プログラミング 14－3
（1997. 8. 21）

同期モデルに基づく自動並列化コンパイラにおけるタスクスケジューリング方式

金山 二郎　飯塚 肇

成蹊大学大学院工学研究科情報処理専攻

概要

一般に、ある問題を解くプログラムを並列化する場合、データ並列性の抽出が不可欠とされる。しかし、問題自体がデータ並列性に乏しい場合や、データ並列性はあってもプロセッサ集合への全体的なマッピングが行なえない場合など、各プロセッサの利用率が低下する傾向にあることが指摘されている。

このような状況に際し、現在のタスクを割り当てられていないプロセッサに異なるタスクをスケジューリングすることで、利用率の向上が見込める。

本稿では、特に分散メモリの並列計算環境におけるタスクの自動並列化について考察し、自動並列化コンパイラ log c におけるタスクスケジューリング方式について示す。

Task scheduling strategy for autoparallelizing compiler based on synchronous parallel computation model

Jiro KANAYAMA　Hajime IIZUKA

Department of Information Sciences,
Graduate School of Engineering,
Seikei University

abstract

In order to parallelize programs for the execution by the multiprocessor, extraction of data parallelism is indispensable. However, the processors can not be fully utilized under certain situations whitch include that data parallelism of the problem is little. Even if the problem itself has a fair amount of data parallelism, a lack of scalability often restricts its utilization.

In such situation, it can be useful to allocate tasks to idol processors.

In this paper, we study autoparallelization of tasks on distributed parallel computation environment and propose a task scheduling strategy for the autoparallelizing compiler log c .

1 まえがき

近年、分散メモリに基づく汎用並列計算環境は、プロセッサ数を増大させる方向で研究開発されている例が少なくない。一方、プログラムは、基本的には問題のデータ並列性を抽出する形で並列化がなされるが、様々な制約により、並列計算環境全体へのプログラムのマッピングが妨げられる場合がある。よって、プロセッサ数の増大に対し、プロセッサの利用率は低下する傾向にある。

この状況を緩和する方法のひとつとして、タスク並列の導入が考えられる。問題を解くプログラムは、ある役割を担う小さなタスクの連続として表現され、並列計算環境においても同様のことが言える。通常、各タスクは他のいずれかのタスクとの依存関係を持つが、中には、プログラム内での順序性を厳密に保持する必要のないものも存在する (図 1-a,b)。よって、データ並列性に基づく並列化を補助する意味で、タスク単位での並列化は効果を挙げる可能性があると考えられる。

ただし、タスクの持つ様々な性質から、単に空きプロセッサにタスクを詰め込む方式では、逆に性能を低下させることになりかねない。また、タスク自体の定義や必要とされる情報の取捨選択も、性能を左右する要素となりうる。本稿では、自動並列化コンパイラ log$c$ のタスクスケジューリング方式について考察し、利点と問題点を明らかにする。

第 2 章ではタスク並列性について概説し、第 3 章で我々が研究開発を行っている並列言語 log$l$ と自動並列化コンパイラ log$c$ について解説する。第 4 章で log$c$ におけるタスク並列化機構について解説し、利点と問題点を明らかにする。最後に、第 5 章でまとめとする。

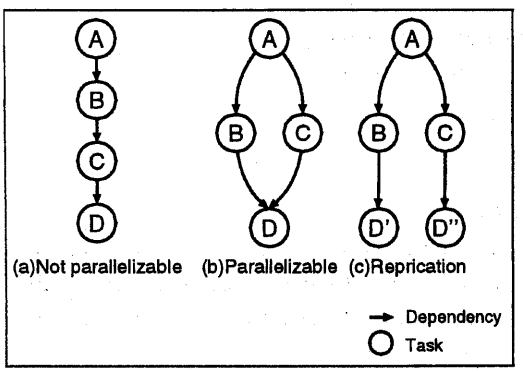

図 1: タスクグラフ

2 タスク並列

2.1 データ並列の限界とタスク並列の利点

データ並列性の抽出に基づくプログラムの並列化は、並列プログラミングの一般的な手法であるが、次のような場合には良好な結果をもたらさない。

データ並列性が得られない場合 問題を解くアルゴリズム自体が強いデータ依存性を持っていて、他のアルゴリズムが見つからなければ、並列化による性能の向上は得られない。

スケーラビリティが得られない場合 データ並列性が得られても、プロセッサ台数が固定だったり、その他の制約があって、スケーラビリティに乏しく、その制約が実際の計算機構成にマッチしない場合には、プロセッサの利用率が著しく減少し、結果として性能の向上は得られない。

