記　号　処　理 72－1
（1994．1．14）

関数型言語の投機的実行と負荷分散

新田善久[†]　　武市正人[‡]

[†]東京大学教養学部　　[‡]東京大学工学部

筆者らは疎結合型 MIMD 計算機 AP-1000 上に遅延評価型の関数型言語処理系 PALADIN (PArallel LAnguage on DIstributed Network) を実装し，投機的実行による負荷分散方式の実験を行なった．本システムでは，投機的実行を制御するために優先度伝播を用いる．すなわち，実行結果が将来必要となるかどうかの見込みをタスクの優先度として表現し，優先度に基づいて計算資源を割り当てる．また優先度およびその変化は計算の依存関係にしたがって伝播させる．本論文ではまず投機的実行による負荷分散について論じ，次にその実装方式について述べる．最後に実験結果の考察を行なう．

Speculative Evaluation of Functional Programs with Load Balancing

Yoshihisa Nitta[†]　　Masato Takeichi[‡]

[†]The College of Arts and Sciences, University of Tokyo
3-8-1, Komaba, Meguro-ku, Tokyo, 153 Japan
[‡]Engineering Department, University of Tokyo
7-3-1, Hongou, Bunkyou-ku, Tokyo, 113 Japan

We implemented a lazy functional language system PALADIN (PArallel LAnguage on DIstributed Network) on MIMD computer AP-1000 to study the effects of speculative work with load balancing. We use priority propagation to control speculative works. In our system, the scheduler tries to assign resources to a task according to its priority which represents the necessity of its result. The priority and its change is propagated according to the dependency of the computation. We describe a way of load balancing by speculative work and show its implementation. Finally, we present the results of the experiments.

1 はじめに

疎結合型 MIMD 計算機において各プロセッサを効率良く動作させるには負荷分散が重要となる．

動的負荷分散方式ではタスクとその計算を行なうプロセッサの組み合わせを動的に決定する．この場合，タスクを投入する対象プロセッサは，各プロセッサ内のタスク数，空きメモリ容量，ネットワークの混み具合など各プロセッサの負荷状態を考慮して決定する．投入するタスクは，あまり他のプロセッサとは通信を行なわずに長く計算を行なうものが望ましい．しかしタスクの性質を詳しく解析するのはそれ自体にコストがかかるので，とりあえず見込みのありそうなタスクを選んで計算を並列に開始し見込みがないことが判明した時点で計算を打ち切る投機的実行方式が有望である．

筆者らは疎結合型 MIMD 計算機 AP-1000 上に遅延評価型の関数型言語処理系 PALADIN (PArallel LAnguage on DIstributed Network) を実装し [1, 2]，投機的実行による負荷分散方式の実験を行なった [3]．本システムでは，投機的実行を制御するために優先度伝播を用いる．すなわち，実行結果が将来必要となるかどうかの見込みをタスクの優先度として表現し，優先度に基づいて計算資源を割り当てる．また優先度およびその変化は計算の依存関係にしたがって伝播させる．本論文ではまず投機的実行による負荷分散について論じ，次にその実装方式について述べる．最後に実験結果をもとに考察を行なう．

2 投機的実行

投機的実行 [4, 5] とは，計算の並列性を高め，負荷分散を行なう方式の一つである．すなわち，将来必要となる見込みの高い計算を空いたプロセッサに早めに投入して計算しておき，全体としての計算速度を高める方法である．したがって，計算には必須 (mandatory) 計算と投機的 (speculative) 計算の 2 種類が存在する．

筆者らは核言語として関数型言語を選び，投機的実行による動的負荷分散の研究を行なった．

関数型言語は計算の副作用がないので，投機的実行で無駄な計算を行なっても，その結果から生じた影響を元に戻す必要はない．また，計算の実行順序の制約がないので，スケジューリングの自由度が高く，タスクを自由にプロセッサに割り当てることができる．

