ハイパフォーマンス
コンピューティング 78－6
（1999. 10. 7）

計算機クラスタ環境における並列処理のための プログラム並列化手法とその評価

朝倉 宏一， 渡邉 豊英
名古屋大学大学院 工学研究科 情報工学専攻
{asakura, watanabe}@nuie.nagoya-u.ac.jp

概要

本稿では，計算機クラスタ環境において効率よく並列処理を行うためのプログラム並列化の手法について述べる．ワークステーションや高性能 PC がイーサネットで接続されている計算機クラスタ環境は，通常の並列処理環境と比較すると，タスクの起動オーバヘッドやタスク間通信オーバヘッドが大きい．我々は，この計算機クラスタ環境の特性を考慮し，プログラム中の関数手続きを単位としたタスク生成手法と，タスクを漸進的に部分実行させることで並列実行の効率を向上させるタスク実行制御手法を開発した．NPB (NAS Parallel Benchmark) を用いた評価実験により，我々の手法は適切な粒度のタスクを生成すること，漸進処理の適用により 15 ～ 40% の処理効率の向上が得られること，などが分かった．

Evaluation of Program Parallelizing Method for Computer Cluster Environment

Koichi ASAKURA and Toyohide WATANABE
Department of Information Engineering,
Graduate School of Engineering, Nagoya University
{asakura, watanabe}@nuie.nagoya-u.ac.jp

Abstract

In this paper, we describe the program parallelizing method for effective parallel processing on the computer cluster environment. This environment has disadvantages in the overhead of task invocation and inter-task communication, since it has no special facilities for parallel processing in comparison with traditional parallel processing environments. In order to make parallel processing in the computer cluster environments successful, we developed the procedure-based task generation method and the task control method by which tasks are executed partially and incrementally. From the experimental results using NPB (NAS Parallel Benchmark), we can conclude that our task generation method generates coarse grain tasks suitable to the computer cluster environments, and that incremental processing improves parallel executability up to 15 ～ 40%.

1 はじめに

近年，対価格性能，可用性，拡張性に優れた計算機クラスタ環境が実用に供されており，並列処理プラットホームとしても注目を浴びている．本稿で対象とする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ネットワークで接続した環境で，特別なハードウェアを装備していないものを指す．すなわち，通常は 1 台 1 台独立な計算機として利用されることを想定した環境であ

り，AP-Net や Myrinet など並列処理のための超高速ネットワークを装備しない環境を対象としている．我々は，このような計算機クラスタ環境において並列処理を行うための，ソフトウェア環境を整備することを研究の目標としている．

本稿では，計算機クラスタ環境で並列処理を手軽に達成するための並列化コンパイラの開発における，プログラム並列化の手法について述べる．本並列化手法は，関数手続きレベルの粒度のタスクを生成するタスク生成手法と，タスク内の並列性を抽出する漸進処理に基づいたタスク実行制御手法から成っている．この手法により，計算機クラスタ環境でのタスクの起動オーバヘッドやタスク間の通信オーバヘッドに対処可能な並列処理が可能となる．また，本並列化手法の有効性を評価するために行った，ベンチマーク・プログラムに対するシミュレーション実験の結果についても述べる．本タスク生成手法により適切な粒度のタスクが生成され，漸進処理によりタスク間の並列性が抽出され，効率よい並列処理が可能になることが分かった．

本稿は以下のような構成になっている．まず，2章では，関連研究として従来のプログラム並列化手法に言及し，従来の手法が計算機クラスタ環境に適合しないことを述べる．そして，我々の並列化手法の概略について述べる．3章では，プログラムからタスクを生成する手法について簡単に述べる．4章では，我々が提案する漸進処理について説明し，漸進処理を行うためにタスクから漸進処理単位を抽出する方法について述べる．5章では，本手法の有効性を確認するために行った実験の結果について述べる．6章でまとめと今後の課題について述べる．