このような状況に際して、タスク並列を導入することは、プロセッサの利用率低下を打開するための有用な対策であると考えられる。タスク並列自体は疎粒度の並列化戦略であり、一般には並列化に優れているとは言いがたいが、データ並列と組み合わせて補助的に用いることで、より高い性能を得られる可能性を生む。

また、通信コストの削減が期待できる。定義にもよるが、我々の想定するタスクはそれ自体が密接に通信を行う。そのような場合、プロセッサのグルーピングは、通信のグルーピングにつながり、通信オーバーヘッドの削減が期待できる。

最後に、並列計算環境がクラスタをなしている場合には、通信のコリジョンの削減も期待できる。ただし、マッピングの方法に関しては専用の機構を要する。

2.2 タスク並列の方法

タスク並列化の戦略としては、主に次の 2 つが考えられる。

異なるタスクの並列化 互いに並列化可能な異なるタスクを並列にスケジューリングする (図 2-a)。

同じタスクの複製 あるタスクが複数のタスクと依存関係を持っている場合、そのタスクを解析し、可能であれば、分割して他のタスクと統合する (図 2-b)。

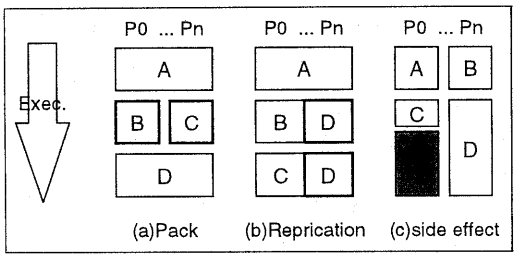

図 2: 2種類のタスク並列と副作用

このうちの後者については、特に高度な依存性解析、タスクグラフの再構成 (図 1-c) に加え、前者との兼ね合いを要する。また、両者は独立に考察することも可能であることから、今回は異なるタスクの並列化のみを扱う。

3 並列言語 log l

並列言語 log l は、同期モデルに基づく並列言語である [1]。基本的な構造は、一般に PRAM アルゴリズムと呼ばれる構造化言語にならっている。PRAM アルゴリズムには特に固定した定義は存在しないが、本稿では [2] のものを採用している。

基本的には共有メモリを持つデータ並列言語として位置付けられる。HPF[3] や VPP Fortran[4] などに代表される並列 Fortran に似た思想のもとに設計されているが、並列実行部に大きな違いを持つ。並列 Fortran が実際の並列計算機を仮定しているのに対し、log l では、プログラムの設計を実際の計算機構成から独立に行えるよう、PRAM[5, 6, 7] に基づく並列計算機モデルを仮定している (図 3)。

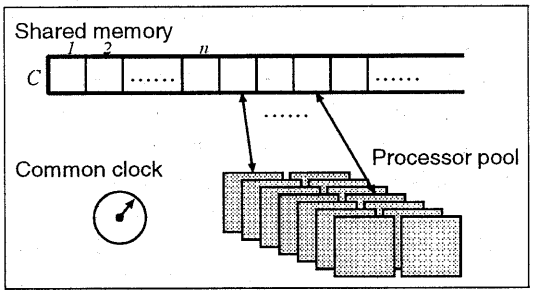

図 3: log l が仮定する並列計算機モデル

また、特に重要な構成要素として、以下のものが挙げられる。

par 構造 並列実行構造である par 構造は次のような文法を持ち、意味的には、「集合 X の各要素 p に対して、*OPERATION*(p) を実行する」。

$$\textbf{par}\ p \in X\ \textbf{do}\ \mathit{OPERATION}(p)$$

メモリモデル log l では共有メモリついて、メモリモデルを複数用意しており、詳細なプログラミングを可能にしている。読み込みと書き込みそれぞれについて、同時アクセスが可能か否かを指定できる。また、同時書き込みについては、アクセス競合の対処法を複数用意している。

3.1 自動並列化コンパイラ log c

log c は log l に基づく自動並列化コンパイラであり、現在、pthread コード、PVM コードを生成することができる [8, 9]。

主な役割として、par 構造を解釈し、実際の並列計算環境に合致する並列コードに変換する。各 par 構造の間はバリア同期がとられ、逐次的に実行される。

ここで、par 構造をタスクとみなした場合に、これまでの実装ではプロセッサの利用率低下への対処がなされておらず、本稿ではこの改善を目指す。

また、HPF のような自動並列化コンパイラでは、タスク制御をプログラマが担当していた。本稿では、言語仕様の変更を行なわず、これを自動化することで、より抽象的なプログラミング環境の提供を行ない、プログラマの負担を軽減することを目標とした。