投機的実行方式では

- 投機的に投入したタスクは本来必要とされる計算を妨げない
- タスクは必要でないことが判明した時点で計算を打ち切ることができる

という機構が必要となる．遅延関数型言語の特徴である評価方式は式の評価を関数閉包を作る段階までで一旦凍結し，その後その関数閉包の値が必要となった時点で計算を再開するというものである．この方式は逐次実行においては不要な計算の除去という意味を持つが，並列実行においては投機的に投入したタスクを停止させる機構として機能する．つまり任意の時点で関数閉包を作ることによってそれ以降の計算を凍結し，必要になった時点で途中までの計算結果を無駄にすることなく再開できることになる．したがって，遅延評価方式は投機的実行を扱うのに適しているといえる．

また，疎結合型並列計算機上ではデータの共有は複製によって行なうが，大きなデータ全体を一度に複製するのは効率的ではなく，要求駆動によって部分的にデータを複製する方式が望ましい．遅延評価ではデータに対する計算が遅延されて要求駆動的に計算が進むので，効率の良い複製を自然に実現できる．このように遅延評価方式は疎結合型計算機上のデータ構造を扱うのにも適している．

2.1 優先度伝播

本システムでは，優先度伝播 (priority propagation) を用いて投機的計算を制御する．すなわち，投機的計算と必須計算をタスクの優先度の違いで表現し，優先度の変化は計算の依存関係にそって伝播させる．

あるタスクが計算する式の結果が将来必要となるかどうかの見込みがそのタスクの優先度であり，計算資源は優先度に基づいて割り当てられる．見込みのある計算ほど優先度が高くなり多くの計算資源が割り当てられるので実行は早く終了する．計算の進行によって見込み情報は変化するが，それに対応してタスクの優先度も動的に変化し，計算の依存関係にしたがって伝播する．

必須計算を行なうタスクは最高レベルの優先度 $priority_{MAX}$ を持ち，投機的計算は $priority_{MAX}-1$ から最低レベル $priority_{MIN}$ までの優先度を持つ．ある一定以上の優先度を持つタスクだけが新しい投機的計算を投入できる．投機的計算は親タスクよりも低い優先度で投入されるので，投機的計算が無限に増殖することはない．投機的実行タスクの優先度は計算結果が必要でないことが判明した時点で $priority_{MIN}$ に落ちる．

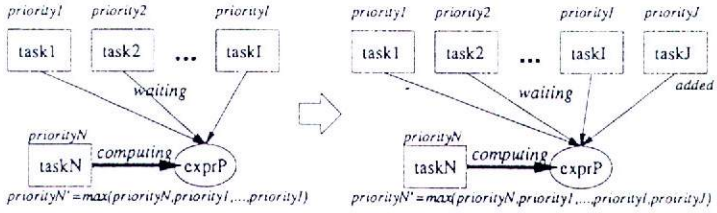

図 1: 優先度の変化

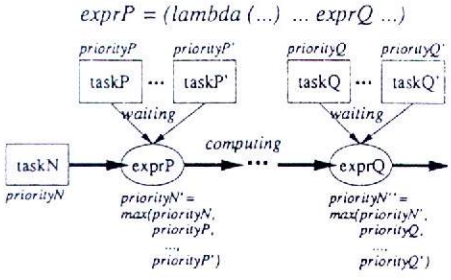

図 2: 優先度の伝播

ある式を計算するタスクの優先度は，その計算結果を待つタスクの優先度の最大値となる (図 1参照)．すなわち式 $exprP$ を計算中のタスク N の優先度は $exprP$ の計算結果を待つタスク $1,\ldots,I$ の優先度と自分自身の優先度の最大値となる．ここで新たにタスク J が $exprP$ の結果を必要とするとタスク J の実行はブロックされ，タスク N の優先度 $priorityN$ はタスク J の優先度も考慮して

$$\max(priorityN, priority1, \ldots, priorityI, priorityJ)$$

を計算した結果となる．古い $priorityN'$ よりも $priorityJ$ が大きい場合はタスク N が一時的に加速されることになる．

タスク N は計算の依存関係にしたがって複数の式を順に評価してゆくが (図 2参照)，タスク N の新しい優先度はこの計算の依存関係にそって伝播する．すなわち，タスク N が式 $exprQ$ の計算している時に式 $exprP$ において優先度の変化が起きた場合は，$exprP$ 以後の式 ($exprQ$ を含む) を評価する優先度はその影響を受ける．タスク N が式 $exprQ$ の評価を終えたときは，$exprQ$ の計算結果を待つタスクからの影響を排除した値に戻る．

