からのオフセット値で表現することにより，解析対象である配列データの次元数に関係なく統一的に操作位置を指示することができる．

一般に，ループ文における配列データの操作位置は，ループ文の実行に従って移動する．このループ文の実行に伴う操作領域の移動を適切にモデル化することが，アクセス・パターンにとって重要である．操作領域の移動をモデル化するとき，移動を捉える粒度とモデルの複雑さがトレード・オフの問題として発生する．つまり，個々の配列操作に対する移動をモデル化するか，イタレーションなどある一定回数の操作に対して移動をモデル化するかにより，表現されるアクセス・パターンの精密性と複雑性が変化する．例えば，個々の配列操作をモデル化すれば，アクセス・パターンとして表現される情報の精密性は向上するが，モデルは非常に複雑になり，アクセス・パターンの利用・応用が困難になる．また，Access Region や LMAD では，ループ文実行中の操作領域の移動をモデル化しないことでモデルを簡潔化しているが，そのため操作領域の移動などの情報は得られない．

我々のアクセス・パターンでは，ループ文の漸進処理における解析のために配列操作をモデル化することが目的である．ループ文の漸進処理では，配列データを共有する複数のループ文を，最外ループ文のイタレーション単位でパイプライン的に並列処理する [58]．したがって，イタレーション単位での操作領域の移動の解析が重要となり，イタレーション内で操作される配列データの操作順序の解析は必要でない．そこで，我々のアクセス・パターンでは，最外ループ文の 1イタレーションの実行における配列データの操作領域と，その操作領域がループ文の実行に伴いどの

ように移動するかの情報によりアクセス・パターンを構成し，ループ文における配列データの操作をモデル化する．

4.2.2 アクセス・パターン表現

我々は，図 4.1 のように正規化されたプログラムにおけるアクセス・パターンを以下のように定義する [81]．

$$(Var, Start, Stride, Num, Velocity, Accuracy)_{ub}$$

ここで，

Var ··· 最外ループ文のループ変数，

$Start$ ··· ループ文の実行で最初に操作される位置，

$Stride$ ··· 操作位置の間隔，

Num ··· イタレーションで操作する要素の個数，

$Velocity$ ··· 操作領域の移動速度，

$Accuracy$ ··· 操作の確実性，

ub ··· ループ変数の最大値，

である．ただし，配列データの先頭アドレスを 0 としている．最外ループ文の 1 イタレーションの実行において操作される配列要素は，$Stride$ と Num でモデル化されている．1 イタレーションで操作される配列要素の個数を Num で，配列要素の規則的な並びを $Stride$ でそれぞれ表現し，1 イタレーションで操作される配列データの操作領域を規定する．そして，$Stride$ と Num で規定された操作領域が，ループ文の実行に従って $Velocity$ で示される速度で移動するとして，ループ文全体の操

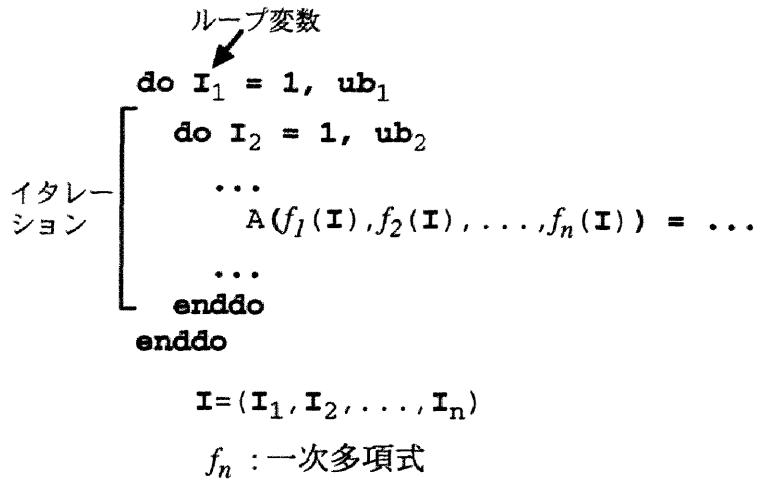


図 4.1: 解析対象のループ文の形式

作領域をモデル化している．また，*Accuracy* は *Stride*, *Num*, *Velocity* で表現される操作領域の正確性を表すパラメータであり，*MAY* か *MUST* のどちらかの値を取る．通常は *MUST* を取るが，ループ文中で実行される配列を操作する文が if 文などにより不確定である場合，*MAY* とすることでその不確定さを表現可能としている．また，後述するアクセス・パターンの概略表現においても *MAY* の値を取る．

図 4.2 にアクセス・パターンにおける各パラメータの意味を模式的に示す．図中の正方形は配列要素を表しており，正方形の横の並びが解析対象になっている配列データを表している．ハッチがかかっている配列要素はイタレーション中で操作されていることを表している．縦方向が時間の経過を表しており，ループ文の実行が進むにつれて操作される配列要素が変化することを表している．最初に操作される配列要素が *Start* で，操作要素間の間隔が *Stride* で，操作要素数が *Num* で，それぞれ示されている．そして，ループ文の実行による操作領域の移動は，各イタレーションにおいて最初に操作される操作要素間の間隔として，*Velocity* パラメータで特徴づけられることが分かる．

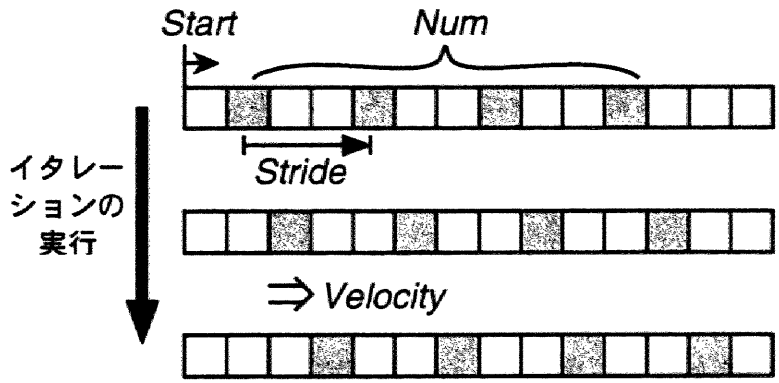


図 4.2: アクセス・パターンの模式図

ループ文においてアクセス・パターンが計算されると，アクセス・パターンのパラメータよりループ文の漸進処理の適用可否を判断することができる．漸進処理では2つのループ文において，それぞれのイタレーションをどのような割合で実行できるかを表すイタレーション依存比の計算が重要である [58]．前ループ文，後ループ文のアクセス・パターンをそれぞれ

- $A_{pred}(Var_p, St_p, Str_p, Num_p, Vel_p, Acc_p)_{ub_p}$,
- $A_{succ}(Var_s, St_s, Str_s, Num_s, Vel_s, Acc_s)_{ub_s}$

とすると，2つのループ文のイタレーション依存比は

$$Vel_s : Vel_p$$

と計算することができる．そして，他のパラメータより前章で述べた漸進処理開始点までのイタレーション実行回数 i_{wait} を計算することで，漸進処理の適用可否の判断，および同期タイミングを決定することができる．

4.2.3 アクセス・パターン間の演算

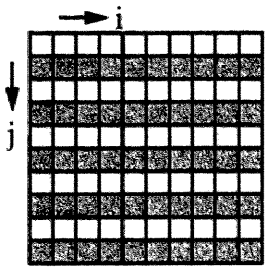
アクセス・パターンを計算するとき，ネストしたループ文の扱いや，1 つのループ文内に存在する複数の配列操作の扱いが重要である．単純なプログラム例を除けば，一般に実プログラムにおけるアクセス・パターンの計算は難しい．ネストしたループ文に対してアクセス・パターンを計算するとき，内側ループ文のアクセス・パターンをまず計算し，それを元に外側ループ文のアクセス・パターンを順次計算することができれば，アクセス・パターンの計算が容易になる．また，1 文中に複数の配列操作が存在する場合も，それぞれの配列操作に対してアクセス・パターンを計算し，それらを合成できれば，アクセス・パターンの計算が容易になる．このようなアクセス・パターン間の演算を定義することで，ネストの深さや配列操作の個数に関わらず，常にアクセス・パターンが解析可能となる．我々は，ネストしたループ文において，内側ループ文のアクセス・パターンを外側ループ文のアクセス・パターンに融合し，外側ループ文のアクセス・パターンを計算することを，畳込み演算と定義する．また，1 つのループ文内に存在する複数の配列操作に対してそれぞれのアクセス・パターンを合成し，1 つのアクセス・パターンを計算することを集約演算と定義する．

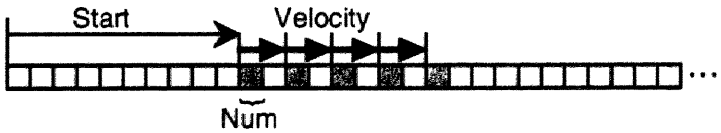
以下，それぞれの演算の例を挙げる．図 4.3 は畳込み演算の例である．図 4.3 のプログラムの二重ループ文で配列を操作している部分の配列添字式から配列データの操作位置を先頭アドレスからのオフセット値で表すと

$$2 \times j + 10 \times (i - 1) - 1 = 10i + 2j - 11$$

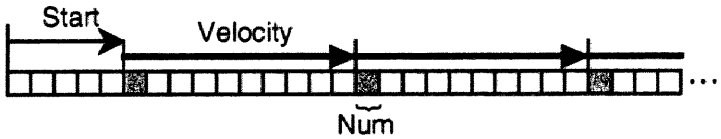
となる．ここで，内側ループ文だけに注目してアクセス・パターンを解析する．ループ文の最初の実行で操作される要素は，配列添字式に $j = 1$ を代入することで得られ，$10i - 9$ となる．また，イタレーション内で実行される文は 1 文のみなので，1 イタレーションで操作される要素は 1 つのみである．さらに，配列添字の j の係数より，内側ループ文の実行に伴う操作領域の移動が 2 であることが分かる．した

```
integer a(10,10)
do i=1, 10
  do j=1, 5
    a(2*j, i) = ...
  enddo
enddo
```

