ハイパフォーマンス
コンピューティング 72－16
（1998. 8. 7）

コンパイラ字句解析の実行時スレッドサイズの決定法に関する研究

大森洋一†

†：高知工科大学 情報システム工学科

マルチスレッドモデルは，特に小規模な並列処理において，その有効性が認識されており，多くの計算機システムがサポートするようになっている．しかし，現在使われているプログラムの多くは，十分マルチスレッド化されていない．そこで，マルチスレッド化による高速化が望める部分を，応用分野ごとに効率的にスレッド化するためのアルゴリズムを提案する．schemeプログラムを対象としたマルチスレッド化を補助するツールを作成し，幾つかの例で検証したところ，応用プログラムのにスレッド生成の結果が大きく依存することが確かめられた．また，効率のよいスレッド生成のためには，システムの特性を考慮したパラメータ選択が不可欠である．

A Determining Method of Run-Time Thread Size at Lexical Analysis

Youichi OMORI†

†：Department of Information System Engineering
Kochi University of Engineering
Miyanokuchi 185, Tosayamada, Kami, Kochi, 782–0003 Japan

E-mail: youiti-o@info.kochi-tech.ac.jp

The thread model is supported by many parallel systems for its good efficiency to small parallelism. But, most of current programs haven't been threaded. We propose a heuristic algorithm to support extracting user level threads. The algorithm can detect the parts of the program which is accelerable by threading. A simple application of this algorithm to some examples of the scheme language said, whether the result of extracted threads' pattern is optimal or not depends on the application area. The choice of scheduling parameters accounting the system's character is inevitable when determing the way to divide the program into threaded.

1 はじめに

マルチスレッドは，特に小規模なメモリ共有型システムにおいて，並列計算の有力なモデルの一つであり，単一プロセッサにおいても有効な場合がある，実装が比較的容易である等の理由により，多くの計算機システムで利用可能となっている．ところが，マルチスレッドが利用可能なシステムにおいても，マルチスレッド化されているプログラムはそう多くない．この理由の一つとして，現時点では，マルチスレッドを使うモデルに慣れたプログラマが少ない点が挙げられる．すなわち，マルチスレッドモデルは、従来のマルチプロセスモデルより並列処理のオーバヘッドが小さく，比較的簡単に利用できるが，常に性能の向上が得られるわけではないという問題がある [3]．そこで，よく使われているプログラミング言語である scheme で書かれた逐次プログラムを対象に，マルチスレッドによる並列評価の効率を上げるために必要な情報を，プログラマに提示するツールである mtassist を作成した．mtassist は共有メモリシステム上のスレッドモデルを対象として，次の機能をもっている．

1. scheme によって書かれたプログラムを読み込み，並列評価の可能性がある部分を検出する．
2. アプリケーション分野の推測に基づいたスレッドサイズに応じて，並列評価のコストを評価する．
3. スケジューリングで利用するための簡単なヒントを生成する．
4. 解析は 1 パスで終了する字句解析の範囲にとどめ，変数の依存解析などは行なわない．

具体的なスレッドの挙動は POSIX スレッドによるインターフェイス定義を想定し，ユーザレベルでスケジューリングするものとした．

mtassist の作成にあたって，いくつかの問題点があったが，本稿ではそれらのうち，スレッド生成についての評価アルゴリズムとコストの評価方法について述べる．以下，2 章では，システムからみた最適なスレッド実行条件の検討，3 章では，スレッド生成アルゴリズムについて述べ，4 章でまとめを行なう．

2 最適なスレッド実行条件

関数型言語によるプログラムは，明示的な制御フローがないので，並列実行に適しているといわれる．具体的には，意味の一貫性が保証されている範囲で，引数の並列評価を利用する方法が用いられる．scheme で記述されたプログラムにおいても，遅延評価の部分を除き，静的に評価順序を決定できて，複雑になりがちな意味解析を用いることなく，かなり高い並列度を得られる [2]．

しかしながら，現在の商用システムでは，対象となる並列計算機のモデルは，プロセッサ数に関するスケーラビリティを確保するために

- プロセッサは無限にある
- スレッド生成，破棄にコストはかからない
- 同期操作以外に通信コストはかからない

といった単純なものがほとんどであり，しかも，細粒度の並列性を効率よく実行するための工夫がなされていない．このため，現実には，最大並列度を得られるプログラム記述が常に最適というわけではない．