2.2 優先度伝播の実装

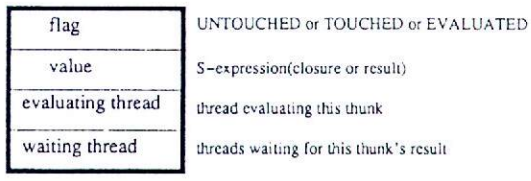

図 3: 優先度伝播を行なう thunk 型データ構造

遅延された計算を表現するために，図 3に示す thunk 型のデータ構造を用いる．thunk 型は，式が既に計算されたか否か計算中かを示すフラグと，式または計算結果を保持するための場所を持つ．さらに，優先度伝播を実現するために，

- 式を計算しているタスク
- 計算結果を待っているタスク群

を記録する場所を持つ．この情報を参照して，式の計算結果を待つタスクや計算中のタスクに変化 (待ちタスクの増減，待ちタスクの優先度変更) の影響を伝播する．

タスクを表す構造は図 4に示すように，

- 計算中の thunk 型データ群
- それぞれの thunk 型データの計算結果を待つタスク群
- それぞれのタスクの優先度

をリストとして保持する場所を持つ．優先度の変化はメッセージとしてタスクに伝達されるので，リスト中から必要な部分を選び出して優先度の再計算を行なう．しかしこの方式では，式の計算を試みる度に thunk を作り出し，その thunk 型データ構造をタスク構造内に記憶する操作が必要となる．これでは計算のオーバーヘッドがかなり大きくなるので，コンパイルなどの前処理の時に他の

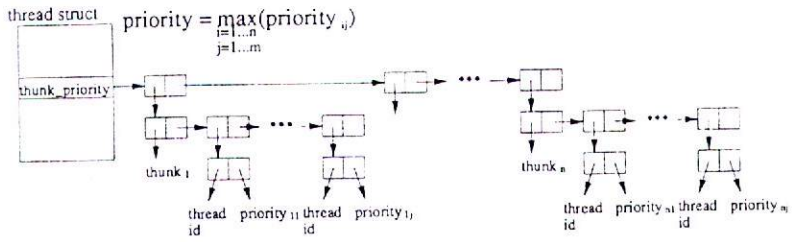

図 4: 優先度伝播を受けるタスク構造

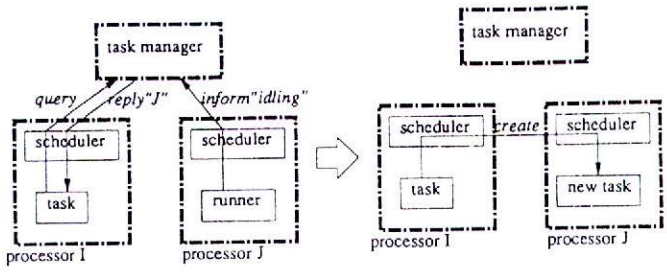

図 5: タスクマネージャ

タスクと共有しない式を検出し，無駄な登録操作を省くようにすべきであろう．

3 負荷計算

計算タスクは各プロセッサ上にスレッドとして実現する．新たにタスクを生成する場合は負荷が軽いプロセッサを選ぶ．プロセッサの負荷情報を厳密に計算するには，以下の情報を基にする必要がある．