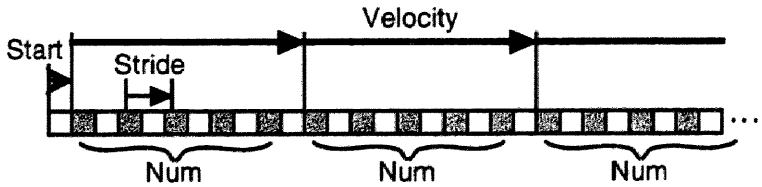




$A_{in}(j, 10i-9, 1, 1, 2, MUST)_5$



$A_{out}(i, 2j-1, 1, 1, 10, MUST)_{10}$



$A(i, 1, 2, 5, 10, MUST)_{10}$

図 4.3: アクセス・パターン間の畳込み演算

```
integer a(100)
do i=1, 91, 10
  a(i) = ...
  a(i+3) = ...
  a(i+6) = ...
enddo
```

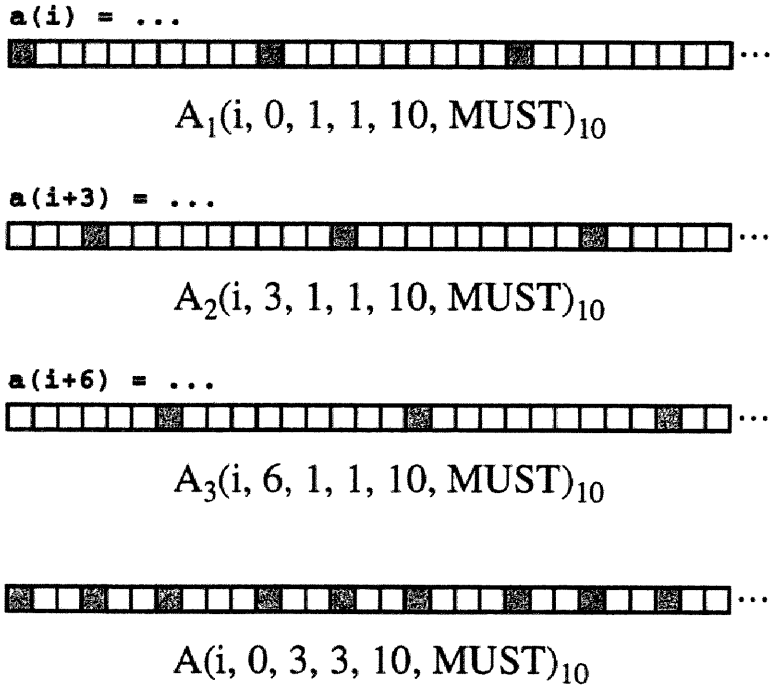


図 4.4: アクセス・パターン間の集約演算

がって，内側ループ文のみに注目したアクセス・パターンは図中の A_{in} のように表される．同様に，外側ループ文だけに注目し，内側ループ文が存在しないとして計算されたアクセス・パターンは A_{out} のように表される．畳込み演算は，2つのアクセス・パターン A_{in}, A_{out} から，二重ループ文に対するアクセス・パターン A を計算するための演算である．同様に図 4.4 に集約演算の例を示す．図 4.4のプログラムのループ文には複数の配列操作が存在している．配列操作はそれぞれ A_1, A_2, A_3 のようなアクセス・パターンを有する．集約演算は，複数のアクセス・パターンより，ループ文のアクセス・パターン A を計算するための演算である．

アクセス・パターンの計算には上記2つの演算が重要であるが，本章ではアクセス・パターンの畳込み演算に焦点を当て，畳込み演算の適用可否判定やアクセス・パターンの計算法について述べる．アクセス・パターンの集約演算は今後の課題である．

4.3 アクセス・パターンの畳込み演算

4.2節でも述べたように，我々のアクセス・パターンではイタレーションで操作される配列データを *Stride* と *Num* で，またその移動を *Velocity* でモデル化している．ネストしたループ文のアクセス・パターンを考えるとき，外側ループ文の1イタレーションの実行での配列データの操作領域は，内側ループ文全体の実行における操作領域となる[1]．したがって，内側ループ文のアクセス・パターンが外側ループ文のアクセス・パターンに畳込み可能となるためには，内側ループ文のアクセス・パターン全体で操作される配列データの領域を，外側ループ文のアクセス・パターン中の *Stride* と *Num* の2つのパラメータで記述可能でなければならない．すなわち，内側ループ文全体の実行で操作される配列データの要素が，*Stride* と *Num* の2つのパラメータで表現可能となるよう，規則的に並んでいなければならない．操作される配列データの要素が規則的に並んでいれば畳込み可能であり，ネストしたルー

[1]簡単化のため，ここではパーフェクト・ループ文を対象としている．

プ文における配列操作がアクセス・パターンで表現可能となる．逆に，操作される配列データの要素が *Stride* と *Num* で記述できなければ，ネストしたループ文のアクセス・パターンを計算することができない．そこで，ネストしたループ文においてアクセス・パターンを解析するとき，2 つのアクセス・パターンが畳込み可能であるか否かの判定が必要となる．

以下，内側ループ文のアクセス・パターンが外側ループ文のアクセス・パターンに畳込み可能となる条件を示し，ネストしたループ文に対する 2 つのアクセス・パターンの畳込み演算について定義する．

4.3.1 畳込み演算の定義

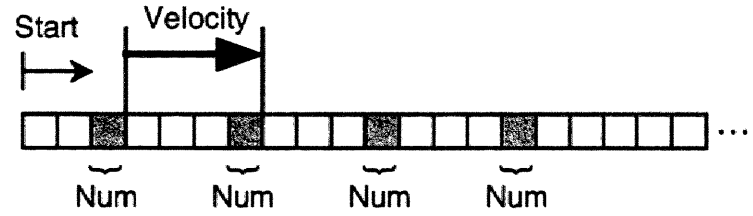
アクセス・パターンが畳込み可能となる条件，すなわちアクセス・パターンで表現される操作領域中の配列データ要素が，*Stride* で表すことのできる一定の間隔で並んでいる条件は以下の 3 通りである．

1. 内側ループ文の 1 イタレーションで操作される配列要素が 1 つだけのとき．

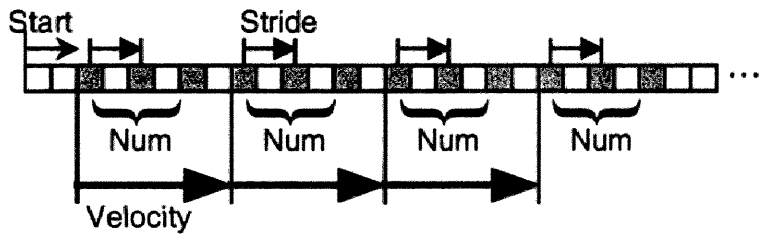
2. 内側ループ文で $Velocity \geq Stride$ の場合，*Velocity* が *Stride* の整数倍となっており，かつ領域が連続する十分な配列要素を操作しているとき．

3. 内側ループ文で $Velocity < Stride$ の場合，*Stride* が *Velocity* の整数倍となっており，かつ領域が連続する十分な配列要素を操作しているとき．

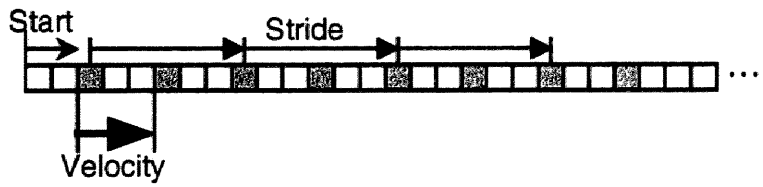
条件 (1) は自明である．図 4.5(a) で示すように，1 イタレーションで操作される配列要素が 1 つだけのとき，ループ文全体の実行で操作される配列要素は *Velocity* の間隔で位置している．また，そのときの配列データの要素数は，ループ変数の上限値 *ub* となる．したがって，*Velocity* の値を *Stride* とし，*ub* の値を *Num* とすれば，ループ文全体で操作される配列データの要素集合を *Stride* と *Num* で表現できる．



(a) Num=1 のとき



(b) Velocity ≧ Stride のとき



(c) Velocity ＜ Stride のとき

図 4.5: 畳込み可能条件

次に，図 4.5(b) により条件 (2) を考える．1 イタレーションの実行で $Stride$ の間隔で並ぶ Num 個の配列要素が操作される．そして，ループ文の実行が進むにつれ，$Velocity$ だけ操作位置が移動する．このような操作領域が $Stride$ と Num の 2 つのパラメータで表現可能となるためには，すべての配列要素が等間隔で並んでいる必要があり，それは $Velocity$ が $Stride$ で割り切れるときである．また，$Velocity$ が $Stride$ で割り切れたとしても，配列要素が隙間なく並んでいなければならない．そのため，$Stride \times Num \geq Velocity$ という関係が成立していなければならない．この関係が成立していないループ文では，操作される配列要素が等間隔とならない．また，このときループ文全体で操作される配列要素数は，図 4.6(a) で示す通りに計算できる．つまり，ループ変数が 1 から $ub-1$ までの部分での要素数は $\frac{Velocity}{Stride}$ であり，ループ変数が ub の部分での要素数は Num である．したがって，ループ文全体で操作される配列要素数は

$$\frac{Velocity}{Stride} \times (ub-1) + Num$$

と計算することができる．

条件 (3) は，条件 (2) において $Velocity$ と $Stride$ の大小関係が異なるだけであるので (図 4.5(c) 参照)，畳込み可能となる条件は条件 (2) と同様である．また，操作される配列要素数も同様に

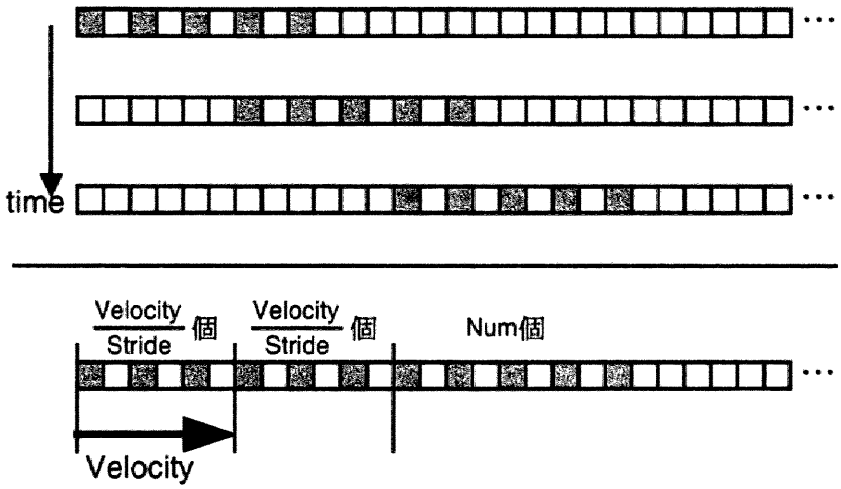
$$\frac{Stride}{Velocity} \times (Num-1) + ub$$