なお、今回は分散環境を想定した PVM コードを生成するバージョンでの考察について示す。

4 log c におけるタスク並列化機構

4.1 指針

タスクの並列化を行うのに、どのようなことを指針とするかは、性能に強く影響する。

本稿では、性能の低下が起こらないことを第一の指針とする。我々のコンパイラにおいては、タスク並列はあくまでも補助的な存在であり、並列化の主軸であるデータ並列を損なわないように配慮する。

また、タスクの実行時間を計測するような、いわゆるプロファイリングは行なわない。これについては、将来的には導入することも考えられるが、我々の場合、

プロファイリングは最終的な手段であり、可能な限り論理構造を重視する。実行時間が未定の状態でタスクを一旦プロセッサに割り当てると、図2-cのようにデッドスペースを生む可能性があるが、本稿では、これを極力回避するような機構を提案する。

4.2 実行形態

まず、コンパイル後の実行形態としては、全体の実行を司るタスク管理プロセスが存在し、後述する**log c タスク管理表**を使ってこれを管理する (図4)。

各プロセッサにはひとつのタスク実行プロセスが常駐し、タスク管理プロセスの指示のもとにタスクを実行する。

4.3 log c タスク

log c では、ひとつの par 構造をひとつのタスクとして定義する。特に区別の必要がある場合、これを**log c タスク**と呼ぶ[1]。

タスクはコンパイル時に静的に定められ、**タスク管理表**と呼ばれる次の情報を持つ。

入力 実行に必要とされる共有メモリの場所と大きさ。

出力 最終的に書き出した共有メモリの場所と大きさ。

発火条件 そのタスクへの入力を持つタスクのリスト。

スケーラビリティ 何台のプロセッサに割り当てて実行することが可能かを示す式。スケーラビリティがない場合は定数となる。

再割当可能フラグ プロセッサに余剰が発生した場合に、実行を中断し、タスクを再び割当てて、途中からまた実行を継続できるかどうかを示すフラグ。

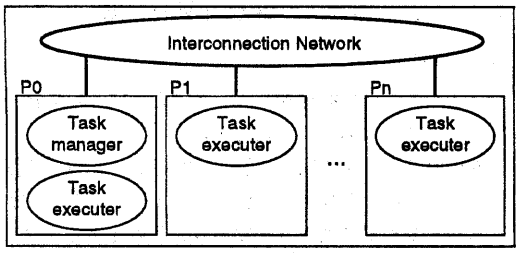

図 4: タスクマネージャを伴うプログラム実行

[1]前述したように、これも将来的には、タスク複製のような概念の導入が考えられる。

この別により、タスクは特に**再割当可能タスク**と**再割当不可能タスク**に分けられる。

また、par 構造以外の逐次ブロックについて、これまでは、タスク管理はあるタスク実行プロセスが担っており、そのプロセスが逐次ブロックを実行していた。今回から、タスク管理の分担が専用のプロセスに割り当てることになったため、逐次ブロックについてもタスク管理プロセスが担当することとした。

4.4 タスク管理プロセスのタスク並列化戦略

以上のような条件と情報のもとに、タスク管理プロセスは動的にタスク並列化を行なう。

並列化は再割当不可能タスクを軸に行なわれる。再割当不可能タスクは、計算機構成上できるだけ大きなスケールで配置を行なう。このため、再割当不可能タスクの実行直前には、そのスケールのタスク実行プロセスはバリア同期を行なう。これは、デッドスペースを生む場合が少なくない。しかし、もし、より小さなスケールでの割当を行なったとしても、再割当が不可能なため、結果としてより大きな空きスペースを生成する可能性がある。

ただし、順序を入れ換えても問題がなく、現在の利用されていないプロセッサ群に割当可能な他のタスクが存在した場合、そのタスクを実行する。

4.5 変換イメージと考察

図5-dが、タスク並列を導入する前の実行イメージであり、図5-eは導入後の実行イメージを表す。いくつかのデッドスペースは残されるが、導入前と比較すると小量になっている。