- スレッド数 — 最近の一定時間内に生成されたスレッド数，現在存在しているスレッド数とその優先度，現在アクティブなスレッド数とその優先度
- 通信量 — 最近の一定時間内に通信されたパケット量，データの輸入・輸出表内に保持されている外部参照ポインタのエントリ数 (今後発生する可能性のある通信量を反映すると予想される)
- メモリ使用量 — 最近の一定時間内にガベージ・コレクタが起動された回数
- データ移動のコスト — 他のプロセッサで計算タスクを生成するにはデータの移動が必要で，それ自体に高いコストがかかる．したがって，プロセッサの負荷にある程度以上の差が存在しない場合にはプロセスの生成は行なうべきでない．

しかし，負荷計算をあまり厳密に行なっていてはそれ自体がボトルネックとなり計算効率を落してしまう．したがって，近似的な計算が必要とされる．

3.1 負荷計算の近似

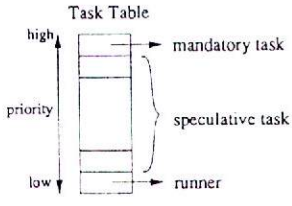

図 6: タスク表

N プロセッサによる並列実行では，各プロセッサの状態の管理のみを行なうプロセッサを 1 個用意し，この上でタスクマネージャを動作させる (図5参照)．タスクマネージャは各プロセッサの負荷状態を保持しており，各プロセッサからの問い合

わせに答える．新たに投機的計算を投入しようとしたタスクはタスクマネージャへ問い合わせを行ない，負荷の軽いプロセッサの番号を受けとる．低負荷のプロセッサが存在しなければ同一プロセッサで実行するように遅延計算の構造を生成する．

タスクマネージャが各プロセッサの状態を常時調べるのはオーバーヘッドとなるので，低負荷プロセッサが自己申告することにした．すなわち，図6に示すように各プロセッサ上に最低の優先度でタスク要求スレッド(伝令 =runner)を動作させておく．他にアクティブなタスクがなくなった時点でこれに制御が移りタスクマネージャにプロセッサの状態を通知する．したがって，タスクマネージャに状態が通知されるのは，すべてのタスクが終了した場合と，すべてのタスクが受信待ちでブロックされた場合である．本来はブロックされているタスクの数と優先度も状態通知の内容に加えることが望ましいが，本研究では負荷計算を簡単にするためにアクティブなタスクがないという事実のみを伝えることにした．

4 投機的計算の投入

他のプロセッサに投入するタスクは，それ以外のタスクとの通信量が少なく寿命の長いタスクであることが望ましい[6]．投入対象タスクの選択は処理系が自動的に判断するのが理想的であるが実装には困難な点が多いので，プログラマが投機的実行の開始をプログラム中にヒントとして記述することにした．プログラマによる記述はあくまでもヒントであり，その記述によって常に投機的実行が投入されるわけではない．投入対象プロセッサは処理系が自動的に選択する．すなわち，投機的実行がヒントとして指定された式の評価は，低負荷のプロセッサが存在すれば投機的計算となり，存在しなければ同一プロセッサ上での遅延評価となる．

投機的実行をヒントとして指定する関数を下に示す．これらの式を評価するタスクの優先度が新しい投機的計算を投入できるほど高く，かつ，低負荷のプロセッサが存在する場合に投機的実行を行なう．

- (`if*` *cond then else*) — *cond* 部, *then* 部, *else* 部を並列に評価する．*cond* 部の計算の優先度はこのS式全体の評価の優先度と等しく，*then* 部, *else* 部の計算の優先度はそれよりも低い．*cond* 部の計算が終った時点で *then* 部と *else* 部の一方の優先度がS式全体と等しくなり，他方は $priority_{MIN}$ に落ちる．
- (`head*` *list*), (`tail*` *list*) — データの構成子 cons で遅延評価を実現しているために，リストの操作時に遅延計算の値を求める計算が行なわれる．リストのある要素を操作したときは，それ以降の要素についても優先度を下げながら順番に投機的に計算を投入する．
- (`map*` *func list*) — *list* の各要素への *func* の適用を投機的に行なう．map 関数はリストの各要素に関数を作用させることが目的であるので並列性を引き出せる可能性が高い．