と計算できる (図 4.6(b) 参照)．

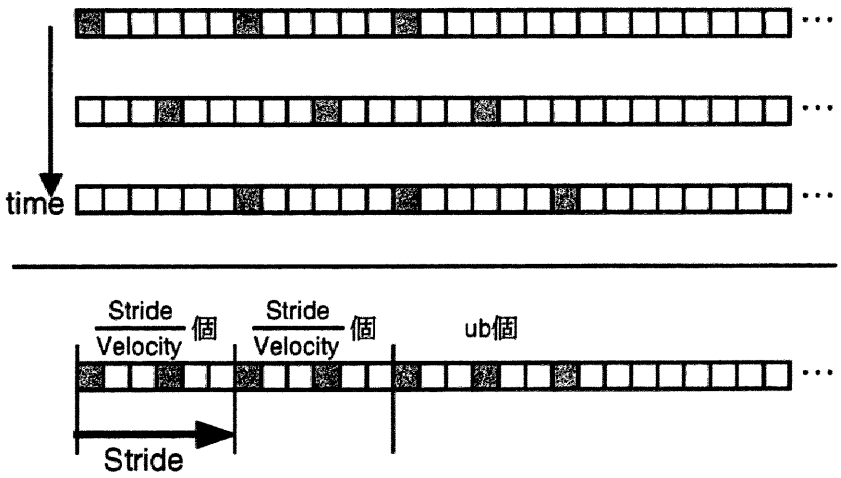
これらより，畳込み演算は以下のように定義される．

[アクセス・パターンの畳込み演算]

二重ループ文において，内側ループ文と外側ループ文のアクセス・パターンをそれぞれ，



(a) Velocity ≧ Stride のとき



(b) Velocity ＜ Stride のとき

図 4.6: 操作される配列要素の個数

内側: $A_{in}(Var_1, St_1, Str_1, Num_1, Vel_1, Acc_1)_{ub_1}$

外側: $A_{out}(Var_2, St_2, Str_2, Num_2, Vel_2, Acc_2)_{ub_2}$

とする．このとき，内側ループ文のアクセス・パターンが外側ループ文のアクセス・パターンに畳込み可能である条件，および畳込みの結果計算されるアクセス・パターン A は以下のように定義される．

1. $Num_1 = 1$ であるとき．

   $A(Var_2, St'_2, Vel_1, ub_1, Vel_2, Acc'_2)_{ub_2}$

2. $Vel_1 \geq Str_1$ で，Vel_1 が Str_1 の整数倍であり，かつ $Str_1 \times Num_1 \geq Vel_1$ であるとき．

   $A(Var_2, St'_2, Str_1, \frac{Vel_1}{Str_1} \times (ub_1 - 1) + Num_1, Vel_2, Acc'_2)_{ub_2}$

3. $Vel_1 < Str_1$ で，Str_1 が Vel_1 の整数倍であり，かつ $Vel_1 \times Num_1 \geq Str_1$ であるとき．

   $A(Var_2, St'_2, Vel_1, \frac{Str_1}{Vel_1} \times (Num_1 - 1) + ub_1, Vel_2, Acc'_2)_{ub_2}$

ここで，

$$St'_2 = St_1 + \left\{ \begin{array}{ll} Str_1 \times (Num_1 - 1) & (Str_1 < 0) \\ 0 & (\text{それ以外}) \end{array} \right\} + \left\{ \begin{array}{ll} Vel_1 \times (ub_1 - 1) & (Vel_1 < 0) \\ 0 & (\text{それ以外}) \end{array} \right\}$$

で $Var_1 = 1$ を代入した値であり，

$$Acc_2' = \begin{cases} MUST & (Acc_1 = Acc_2 = MUSTのとき) \\ MAY & (それ以外) \end{cases}$$

である．

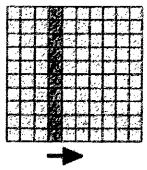
4.3.2 畳込み演算の例

4.3.1 節に述べた畳込み演算の計算例を図 4.7 に挙げる．図 4.7(a) は二次元配列データを全走査するプログラムである．内側ループ文のアクセス・パターンにおいて $Num = 1$ となるので，この内側ループ文のアクセス・パターンは外側ループ文のアクセス・パターンに畳込み可能である．アルゴリズムに従ってアクセス・パターンを計算すると，$A(i, 0, 1, 10, 10, MUST)_{10}$ となり，プログラムでの操作順序が正確に解析できていることが分かる．図 4.7(b) のように操作順序が異なったプログラムに対してもアクセス・パターンは解析可能である．どちらのプログラムも同じ操作領域を有するが，操作順序の違いがパラメータにより正確に表現されている．

複雑な配列操作のプログラム例として，波頭 (wave-front) 型に配列を操作するプログラムを図 4.7(c) に示す．このプログラムも内側ループ文のアクセス・パターンにおいて $Num = 1$ であるので，条件 (1) により畳込み可能である．図 4.7(d) は三次元配列データを操作する三重ループ文のプログラム例である．内側の二重ループ文に関しては他のプログラム例と同様に条件 (1) により畳込み可能となり，内側二重ループ文のアクセス・パターンが $A_{kj}(j, 100i-99, 2, 5, 10, MUST)_{10}$ と解析されている．そして，最外ループ文への畳込みに関しては，条件 (2) を満たしており畳込み可能である．したがって，最終的にアクセス・パターンが $A(i, 1, 2, 50, 100, MUST)_{10}$ と正確に計算されている．

以上に示したように，我々が提案した畳込み演算によって，ネストしたループ文に対するアクセス・パターンが正確に解析可能であることが分かった．最後の例の

```
integer a(10,10)
do i=1,10
  do j=1,10
    a(j,i) = ...
  enddo
enddo
```



内側アクセス・パターン

$A_j(j, 10i-10, 1, \underline{1}, 1, MUST)_{10}$

畳込み可能

外側アクセス・パターン

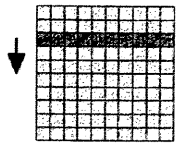
$A_i(i, j-1, 1, 1, 10, MUST)_{10}$

アクセス・パターン

$A(i, 0, 1, 10, 10, MUST)_{10}$

(a) 二重ループ文

```
integer a(10,10)
do i=1,10
  do j=1,10
    a(i,j) = ...
  enddo
enddo
```



内側アクセス・パターン

$A_j(j, i-1, 1, \underline{1}, 10, MUST)_{10}$

畳込み可能

外側アクセス・パターン

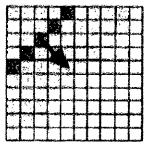
$A_i(i, 10j-10, 1, 1, 1, MUST)_{10}$

アクセス・パターン

$A(i, 0, 10, 10, 1, MUST)_{10}$

(b) 異操作順序の二重ループ文

```
integer a(10,10)
do i=1,10
  do j=1,i
    a(i-j+1,j) = ...
  enddo
enddo
```



内側アクセス・パターン

$A_j(j, i-1, 1, \underline{1}, 9, MUST)_i$

畳込み可能

外側アクセス・パターン

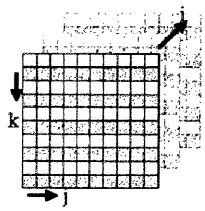
$A_i(i, 9j-9, 1, 1, 1, MUST)_{10}$

アクセス・パターン

$A(i, 0, 9, i, 1, MUST)_{10}$

(c) 波頭ループ文

```
integer a(10,10,10)
do i=1,10
  do j=1,10
    do k=1,5
      a(2*k,j,i) = ...
    enddo
  enddo
enddo
```



k ループ文アクセス・パターン

$A_k(k, 100i+10j-109, 1, \underline{1}, 2, MUST)_5$

畳込み可能

j ループ文アクセス・パターン

$A_j(j, 100i+2k-101, 1, 1, 10, MUST)_{10}$

内側二重ループ文アクセス・パターン

$A_{kj}(j, 100i-99, \underline{2}, \underline{5}, \underline{10}, MUST)_{10}$

畳込み可能

i ループ文アクセス・パターン

$A_i(i, 10j+2k-11, 1, 1, 100, MUST)_{10}$

アクセス・パターン

$A(i, 1, 2, 50, 100, MUST)_{10}$

(d) 三重ループ文

図 4.7: 畳込み演算の例

ように，内側ループ文から順次畳込み演算を施すことで，ネストの数に関係なく様々なループ文においてアクセス・パターンを計算することができる．

4.4 評価実験

本節では，アクセス・パターンの畳込み演算が実プログラム中に現れる配列操作に対してどの程度適用可能かを評価するために実施した実験について述べる．

4.4.1 実験内容

本実験では NAS ベンチマーク・プログラムの逐次版 (NPB2.3-serial) を用いて，ベンチマーク・プログラムに含まれる配列操作に対してアクセス・パターンを解析した [82]．畳込み演算アルゴリズムのみの評価のため，本実験ではそれぞれの配列操作が独立のループ文中に存在するとした．すなわち，図 4.8 に示すように複数の配列操作が存在するプログラムの場合，独立なループ文にそれぞれの配列操作が存在したプログラムであると捉え，アクセス・パターンの畳込み演算を適用した[2]．また，プログラムにはループ文内の定数伝搬処理，変数の展開処理などを事前に施した．アクセス・パターンの解析には，我々が現在開発中の並列化コンパイラ・ツールキットを用いた．並列化コンパイラ・ツールキットは SUIF[83] を元に開発されており，並列化コンパイラの開発を容易にするためプログラム中の文間の依存関係の解析などが可能となっている [84]．本実験では，並列化コンパイラ・ツールキットのプログラム解析部を用いて，アクセス・パターンを解析するプログラムを実装した．