2 プログラム並列化の戦略

2.1 関連研究

プログラムの自動並列化の試みは数多くある．もっとも適用例が多いのは，プログラム中のループ文 (FORTRAN の DO 文，C 言語の for 文など) に対する並列化である [1–3]．ループ文内のイタレーション間でのデータ依存関係を解析し，各イタレーションを並列実行するためのコードを生成する．並列化の対象となるプログラムが数値計算を主とするものであったため，プログラム中のループ文での計算を高速化することにより，プログラム全体の処理効率の向上を計ることができた．

また，ループ文以外の部分の並列性を抽出するため，基本ブロックを並列化の単位としたプログラム並列化手法も開発されている．この手法では，マクロタスクグラフ [4,5] や HTG (Hierarchical Task Graph)[6] などを用いてプログラムを解析し，基本ブロック，ループ文，関数手続きなど，プログラム中の様々なレベルでの並列性抽出を目的とした．これにより，様々な粒度のタスクによる並列処理を可能とした．

このように，数多くの並列化手法が提案されているが，これらは並列計算機上での並列処理を目的として開発された．すなわち，これらの手法は高速なネットワーク，ランタイム・ルーチンなどを装備し，並列計算のために設計された並列計算機上での並列処理を想定して開発され，我々が想定する計算機クラスタ環境に対しては，そのままでは適合しないという問題点がある．

我々が対象とする計算機クラスタ環境は，並列処理のための特別な装備を持たないワークステーションや高性能 PC をイーサネットで接続した，いわゆる分散環境である．また，各計算機では OS として一般的な UNIX が動作している．したがって，本計算機クラスタ環境で並列処理を行うとき，計算機間の通信オーバヘッドと各計算機におけるタスク起動時の OS オーバヘッドに対して特に注意を払わなければならない．計算機クラスタ環境の特性を考慮して効率のよい並列処理を達成するためには，

1. タスク起動時の OS オーバヘッドを考慮したタスク粒度，
2. 通信オーバヘッドを考慮したタスク間通信の頻度，

が並列化手法開発の重要なポイントとなる．

2.2 計算機クラスタ環境での並列処理のための並列化手法

従来の並列化手法は基本ブロックレベルの粒度のタスクで並列処理が可能な環境を前提としている．したがって，我々の対象とする計算機環境ではタスク起動時のオーバヘッドが大きいので，基本ブロックレベルのタスクは粒度が細かすぎ，関数手続きレベルのタスクが必要となる．もちろん，マクロタスクグラフや HTG を用いた手法でも関数手続きレベルのタスクが生成可能であるが，基本ブロックレベルのタスクも依然として生成されてしまう．つまり，従来の並列化手法は基本的には基本ブロックレベルのタスクによる並列処理を原則としており，関数手続きレベルのみのタスク生成に対して積極的ではない．したがって，計算機クラスタ環境での並列処理のためには，基本ブロック，ループ文レベルの粒度のタスクが混在せず，関数手続きレベルの粒度を持つタスクのみを生成する並列化手法が必要となる．関数手続きレベルのタスクにより，OS の起動オーバヘッドが削除され，効率のよい並列処理が期待できる．

タスクとして関数手続きレベルの粒度を採用する場合，タスクの実行制御手法についても注意を払う必要がある．従来のタスク実行制御では，タスク

の実行終了を機に新しいタスクが起動される．つまり，あるタスクの実行が終了したとき，各タスクに付加された最早実行条件[7]を評価することにより，まだ実行されていないタスクの中から実行可能なタスクを選択し，新しいタスクとして起動する．基本ブロックレベルのタスクの場合，タスクの粒度が細かいため，タスクの実行終了と最早実行条件の評価のタイミングがほぼ一致していた．すなわち，最早実行条件の再評価が必要となるのはデータ依存・制御依存関係にある文の実行が終了したときであるが，基本ブロックを単位としたタスクの場合，最早実行条件の再評価のタイミングとタスクの実行終了のタイミングがほぼ等しい．しかし，関数手続きレベルのタスクの場合，タスクの粒度が粗いのでタスクの実行終了時での再評価では並列性を十分抽出することができない，という問題が発生する．

