ハイパフォーマンス
コンピューティング 74−16
（1998．12．11）

分散並列関数型言語における動的なタスク粒度の選定手法

吉池 久夫[†] 中西 正和[††]

[†] 慶應義塾大学大学院 理工学研究科 計算機科学専攻
[††] 慶應義塾大学 理工学部 情報工学科

キューマシン方式並列実行では，分散メモリ型並列計算機上の関数型言語において効率良くタスクの粒度選定を行い優れた並列実行を可能としている．しかしながらタスクの粒度はセグメントのサイズに依存してしまい，効率良く実行するためには適切なサイズのセグメントで実行しなければならないが，そのサイズはアプリケーションに依存する．そこで本論文ではAP1000+上に実装された並列関数型言語処理系において，実行時に要素プロセッサのタスクリストの長さとタスク要求信号の有無によってセグメントサイズの切り替えを行った．これにより，最大粒度を状況に応じて変動させることが可能となり，本方式を用いて実行した3つのアプリケーションで，いずれも最も実行効率の良かった固定長のセグメントサイズで実行したのと同じ速度向上が得られていることを確認した．

Dynamic Granularity Control for Parallel Functional Language on Distributed Memory Machine

Hisao YOSHIIKE[†] Masakazu NAKANISHI[†]

[†] Department of Computer Science,
Keio University

In this paper, we propose a queue-machine-based parallel system with dynamic changing length of segment on distributed memory machine "AP1000+". In usual queue-machine-based parallel execution, length of segment is fixed, thus it is hard for efficient performance to set a optimum length of segment before execution. In our approach, we control granularity of task by changing length of segment to get efficient performance independent on applications. In our implementation, length of segment is changed by existence of a task-request-message from other processor and length of local task queue. As a result, our implementation can get performance as same as execution on optimum length of segment in 3 applications we examine.

1 はじめに

副作用の無い関数型言語は引数間に依存関係が存在しないため，引数を並列に評価するとによって，高い並列性を抽出することが可能である．しかしながら関数呼び出しを1つのタスクに対応させたのでは，その粒度は非常に小さなものとなり，並列化に伴うオーバヘッドが処理効率の向上を上回ってしまい，逐次実行よりも処理効率が悪化する場合がある．そのため，並列関数型言語においてタスクの粒度選定は必要不可欠であるとされてきた [1].

キューマシン方式並列実行では，計算木で同一階層にある複数の関数呼び出しのフレーム (以下，フレームと呼ぶ) を**セグメント (segment)** と呼ばれる固定長の連続領域にまとめて1つのタスクとして併合する．これにより，タスクの粒度をある程度大きくすることが可能となり，細粒度タスクの並列化を抑制できる [3, 4]．しかしながら，この手法においては最適な実行効率を得るためのセグメントサイズはアプリケーションに依存して一意に定まらないという欠点がある．

そこで本研究では，事前にセグメントサイズを設定することなく効率の良い並列実行の獲得を目的とし，実行中にセグメントサイズを状況に応じて切り替える並列 Lisp 処理系を AP1000+上に実装した．

2 動的粒度選定

並列関数型言語においては，再帰的な関数呼び出しを用いることがあるため，実行経路はデータに依存して大きく変わってしまい，静的にタスクの粒度選定を行うことは困難である．よって並列関数型言語においては動的な粒度選定が不可欠となる．

代表的な動的粒度選定の手法として **Lazy Task Creation(LTC)**[5] があげられる．この手法ではフレームをプロセッサにローカルなスタックで管理し，アイドルプロセッサは他のプロセッサのスタックを調べ，実行可能なフレームがあればスティールして実行する．スティールの際はタスクの粒度が大きくなるようにスタックの根元にあるフレームをスティールする．

この手法は共有メモリ上においては，ある程度効率良く働くが，分散メモリ上においては並列化にかかるコストが大きい点とプロセッサ数が多くなるため，細粒度タスクのスティール頻度が多くなり実行効率が低下する．

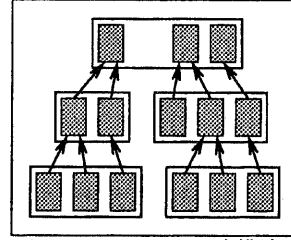
図 1: セグメントの木構造

3 キューマシン方式並列実行

LTC を分散メモリ上で実装する場合，スティールの対象となるタスクが1個のフレームであるために，2節で述べた細粒度タスクのスティールによって実行効率の向上は望めない．そのためにスティールの対象となるタスクに十分粒度を持たせる必要がある．