5 実験

```
(define limit 7)
(define QUEENS (lambda (n) (queens n n nil)))
(define COLLECT (lambda (l) (collect l (length l) nil)))
(define queens (lambda (m num board)
  (if (eq m 0) board
    (if (> m limit)
        (map*
          (lambda (n)
            (if (safe? board n 1)
                (queens (- m 1) num (cons! n board))
                nil))
          (mklist 1 num))
        (map!
          (lambda (n)
            (if (safe? board n 1)
                (queens (- m 1) num (cons! n board))
                nil))
          (mklist 1 num))))))
(define safe? (lambda (p q n)
  (if (null p) t
      (and (neq (head p) q)
           (neq (head p) (- q n))
           (neq (head p) (+ q n))
           (safe? (tail p) q (+ n 1))))))
(define collect (lambda (l n ans)
  (concat
    (filter!
      (lambda (x)
          (if (eq n 1) (if (null x) ans (cons! x ans))
          (collect x (- n 1) ans)))
      l))))
(define mklist (lambda (s e)
  (if (> s e) '()
      (cons s (mklist (+ s 1) e)))))
(define map (lambda (f l)
  (if (null l) '()
      (cons (f (head l)) (map f (tail l))))))
(define map! (lambda (f l)
  (if (null l) '()
      (cons! (f (head l)) (map! f (tail l))))))
(define filter! (lambda (f l)
  (if (null l) '()
      (let ((x (cons! (f (head l)) (filter! f (tail l)))))
        (if (null (head x)) (tail x) x)))))
```

図 7: 8-Queens の全解探索プログラム

図7に示すプログラムを実行し 8-Queens の全解探索 (`COLLECT (QUEENS 8)`) にかかる時間を計測した．プログラムからわかるように実験ではmapなどの本来組み込みであるべき関数も解釈系で実行させている．`cons!` は遅延型でないリストの構成子である．関数 `QUEENS` は N-Queens を計算

表 1: 8-Queens 問題の全解探索 (リストへの収集時間も含む) (単位:sec)

マシン (プロセッサ数)	逐次処理	並列処理		
	$limit \geq 8$	$limit = 7$	$limit = 6$	$limit = 5$
AP-1000(1)	71.07* + 30.47** = 101.54	—	—	—
AP-1000(3)	—	101.57	104.60	101.93
AP-1000(4)	—	92.21	101.37	103.60
AP-1000(8)	—	58.56	84.58	73.17
AP-1000(10)	—	39.24	80.67	69.17
AP-1000(16)	—	39.24	75.47	74.34
AP-1000(18)	—	39.94	64.67	75.11
AP-1000(24)	—	39.24	58.19	76.23
AP-1000(32)	—	39.24	41.55	78.68
AP-1000(40)	—	39.24	31.77	71.66
AP-1000(48)	—	39.24	31.35	70.03
AP-1000(54)	—	39.24	31.48	68.00
AP-1000(64)	—	39.24	31.36	51.25
SS10	29.15* + 12.43** = 41.58	—	—	—
SS/IPX	54.00* + 23.47** = 77.47	—	—	—
Sun3	301.32* + 118.21** = 419.53	—	—	—

