プログラミング 18－18
(1998. 3. 23)

並行オブジェクト指向言語における再帰にともなうデッドロックの回避機構の設計と実装

柳川 和久　佐藤 直人　大澤 範高　弓場 敏嗣
電気通信大学大学院情報システム学研究科
〒 182–8585 東京都調布市調布ヶ丘 1–5–1
Tel 0424–83–2161
Email katze@yuba.is.uec.ac.jp

あらまし

オブジェクト単位の相互排除を行なう並行オブジェクト指向言語では, 再帰的なメソッド呼出しによりデッドロックが起こり得る. 我々はこの再帰にともなうデッドロックを排除するプログラマに対して透過な仕組み—キー/ロック法—を提案する. キー/ロック法ではメッセージに排他的なメソッドに一意に対応する鍵を与え, この鍵を持ったメッセージを特別に受理する. この手法は既存の研究と比べメッセージ送受のオーバヘッドを小さくでき, 特にスレッドの分岐数が多い場合に有効である. キー/ロック法を並行言語 SR により実装し, 性能評価を通して同手法の有効性を検証した.

キーワード　並行オブジェクト, 再帰, デッドロック

Evaluation of Avoidance Method of Recursive Deadlock in Concurrent Object-Oriented Programming

K. Yanagawa　N. Sato　N. Osawa　T. Yuba
Graduate School of Information Systems, the University of Electro-Communications
1–5–1 Chofugaoka, Chofu, Tokyo 182–8585
Tel 0424–83–2161
Email katze@yuba.is.uec.ac.jp

Abstract

This paper proposes a new scheme, called *Key/Lock* scheme, for handling recursive method calls in concurrent object-oriented programming. The Key/Lock scheme associates each method call with an unique *key*, so that recursive method calls with the key can be identified and accepted by the receiver objects. Compared with the existing schemes, the Key/Lock scheme needs less overheads in adding and extracting extra informations, attached to messages, for identifying recursive method calls. We have integrated our scheme to the concurrent object-oriented language SR, and proved its effectiveness through performance evaluation.

Keywords　Concurrent Object, Recursion, Deadlock

1 はじめに

多くの並行オブジェクト指向言語オブジェクト単位で相互排除を行なう．これらの言語では排他的なメソッドの実行中にメッセージが到着すると，メソッドが終了するまで到着したメッセージは受理されない．このため再帰的なメソッド呼出しはデッドロックを引き起こす．例えば複数の小データベースから構成されるデータベースシステムを考える（図 1）．ここで各小データベースがある人の年齢，性別，職業，家族構成を保持しているものとする．ある人の情報を家族の情報を含めて得るために問い合わせを行なうと，E は家族の情報を得るために A に問い合わせを行なう．この問い合わせはブロックされデッドロックが生じる．

このような再帰にともなうデッドロックを防ぐ手法は二通りある：一つはプログラマが再帰的なメッセージの受理を記述する方法である．実際にはプログラム中に再帰的なメッセージを特別に受理する，もしくは相互排除を解除して任意のメッセージを受理するようにするというコードを埋め込む．しかしプログラマが再帰的なメッセージへの対処を記述する場合，入力に依存して動的に変化する複雑な再帰パターンがあるとプログラムが困難になるという問題がある．図 1 の例で言うと，問い合わせの結果データベースの書き換えが起こり得るので，一つの問い合わせが終るまで他の問い合わせの受け付けを待たせる必要がある．よって再帰的なメッセージのみを識別できるようにし，特別に受理をするコードが必要になる．このためプログラムは複雑になる．

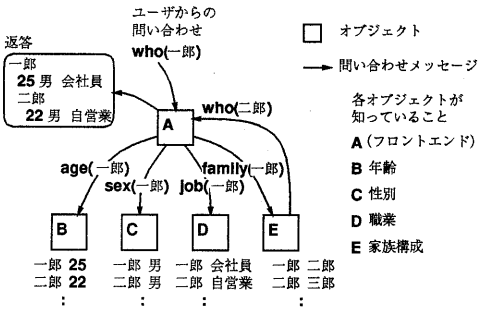


図 *1* 再帰的に問い合わせをするデータベース：家族の情報を得るために E は A に再帰的な問い合わせを行なう

もう一つはシステムがプログラマに対して透過に再帰への対処を行なう方法である．システムはメソッド呼出しの履歴を保存しておき，この情報をもとに再帰的なメッセージを識別し，特別に受理する．プログラマは逐次プログラミングの場合と同様にプログラムを書ける．しかし [1] を始めとする既存の手法ではオーバヘッドが大きく，性能的に問題がある．我々は