実験では，アクセス・パターンの解析率について調査した．プログラムに畳込み演算を適用するとき，3 種類の結果が生じる．すなわち，最外ループ文に対するアクセス・パターンが解析できる場合 (完全解析) と，途中のネストまでアクセス・パ

[2]したがって，本実験は NPB2.3-serial に対する提案手法の直接の有効性を示すものではない．実プログラムによく出現する配列操作に対して提案手法がどの程度適用可能かを示すために，例として NPB2.3-serial に出現する配列操作を利用した実験である．

```
integer a(10,10)

do i=1, 10
  do j=1, 10
    a(j,i) = a(j,i) + a(i,j)
  enddo
enddo
```

```
integer a(10,10)

do i=1, 10
  do j=1, 10
    a(j,i) = ...
  enddo
enddo

do i=1, 10
  do j=1, 10
    ... = a(j,i)
  enddo
enddo

do i=1, 10
  do j=1, 10
    ... = a(i,j)
  enddo
enddo
```

図 4.8: プログラムの変換

ターンが解析できる場合 (部分解析) と，アクセス・パターンがまったく解析できない場合 (解析失敗) である．プログラム中の全配列操作に対して上記の場合の割合を調査することで，畳込み演算アルゴリズムの適用可能性を評価した．

4.4.2 実験結果

表 4.1に実験結果を示す．NPB2.3-serial の各プログラムに対するアクセス・パターンの解析結果を示している．表では，各プログラム内の配列操作の総数，解析成功数，解析失敗数が示されている．また，解析成功数の内訳として完全解析数と部分解析数が，解析失敗数の内訳として失敗の原因が，それぞれ示されている．表を見ると解析失敗数，すなわちアクセス・パターン解析の対象外となる配列操作は 4.1% であり，プログラム中のほとんどすべての配列操作がアクセス・パターンを解析可能であることがわかる．解析失敗となった配列操作は，ループ文の上限・下限値が変数で不定であるループ文内に存在した配列操作であったり，配列のサイズが不定であるといったコンパイル時の静的解析では対処できないものや，while ループ文など解析の対象外であるものがほとんどである．配列添字式が複雑で解析できなかったものは，`a(b(i))` のような添字式に配列が存在するもののみであり，ほとんどすべての配列操作中の配列添字式は我々の規定した制限内のものであった．このことより，静的解析で対処可能な配列操作に対してアクセス・パターンをほぼ解析可能であることがわかる．解析成功の中では，BT, LU, SP 以外のプログラムではすべての配列操作が完全解析されている．部分解析のうち，解析できたネスト数をまとめたものを表 4.2 に示す．総ネスト数の 64.8% のネストが解析されていることがわかる．また，解析結果の詳細を表 4.3に示す．表ではプログラム中のネスト・ループ文において解析できたネスト数が示されている．縦方向の数字がループ文が何重ネストであるかを，横方向の数字が解析できたネスト数を，それぞれ表している．対角線部分の数字は完全解析となったループ文の数を，それ以外は部分解析となったループ文の数を，それぞれ表している．これを見ると，まったく畳込みが適用でき

表 4.1: 実験結果

プログラム名	総配列操作数	解析成功数	完全解析	部分解析	解析失敗数	while ループ	添字式規定外	上限・下限値	可変サイズ
BT	1698	1698 100.0%	439	1259	0 0.0%	0	0	0	0
CG	54	52 96.3%	52	0	2 3.7%	0	0	2	0
EP	10	10 100.0%	10	0	0 0.0%	0	0	0	0
IS	53	40 75.5%	40	0	13 24.5%	0	3	0	10
LU	1648	1581 95.9%	1102	479	67 4.1%	0	0	0	67
MG	247	122 49.4%	122	0	125 50.6%	49	6	0	70
SP	1392	1392 100.0%	513	879	0 0.0%	0	0	0	0
合計	5102	4895 95.9%	2278 46.5%	2617 53.5%	207 4.1%	49	9	2	147

注釈
- whileループ … whileループ文中の配列操作でループ変数が存在しない
- 添字式規定外 … 配列添字式が複雑で解析対象外
- 上限・下限値 … ループ文の上限値，あるいは下限値が不定
- 可変サイズ … 配列サイズが不定

表 4.2: 部分解析における解析ネスト数

プログラム名	総ネスト数	解析成功ネスト数	解析失敗ネスト数
BT	3849	2563 66.6%	1286 33.4%
LU	1515	987 65.1%	528 34.9%
SP	2838	1764 62.2%	1074 37.8%
合計	8202	5314 64.8%	2888 35.2%

ない，すなわち横方向の数字が 1 である欄に属する部分解析となった配列操作文は存在せず，1 重ループ文を除くすべての配列操作で畳込み演算が適用できたことが分かる．この部分解析の多くは 4.5.2 節で述べる *Accuracy* パラメータを用いた対処により完全解析可能であり，これらの実験結果より，アクセス・パターンの畳込み演算はほとんどの配列操作に対して適用可能であることがわかる．

4.5 議論

4.5.1 関連研究

ループ文において操作される配列データに対する解析として，我々のアクセス・パターンに類似したものに LMAD がある [60]．LMAD は複雑な添字式を有する配列操作を持つループ文を実行したときの操作領域を解析するために開発された．ループ文全体で操作される領域を解析することで配列データのプライベート化を図り，ループ文間の並列性を抽出することを目的としている．したがって，ループ文の全体の実行中の操作領域のみをモデル化し，その配列データ要素の操作順序はモ

表 4.3: 部分解析結果の詳細

BT	1	2	3	4	5
1	43				
2	0	39			
3	0	1202	320		
4	0	15	27	37	
5	0	3	6	6	0

LU	1	2	3	4	5
1	50				
2	0	877			
3	0	406	145		
4	0	44	24	30	
5	0	0	5	0	0

SP	1	2	3	4
1	43			
2	0	132		
3	0	678	301	
4	0	195	6	37

… 完全解析

□ … 部分解析

デル化の対象としていない．LMAD では，*span* と *stride* のパラメータの組を複数用いることで，配列データの操作領域をモデル化している．*span* により領域の幅，すなわち操作領域の両端間の距離を表し，*stride* によりその領域内の操作される配列要素の間隔を表している．これは，我々のアクセス・パターンにおいて 1 イタレーションで操作される領域を $Stride$ と Num で表現しているのと同様な考え方である．多くの場合，

$$span = Stride \times (Num - 1)$$

と表現することができ，我々のアクセス・パターンは LMAD と同様の表現力を有している．

LMAD は操作領域内の操作順序をモデル化していない．すなわち，ループ文中での操作順序に関係なく，操作する領域が同じであれば，そのループ文に対する LMAD は同値となる．例えば，図 4.7(a),(b) のようなループ文の場合，LMAD はそれぞれ $A_{10}{}^{1}_{100}{}^{10}+0$, $A_{100}{}^{10}_{10}{}^{1}+0$ となる．このとき，同値の定義によりこれら 2 つの LMAD は同一のものとして計算可能となっている．このように LMAD では配列操作を簡略化してモデル化しているため，LMAD では我々のアクセス・パターンの畳込み演算と同様な coaleseable 演算の他，集約演算も定義可能となっている．これに対し，アクセス・パターンでは操作領域内の配列データの操作順序を表現するための *Velocity* パラメータが導入されており，同様な配列領域を持つループ文でも様々なアクセス・パターン表現が考えられる．そのため，畳込み可能な条件も LMAD と比較すると複雑になっている．しかし，畳込み演算の定義により，LMAD と同様にネストしたループ文に対する解析が可能となっている．

つまり，まとめると，LMAD はループ文全体の実行における配列データの操作領域を解析するために開発され，*span* と *stride* のパラメータの組により構成されている．シンプルなモデルなので表現力は弱いが，様々な演算が定義されており，複雑な配列添字式が解析可能となっている．それに対して，我々のアクセス・パターンはループ文の実行に伴う配列データの操作領域の移動を解析可能なように拡張さ

れている．そのため，演算は複雑になっているが，アクセス・パターンの解析に最低限必要な畳込み演算が定義されており，ループ文中の配列操作をより柔軟にモデル化することができる．

4.5.2 部分解析に対する対処

実験においては，解析成功数のうち約 50% が部分解析であった．部分解析とは，途中のネストまで畳込み演算が適用できたが，最外ループ文のアクセス・パターンは計算できなかったものを指す．これら部分解析となった配列操作の多くは，配列データの一部分のみを操作し，配列領域が連続しないことが原因であった．つまり，図 4.9に示すようにループ文の上限値や下限値が配列データ全体を操作しないプログラムの場合，畳込み演算適用判定において $Str \times Num \geq Vel$ や $Vel \times Num \geq Str$ が成立せず，アクセス・パターンを計算することができないことが，部分解析の主な原因であった[3]．

ここで，アクセス・パターンの $Accuracy$ パラメータを用い，アクセス・パターンを概略的に表現することで，アクセス・パターンが計算可能となる．すなわち，アクセス・パターンに対して以下の変換規則を用いることで畳込み演算を適用することが可能となる場合がある．