2.1 スレッドの動作モデル

POSIX スレッドは，図 1 に示すような 4 状態からなる．それぞれ，Active は実行状態，Ready は処理装置の割り当てを待っている状態，Block はロックや同期による資源の解放を待っている状態，Zombie は合流等による資源の回収を待っている状態である．

さらに，状態間の遷移を行なうスケジューリング方式により，カーネルレベルのスレッドとユーザレベルのスレッドに大別できる．カーネルレベルのスレッドは，システムによるスケジューリング結果に基づいて，実行状態が変更されるようなモデルである．これに対して，ユーザレベルのスレッドは，プログラマが，生成や破棄も含めたスレッドの状態変更命令をプログラムに埋め込み，すべての動作を対象プログラムの空間で行なう．ユーザレベルのスレッドモデルでは，システムの情報が分かれば，理論的に最適化が可能である．

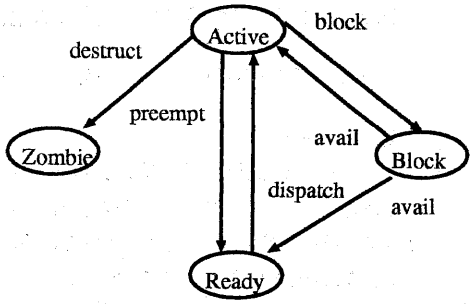

図 1: POSIX スレッドの状態遷移

2.2 スレッドのサイズ

スレッドモデルは計算機資源の使用量が比較的小さいが，対象となる問題が大型になると，記憶領域の大きさや同期のコストが無視できない．POSIX スレッドは，次のような構造により表現される [1].

入出力	
引数	起動時の引数
レジスタ状態	汎用レジスタの値
スタック	
スタックサイズ	
スタックアドレス	ハードウェアスタックとは限らない
スケジューリング	
優先度	整数値
ポリシー	
タイムアウト	同期時のタイムアウト時間
シグナルハンドラ	
シグナルマスク	
シグナルカウンタ	トラップしたシグナル数
状態管理	
スレッドの状態	
Ready キュー	待ちキューへのポインタと ロック変数
Block キュー	待ちキューへのポインタと ロック変数
データ	
内部変数	ローカル変数
固有エラー	エラー番号

表 1: スレッド構造体

システム内の計算資源が競合しておらず，同期や排他制御のオーバヘッドがない時，任意のスレッドは実行状態になることができて，理想的にはシステムのもつ並列度に応じて，線形に高速化される．

こうしたスレッドの実行効率に大きな影響があるのは，表 に挙げたスレッドのデータ構造のうち，他のスレッドと共有する資源であるメモリ使用領域とスレッドの実行時間である．メモリ使用領域は，

$$(\text{管理構造体} + \text{スタックサイズ} + \text{内部変数}) \times \text{ロードされているスレッド数} \quad (1)$$

で計算される．

また，同期をスピンウェイトで実現した場合、スレッド長を n ステップ，同期変数の 1 ステップあたりの出現頻度を p_s，同期操作でスピンウェイトの頻度を p_w，平均待ち時間を t_w，タイムアウト時間を t_{max}，タイムアウトによりブロックする確率を p_b とすると，スレッドの平均実行ステップ数 n_{avg} は，

$$\begin{aligned} n_{avg} &= n(1-p_s) + np_s[(1-p_w) \\ &\quad + p_w\{t_w(1-p_b) + t_{max}p_b\}] \\ &= n - np_sp_w\{1 - t_w(1-p_b) - t_{max}p_b\} \end{aligned} \quad (2)$$

となる．これらの式 (1) と (2) から次のことが分かる．

1. スレッドの実行時間を短くするには，同期変数の出現頻度 p_s を小さくすればよい．スレッド長が長いと同期すべき変数が増え，スピンウェイトあるいはブロックしやすくなる．このため，スレッド長が長すぎない方がよい．
2. スピンウェイトの確率 p_s は小さい方がよい．変数の依存関係は，保たれているのが望ましい．また，スケジューリング・アルゴリズムにも大きく左右される．
3. 並列度が小さい場合，式 (1) から，メモリ使用領域などの必要な資源が減り，スケジューリングのコストも小さくなる．ただし，並列度は下がるため，可搬性は低下する．

2.3 スレッドの表現

前節の検討から， mtassist はスレッドのサイズ n および同期変数の出現確率 p_s を引数として受取り，タイムアウト時間 $n/2$，スピンウェイトの確率やブロックの確率などを乱数によって決め，目