我々はプログラムの読みやすさを取ってシステムが再帰への対処を行なう手法を取ることにし，この問題を解決するための別の手法を提案した [7]．この手法は既存の手法よりオーバヘッドが小さい．今回はこのデッドロック回避手法の予備的な実装を行ない，性能評価を行なった．以下 2 節で従来の研究の問題をまとめ，3 節で提案する再帰にともなうデッドロックの回避手法を説明する．4 節で予備的な実装による評価の結果を示した後 5 節で評価の結果を踏まえた考察をする．最後にまとめと今後の課題を述べる．

2 再帰を許す並行オブジェクト指向言語

本節では再帰を許す並行オブジェクト指向言語を概観し，従来の研究の問題点をまとめる．

まず再帰を次の三つに分類する．

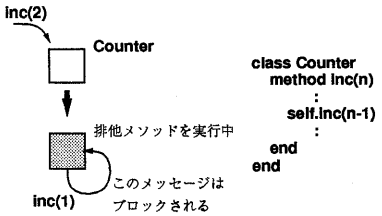
- 直接再帰：自分自身のメソッドの呼出し（図 2a）．
- 間接再帰：他のオブジェクトを経由したメソッド呼出し（図 2b）．
- 分岐を含む間接再帰：間接再帰で並行したメソッド呼出しを含む場合（図 2c）．

相互排除を解除して任意のメッセージを受理する手法にはプログラムをわかりにくいものにするという問題がある．これに対して直接再帰のみを特別に扱う Obliq [2] やネストしたコールを一連の論理的なスレッドとして捉える Hybrid [5], Java [4] などは分岐を含む間接再帰には対処できない．

分岐を含む間接再帰に対応する手法として multi-ported object [1], named-thread [1] が提案されている：Multi-ported object ではメソッド起動毎に新たにメッセージを受信するポートを作り，メッセージに自分自身と最新のポートの組を添付する（ポートバインディングマップ）．送信オブジェクトは受信オブジェクトの最新のポートに向けてメッセージを送り，受信オブジェクトは最新のポートに到着するメッセージのみを処理することで再帰的なメッセージを特別に処理できる．しかし送り先のオブジェクトの最新のポートを検索する時間はメソッドコールの深さに比例して長くなる（図 3）．

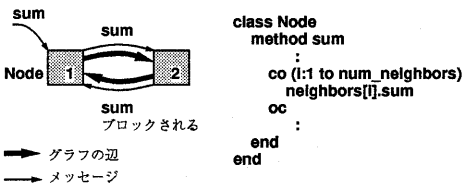
Named-thread では Hybrid と同様にネストしたコールを一連の論理的なスレッドと考える．各スレッドには一意なスレッド識別子（以下スレッド ID）が与えられる．スレッドが分岐した場合にはスレッド ID が拡張される．この手法ではスレッド間の関係を確認するために，

再帰によって値を書き換えるカウンタ.

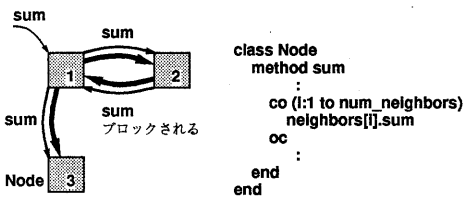


a 直接再帰の例: 再帰するカウンタ

グラフ上のノードが持つ値全ての和を求める.



b 間接再帰の例: 閉路があるグラフでの Graph Sum



c 分岐を含む間接再帰の例: 閉路があるグラフでの Graph Sum

図 *2* 再帰にともなうデッドロックの分類

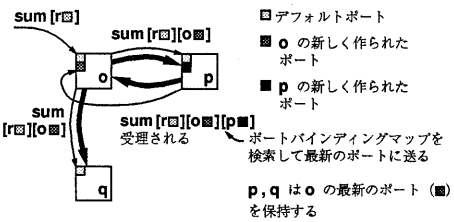


図 *3* Multi-Ported Object による Graph Sum の例

メッセージ受信時にスレッド ID の比較を行なう. スレッド ID はスレッドの分岐毎に拡張されるので, この比較時間はスレッドの分岐回数に比例する（図 4）.