そこで我々は，タスク内の文のデータフロー・制御フローを解析し，タスクの実行終了時でなくタスクの実行中に他タスクを起動する実行制御手法を採用する．タスクの粒度は粗くしたまま，最早実行条件の評価のタイミングは細かくすることにより，関数手続きレベルのタスクにより効率よい並列処理を達成する．

さらに我々は，タスクの一部を漸進的に部分実行させることにより，より効果的な並列実行を達成するタスク実行制御手法を開発した．関数手続きレベルのタスクの場合，タスクの実行にはその関数手続きのパラメータ変数と使用する大域変数がすべて準備されることが必要である．最早実行条件にはそれらすべての変数に関するデータ依存関係が含まれている．しかし，タスク内の文で計算にすべての変数を必要とするものは少ない．一般的には，一部の変数が準備されれば計算可能な文が存在する．そのような文は，実際には実行可能であってもすべての変数が準備されるまで実行が遅延されてしまう．この実行の遅延を避けるために，我々の漸進処理に基づいたタスク実行制御手法ではタスクを漸進的に部分実行させる．タスク内の文間のデータ依存関係より，タスク内のデータフローを解析する．この解析により，一部の変数が準備されたときに実行可能となる文を抽出することができる．タスクの各パラメータ変数や大域変数に対して実行可能な文を抽出し，それぞれを漸進処理単位として再構成する．そして，実行中のデータの準備状況に従って漸進処理単位を実行させることで，タスクの漸進処理が達成され，実行遅延を減少させることができる．

まとめると，我々のプログラム並列化手法は以下のようになる．

1. タスクの生成を関数手続きレベルで行い，粗粒度タスクを生成する．
2. タスクの最早実行条件をタスクの実行中に評価することにより，並列性を抽出する．
3. タスク内のデータフロー解析により，パラメータ変数や大域変数の準備状況に従って実行可能な漸進処理単位を生成し，タスク内でデータ駆動的に実行させる．

1. により，計算機クラスタ環境におけるタスク起動のオーバヘッドに対処可能な粒度のタスクを生成する．また，2. により，関数手続き単位のタスクでは十分な並列性が抽出できない，という状況を回避する．さらに，3. により，タスクの実行遅延状態を減少させ，漸進処理によるタスクの部分実行により，さらなる並列性の抽出を可能とする．

3 タスク生成手法[8]

タスクはプログラム中の関数手続き単位で生成される．関数手続き単位でタスクを生成することで，粒度の粗いタスクを生成できる．また，各タスクの変数空間が独立となるため，タスク間の通信頻度も低くすることができる．しかし，計算機クラスタ環境で適切に動作するタスクを生成するためには，すべての関数手続きをタスクとして生成するのではなく，並列実行により処理効率が向上すると期待される関数手続きをタスクとして生成しなければならない．そのためには，関数手続き間の並列実行性を評価することが必要となる．

我々のタスク生成手法の概要を以下に示す．

1. プログラム中の関数手続きの呼出し関係を解析し，関数呼出しグラフを生成する．関数呼出しグラフは，プログラム内の各関数手続きを節で，関数手続き呼出し文を有向辺で表したグラフである．
2. 関数呼出しグラフの中の各有向辺において，両端の関数手続き間の並列実行性を依存度として解析する．依存度は，二つの関数手続きをタスクとして実行したとき，どの程度並列実行できるかを表す評価尺度である．
3. 依存度により，それぞれの関数手続きをタスクとして生成するか否かを決定する．

図1に本手法の摸式図を示す．依存度が低い，すなわち並列実行性が高いと判断された関数手続き呼出し文の有向辺を切断することにより，タスクが生成される．依存度による並列実行性抽出により，計算機クラスタで効率よく実行可能なタスクが生成される．