ここでは本研究の基となるキューマシン方式並列実行 [3, 4] における粒度選定手法について述べる．

3.1 キューマシン方式粒度選定

キューマシン方式並列実行とは，同一階層にある複数のフレームをセグメントと呼ばれる固定長の連続領域に割り当て，そのセグメントを1つのタスクとして並列実行を行う手法である．フレームをセグメントに割り当てる際は同一セグメント中の全てのフレームは同じ親フレームに値を返すように割り当て，同期操作回数の削減を行っている (図 1)．このような手法によってタスクの粒度を大きくすることが可能となる．

セグメントはその中にあるフレームが全て実行可能になればタスクとして実行することが可能となり，その実行可能セグメントはプロセッサにローカルなセグメントリストに保持されLTCと同様のポリシーでスケジューリングされる．この時スティールされるセグメントは十分に粒度が大きいため細粒度タスクの並列化を抑えることが可能となる．

3.2 分散メモリ上への応用

従来のLTCで，細粒度タスクがプロセッサ間を移動する場合，通信オーバヘッドによって効率が低下するという問題があった．しかしながらキューマシン

方式並列実行では，移動するタスクは複数のフレームをまとめたセグメントであるために粒度は十分に大きく，そのため通信オーバヘッドによる効率低下は起こりにくくなるという利点がある [6].

4 可変長セグメントによる粒度選定

本節ではキューマシン方式並列実行において固定長セグメントを用いて実行した際の実行効率とセグメントサイズの相関関係について述べた後，可変長セグメントを用いた手法について述べる.

4.1 セグメントサイズと実行効率

キューマシン方式並列実行では同一階層のフレームを併合するために，実行効率は実行時に動的に生成される計算木の形によって大きく依存する．従って効率良く実行するためにはアプリケーションに適したセグメントサイズで実行することが重要であると言える.

セグメントサイズと実行効率の相関を示すためにAP1000+上でキューマシン方式により粒度選定を行う並列 Lisp 処理系 [6] で次に述べるベンチマークを実行し，セグメントサイズと実行効率の相関を調べた.

fib: 40番目のフィボナッチ数を求めるベンチマーク．並列度の上限は非常に高い.

tak: Lisp の標準的なベンチマークである tak 関数．引数として 27, 18, 9 を用いた.

mall: 乗法的加法的線形論理のシーケント計算 [8] を縦型探索で行うことにより論理式の自動証明を行う．ここでは生成タスク数を一定にするため証明不可能な式の反駁を行っている．並列度は非常に低い

以上のベンチマークのセグメントサイズと実行効率の相関関係を，それぞれ図2から図4に示す．ただし上のベンチマークにおいて実行初期状態の並列性が損なわれないようにするため，計算開始から定数個のセグメントを処理するまではセグメントサイズを意図的に小さくしている．また図2から図4において図中に記載してある数値は切り替え後，固定したセグメントのサイズ (単位は byte) を表している．図の縦軸は64台で実行したときに最も実行時間が短かったセグメントサイズで逐次実行したときの実行時間を基準とした速度向上を表している．また，図5に各アプリケーションの平均的な粒度を示す．ただし，粒度はセグメントとして使用した記憶領域におけるフレーム領域の占める割合で表している.

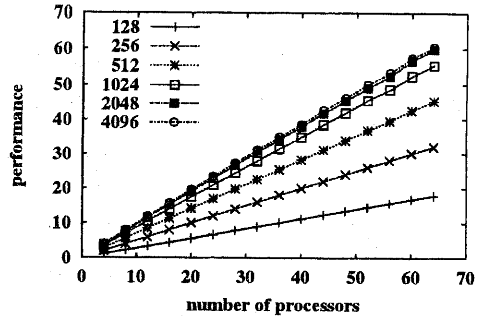

図 2: セグメントサイズと実行効率 (fib)

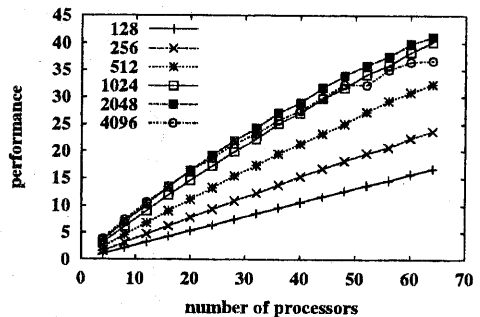

図 3: セグメントサイズと実行効率 (tak)