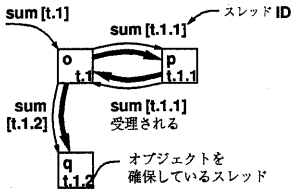


図 *4* Named-Thread による Graph Sum の例

Multi-ported object, named-thread はともオーバヘッドについての考察が十分でない. 特に named-thread ではメッセージ受信時のオーバヘッドがスレッドの分岐回数のみに比例するため, スレッドが分岐する間接再帰の場合, メッセージ受信時のオーバヘッドは再帰の深さに比例する. 我々はよりオーバヘッドの小さい効率的なアルゴリズムを提案する.

3 キー/ロック法によるデッドロック回避

本節では分岐を含む間接再帰を可能にするキー/ロック法の仕組みを概説する. なお以下の議論で排他メソッドとは相互排除を必要とするメソッドであり, コールとは返答待ちを行なう同期型のメッセージ送信の事である.

3.1 アルゴリズムの概略

デッドロック回避法キー/ロック法は以下のようなアルゴリズムである:

1. 排他メソッドの起動毎に一意な鍵が生成される.
2. 排他メソッドからの送信メッセージには鍵を添付し, この鍵を持ったメッセージが到着した場合特別に受理する.

この様子を先のカウンタの例を用いて 図 5 に示す. キー/ロック法ではオブジェクトは次の二つの状態のいずれかとする:

- 初期状態: 実行中の排他メソッドが存在しない状態.
- 排他状態: 排他メソッドを実行中の状態.

初期状態のオブジェクトが排他メソッドをコールされると

1. 排他メソッドの起動時にメソッドに一意に対応する鍵を生成し,
2. これをスタックに保存し,
3. 排他状態へ遷移する.
4. 排他メソッドが送信するコールメッセージにはスタックの先頭の鍵を添付する.

表 1 関連研究

手法	特徴
ABCL/f [6] 等: メソッド実行中の適当な時点で相互排除を解除し任意のメッセージを受理する	分岐を含む間接再帰が可能. 論理的なスレッドを認識できない.
Obliq [2] 等: 直接自己再帰を特別に処理する	直接再帰のみが可能.
Hybrid [5], Java [4] 等: ネストしたコールを一つの論理的なスレッドと考えスレッド単位で排他制御をする	分岐を含まない間接再帰のみが可能.
Multi-Ported Object [1]	分岐を含む間接再帰に対応. メッセージ送信時のオーバヘッドが大きい.
Named-Thread [1]	分岐を含む間接再帰 に対応. メッセージ受信時のオーバヘッドが大きい.

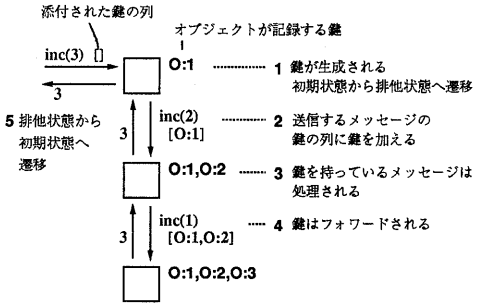


図 5 キー/ロック法のアルゴリズム

一方排他状態のオブジェクトはコールを受けると

1. 受信したメッセージがスタックの先頭の鍵を持つかどうかを調べ,
2. 鍵を持っている場合メッセージを受理し,
3. メソッドを起動する. メソッドが排他メソッドであった場合にはさらに鍵の生成と記録を行なう.
4. 全ての排他メソッドが終了すると初期状態へ遷移する.

各メソッドはコールされた時に添付されていた鍵をフォワードするので, 間接的な再帰にも対処できる.

スレッドが分岐する場合, 分岐したスレッドはそれぞれ同じ鍵を持つ. しかしあるスレッドがオブジェクトに入った場合, そのオブジェクトでは新たな鍵が生成されスタックの先頭に入るので分岐したスレッド同士は干渉しない.

3.2 アルゴリズムの効率化

キー/ロック法を素朴に実装した場合各種のオーバヘッドが大きく, 性能的に問題になってくる. なかでも特に以下の四つが問題である:

メッセージへの付加情報の増大
排他メソッドを起動するたびに鍵が生成され, これがフォワードされていく. この結果排他メソッド呼出しの深さに応じてメッセージへの付加情報の量が大きくなる.