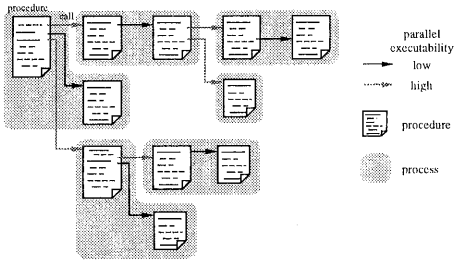
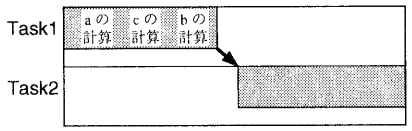
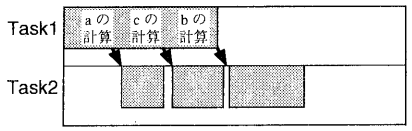


図 1: 関数手続きに基づいたタスク生成



(a) 従来の実行制御



(b) 漸進処理による実行制御

図 2: 漸進処理の摸式図

4 漸進処理に基づいたタスク実行制御手法 [9]

関数手続きを単位とした粗粒度タスクにより，タスク起動時のオーバヘッドが削減され，計算機クラスタ環境でも適切に並列実行が可能となる．しかし，このままではプログラム中の並列性を十分抽出していない．

タスクが起動可能な時点は最早実行条件により与えられる．タスクの実行経過に従って，最早実行条件を評価し，条件を満たすタスクを起動することで，計算結果に矛盾がなく並列実行が達成される．しかし，タスクの粒度が粗くなるにつれ，タスク間でのデータ依存・制御依存関係が複雑になり，最早実行条件も複雑化する．タスクの最早実行条件が成立し，計算に必要なすべての変数が使用可能となってからタスクが起動されるので，タスクの起動のタイミングが遅くなってしまう．これにより，タスクの実行遅延が発生し，プログラム中の並列性が抽出できず，効率よい並列処理が達成できない．

我々は，この実行遅延を減少させ効果的な並列処理を達成するために漸進処理を導入する．図 2に漸進処理を適用したときのタスクの実行過程を示す．Task2 は Task1 内で計算される変数 a, b, c を使用するので，Task1 と Task2 はデータ依存関係にある．図 2(a) に示すように，従来までの実行制御ではすべての変数が使用可能な状態になることで最早実行条件が成立し，Task2 が起動される．したがって，すべての変数が準備されるまで Task2 の実行は延期される．図 2(b) が漸進処理を適用した場合を示している．変数がいずれか一つでも使用可能となれば実行可能となる文を Task2 内より抽出する．そして，Task1 での変数の準備状態に従って，それらの文を漸進的に実行させる．この実行制御により，Task2 の部分実行が達成され，実行遅延が減少され，効率よい並列実行が可能となる．

この漸進処理を行うためには，タスク内で部分的に実行可能な漸進処理単位を抽出しなければならない．漸進処理単位は，タスク内の計算で必要となる変数に対してデータ駆動的に実行される．つまり，ある変数が使用可能となったとき，その変数を使用する文が実行可能となる．したがって，タスク内でパラメータ変数や大域変数がどのように使用されるかを解析し，各パラメータ変数や大域変数の到達可能性解析処理により，各変数が使用可能となった場合に実行可能な文を抽出する．これを漸進処理単位として再構成し，変数の準備状況に従い実行されるようタスク内を書き換える．