- $A(Var, St, Str, Num, Vel, Acc)_{ub}$

  $\downarrow$

  $A'(Var, St, Str, Num', Vel, MAY)_{ub}$

  ただし， $Num' = Num + \alpha \quad (\alpha > 0)$

同様に， $Stride, Velocity$ パラメータに対しても以下の変換規則が定義可能である．

- $A(Var, St, Str, Num, Vel, Acc)_{ub}$

  $\downarrow$

[3]図のような配列操作は，配列の境界と内部での計算方法が異なる場合や，袖領域を用いた場合などによく現れる．

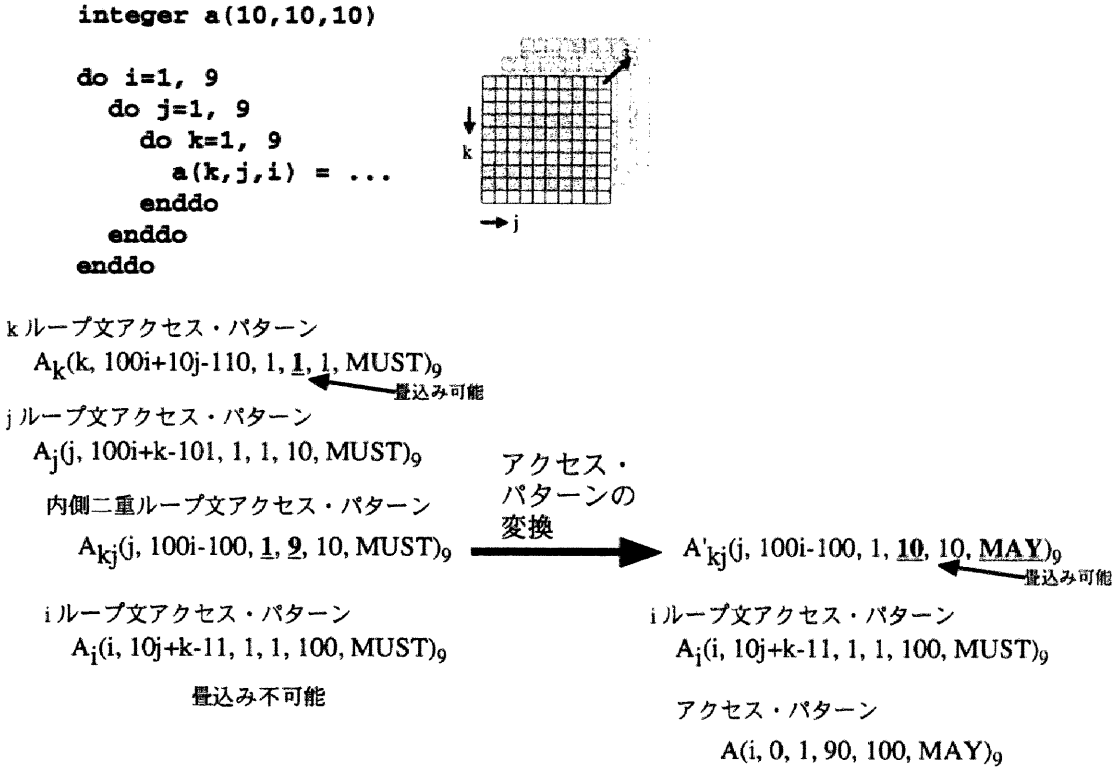


図 4.9: Accuracy パラメータを用いたアクセス・パターン変換

$A'(Var, St, Str', Num', Vel, MAY)_{ub}$
ただし，Str'はStrの約数，$Num' = Num \cdot \frac{Str}{Str'}$

- $A(Var, St, Str, Num, Vel, Acc)_{ub}$
  ↓
  $A'(Var, St, Str, Num, Vel', MAY)_{ub'}$
  ただし，Vel'はVelの約数，$ub' = ub \cdot \frac{Vel}{Vel'}$

これらの演算により，部分解析となった配列操作に対してアクセス・パターンを解析することが可能となる．例えば，図 4.9 に示すように，内側のアクセス・パターン A_{kj} を

$$A_{kj}(j, 100i - 100, 1, 9, 10, MUST)_9$$
↓
$$A'_{kj}(j, 100i - 100, 1, 10, 10, MAY)_9$$

と変換することで，最外ループ文のアクセス・パターンが $A(i, 0, 1, 90, 100, MAY)_9$ と計算でき，完全解析が可能となる．

しかし，これらの変換規則を適用することによりアクセス・パターンの正確性が減少するので，変換規則の適用は状況により判断する必要がある．*Accuracy* パラメータを用いたアクセス・パターンの変換と，その適用範囲については今後の課題として十分に検討する必要がある．

4.6 おわりに

本章では，ループ文の漸進処理適用のためのアクセス・パターンの計算手法として畳込み演算アルゴリズムを示した．前章でもアクセス・パターンについて述べたが，これは二次元配列を操作する二重ループ文に特化した表現であり，汎用性に問題があった．本章で提案したアクセス・パターンは，ループ文中の配列操作を 1 イタ

レーションで操作する領域とループ文の実行に伴う領域の移動でモデル化することにより，任意の配列を操作する任意のループ文の解析を可能とした．本アクセス・パターンはループ文における配列データの操作領域だけでなくループ文実行中の操作領域の移動をモデル化しているので，従来提案されてきたループ文での操作領域をモデル化する手法では表現できない特性を表現可能とした．畳込み演算は，ネストしたループ文中の内側ループ文のアクセス・パターンを外側ループ文のアクセス・パターンに融合し，ネストしたループ文におけるアクセス・パターンを計算する演算である．アクセス・パターンが畳込み可能である条件を示し，畳込みの結果得られるアクセス・パターンの計算アルゴリズムについて述べた．畳込み演算を定義したことで，ネストしたループ文でのアクセス・パターンが解析可能となった．

第 5 章

結論

5.1 本論文のまとめ

本論文では，計算機クラスタ環境において効率よい並列処理を実現するためのプログラム並列化手法について述べた．計算機クラスタ環境を並列処理環境として活用し並列処理を実現するためには，従来の並列計算機環境との相違を考えることが重要である．また，その相違点に対処したプログラム並列化手法や並列処理手法が必要となる．従来の並列計算機環境と計算機クラスタ環境の相違点は，プロセッサ間の通信処理がプロセッサの計算処理と比較すると相対的に低速なことである．すなわち，プロセッサにおける計算処理とネットワークにおける通信処理の処理時間の差が大きいということである．したがって，この相違点のため，計算機クラスタ環境において並列処理を実現するには，以下の 2 つの方針が重要である．

1. プロセッサ間通信がなるべく発生しないような並列処理を実現する．
2. タスクの実行スケジューリング，同期タイミングを調整し，プロセッサ間通信をプログラム全体の実行効率に影響させないような並列処理を実現する．

これらの方針に基づいて，本研究では計算機クラスタ環境における並列処理に適したプログラム並列化手法を提案した．

まず，前者のプロセッサ間通信をなるべく発生させない並列処理として，我々はプログラム中の関数単位で並列処理を実現するための関数呼出し文並列化手法を開発した．関数をタスク生成の単位とすることにより，変数空間をタスクに局所化し，タスク間の余分な通信処理を削減することができる．また，プログラム中の各関数呼出し文において callee/caller 関係にある 2 つの関数間の並列実行性を依存度として評価し，タスクを生成するアルゴリズムを開発した．このアルゴリズムにより，タスク起動時のオーバヘッドを削減することができ，効率よい並列処理を計算機クラスタ環境で実現することが可能となった．

次に，後者のプロセッサ間通信をプログラム全体の実行効率に影響させない並列処理として，計算と通信のオーバラップによりループ文を並列実行させるループ文漸進処理手法を開発した．ループ文漸進処理手法では，同一配列データを共有するループ文に対して，各イタレーションの実行と計算結果の通信をオーバラップさせることで，タスク間通信に起因するタスクの実行中断状態を回避し，ループ文を並列実行させる．イタレーションの同期タイミングを調整するために，2 つのループ文における配列データへの操作状況を解析し，イタレーション依存比を計算する．イタレーション依存比により，計算と通信のオーバラップのときイタレーションに実行をどのように調整すればよいかが解析可能となり，細粒度タスクによる並列処理に合致しない計算機クラスタ環境において効率よいループ文漸進処理が実現できる．また，ループ文漸進処理の適用のため，ループ文における配列データに対する操作状況をモデル化するアクセス・パターンを提案した．アクセス・パターンは多次元ループ文中で多次元配列データを操作する文における操作状況を解析するために提案され，配列操作文での配列データの操作領域はもちろん，ループ文の実行にともなう配列データの操作領域の移動をモデル化している．ネスト・ループ文においてアクセス・パターンを計算するための，アクセス・パターン間の畳込み演算を提案した．ネスト・ループ文において，それぞれのループ文に対するアクセス・パターンをまず計算し，それらを畳込み演算により統合することでネスト・ループ文におけるアクセス・パターンを計算可能とする．アクセス・パターンを定義することで，

任意のループ文に対する漸進処理の適用可否を判断することができる.

これらの並列化手法により，計算機クラスタ環境において効率よい並列処理を実現することができ，評価実験においてこれら提案手法の有効性，妥当性を評価した.

5.2 今後の課題

本研究における今後の課題について述べる.

5.2.1 関数呼出し文並列化手法

2章では大域変数やポインタ変数の扱いについては特に触れていなかった．これは，大域変数やポインタ変数の問題はタスク生成時ではなく，タスク実行時の実行順序制御機構で解決されるべき問題であるという考えからである [85]．また，基本的には大域的データフロー解析やエイリアス解析により，局所変数と同様に対処可能である [86–89]．しかし，ポインタ変数は実行時に決定される値などにより操作されることが多く，プログラムのコンパイル時では概略的な解析にとどまることが多い．ポインタ変数の値により関数間の並列実行可能性が変化する場合もあるので，大域変数，ポインタ型変数を考慮に入れたタスク生成手法の検討は今後の課題である．コンパイラにおいて実行時に決定される変数などに対する対処は非常に困難であるので，このような問題に対してプログラムのコンパイル時にどの程度まで対処するかは大きな課題となっている.