図2から図4より明らかな通り，最良の実行効率を得るためのセグメントサイズはアプリケーションによって異なることがわかる.

fibのように並列度の上限が非常に高いアプリケーションにおいてはセグメントサイズを大きくした方が実行効率は良くなっている．tak に関しては，セグメントサイズが大きすぎる場合 (4096byte)，速度向上が低下している．これはある時点で並列化可能なフレーム数が fib と比較して少なくなるために，セグメントサイズを大きくしすぎると並列度が抑制されてしまうためである.

mall に関しては元々，並列度が低いため，計算木の同一階層におけるフレーム数がわずかであり，併合されるフレームの数が制限されてしまう．従って，セグメントのサイズを大きくすると，図5が示すようにそれだけメモリの利用効率は悪くなり，実行効率も低下する.

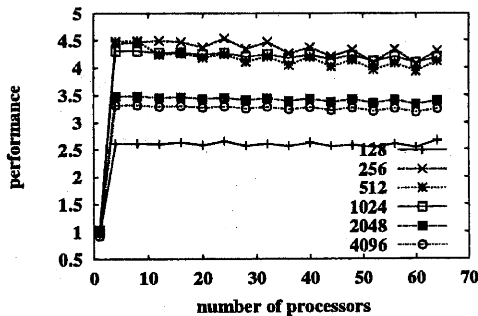

図 4: セグメントサイズと実行効率 (mall)

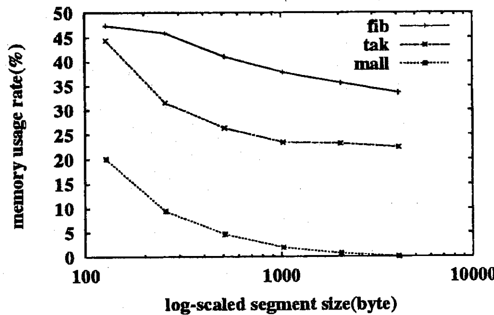

図 5: 平均粒度 (セグメント領域利用率)

4.2 可変長セグメントによる並列実行

以上述べて来たようにキューマシン方式並列実行においては，同じアプリケーションでもセグメントサイズによって実行効率が変わってしまうという問題がある．また実行中に並列度が変動する場合，セグメントサイズを一意に固定してしまうと効率の良い並列実行は期待できない．

そこで本研究では実行中にセグメントサイズを切り替えることにより事前にセグメントサイズを設定することなく効率の良い並列実行の実現を目指した．

本研究ではセグメントのサイズを切り替える要因として，以下のものを取り上げた．

並列度:
キューマシン方式並列実行ではローカルのセグメントリストでセグメントをスケジューリングしている．そのためセグメントリストに十分な数のセグメントが無く，かつセグメントのサイズが大きいと並列性を無駄に抑制してしまう可能性がある．

他プロセッサからのタスク要求:
実行すべきタスクが無くなったプロセッサは他のプロセッサにタスク要求の信号を送付し，その信号を受信したプロセッサは余分なタスクがあれば，そのタスクを送信する．タスク要求の信号が来ない場合，システム全体的に駆動状況にあると考えられるため，その場合，無理に並列度を上げる必要は無いと考えられる．

以上の要因を考慮に入れ本研究では以下に示す通りの方法によってセグメントサイズの切り替えを行った．

1. ローカルなセグメントリスト中にセグメントが1つしかなく，かつ他プロセッサからタスク要求があった場合，現行のセグメントサイズを半分に縮小．
2. ローカルなセグメントリスト中に複数のセグメントがあり，またタスクのスティール要求が無い場合は現行のセグメントサイズを 2 倍に拡大．

以上述べたような手法では，セグメントリストにセグメントが複数保持されるようになったらすぐ，セグメントサイズを拡大することになり，あまり効果がでないように思われるが，並列化可能なタスクが無くなったときにセグメントサイズを縮小し，含むことのできるフレーム数の上限を下げることにより図 6 の右側のセグメントツリーのように並列度を増加させることが可能となる．またサイズを小さくしてから次のタスク要求が来るまでセグメントのサイズが大きくならない場合，無駄に細粒度タスクを並列化してしまい，実行効率が低下してしまう可能性がある．そのために，セグメントが複数保持されるようになったら，セグメントサイズを拡大することが重要となる．