* QUEENS の実行時間, ** COLLECT の実行時間

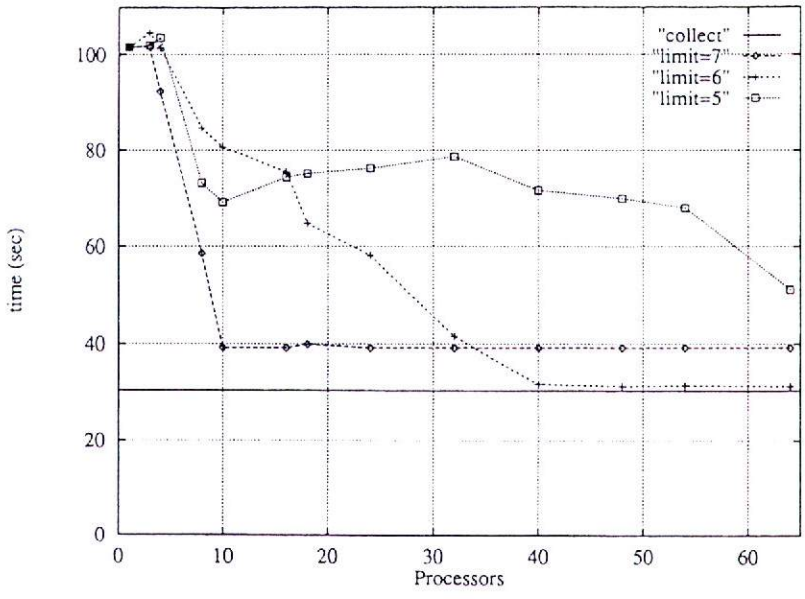

図 8: 8-Queens 問題の全解探索 (並列版) の結果グラフ

し，関数 COLLECT は計算されたデータを収集してリストにする．

単一プロセッサでの実験では空きプロセッサの探索は行なわない．プロセッサ数が 2 のときは一方のプロセッサはタスクマネージャになるので空きプロセッサは存在せず投機的実行は投入されない．したがってこの場合の空きプロセッサの探索は単にオーバーヘッドとなる

AP-1000 の単一プロセッサ上での実験ではQUEENS に 71.07 秒，COLLECT に 30.47 秒かかった．関数 COLLECT は逐次的に動作するので，複数プロセッサ上の分散データに対して実行した場合にも

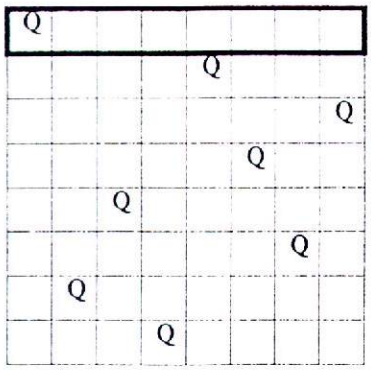

図 9: チェス盤

同程度の時間がかかると予想される.

並列プロセッサによる実験結果が表 1であり, 縦軸に計算時間 (ゴミ集めは含まない), 横軸にプロセッサ数をとったグラフが 図 8 である.
並列プロセッサによる実験では QUEENS の計算と COLLECT の計算が並行して進むので両者の計算にかかる時間を分離するのは困難である. そこでグラフでは単一プロセッサ時に COLLECT にかかった時間を実線で示し, その差で QUEENS の計算時間を近似値として示すことにした.

図 9に太線で示すように, プログラムでは盤の横方向のどれに駒を置くかを並列に調べる. limit の値で上から何行目の配置までを並列に調べるかを制御している. limit の値が 7, 6, 5, ... と減少するにしたがって, 盤の 1 行目, 2 行目, 3 行目と並列に配置する行が増えるので, 投機的に投入できるタスク数は 8, 8^2, 8^3, ... と増加する. 並列実行の粒度はそれぞれ $70.07/8 = 8.76$ 秒, $70.07/8^2 = 1.09$ 秒, $70.07/8^3 = 0.14$ 秒, ... 程度となる.

limit=7 ではプロセッサ数が増えるにしたがって計算時間は減少するが, プロセッサ数 10 で速度最大になり, それ以降は変化しない. プロセッサ数 10 のときは 8 個の投機的実行タスクがすべて別プロセッサに投入された状況である.

limit=6 ではプロセッサ数が増えるにしたがって計算時間はゆっくり減少し, 32 個で limit=7 の最良値と並ぶ. プロセッサ数 40 でほぼ最高の速度となり, それ以降はほとんど速度変化しない.

limit=7 および 6 の振舞いの差は, 並列タスクの粒度と並列度の組み合わせによっておきる. どちらの場合でも最高速度に達した時には QUEENS と COLLECT の計算にかかる時間が,

粒度 + COLLECT にかかる時間

となるので妥当な値であると考えられる.