また、タスクBの終了時、タスクEは小さなスケールであれば割り当てることが可能だが、再割当不可能なことから、実行効率の損失を考慮し割り当てられていない。

本稿で用いたスケジューリング方式は保守的なものであり、潜在する並列化の可能性のいくつかを無視している。例えば、各タスクのスケーラビリティから、タスクの組合せによって、より効率的な実行を実現するマッピングが考えられる。しかし、各タスクの実行時間が未定である以上、この方式については、なんらかのプロファイリングか、タスク複製を許すような戦略を要する。

また、入出力に依存してどのプロセッサに割り当て

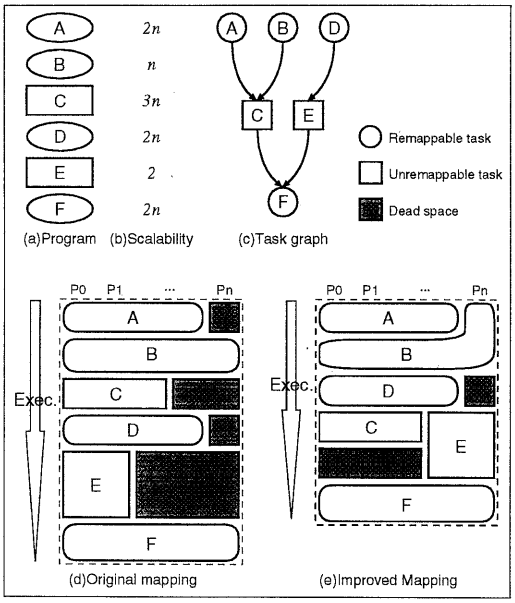

図 5: 変換イメージとマッピングの比較

るかを考慮すれば、通信量の削減が図られるが、逆にタスク並列には不利に働く場合も考えられる。この点についても、導入にあたっては、通信性能に加え、上記のプロファイリングを考慮する必要がある。

ただし、通信性能を含めた並列計算環境については、本稿では、分散環境ということだけを仮定しており、これより詳細な機構については触れていない。これは、最近の高速なネットワークの台頭により、通信性能をある程度後回しに考えても差し支えないと思われるためである。

最後に、log c タスクでは、タスクの実行時中断は想定しない。このような状況は、例えば、GUI を伴うようなプログラムで考えられるが、このようなプログラムをサポートするためには特別な機構を要する。

5 あとがき

本稿では、自動並列化コンパイラ log c におけるタスクスケジューリング方式についての設計と考察を行なった。現在、設計に基づいた試作を行なっている。

今回のタスク並列は複製を行なっていないため、データ並列を補助する保守的なものとして位置付けられるが、将来的に複製を許した場合には、データ並列と並んで並列化の主軸をなすものと思われる。

抽象言語によるプログラムの自動並列化自体については、メッセージパッシングライブラリのような低レベルな環境を直接利用するプログラムと比較すると、やはりある程度の不利が生ずる。ただし、ソフトウェア分散共有メモリのような環境と比較すると、動的な部分がより少ないだけ、効率的に優位を保つ可能性がある。また、ヒューリスティックな解法や特殊な高速化技術については、ライブラリという形での導入が考えられる。

今後の課題としては、実際の性能評価による問題点の抽出と解消、タスク複製の導入が挙げられる。

参考文献

[1] 金山二郎, 飯塚肇：同期モデルに基づく自動並列化コンパイラ, 情報処理学会第 7 回プログラミング研究会 (1997).

[2] 宮野悟：並列アルゴリズム, 近代科学社 (1993).

[3] 妹尾義樹：HPF 言語の現状と将来, 情報処理, Vol. 38, No. 2, pp. 90–99 (1997).

[4] 岩下英俊：HPF からみた VPP Fortran, 情報処理, Vol. 38, No. 2, pp. 114–120 (1997).

[5] Fortune, S. and Wyllie, J.: Parallelism in random access machines, in *Proc. 10th ACM Symposium on Theory of Computing*, pp. 114–118 (1978).

[6] Savitch, W. J. and Stimson, M.: Time bounded random access machines with parallel processing, *J. ACM*, Vol. 26, pp. 103–108 (1979).

[7] Goldshlager, L. M.: A universal interconnection pattern for parallel computers, *J. ACM*, Vol. 29, pp. 1073–1086 (1982).

[8] 金山二郎, 飯塚肇：PRAM プログラムから pthread プログラムへの変換, 情報処理学会第 52 回全国大会 (1996).

[9] 金山二郎, 飯塚肇：PRAM プログラムから PVM プログラムへの変換, 情報処理学会第 54 回全国大会 (1997).