従来の方式では，セグメントのサイズは固定長であったため，不要になったセグメントを回収し，再利用することは容易であった．しかしながら本方式で未使用セグメントの割り当てを行うときに，最適なサイズのセグメントを見つけ出して確保するようにしたのでは，そのコストは大きなものとなってしまう．そこで，本研究ではセグメントを最大サイズ (4096byte) に固定し，フレームを割り当てる際に使用可能領域を制限することによって，可変長セグメントを実現した (図 7)．この方式によりセグメントの割り当て，解放にかかるコストの削減を行っている．

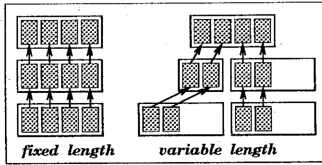

図 6: セグメント縮小の効果

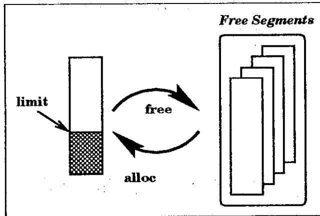

図 7: 可変長セグメントの実現

5 実験及び考察

本研究では 4.2 節で述べた方針によってセグメントサイズを動的に変更する並列 Lisp 処理系を富士通 AP1000+ 上に実装した. 実装の詳細は以下の通りである.

- ヒープ管理
  それぞれのセルのヒープ領域をローカルヒープとキャッシュヒープの 2 種類に分けリモートセルのローカルヒープ中にあるコンスセルを参照した場合は, 自分のキャッシュヒープにコピーし次回の参照からはキャッシュヒープ中のコンスセルを参照する方式 [2] を用いた.
- 負荷分散方式
  アイドル状態に陥ったプロセッサがランダムに他のプロセッサを選択し, タスク要求を送付するという動作をタスクが得られるまで繰り返す**ランダムポーリング方式** [7] によって実現した.

また, 実験環境に関しては使用したセル[1]は最大 64 台で, その構成は以下に示す通りである.

- CPU: SuperSPARC 50MHz
- メインメモリ: 64MB
- 2 次キャッシュ: 36KB

5.1 実験結果

実験のベンチマークとしては 4.1 節で述べた `fib`, `tak`, `mall` を用い, セグメントサイズを 128byte から 4096byte の範囲で切り替え, 速度向上を測定した. また本手法の有効性を示すために従来手法で最も実行効率の良かったセグメントサイズでの実行結果との比較を行った. その結果は `fib`, `tak` は図 8, `mall` は図 9 に示す通りになった. 図中で "dynamic" は本手法による速度向上を示している. また各ベンチマークでサイズ別に処理したセグメントの数は表 1 に示す通りであった.

[1] AP1000+において単位プロセッサのことをセルと呼ぶ.

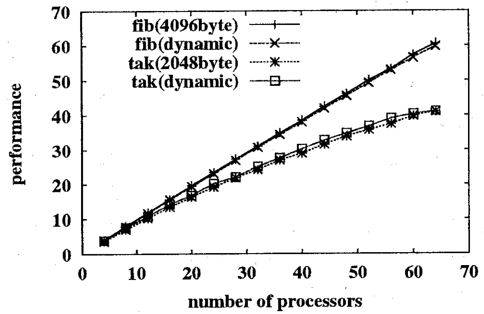

図 8: 実験結果 (`fib` 及び `tak`)

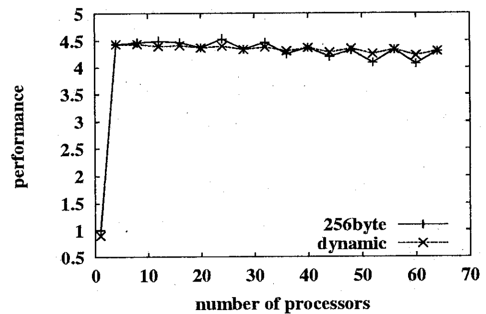

図 9: 実験結果 (`mall`)

表 1: 処理セグメント数 (セル 64 台による測定結果)

	fib	tak	mall
128 byte	670	681	27075
256 byte	64	64	30994
512 byte	64	134	30014
1024 byte	64	951	27291
2048 byte	6929	18399	25387
4096 byte	2999226	355146	15994
合計	3007017	375375	156755

5.2 実験結果の考察

図8, 9からもわかるようにどのアプリケーションでも最適なサイズで実行した場合と同等の性能向上が得られていることがわかる.