標とする平均スレッド長を求める．この結果に基づいて，schemeプログラムのスレッドの目安を表示する．まず，最上位節は無条件で分割し，さらにcond節をまたがったスレッドは作らないようにし，そろぞれの分岐についてカウントするものとする．さらに，それまでの平均スレッド長が目標スレッド長よりも長ければ短めに，目標スレッド長よりも短ければ長めにスレッドに抽出する．

スレッドの区切りには，schemeのコメント記号である';'と改行コードを出力するようにし，そのままschemeインタプリタで処理できるようにした．これにより，スレッド数も簡単にかぞえることができるようになる．また，コメント中に同じレベルでのスレッド数を出力している．

3 スレッドの判定アルゴリズム

スレッドが有効な応用分野として，次のような例が挙げられる．

1. クライアント-サーバシステムにおける入出力の多重化
2. ユーザーインターフェイスなどのイベントループの応答速度の向上
3. マルチプロセッサによる並列処理

表 2: スレッドが有効な例

現在広く用いられているスレッドプログラミング法は，従来の逐次プログラムないしアルゴリズムの書き換えを前提としており，表3.1の例に共通する特徴として，中心となる部分がループで構成されており，似たような作業を一種のポーリングにより，処理していく点が挙げられる．ポーリングされたそれぞれの作業は，分岐やシグナルによって決まる独立した処理であり，それぞれをスレッド化することにより，ブロックを避けられる可能性が高くなる [5].

これらは，マルチスレッド化による改善が望める部分であり，本章では，第2章のような機械的なスレッド抽出ではなく，よりヒューリスティックなアルゴリズムを提案する．

3.1 応用の推測に基づくアルゴリズム

表3.1の例は，それぞれ次のように具体化できる．

1. 入出力の多重化
   プロセス単位の並行処理では，非同期な入出力装置などでブロックが起きると，他のプロセスでも排他制御によるブロックが生じる．そこで，構造化されたロックを用いて，ブロックしたスレッドから他の実行可能なスレッドへ実行権を移すことにより，簡単に入出力の多重化が実現できて，外部との通信応答速度や資源の利用効率が向上する．
   プロセスをほぼそのまま置き換えるため，スレッドのサイズは比較的大きく，資源の排他制御以外のスレッド間の同期は少ない．
2. 応答速度の向上
   GUIの普及にともないイベント駆動型のプログラムが一般化しており，こうした細粒度のイベントの処理にスレッドを応用すると，連続した入力に対する応答性が向上する．また，プログラム内部の通信においても，イベント駆動が本質的にもつ並列性を有効に利用できる．この場合，スレッドのサイズは比較的小さく，同一資源へのアクセスが多いため，スレッド間の同期は多い．
3. マルチプロセッサによる並列処理
   マルチスレッドで記述した数値演算を，マルチプロセッサ上で並列実行することにより，高速化や大規模化が図れる．問題により，大きく性質が異なり一般化は難しいが，入出力や分岐が少なくループによる繰り返しを特徴とする行列演算と，イベント・シミュレーションや探索問題など分岐が多数生じる不規則演算に大別できる．
   前者は，スレッド長が長く共有変数も多いが，規則性を利用して実行効率を改善できる．後者は，スレッドを長くできず，しかも変数を共有する多くのスレッドができるため，バリヤなどが必要になり，同期のコストが高くなる．

こうした特徴を利用して，特に表の4種類の応用分野について，それぞれに適したスレッド化を行なう次のようなアルゴリズムを提案する．ただし，サブルーチン呼び出しや再帰構造は考慮して

いない.

1. 最長ループを探す.
   これらは，ループを特徴としており，その中でもっとも大きなものをプログラムの本体とみなす.
2. 本体中の分岐を探す.
   基本的な制御フローが単一ならば，規則的な行列演算 (C) と推測する．この場合の分岐には，制御文のみでなく，シグナル処理も含む.
3. 分岐の大きさを比較する.
   分岐後の処理がループのサイズと比べて，大きければサーバ (A) と推測する.
4. それぞれの大きさを比較する.
   分岐後の処理がループのサイズと比べて，小さくかつそれぞれのサイズが一定ならば，イベント処理 (B) と推測する.
5. 分岐の大きさを比較する.
   分岐後の処理がループのサイズと比べて，小さくかつそれぞれのサイズがまちまちならば，不規則演算 (D) と推測する.
6. 規則的演算の分類
   (A) または (C) ならば，スレッド長をできるだけ長くする．これは，分岐がないか，あってもあまり実行されないと予想されるからである.
7. 不規則演算の分類
   (B) または (D) ならば，スレッド長にはこだわらず，分岐を基本としてスレッド化する.