オブジェクトへの付加情報の増大
オブジェクトは有効な鍵を記録しておく必要がある. 一般の再帰を許している事を考えると, 再帰の深さに応じてオブジェクトが記録する鍵の量が増える.

メッセージ受信時のオーバヘッド
メッセージ受理の際, 鍵の有効性を調べるという動作が加わる. メッセージに付加された鍵が増えるに従いこの処理にかかる時間は増加する.

メッセージ送信時のオーバヘッド
メッセージ送信の際, 添付されている鍵の列を複製する事が必要になる. メッセージに付加された鍵が増えるに従いこの処理にかかる時間は増加する.

実際にキー/ロック法を利用するにはアルゴリズムの効率化を図り, これらのオーバヘッドを削減する必要がある.

メッセージへの付加情報の削減

コールは返答待ちを行う同期型のメッセージ送信である. このため返答待ちメソッドのスタックの先頭にあるメソッドに対応する鍵を持つメッセージのみが受理され得る. よってある時点で有効な鍵は各オブジェクトについて高々一つだけである. このことを利用するとメッセージにつける鍵を減らせる. 具体的には, メッセージは各オブジェクトで最後にコールを行ったメソッドに対する鍵だけを保持すれば良い.

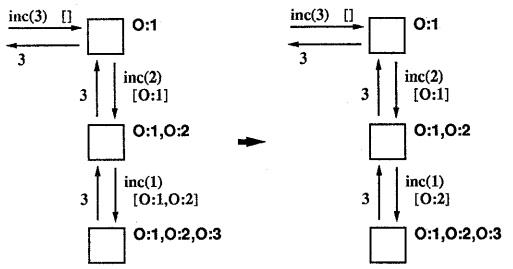


図 6 メッセージへの付加情報の削減

オブジェクトへの付加情報の削減

コールが同期型である事から, 返答待ちメソッドのスタックの深さとオブジェクトの識別子の組はある時点でシステム中で一意である事が保証される. 鍵をオブジェクトの識別子とスタックの深さの組で表す事にすると, 鍵の識別は現在のスタックの深さとの比較で行える. この場合オブジェクトは返答待ちスタックの深さだけを記録しておけば良い.

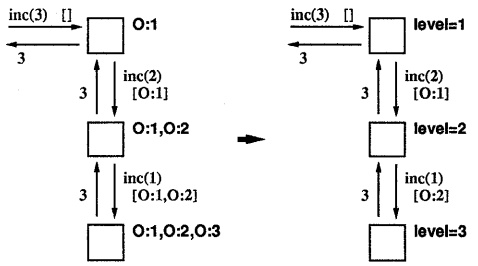


図 7 オブジェクトへの付加情報の削減

鍵の識別時間の削減

メッセージに添付する鍵の表現を工夫する事により, 鍵の識別時間は定数時間にできる. メッセージに新たに鍵を付け加える場合, 新しい鍵を今までの鍵の列の最後尾につなげるものとする. こうすると一度メッセージに加えられた鍵は, その鍵が無効になるまでメッセージ上で同じ位置に存在する. この性質は メッセージサイズの削減の節で述べた手法を適用しても成立する. よってオブジェクトが自分が鍵を加えた位置を記録しておく事で, 一定の時間で鍵の識別を行える.

残る問題—鍵の複製時間

鍵の識別を定数時間にするために, 鍵の列は配列で実現する. このため新たな鍵の添付や鍵の列の書き換えを行なうには, メッセージ送信の際に鍵の列を複製する事が不可欠となる. 他の手法ではこの付加情報の複製を本質

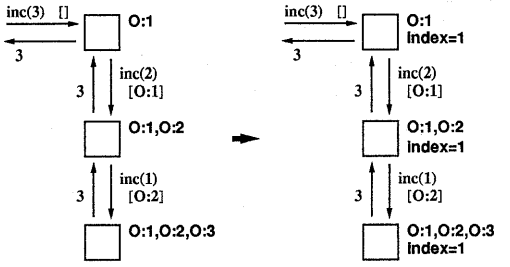


図 8 鍵の識別時間の削減

的には必要としない. このためスレッドの分岐が少なくコールのネストが深い場合キー/ロック法が性能的に不利になる.

4 性能評価

キー/ロック法を組み込んだオブジェクト指向言語 SR/KL を実装し, そのオーバヘッドを評価した. 以下に実装の概要と性能評価の結果を示す.