また，2章で提案したタスク生成アルゴリズムでは，タスク間で授受されるデータ量に関しては考慮していない．すなわち，タスク間通信が発生するか否かは依存度によってタスク生成時に評価されているが，タスク間通信時のデータ量については評価していない．これは，計算機クラスタ環境においては通信処理を開始するコストが非常に大きいので，通信オーバヘッドの評価には，タスク間通信が発生するか否かの方が，タスク間通信時に授受されるデータ量よりも支配的であるという理由からである．しかし，タスク間通信時に授受されるデータ量は，そのタスク間の

並列実行可能性を決定する大きな指標の一つである．したがって，タスク生成アルゴリズムにおいてタスク間通信時のデータ量を評価することも重要であり，今後の課題である．

関数呼出し文並列化アルゴリズムは，タスクの生成に関し実行時の動作環境については注意を払っていない．すなわち，対象とする計算機クラスタ環境内の計算機の台数や性能の差違を考慮して生成タスク数を変更することはできない．実際，計算機クラスタ環境ではそれぞれの計算機の特性[1] が異なっている場合が多く，計算機の台数や特性の違いを十分に生かした処理が大切である．さらに，実行時の環境の挙動には不確定性が含まれている．例えば，タスクが各計算機に均等に割り当てられずに負荷の不均一が生じたり，ネットワークの混雑に従い通信時間にばらつきが生じたりする．今後は，計算機クラスタ環境の特徴を生かした機能分散的な並列処理を可能とするタスク生成アルゴリズム，実行時環境の不確定性に対処する手法について研究を進めていく必要がある．

5.2.2 ループ文漸進処理手法

3 章の議論では，ループ文のみに注目し，最適な同期タイミングの計算方法について述べた．すなわち，実プログラムからループ文のみを取り出し，プログラム中にループ文のみが存在するとして議論した．しかし，実際のプログラムの並列実行においては，すべてのループ文に対して漸進処理を適用する必要はない．すなわち，漸進処理を適用することでプログラム全体の処理効率が向上するループ文のみに漸進処理を適用すればよい．したがって，プログラムから漸進処理の適用が必要なループ文を抽出しなければならない．我々は，プログラムを並列実行したとき，プログラム全体の実行時間を規定するタスクを抽出し，そのタスク間にループ文の漸進処理を適用するアルゴリズムを開発中である．漸進処理を適切なループ文に適用し，プログラム全体の処理効率を向上させる手法を開発することが今後の課題である．

[1] プロセッサの速度，メモリ容量，ディスクの有無など．

また，3章で対象としなかったループ文に対するアクセス・パターンの解析，漸進処理の適用も今後の課題である．実用性を考慮すると，開始点の違いにより解析できないループ文などに対処する必要がある．

5.2.3 ループ文のアクセス・パターン

4.2.3節で述べたように，アクセス・パターンの解析において重要な演算には，4章で提案した畳込み演算の他に，集約演算がある．集約演算は，同一ループ文内に複数存在する配列操作やループ文のアクセス・パターンを融合したアクセス・パターンを計算する演算である．4章では，簡単化のためパーフェクト・ループ文を対象としたが，並列化の対象となるプログラムではループ文中に複数のループ文が存在することが多い．集約演算を定義することで，パーフェクト・ループ文でないループ文のアクセス・パターンも解析することができ，より広範囲のプログラムに対して柔軟な解析が可能となる．したがって，アクセス・パターンに対して集約演算を定義することが今後の課題である．

また，我々のアクセス・パターンは配列添字式としてループ変数の一次多項式を対象としているが，LMAD は複雑な配列添字式を解析対象としている．複雑な配列添字式を対象とするためのアクセス・パターンの拡張も今後の課題として挙げられる．

付録 A

関数の実行時間推定処理

主な字句の重みを付表に示す．(,) や ; などの計算に関係しない字句の重みは 0 である．[] は配列参照の場合のアドレス計算を考慮して重みが与えられている．*, /, % は他の重み 1 の演算子との計算の複雑さを考慮して，重みを 2 としている．
[例 1]

```
if (a==1) {
    a++;
    c++;
} else {
    b++;
}
```

条件分岐確率は未知であり，0.5 とする．条件節の “a==1”， then 節の “a++; c++;”，else 節の “b++;” の実行ステップ数はそれぞれ 3, 10, 5 である．したがって，この if 文の推定実行ステップ数は

$$3 + 0.5 \times (10 + 5) = 10.5$$

となり，端数切上げで 11 となる．

付表 主な字句の重み

字句	重み	備考
(,), ;, auto, int, ···	0	
識別子	1	
+, -, [], ···	1	
*, /, %	2	
+=, -=, ···	3	a += 2 は a = a + 2 と同じ計算量のため.
++, --	4	a++ は a = a + 1 と同じ計算量のため.

[例 2]

```
for (i=0; i<100; i++) {
    a++;
}
```

繰返し回数は 100 である. 推定実行ステップ数は,

$$ET(\text{“i=0”}) + 100 \times (ET(\text{“i<100”}) + ET(\text{“i++”}) + ET(\text{“a++”}))$$
$$= 3 + 100 \times (3 + 5 + 5) = 1303$$

となる.

付録 B

サンプル・プログラム

2章の実験で用いた行列の乗算とノルムを求めるプログラムを以下に示す.

```
/*
 * matrix.c
 *      行列のかけ算，ノルムを計算するプログラム
 */
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

#define MAT             550
#define SMAT            11
#define MAT_A           "matrix_a.dat"
#define MAT_B           "matrix_b.dat"

#define CALC_NORM(X,Y)  \
    int i, j, k, l, pivot; \
    int piv[X], res[X]; \
    int result; \
    int count; \
    int mat[X-1][X-1]; \
\
    count = 0; \
\
    for (l = 0; l < X; l++) { \
        pivot = l; \
        k = 0; \
        for (i = 0; i < X; i++) { \
```

```
                if (i == pivot) \
                    continue; \
                for (j = 1; j < X; j++) \
                    mat[k][j] = rp[i][j]; \
            } \
            piv[count] = rp[pivot][0]; \
            res[count] = Y(mat); \
            count++; \
        } \
    \
        result = 0; \
        for (l = 0; l < X; l++) \
            if ( l%2 == 0) \
                result += piv[l]*res[l]; \
            else \
                result -= piv[l]*res[l]; \
    \
        return result

    /**/
    /**/

    void calc_matrix_mult(void);
    void calc_row(int *, int);
    int calc_mult(int, int);
    void calc_compress(int[SMAT][SMAT], int[MAT][MAT], int[MAT][MAT]);
    void load_data(int *, char *);

    int calc_norm15(int[15][15]);
    int calc_norm14(int[14][14]);
    int calc_norm13(int[13][13]);
    int calc_norm12(int[12][12]);
    int calc_norm11(int[11][11]);
    int calc_norm10(int[10][10]);
    int calc_norm9(int[9][9]);
    int calc_norm8(int[8][8]);
    int calc_norm7(int[7][7]);
    int calc_norm6(int[6][6]);
    int calc_norm5(int[5][5]);
    int calc_norm4(int[4][4]);
    int calc_norm3(int[3][3]);

    /**/
    /**/
```

```
int matA[MAT][MAT];
int matB[MAT][MAT];
int matC[MAT][MAT];
int matD[SMAT][SMAT];

/**/
/**/

main(int argc, char *argv[])
{
    int i;
    int result;
    time_t start, elasp;

    /* データの読込み */
    fprintf(stderr, "data read start\n");
    load_data(matA, MAT_A);
    load_data(matB, MAT_B);
    fprintf(stderr, "data read end\n");

    /* 行列のかけ算 */
    fprintf(stderr, "calculation mult start\n");
    start = time((time_t)0);
    calc_matrix_mult();
    elasp = time((time_t)0);
    fprintf(stderr, "calculation mult end\n");
    fprintf(stderr, "time = %d\n", (int)(elasp - start));

    /* 行列の圧縮 */
    fprintf(stderr, "calculation compress start\n");
    start = time((time_t)0);
    calc_compress(matD, matA, matB);
    elasp = time((time_t)0);
    fprintf(stderr, "calculation compress end\n");
    fprintf(stderr, "time = %d\n", (int)(elasp - start));

    /* ノルムの計算 */
    fprintf(stderr, "calculation norm start\n");
    start = time((time_t)0);
    result = calc_norm11(matD);
    elasp = time((time_t)0);
    fprintf(stderr, "calculation norm end\n");
    fprintf(stderr, "time = %d\n", (int)(elasp - start));
```

```
    }

    /**/
    /**/

120 void calc_matrix_mult(void)
    {
        int i;

        for (i = 0; i < MAT; i+=50)
            calc_row(&matC[i], i);
    }

    void calc_row(int *rp, int pos)
    {
130     int i, j;

        for (i = pos; i < pos + 50; i++)
            for (j = 0; j < MAT; j++)
                *rp++ = calc_mult(i, j);
    }

    int calc_mult(int row, int column)
    {
        int i;
140     int result;

        result = 0;
        for (i = 0; i < MAT; i++)
            result += (matA[row][i] * matB[i][column]);

        return result;
    }

    /**/
150 /**/

    void calc_compress(int r[SMAT][SMAT], int a[MAT][MAT], int b[MAT][MAT])
    {
        int i, j, i0, j0;
        int count;
        int limit;
        int ir;
```

```
        count = 0;
160     ir = 0;
        limit = MAT / SMAT;


        for (i0 = 0; i0 < SMAT; i0++) {
            for (j0 = 0; j0 < SMAT; j0++) {
                ir = 0;
                count = 0;
                for (i = 0; i < limit; i++) {
                    for (j = 0; j < limit; j++) {
170                     ir += a[i0 * SMAT + i][j0 * SMAT + j]
                            + b[i0 * SMAT + i][j0 * SMAT + j];
                        count++;
                    }
                }
                r[i0][j0] = ir / count;
            }
        }
    }

180 /**/
    /**/

    int calc_norm15(int rp[15][15])
    {
    CALC_NORM(15,calc_norm14);
    }

    int calc_norm14(int rp[14][14])
    {
190 CALC_NORM(14,calc_norm13);
    }

    int calc_norm13(int rp[13][13])
    {
    CALC_NORM(13,calc_norm12);
    }

    int calc_norm12(int rp[12][12])
    {
200 CALC_NORM(12,calc_norm11);
    }
```

```
int calc_norm11(int rp[11][11])
{
CALC_NORM(11,calc_norm10);
}

int calc_norm10(int rp[10][10])
{
CALC_NORM(10,calc_norm9);
}

int calc_norm9(int rp[9][9])
{
CALC_NORM(9,calc_norm8);
}

int calc_norm8(int rp[8][8])
{
CALC_NORM(8,calc_norm7);
}

int calc_norm7(int rp[7][7])
{
CALC_NORM(7,calc_norm6);
}

int calc_norm6(int rp[6][6])
{
CALC_NORM(6,calc_norm5);
}

int calc_norm5(int rp[5][5])
{
CALC_NORM(5,calc_norm4);
}

int calc_norm4(int rp[4][4])
{
CALC_NORM(4,calc_norm3);
}

/**/
/**/

int calc_norm3(int r[3][3])
```

```
{
    int result;

    result = 0;
    result += r[0][0] * r[1][1] * r[2][2];
    result += r[0][2] * r[1][0] * r[2][1];
    result += r[0][1] * r[1][2] * r[2][0];
    result -= r[0][2] * r[1][1] * r[2][0];
    result -= r[0][0] * r[1][2] * r[2][1];
    result -= r[0][1] * r[1][0] * r[2][2];

    return result;
}

/**/
/**/

void load_data(int *datap, char *fname)
{
    FILE *fp;
    int i, j;

    fp = fopen(fname, "r");

    for (i = 0; i < MAT; i++)
        for (j = 0; j < MAT; j++) {
            *datap = (int) fgetc(fp);
            datap++;
        }
    fclose(fp);
}
```