3.2 Scheme への応用

scheme プログラムを対象として，第節のアルゴリズムを mtassist に組み込み，表 3 のプログラム群に適用した．scheme では，明示的なループ構造は存在せず，末尾再帰によってループ処理が行なわれる.

プログラム名	プログラムの説明	大きさ (ステップ)
quicksort	クイックソート	22
mult	行列積の計算	14
hash	ハッシュの計算	18
view	jacal シェル	60

表 3: scheme プログラムの例

この結果，マルチスレッド化の対象とする応用プログラムの性質による分類に成功し，スレッド生成の結果が大きく依存することが分かった．ただし，各アルゴリズム中の最大ループ中の n 個の分岐後の長さ $bs_i(i = 1, 2, \ldots)$ と最大ループのサイズ ls の比による区別は，$\delta(bs_i) - (bs_{max} - ls/n)$ の符号によって分類した．ただし，$(bs_{max}$ は対象ループ中の分岐後の最大長である.

実験の一部を簡単にまとめると，表 4 のようになる.

プログラム名	判定	スレッド長	最大ループ長
quicksort	(D)	4	9
mult	(C)	10	10
hash	(C)	5	8
view	(C)	9	30

表 4: 応用分野の推測結果

同時に，効率のよいスレッド生成のためには，システムの特性を考慮したスケジューリングが前提となる．各スレッドの状態として，図 1 を想定したとき，スレッド実行時間を T_{active}，スレッド生成時間を T_{create}，スレッド破棄にかかる時間を $T_{destruct}$，スレッドの状態遷移にかかる時間を T_{switch}，ブロックしている時間を T_{block}，同期処理の時間を T_{wait} とすると，システム側からみたスレッドの実行時間はこれらの和になる.

一般にスレッド生成はプログラムの起動時に一度行なわれるだけなので，スケジューリングが必要となるような大きさの問題では無視できる．スレッドの破棄も終了時の一度だけであり，同様である．この結果，

$$T_{thread} = T_{active} + T_{block} + T_{switch} + T_{wait}$$

となる.

スケジューリングポリシーによらず、処理装置の数に比べて，スレッドが十分にあれば，ブロックはしなくてすむ．さらに，現在の商用計算機で用いられている処理装置では，スレッドの切り替え時間と，無限時間ウェイトにより同期するまでの時間の平均は，前者の方が圧倒的に長い.

このため，スレッド切り替え時間を決定ずける，レジスタ数，スタックに関するメモリ管理，内部変数のサイズといったパラメータの選択が重要である [4].

4 おわりに

今後の研究として，より現実的なサイズのプログラムへの応用と，その場合のパラメータ設定がある．現在の評価は，単機能のサブルーチンレベルのプログラムを対象としており，複数の機能をもつようなプログラムにおけるスレッド抽出法を検討する必要がある．

より高い性能を引き出すためには，コンパイラ形式にして，詳細なシステム情報を利用できるバックエンドでの並列化が望ましい．また，並列化の過程をユーザに見せないことで，より使いやすいシステムとなると考えられる．

同時に，バックエンドの抽象化によって，より可搬性の高い並列記述を可能にしていく必要がある．特定のハードウェアに依存した最適化では，システムの進化についていけなくなるのは，歴史の示すところである．

参考文献

[1] Kleiman, S., Shah, D.,and Smaalders, B.: "実践マルチスレッドプログラミング," アスキー出版局, 1998.

[2] 田中哲朗, 岩崎英哉, 武市正人: "Commited Choiceによる関数プログラムの並列実行," 日本ソフトウェア科学会第 9 回大会, pp. 85–89, 1989.

[3] Feautrier, P.: "Fine-Grain Scheduling under Resource Constraints," *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, pp.1–15, 1994.

[4] Sterling, T. T., Musciano A. J., Becker D. J., and Osborne, R. B.: "Multiprocessor Performance Measurement Using Embedded Instrumentation," *Proceedings of the 1988 International Conference on Parallel Processing*, pp.156–165, 1988,

[5] Feng, H.-W., Fujimoto, R. M.: "A Shared Memory Algorithm and Performance Evaluation of the Generalized Alternative Construct in CSP" *Proceedings of the 1988 International Conference on Parallel Processing*, II, pp.176-179, 1988