4.1 キー/ロック法を付加した言語 SR/KL の実装

実装はオブジェクトベースの並行言語 SR [3] へのトランスレータという形をとった. SR では他の言語のモジュールに相当する 'resource' という単位でプログラムが分割され, 計算の実行は相手 resource で定義されたオペレーション（手続き）の呼び出しという形をとる. そこで各リソースをオブジェクト, オペレーションをメソッドと捉え, メソッドのコードの前/後処理としてキー/ロック法を実装した. 鍵の添付は鍵の配列を用意し, これをメソッドコールの引数に加える事で実現している.

SR/KL で書いた graph sum プログラムの例を 付録 A に示す. このコードは次の性能評価で用いたものである.

4.2 性能評価

性能評価には graph sum の例を用いた. グラフのトポロジは 図 9 の通りである. 各ノードのコードは 付録 A

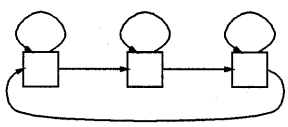


図 9 グラフのトポロジ

の通りである. コード中で co...oc で囲まれた部分が並行コールを行なう部分で, スレッドの分岐点になる.

性能比較の対象には multi-ported object と named-thread を選んだ. 比較対象の実装は [1] のアルゴリズム

を変形したものとなっている．比較したのは以下の二つの時間である:

- メッセージ送信時のオーバヘッド
  - 識別情報の複製時間
  - 最新ポートの検索時間（multi-ported object のみ）
- メッセージ受信時のオーバヘッド
  - メッセージ識別時間

さらに全ての手法について全実行時間を計測し，相互排除を解除して任意のメッセージを受理する手法と比較した．

表 *2* 各ノードでのオーバヘッド

	K/L	NT	MPO
送信時	$O(n)$	$O(1)$	$O(n)$
受信時	$O(1)$	$O(n)$	$O(1)$

n: グラフの総ノード数
K/L: キー/ロック法
NT: Named-Thread
MPO: Multi-Ported Object

ノード一つあたりの定性的なオーバヘッドを 表 2 に示す．図 9 のようなトポロジでは全てのノードでスレッドの分岐が起こるため，各手法ともメッセージへの付加情報の量はグラフ上の総ノード数 n に比例する．よって各ノードでのオーバヘッドの全体はどの手法でもグラフ上の総ノード数 n に対し $O(n)$ となる．

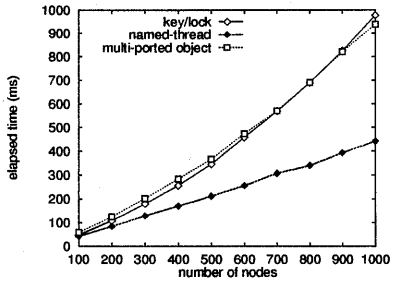


図 *10* 送信時のオーバヘッド

送信時のオーバヘッドの測定結果を 図 10 に示す．キー/ロック法では各オブジェクトでコールの深さに比例する鍵の列の複製時間がかかる．また multi-ported object ではポートバインディングマップの拡張にかかる一定の時間と，最新のポートを検索するのにかかるコールの深さに比例する時間の合計になる．このため送信時のオーバヘッドは 図 9 のグラフではノード数の二乗に比例する．Named-thread ではスレッド ID の拡張を行なうある一定の時間がかかる．よってオーバヘッドは 図 9 のグラフではノード数に比例する．

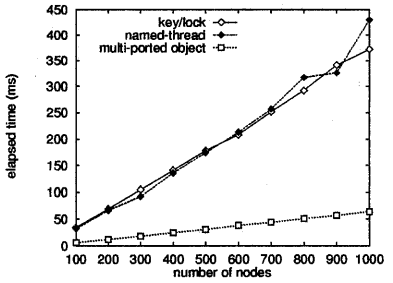


図 *11* 受信時のオーバヘッド

受信時のオーバヘッドの測定結果を 図 11 に示す．キー/ロック法と multi-ported object ではメッセージの識別に一定の時間がかかる．このため受信時にノード数に比例したオーバヘッドがかかる．Named-thread ではスレッド ID を比較してスレッド間の関係を調べるのにスレッドの分岐回数に比例する時間がかかる．よってノード数の二乗に比例する時間がかかる．ただし今回の測定ではトポロジの特性のためはっきりとした結果が出ていない[1]．