limit=5 ではプロセッサ数が 8 ～ 10 の時は, limit=6 のときよりも計算時間が減少する. これは不充分な数のプロセッサしかないので limit=6 では 1.09 秒粒度のタスクの偏りができてしまうが, limit=5 では 0.14 秒粒度のタスクが多数あるのでたとえ分布が偏っても 1.09 秒粒度のタスクよりも適切に配置されたためと推測される. limit=5 ではプロセッサ数が 10 ～ 50 の時は同条件のプロセッサ数 8 ～ 10 の時よりも計算速度は悪くなる. これは, 0.14 秒粒度のタスクの数に対してプロセッサ数が中途半端に多いため, 0.14 秒粒度のタスクがプロセッサ間で交互に生成され, 通信のコストのオーバーヘッドがかかっているためである. プロセッサ数が 50 以上では 0.14 秒粒度のタスクが一様に分布できるようになり計算時間は小さくなっていく.

以上の結果から, 負荷分散を効率的に行なうためにはタスクの粒度とプロセッサの数を適切に対応させる必要のあることがわかる.

小粒度のタスクは並列実行によって得られる利得と生成に必要なコストとの差が小さいので, 充分なプロセッサ数が存在しない限り生成するべきではない. また, 小粒度の並列タスクを多数生成すると, 通信のコストが増えるのでプロセッサ数が多少増えても速度は改善されなくなることもわかる.

6 まとめ

疎結合型 MIMD 計算機上で遅延評価型の関数型言語の負荷分散を投機的実行によって行なう方式を提案し, 基本的な実験を行なった. 必須計算と投機的計算を優先度という概念で統一的に扱うことができること, および, 遅延評価機構を拡張して優先度伝播を自然に実現できることを示した.

投機的実行による動的負荷分散の実験では, 8-Queens の全解探索にかかる時間を並列実行単位の粒度とプロセッサ数を変化させて計測した. 提案した方式による投機的実行によって, プロセッサ数と粒度の関係を適切に選べばプログラムの評価を高速化できることを示した.

現在はタスクマネージャはアクティブなタスクが存在しないプロセッサを無条件に投機的計算の投入が可能であるとみなしている. しかし, プロセッサ上でブロックされているタスクの状態なども評価に入れたり, プロセッサ間の負荷の差を考慮するなどして, より適切な負荷計算を行なう必要がある. そのためには, 計算に用いる要素が増

えても速度が落ちないような適切な近似計算を開発する必要がある.

また，優先度伝播の処理のために低下した処理系の実行効率を向上させるためには，コンパイル時などに他のタスクと共有しない式を検出して無駄な操作を省き，不必要なオーバーヘッドを減らす必要がある.

さらに，現在はプログラマが投機的実行のヒントを記述しているが，将来はプログラムの性質をシステムが調べてプロセッサ数に適した粒度を自動的に判別し，投機的実行を投入できるように改善してゆく必要もあろう.

7 謝辞

本研究の一部は富士通並列処理研究センターを利用して行なったものである．AP1000 およびその開発環境を提供していただき感謝いたします.

参考文献

[1] 新田善久，武市正人: 疎結合型並列計算機における遅延関数型言語の効率的実行に関する研究, 日本ソフトウェア科学会関数プログラミング研究会, 1992.12.16.

[2] 新田善久，武市正人: 疎結合型並列計算機における遅延関数型言語の効率的実行に関する研究 (II), 日本ソフトウェア科学会関数プログラミング研究会, 1993.3.5.

[3] 新田善久，武市正人: 疎結合型 MIMD 計算機における関数型プログラムの投機的実行と負荷分散, 情報処理学会論文誌 (投稿中)

[4] R. B. Osborne: *Speculative Computation in Multilisp*, Parallel Lisp: Languages and Systems, T. Ito, R. H. Halstead, Jr. (Eds), Springer-Verlag, 1990.

[5] R. H. Halstead, Jr.: *New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools*, Parallel Lisp: Languages and Systems, T. Ito, R.H. Halstead, Jr. (Eds), Springer-Verlag, 1990.

[6] 前川守・所真理雄・清水健太郎編: 分散オペレーティングシステム — *Unix* の次にくるもの, 共立出版, 1991.