謝辞

本研究は，名古屋大学大学院工学研究科情報工学専攻 渡邊豊英教授の御指導の元で行われました．著者が名古屋大学工学部情報工学科の 4 年生となり，渡邊先生の研究室に配属されて以来，名古屋大学大学院工学研究科情報工学専攻博士課程前期課程，同後期課程，同専攻助手の今日までの長い間，終始懇切丁寧な御指導と多大なる御配慮を賜りました．渡邊先生に対し，ここに心から感謝の意を表し，深く御礼申し上げます．

同時に，本論文をまとめるにあたって，御多忙の中，多くの有益かつ適切な御教示と注意深い検討を頂きました名古屋大学大学院工学研究科情報工学専攻 阿草清滋教授，同専攻 坂部俊樹教授に厚く御礼申し上げます．

また，学生時代から今日まで，日々の研究室での生活において，いろいろ御配慮頂いた杉野花津江先生に感謝いたします．

本研究に関して熱心な御意見，御討議頂いた渡邊研究室の皆様に感謝いたします．特に，貴重な御意見を頂いた名古屋大学大学院工学研究科情報工学専攻 加藤ジェーン助教授，現在名城大学理工学部情報科学科 佐川雄二講師，現在九州工業大学情報工学部機械システム工学科 高田修助教授，現在九州芸術工科大学芸術工学部芸術情報設計学科 牛尼剛聡助手に厚く御礼申し上げます．

本研究の一部は，渡邊研究室の DP 研究グループの PD 班の方々と実施しました．特に，ループ文漸進処理手法に関する研究は，現在株式会社豊田中央研究所システム・エレクトロニクス分野第 21 研究領域 三田勝史君と共に取り組みました．こ

こに感謝の意を表します.

最後に，暖かく見守ってくれ心の支えとなってくれた両親と家族に感謝します.

参考文献

[1] 横川三津夫: “地球シミュレータの完成を迎えて”, 並列処理シンポジウム JSPP2002, pp. 329–332 (2002).

[2] 高橋 義造 (編): “並列処理機構”, 丸善株式会社, P. 272 (1989).

[3] 渡辺勝正: “並列処理概説”, 並列処理シリーズ, 第 1 巻, コロナ社, P. 205 (1991).

[4] 奥川峻史: “並列計算機アーキテクチャ”, 並列処理シリーズ, 第 2 巻, コロナ社, P. 175 (1991).

[5] 鈴木則久, 清水茂則, 山内長承: “共有記憶型並列システムの実際”, 並列処理シリーズ, 第 16 巻, コロナ社, P. 208 (1993).

[6] 石畑宏明, 稲野聡, 堀江健志, 清水俊幸, 池坂守夫: “高並列計算機 AP1000 のアーキテクチャ”, 電子情報通信学会論文誌 D-I, Vol. J75-D-I, No. 8, pp. 637–645 (1992).

[7] 林憲一, 土肥実久, 堀江健志, 小柳洋一, 白木長武: “AP1000+: 並列化コンパイラをサポートするアーキテクチャ”, *Proc. of SWoPP'94* (1994).

[8] 今村伸貴, 藤崎直哉, 石畑宏明, 池坂守夫: “AP1000+ における Microkernel IPC の実装と評価”, 情報処理学会 OS 研究会, Vol. 95-OS-70, No. 9, pp. 65–72 (1995).

[9] 笠原博徳, 成田誠之助, 橋本親: "OSCAR (Optimally Scheduling Advanced Multiprocessor) のアーキテクチャ", 電子情報通信学会論文誌, Vol. J71-D, No. 8, pp. 1440–1445 (1988).

[10] P. Steenkiste: "Network-Based Multicomputers: A Practical Supercomputer Architecture", *IEEE Trans. on Parallel and Distributed Systems*, Vol. PDS-7, No. 8, pp. 861–875 (1996).

[11] T. Sterling, D. Becker, D. Savarese, J. Dorband, U. Ranawake and C. Packer: "Beowulf: A Parallel Workstation for Scientific Computation", *Proc. of Int'l Conf. on Parallel Processing*, Vol. 1, pp. 11–14 (1995).

[12] C. Reschke, T. Sterling, D. Ridge, D. Savarese, D. Becker and P. Merkey: "A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstation", *Proc. of the Fifth IEEE Int'l Symp. on High Performance Distributed Computing* (1996).

[13] D.E. Culler, A.A. Dusseau, R.A. Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa and F. Wong: "Parallel Computing on the Berkeley NOW", *Proc. of JSPP'97* (1997).

[14] 笠原博徳: "並列処理技術", コロナ社, P. 198 (1991).

[15] S. Brawer, 大森健児 (訳): "並列プログラミングの基礎", 丸善, P. 376 (1990).

[16] A. Silberschatz and P.B. Galvin: "*Operating System Concepts 5th Ed.*", Addison Wesley, P. 888 (1998).

[17] 丸山勝己: "並列オブジェクト指向言語 COOL", 情報処理学会論文誌, Vol. 34, No. 5, pp. 963–972 (1993).

[18] 湯浅太一, 貴島寿郎, 小西浩: "データ並列言語のための拡張 C 言語 NCX", 電気情報通信学会論文誌 D-I, Vol. J78-D-I, No. 2, pp. 200–209 (1995).

[19] P. J. Hatcher, A. J.Lapadula, R. R. Jones, M. J. Quinn and R. J. Anderson: "A Production-Quality C* Compiler for Hypercube Multicomputers", *Proc. of the 3rd ACM SIGPLAN Symp. on Principle & Practice of Parallel Programming*, pp. 73–82 (1991).

[20] N. Carriero and D. Gelernter: "Linda in Context", *Communications of the ACM*, Vol. 32, No. 4, pp. 444–458 (1989).

[21] R. Cohen: "Optimising Linda Implementations for Distributed Memory Multicomputers", *Proc. of the 1st Annual Users' Meeting of Fujitsu Parallel Computing Research Facilities*, pp. ANU–5 (1992).

[22] 進藤達也, 岩下英俊, 土肥実久, 萩原純一, 金城ショーン: "FLoPS: 分散メモリ型並列計算機を対象とした並列化コンパイラ", 情報処理学会論文誌, Vol. 37, No. 11, pp. 2030–2038 (1996).

[23] 村岡洋一: "超並列処理コンパイラ", コロナ社, P. 176 (1990).

[24] D. E. Maydan, J. L. Hennessy and M. S. Lam: "Efficient and Exact Data Dependence Analysis", *Proc. of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, pp. 1–14 (1991).

[25] G. Goff, K. Kennedy and C.-W. Tseng: "Practical Dependence Testing", *Proc. of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, pp. 15–29 (1991).

[26] J. Ferrante, K. J. Ottenstein and J. D. Warren: "The Program Dependence Graph and Its Use in Optimization", *ACM Trans. on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319–349 (1987).

[27] 山名早人, 安江俊明, 石井吉彦, 村岡洋一: "並列処理システムにおけるマクロタスク間先行評価方式", 電子情報通信学会論文誌 D-I, Vol. J77-D-I, No. 5, pp. 343–353 (1994).

[28] M. Girkar and C.D. Polychronopoulos: "Extracting Task-Level Parallelism", *ACM Trans. on Programming Languages and Systems*, Vol. 17, No. 4, pp. 600–634 (1995).

[29] M.W. Hall, B.R. Murphy, S.P. Amarasinghe, S.-W. Liao and M.S. Lam: "Interprocedural Analysis for Parallelization", *Proc. of the 8th Int'l Workshop on Languages and Compilers for Parallel Computing*, pp. 61–80 (1995).

[30] U. Banerjee: "*Loop Transformations for Restructuring Compilers -The Foundations-*", Kluwer Academic Publishers, P. 305 (1993).

[31] 中田育男: "コンパイラの構成と最適化", 朝倉書店 (1999).

[32] D. A. Padua and M. J. Wolfe: "Advanced Compiler Optimizations for Supercomputers", *Communications of the ACM*, Vol. 29, No. 12, pp. 1184–1201 (1986).

[33] S. P. Midkiff and D. A. Padua: "Compiler Generated Synchronization for Do Loops", *Proc. of 1986 Int'l Conf. on Parallel Processing*, pp. 544–551 (1986).

[34] R. Cytron: "Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract)", *Proc. of 1986 Int'l Conf. on Parallel Processing*, pp. 836–844 (1986).

[35] 高畠志泰, 本多弘樹, 大澤範高, 弓場敏嗣: "Doacross ループの sandglass 型並列化方法とその評価", 情報処理学会論文誌, Vol. 40, No. 5, pp. 2037–2044 (1999).

[36] 丸島敏一, 村岡洋一: "ループ並列処理方式 doalong", 電子情報通信学会論文誌, Vol. J71-D, No. 8, pp. 1511–1517 (1988).

[37] M. J. Wolfe: "Advanced Loop Interchanging", *Proc. of 1986 Int'l Conf. on Parallel Processing*, pp. 536–543 (1986).

[38] C.-M. Wang and S.-D. Wang: "Efficient Processor Assignment Algorithms and Loop Transformations for Executing Nested Parallel Loops on Multiprocessors", *IEEE Trans. on Parallel and Distributed Systems*, Vol. PDS-3, No. 1, pp. 71–82 (1992).

[39] 石崎一明, 小松秀昭: "分散メモリ並列計算機のためのコンパイラによる通信遅延隠蔽アルゴリズム", 情報処理学会論文誌, Vol. 38, No. 9, pp. 1849–1858 (1997).

[40] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung and D. Schouten: "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors", *Proc. of 1989 Int'l Conf. on Parallel Processing*, Vol. 2, pp. 39–48 (1989).

[41] S. Ramaswamy, S. Sapatnekar and P. Banerjee: "A Framework for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers", Technical Report CRHC-94-10, Center for Reliable and High Performance Computing, University of Illinois at Urbana-Champaign (1994).