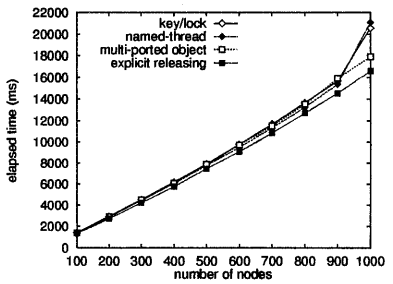


図 *12* 全実行時間

全実行時間の測定結果を 図 12 に示す．相互排除を解除して任意のメッセージを受理する手法がもっとも良い性能を示す．キー/ロック法, named-thread, multi-ported object はほぼ同定度の実行時間で，$n = 500$ で相互排除を解除する手法の 1 割程度速度が低下している．

5 考察

今回の比較ではキー/ロック法の優位性を示すことができなかった．これは実装の問題とともに，例としたプログ

[1] 実装の特性により実際に $O(n)$ の時間がかかるのがはじめのノードだけになっている．

ラムが他の手法に有利なものであったことが原因であると考えられる．以下ではキー/ロック法が有利になる場合について考察する．

コールのネスト中に含まれる異なるオブジェクトの数を n, スレッドの分岐回数を k, 各分岐でのスレッドの分岐数を s とする．named-thread の場合，メッセージの識別のため分岐のたびに拡張されるスレッド ID の比較を行なうので受信時のオーバヘッドは $O(k)$ になる．送信時のオーバヘッドはスレッドの分岐数分スレッド ID 拡張を行なうので $O(s)$ である．Multi-ported object ではメッセージ受信時のオーバヘッドは最新のポートあてであるかどうかの判定のみであるので $O(1)$, 送信時は長さ n のポートバインディングマップの検索を，スレッドの分岐数回行なうので $O(ns)$ になる．一方キー/ロック法ではスレッドの分岐数に関わらず受信時のオーバヘッドは $O(1)$, 送信時は $O(n)$ で変化しない（表 3）．

例えば 1 節の 図 1 の例で考える．送信の単位オーバヘッドを t_S, 受信の単位オーバヘッドを t_R とおく．まずはじめに A で問い合わせを受けるのに t_R, B, C, D, E への送信に t_S のオーバヘッドがかかる．さらに B, C, D, E での受信にそれぞれ t_R の時間がかかる．再帰段階では A が E からの問い合わせを受けるのに t_R, 再度の送信のために t_S のオーバヘッドがかかる．B, C, D, E では受信のために各 t_R オーバヘッドがかかる（図 13）．スレッドの分岐数 s は小データベースの数に等しい．

クリティカルパスでの総オーバヘッドを考える．キー/ロック法の場合送信のオーバヘッドはスレッドの分岐数に依存しないので，小データベースの数が s で再帰が一回だけ起こる場合，オーバヘッドの総計は $3t_R + 5t_S$ となる．

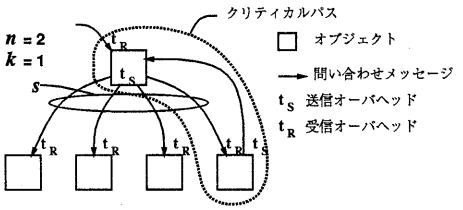
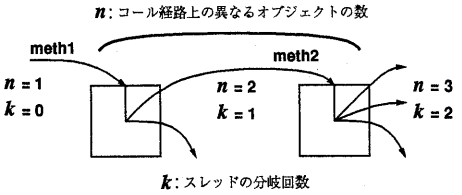


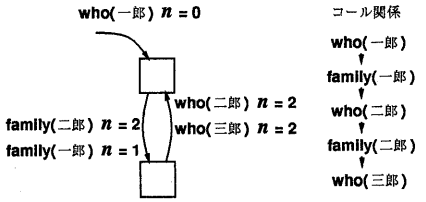
図 *13* データベースシステムの各オブジェクトでのオーバヘッド

Multi-ported object の場合送信のオーバヘッドはスレッドの分岐数とコール中の異なるオブジェクトの数に比例するのでオーバヘッドの総計は $3t_R + (2+3s)t_S$ となり，named-thread では分岐回数に比例する受信オーバヘッドとスレッドの分岐数に比例する送信オーバヘッドのため $3t_R + (1+2s)t_S$ となる．実測値から求めた t_S, t_R を用