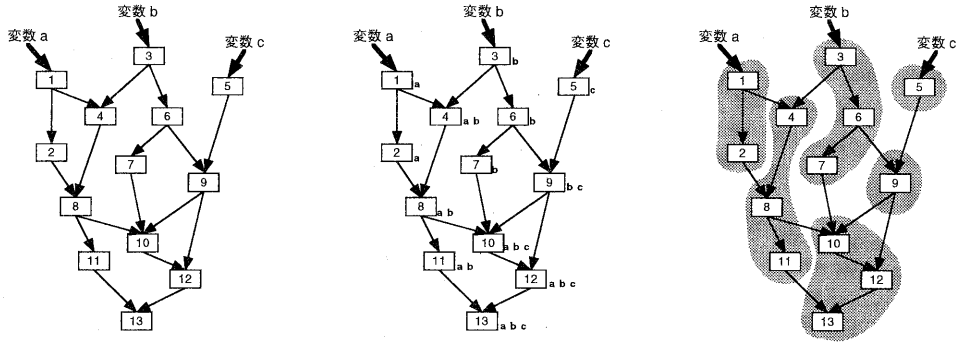
漸進処理単位の抽出手法について以下に述べる．

1. タスク内のデータフローを解析し，節とラベル付有向辺で構成されるデータフロー・グラフを生成する．グラフ内の節はタスク内の文の集合を表し，有向辺がデータフローを表す．有向辺のラベルはそのデータフローを構成する変数である．ここで，節として表されるタスク内の文の集合は節間に制御依存関係が発生しないような集合として構成する．

2. 漸進処理の対象となるパラメータ変数，大域変数の到達可能性を解析し，各節にラベル付けする．変数が到達する節にはその変数のラベルが付加される．

3. 節に付加されたラベルの種類により節をグループ化する．このグループ化により構成される節の集合が漸進処理単位となる．

漸進処理単位の抽出例を図 3に示す．図 3(a) は変数 a, b, c をパラメータ変数とするタスクのデータフロー・グラフを表している．各変数の到達可能性の解析によりラベル付けされた結果を図 3(b) に示す．このラベルに従い節をグループ化した結果が図 3(c) である．例えば，節 1, 2 は変数 a により駆動される漸進処理単位として抽出されている．これらの漸進処理単位を変数の準備状況により駆動させ，タスクの部分実行を実現する．



(a) データフロー・グラフ　(b) 到達可能性の解析結果　(c) 漸進処理単位

図 3. 漸進処理単位の抽出

5 評価実験

本稿で述べた並列化手法の有効性を評価するため，シミュレーション実験を行った．本節では実験結果について述べる．

5.1 対象プログラム

並列化の対象として NPB (NAS Parallel Benchmark) 2.3 を用いた．NPB は NASA の Numerical Aerodynamics Simulation グループが開発したプログラムである [10]．基本的な数値計算プログラムであり，EP, MG, CG, FT, IS, LU, SP, BT の 8 種類のプログラムから構成されている．NPB は最初から MPI を用いた並列プログラムとして公開されているが，本実験では，NPB を逐次実行できるよう書き換えた NPB-serial を使用した．プログラムが使用するメモリ量，データ量，計算量を規定する問題クラスは W (ワークステーション) を用いた．NPB はほとんどの部分が FORTRAN で記述されているが，我々の解析ツールは C 言語を対象としている．したがって，FT 以外の FORTRAN プログラムを C 言語に変換し，実験に使用した[1]．

5.2 実験内容および実験結果

各ベンチマーク・プログラムに対して関数手続きに基づいたタスク生成法を適用し，生成されるタスクの粒度，タスク数について調査した．次に，生成されたタスクに対して漸進処理を適用した場合の並列実行効率の変化をシミュレーションにより検討した．

表 1: NPB ベンチマークによる実験結果

名前	タスク数	実行時間 (秒)	漸進処理での実行時間 (秒)	効果
BT	10	1292	916	1.41
CG	3	35	34	1.03
EP	1	97	–	–
IS	1	13	–	–
LU	6	2240	1712	1.31
MG	9	162	139	1.17
SP	11	2460	1719	1.43

実験結果を表 1 に示す．表で，実行時間はタスクを並列実行したときの時間を，漸進処理での実行時間は漸進処理を適用したときの時間を，効果は漸進処理による処理時間の向上比を，それぞれ表している．実行時間は，シミュレーションにより求めた時間である．EP や IS はタスクが一つしか生成されず，並列実行ができない結果となった[2]．しかし，その他のプログラムに対しては適切な粒度のタスクが生成された．しかし，生成されたタスク数は少なく，並列処理による処理効率の向上はあまり得られそうにない結果となった．これは，ほとんどの対象プログラムが関数手続き内に巨大なループ文を持ち，そのループ文における並列性が抽出されなかった結果である．