[42] S. Pande, D.P. Agrawal and J. Mauney: "A Scalable Scheduling Scheme for Functional Paralleism on Distributed Memory Multiprocessor Systems", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6, No. 4, pp. 388–399 (1995).

[43] 本多弘樹, 水野聡, 笠原博徳, 成田誠之助: "OSCAR 上での Fortran プログラム基本ブロックの並列化手法", 電子情報通信学会論文誌 D-I, Vol. J73-D-I, No. 9, pp. 756–766 (1990).

[44] 本多弘樹, 岩田雅彦, 笠原博徳: "Fortran プログラム粗粒度タスク間の並列性検出手法", 電子情報通信学会論文誌 D-I, Vol. J73-D-I, No. 12, pp. 951–960 (1990).

[45] 笠原博徳, 合田憲人, 吉田明正, 岡本雅巳, 本多弘樹: "Fortran マクロデータフロー処理のマクロタスク生成手法", 電子情報通信学会論文誌 D-I, Vol. J75-D-I, No. 8, pp. 511–525 (1992).

[46] 岡本雅巳, 合田憲人, 宮沢稔, 本多弘樹, 笠原博徳: "OSCAR マルチグレインコンパイラにおける階層型マクロデータフロー処理手法", 情報処理学会論文誌, Vol. 35, No. 4, pp. 513–521 (1994).

[47] 山名早人, 佐藤三久, 児玉祐悦, 坂根広史, 坂井修一, 山口喜教: "並列計算機 EM-4 におけるマクロタスク間投機的実行の分散制御方式", 情報処理学会論文誌, Vol. 36, No. 7, pp. 1578–1588 (1995).

[48] 吉田明正, 越塚健一, 岡本雅巳, 笠原博徳: "階層型粗粒度並列処理における同一階層内ループ間データローカライゼーション手法", 情報処理学会論文誌, Vol. 40, No. 5, pp. 2054–2063 (1999).

[49] 酒井淳嗣, 鳥居淳, 近藤真己, 市川成浩, 小俣仁美, 西直樹, 枝廣正人: "制御並列アーキテクチャ向け自動並列化コンパイル手法", 情報処理学会論文誌, Vol. 40, No. 5, pp. 2045–2053 (1999).

[50] A. Geist, A. Befuelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam: "PVM3 User's Guide and Reference Manual", Technical Report ORTL/TM-12187, Oak Ridge National Laboratory (1993).

[51] W. Gropp, E. Lusk and A. Skjellum: "*Using MPI: Portable Parallel Programming with the Message-Passing Interface*", The MIT Press, P. 371 (1999).

[52] W. Gropp, E. Lusk and R. Thakur: "*Using MPI-2: Advanced Features of the Message-Passing Interface*", The MIT Press, P. 382 (1999).

[53] 朝倉宏一, 渡邉豊英: "AP1000 を用いた学部学生に対する並列処理教育", 並列処理シンポジウム JSPP'98, p. 149 (1998).

[54] 朝倉宏一, 渡邉豊英: "情報工学系学部学生に対する並列プログラミング演習教育", 情報処理学会論文誌, Vol. 40, No. 5, pp. 2235–2245 (1999).

[55] J.E. Moreira, D. Schouten and C. Polychronopoulos: "The Performance Impact of Granularity Control and Functional Parallelism", *Proc. of the 8th Int'l Workshop on Languages and Compilers for Parallel Processing*, pp. 581–597 (1995).

[56] 日高康雄, 小池汎平, 田中英彦: "バンバン粒度制御: 高並列汎用処理における最適粒度制御", 情報処理学会論文誌, Vol. 7, No. 1344-1354 (1996).

[57] 朝倉宏一, 渡邉豊英: "ワークステーション・クラスタ環境における並列化コンパイラの構成", 電気学会論文誌 C, Vol. 118-C, No. 4, pp. 558–568 (1998).

[58] 三田勝史, 朝倉宏一, 渡辺豊英: "配列データを共有したループ文の並列実行のための漸進処理手法", 情報処理学会論文誌, Vol. 42, No. 4, pp. 847–859 (2001).

[59] Y. Paek, J. Hoeflinger and D. Padua: "Access Regions: Toward a Powerful Parallelizing Compiler", Technical Report 1508, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev. (1996).

[60] Y. Paek, J. Hoeflinger and D. Padua: "Simplification of Array Access Patterns for Compiler Optimizations", *Proc. of SIGPLAN Conference on Programming Language Design and Implementation*, pp. 60–71 (1998).

[61] 朝倉宏一, 渡辺豊英: "ループ文アクセス・パターン解析における畳込み演算", 並列処理シンポジウム JSPP2002, pp. 253–260 (2002).

[62] 朝倉宏一, 渡辺豊英: "ループ文アクセス・パターン解析における畳込み演算", 情報処理学会論文誌: ハイパフォーマンスコンピューティングシステム, Vol. 43, No. SIG6(HPS5), pp. 77–87 (2002).

[63] K. Castagnera, D. Cheng, R. Fatoohi, E. Hook, B. Kramer, C. Manning, J. Musch, C. Niggley, W. Saphir, D. Sheppard, M. Smith, I. Stockdale, S. Welch, E. Williams and D. Yip: "NAS Experiences with a Prototype Cluster of Workstations", *Proc. of Supercomputing '94*, pp. 410–419 (1994).

[64] G. Burns, R. Daoud and J. Vaigl: "LAM: An Open Cluster Environment for MPI", Technical report, Ohio Supercomputer Center (1994).

[65] J.C. Jacob and S.-Y. Lee: "Task Spreading and Shtinkinh on a Network of Workstation with Various Edge Classes", *Proc. of ICPP'96*, Vol. 3, pp. 174–181 (1996).

[66] 小林真也, 小川智之, 渡辺尚: "マルチコンピュータシステムにおける自律的負荷分散方式", 電子情報通信学会論文誌 D-I, Vol. J79-D-I, No. 11, pp. 903–915 (1996).

[67] K. Asakura, T. Watanabe and N. Sugie: "C parallelizing Compiler on Local-network-based Computer Environment", *Proc. of the 7th Int'l Parallel Processing Symp.*, pp. 849–853 (1993).

[68] M. Girkar and C. D. Polychronopoulos: "Automatic Extraction of Functional Parallelism from Ordinary Programs", *IEEE Trans. on Parallel and Distributed Systems*, Vol. PDS-3, No. 2, pp. 166–178 (1992).

[69] H. Kasahara, H. Honda, M. Iwata and M. Hirota: "A Compilation Scheme for Macro-dataflow Computation on Hierarchical Multiprocessor System", *Proc. of 1990 Int'l Conf. on Parallel Processing*, Vol. 2, pp. 294–295 (1990).

[70] 福田晃: "並列オペレーティング・システム", 情報処理学会学会誌, Vol. 34, No. 9, pp. 1139–1149 (1993).

[71] 福田晃: "並列オペレーティングシステム", 並列処理シリーズ, 第7巻, コロナ社, P. 200 (1997).

[72] 乾和志, 菅原圭資: "分散 OS Mach がわかる本", 日刊工業新聞社, P. 223 (1992).

[73] A.V. エイホ, J.D. ウルマン, (土居範久訳): "コンパイラ", 培風館, P. 543 (1986).

[74] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh and A. Gupta: "The SPLASH-2 Programs: Characterization and Methodological Considerations", *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture*, pp. 24–36 (1995).

[75] M. Wolfe: "*High Performance Compilers for Parallel Computing*", Addison-Wesley Publishing Company, P. 570 (1991).

[76] W. Pugh: "A Practical Algorithm for Exact Array Dependence Analysis", *Communications of the ACM*, Vol. 35, No. 8, pp. 102–114 (1992).

[77] J. Hoeflinger, Y. Paek and D. Padua: "Region-based Parallelization Using the Region Test", Technical Report 1514, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev. (1996).

[78] 三田勝史, 朝倉宏一, 渡邉豊英: "配列データ分割による漸進的並列処理手法の適用", 並列処理シンポジウム JSPP'98, p. 147 (1998).

[79] K. Sanda, K. Asakura and T. Watanabe: "Access Patterns Analysis between Intertask-dependent Arrays", *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, Vol. VI, pp. 2746–2752 (1999).

[80] H. Zima and B. Chapman: "*Supercompilers for Parallel and Vector Computers*", Addison-Wesley Publishing Company (1991).

[81] 朝倉宏一, 渡辺豊英: "ループ文アクセス・パターン表現による配列操作モデル", 2001 年度 電気関連学会東海支部連合大会, p. 294 (2001).

[82] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo and M. Yarrow: "The NAS Parallel Benchmarks 2.0", Technical Report NAS-95-020, NASA Ames Research Center (1995).

[83] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S-W. Liao, C. Tseng, M.W. Hall, M.S Lam and J.L. Hennessy: "SUIF: An Infrastructure for Research on Paralllelizing and Optimizing Compilers", *SIGPLAN Notices*, Vol. 29, No. 12, pp. 31–37 (1994).

[84] 朝倉宏一, 渡辺豊英: "並列化コンパイラ・ツールキットにおけるプログラム並列化処理とタスク生成処理について", 情報処理学会 第 62 回全国大会論文誌, Vol. 1, pp. 169–170 (2001).

[85] K. Asakura, T. Watanabe and N. Sugie: "An Execution Order Control Method of Distributed Processes for Sharing Global Variables", *Proc. of 1994 IEEE TENCON*, Vol. 1, pp. 156–160 (1994).

[86] V. A. Guarna Jr.: "A Technique for Analyzing Pointer and Structure References in Parallel Restructuring Compilers", *Proc. of 1988 Int'l Conf. on Parallel Processing*, Vol. 2, pp. 212–220 (1988).

[87] L.-C. Lu and M. Chen: "Parallelizing Loops with Indirect Array References or Pointers", *Proc. of 4th Int'l Workshop on Languages and Compilers for Parallel Computing*, pp. 201–217 (1991).

[88] R.P. Wilson and M.S. Lam: "Efficient Context-Sensitive Pointer Analysis for C Programs", *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Inplementation*, pp. 1–12 (1995).

[89] D.-S. Han and T. Tsuda: "Non-Graph Based Aproach on the Analysis of Pointers and Structures", *IEICE Trans. on Information and Systems*, Vol. E80-D, No. 4, pp. 480–488 (1997).