表 *3* 各オブジェクトでのオーバヘッド

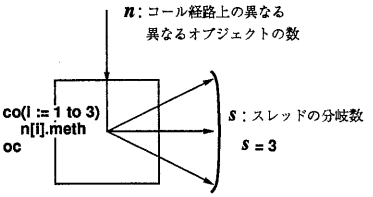
	K/L	NT	MPO
受信時	$O(1)$	$O(k)$	$O(1)$





	K/L	NT	MPO
送信時	$O(n)$	$O(s)$*	$O(ns)$

* ただし実装したアルゴリズムでの上限は $O(n)$



いた結果を に示す

表 *4* 単位オーバヘッド

	K/L	NT	MPO
t_R (ms)	0.35	0.19	0.06
t_S (ms)	0.00195	0.22	0.00188

6 まとめ

並行オブジェクト指向言語における再帰にともなうデッドロックの回避手法としてキー/ロック法を提案した．同手法は排他メソッドに対応した鍵をメッセージに添付するという方法でスレッドの分岐を含む間接再帰にも対応

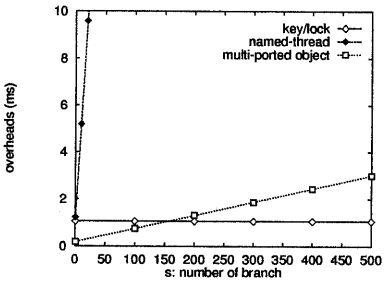


図 *14* s に対するオーバヘッドの総計

する. また再帰の識別に必要な鍵の識別を定数時間でで き, オブジェクトは一定の情報を持っているだけで良い. 時間的なオーバヘッドの上限はスレッドに含まれる独立したオブジェクトの数にのみ比例するのでスレッドの分岐が多い場合に効果的である.

予備的な実装による評価ではメッセージの識別が定数時間でできていることが確認できた. しかしそのために必要となる鍵の列の複製にコールの深さに比例した時間がかかっているため, 従来の研究に対し常に優位に立つことはできなかった. オーバヘッドはプログラムによってかわり, キー/ロック法により有利な場合がある. このような場合にどれだけ優位になるかを調べる必要がある.

考察では各オブジェクトが並列動作する場合を考えオーバヘッドの総計を産出した. しかし現在の実装は単一プロセッサシステム上のもので, 並列動作はしていない. キー/ロック法の有効性を示すためには SR/KL の並列化が必要である. また現在の実装にはいくつか問題がある. 今後これらの問題を解決し, より実践的な例でもって評価を行なうことが必要である.

参考文献

[1] E. A. Brewer and C. A. Waldspurger. Preventing Recursion Deadlock In Concurrent Object-Oriented Systems. Technical Report MIT-LCS//MIT/LCS/TR-526, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1992.

[2] L. Cardelli. A Language with Distributed Scope. http://www.research.digital.com/SRC/personal/Luca_Cardelli/Papers/Obliq.A4.ps, May 1995.

[3] S. J. Hartley. *Operating Systems Programming.* Oxford University Press, 1995.

[4] D. Lea. *Concurrent Programming in Java.* Addison Wesley, 1997.

[5] O. Nierstrasz. Active Objects in Hybrid. In *OOPSLA '87 Proceedings*, Vol. 22, pp. 243–253, December 1987.

[6] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation –. In G. Blelloch, M. Chandy, and S. Jagnnathan, edited, *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[7] 柳川和久. 並行オブジェクト指向言語における再帰にともなうデッドロックの回避機構の実装と評価. Master's thesis, 電気通信大学大学院情報システム学研究科, 1998.

付録 A Graph Sum のノードのコード

```
class Node
  op query() returns r:int
  # 以下のメソッドの詳細は省略
  op set_neighbor(node: cap Node)
  op set_value(v:int)
  op reset()
body Node()
  const MaxNeighbors := 8
  var neighbors[1:MaxNeighbors] : cap Node
  fa i := 1 to MaxNeighbors ->
    neighbors[i] := null
  af
  var num_neighbors:int := 0
  var value:int := 0
  var visited:bool := false

  proc query() returns r
    var ans[1:num_neighbors]:int

    if visited ->
      r := 0
      #release
    [] else ->
      visited := true
      #release
      co (i := 1 to num_neighbors)
        ans[i] := call neighbors[i].query()
      oc
      r := value
      fa i := 1 to num_neighbors ->
        r +:= ans[i]
      af
    fi
  end query
end Node
```