次に，漸進処理の適用に関してであるが，CG を除いては漸進処理の導入により実行効率が 15 ～ 40% 向上することが分かった．CG プログラムはタスク数が少ないため，漸進処理の効果が得られ

[1] FT には複素数が使用されていたので，今回の実験には使用しなかった．

[2] EP は Embarrassingly Parallel の略で，並列処理が非常に困難なソート・プログラムである．また，IS は関数手続き呼出しが二回しかなく，並列化できなかった

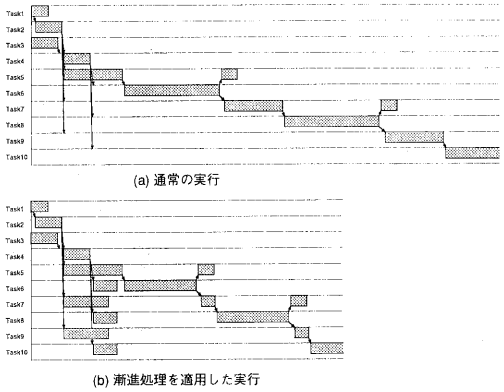


図 4: BT プログラムの実行経過

なかったと考えられる．漸進処理適用の例として，BT プログラムの実行過程の一部を Gannt チャートで表したものを図 4に示す．図から分かるように，Task6 以降のタスクにおいて効果的に漸進処理単位が部分実行され，並列実行性が向上している．

本実験により，本稿で述べた並列化手法により計算機クラスタ環境に適切な粒度のタスクが生成されることが分かった．しかし，生成されるタスク数がそれほど多くなく，このままではプログラムの並列性が十分には抽出されていないことが分かった．また，漸進処理の適用により，15 ～ 40% の並列実行性の向上が確認され，漸進処理に基づくタスク実行制御手法の有効性を確認できた．

6 おわりに

本稿では，特別なハードウェアを装備しない計算機クラスタ環境において，効率よい並列処理を行うための，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でのタスク起動オーバヘッド，タスク間通信オーバヘッドを考慮して，関数手続き単位での並列化手法を検討し，粗粒度タスク生成手法を開発した．また，関数手続き単位でのタスクでは十分な並列性が抽出されないという問題に対して，タスク内の文を漸進的に部分実行させ，並列実行性を高める漸進処理手法を提案した．最後に，本稿で述べた手法の有効性を確認するために行った実験において，適切な粒度のタスクが生成され，また漸進処理により並列性が抽出されることが確認された．

今後の課題としては，プログラムからのさらなる並列性の抽出が挙げられる．実験結果でも述べたように，適切な粒度のタスクを生成することができたが，タスク数は少なく，あまり効果的な並列実行が行われないという問題があった．これは，特に今回使用した数値計算プログラムにおいては，関数手続き内に巨大なループ文が複数存在し，その部分の並列性が十分に抽出されないからである．今後，より並列性が抽出できるタスク生成手法について検討しなければならない．また，漸進処理単位の生成においてデータフロー・グラフを生成するが，このとき節間の制御依存関係を削除するために文の集合を節として採用した．データ依存と制御依存を考慮して漸進処理単位を生成し，より効率よい漸進処理を目指す必要がある．

参考文献

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

[2] 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).

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

[4] 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, Vol. J73-D-I, No. 9, pp. 756–766 (1990).

[5] 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, Vol. J75-D-I, No. 8, pp. 511–525 (1992).

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

[7] 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, Vol. J73-D-I, No. 12, pp. 951–960 (1990).

[8] 朝倉宏一, 渡邊豊英: "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).

[9] K. Asakura and T. Watanabe: "Fine Grain Execution Control of Procedure-oriented Tasks on Computer Cluster Environment", *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, Vol. VI, pp. 2739–2545 (1999).

[10] 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).