`fib` はもともと並列度の上限が高いため, セグメントが一定サイズより大きくなれば実行効率にさほど違いは見られず (図2) に実際, 本手法で最も多く処理したセグメントの殆んどが4096byteと最大サイズのものを処理していた (表1).

`tak` も並列度は十分に高いが, ある時点で並列化可能なフレーム数が減少し, セグメントサイズが大きすぎると無駄に並列性が抑制されてしまう. しかしながら, 本研究での提案手法では, 並列化可能なタスクが無い状態でタスク要求を受信した場合, セグメントサイズを縮小し並列度の増加を図る. その結果, 並列性が低下する場合でも効率良く実行できることが示された.

`mall` は並列性が出るのは計算の初期段階のみで, それ以降は並列性は皆無である. そのような場合, アイドルプロセッサが多発し, タスク要求のメッセージ数は多くなる. そのため, 並列度が無くてもメッセージチェックを行うときは, ほぼ毎回タスク要求が届いているため, セグメントサイズの縮小頻度が多くなり (表1), 結果として256byteで実行した時と同様の速度向上が得られている.

6 結論及び今後の課題

本稿の提案手法はキューマシン方式並列実行において, 実行中に並列化可能なタスクが無くなった時点でセグメントサイズを縮小することによってタスク粒度の上限を下げ事前にセグメントサイズを設定することなく効率良く実行が可能になることを示した. また, 本手法では並列度が高いアプリケーションにおいてもセグメントリストに複数セグメントが保持されるようになったときにセグメントサイズを拡大するため, 細粒度タスクの並列化を抑制し, 効率良く実行できることを示している.

本稿ではセグメントサイズを切替える際は, 拡大時に2倍にし, 縮小時には半分にしている. この縮小率, 拡大率を実行時の状況に応じて正しく変動させることによって, より効率良く実行することが可能になると考えられる. また, アプリケーションによってはどんなにセグメントサイズを拡大しても割り当てられるフレーム数が数個しかない場合, セグメントサイズを大きくしてしまうとメモリの利用率は悪くなり, 実行効率は低下する. 従ってセグメントサイズの拡大時に無駄に大きくならないように最大サイズを制限するようにすることも重要である.

今後は, 様々なアプリケーションを実行して, その特性を調べ, 上で述べたセグメントサイズの変動条件, 変動率を実行時に的確に選定することにより, より効率の良い並列実行を目指す.

謝辞

本研究を行うにあたり, 実験環境を提供して頂いた富士通研究所並列処理研究センターの皆様に深く感謝致します.

参考文献

[1] G. Aharoni, D. G. Feitelson, and A. Barrak. A Run-Time Algorithm for Managing the Granularity of Parallel Functional Programs. *Journal of Functional Programming*, Vol. 2, No. 4, pp. 387–405, 1992.

[2] Cormac Flanagan and Rishiyur S. Nikhil. pHluid: The design of a parallel functional language implementation on workstations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 169–179, May 1996.

[3] 前田敦司, 中西正和. キューマシン方式による並列Lisp処理系のスケジューリング手法. 電子通信情報学会論文誌 D-I, Vol. J80-D-I, No. 7, pp. 624–634, Sep. 1997.

[4] 前田敦司, 中西正和. 新しい計算モデル キューマシンとその並列関数型言語への応用. 情報処理学会論文誌, Vol. 38, No. 3, pp. 574–583, Mar. 1997.

[5] Eric Mohr, David Kranz, and Robert H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280, 1991.

[6] 西出康司, 前田敦司, 中西正和. 関数型言語の分散メモリ上での並列実行における粒度の選定. 情報処理学会プログラミング研究会報告, Vol. 96, No. 107, pp. 19–24, November 1996.

[7] Peter Sanders. A detailed analysis of random polling dynamic load balancing. In *IEEE International Symposium on Parallel Architectures, Algorithms, and Networks*, Kanazawa, Japan, December 1994.

[8] 竹内外史. 線形論理入門. 日本評論社, 1995.