適応的オブジェクトによる並列処理のボトルネック解消

[1]江口 重行　[2]可児 亜祐美　[3]八杉 昌宏　[4]瀧 和男

概要

本論文では，並行アクセスに適応可能なオブジェクトにより，木構造のルートなどにおけるボトルネックを解消する手法を提案する．従来の並列処理系はオブジェクトの振舞いを静的に定めていたため，一定の処理の後変化しなくなったオブジェクトの排他制御を緩和するようなことは行なえなかった．提案する実行時メソッド置換は，読み出し専用メソッドの実行時増加，オブジェクトの不変部分の実行時増加によって，排他制御や遠隔通信を削減するものである．提案方式を共有メモリ型および分散メモリ型並列計算機に実装，評価を行い，その効果を確認した．

Eliminating Bottlenecks on Parallel Systems using Adaptive Objects

[1]Shigeyuki EGUCHI　[2]Ayumi KANI　[3]Masahiro YASUGI　[4]Kazuo TAKI

Abstract

This paper proposes a technique to eliminate bottlenecks (e.g., around a root of tree structure) by adaptive objects for conccurent accesses. Since the conventional parallel processing systems fix objects' behaivors, they cannot relax the mutual exclution for objects which have finished their initialization and become immutable. The proposed dynamic method replacement eliminates the frequency of mutual exclusion and remote message passing by dynamically increasing the number of read only methods and the immutable part of objects. This paper also presents the implementation and the results of performance measurements on both shared memory and distibuted memory parallel architectures, which indicate the effectiveness of our approach.

1　はじめに

並列処理では，しばしばシステムの一部でのボトルネックによる全体の性能低下が問題になる．このようなボトルネックの多くは，アクセスが集中するオブジェクトにおける状態の一貫性保持のための排他制御によって発生する．これに対して，オブジェクトの並行読み出しを許すなどの排他制御規則の緩和，オブジェクトの不変部のみを利用するメソッドの排他制御なしでの実行，不変部分の複製といった手段により，排他制御が必要となる区間，回数を削減できる．

しかし，現在の並列処理系では，このような技法は静的にしか適用されていない．このため，木構造やニューラルネットワークなど，部品となるオブジェクトの状態が構築フェーズと利用フェーズで変化する場合などに，それぞれの状態に応じた排他制御規則を用意するといったことは行なえない．

このような問題に対処するため，本論文では並行アクセスに関して適応可能なオブジェクトと，その記述手段として実行時メソッド置換を提案する．これにより，変数の読み出しのみを行なうメソッドの数及びオブジェクトの不変部分を，実行時に増加させていくことが可能になる．実行時メソッド置換はオブジェクト指向におけるクラスの継承機構を拡張して実現可能である．本論文ではさらに，実行時にオブジェクトの不変部分のみを利用するようになったメソッドの（排他制御や遠隔通信の有無を含む）実行手順を切替えるのに十分な条件を，実行時メソッド置換によるメソッド状態遷移の情報などから静的に解析するアルゴリズムを示す．

以下に本論文の構成を示す．2節では，我々が開発中のオブジェクト指向並列言語 OPA [9] の排他制御モデルについて述べる．3節では，実行時メソッド置換について述べる．4節では，排他制御不要メソッド (free メソッドと呼ぶ) の検出方法を示す．5節では共有メモリ並列アーキテクチャ上の実装について，6節では分散メモリ並列アーキテクチャ上の実装について述べる．7節では実装した処理系の性能評価結果を示す．8節で関連研究について述べる．

---

[1] 神戸大学大学院　自然科学研究科 The Graduate School of Science and Technology, Kobe University, email:`eguchi@chagall.cs.kobe-u.ac.jp`

[2] 富士通 (株) Fujitu Limited

[3] 京都大学大学院 情報学研究科 The Graduate School of Informatics, Kyoto University, email: `yasugi@kuis.kyoto-u.ac.jp`

[4] 神戸大学工学部 情報知能工学科 Dept. of Computer and Systems Engineering, Faculty of Engineering, Kobe University, email:`taki@seg.kobe-u.ac.jp`

2　オブジェクト指向並列言語 OPA

本節では，オブジェクト指向並列言語 OPA の排他制御モデルについて述べる．OPA は Java 言語 [2] を並列に拡張したもので，Java のスレッドの代わりに，構造化された並列構文，同期機構，提案する排他制御モデルを組み込んだものである．

排他制御の緩和にむけて，OPA では各オブジェクト上でインスタンス変数を読み出すのみのメソッドの並列実行を許す．また，変化しなくなった変数を読み出すのみのメソッド (free メソッド) をできるだけ排他制御なしで実行することで処理効率の向上をはかる．

2.1　instant メソッド

OPA では，キーワード instant をつけて宣言されたメソッドを，メソッドのインスタンス変数への文面上のアクセスに応じて，読み出し専用 (RO) と読み書き両用 (RW) に分類する．

instant メソッドの動作を図 1に示す．instant メソッドはまず最初に必要なオブジェクトの変数を暗黙の局所変数に一括読み出し (コピー) し，これを利用してメソッドを実行する．RW メソッドで変更された局所変数の値は，Schematic [8] と同様に，オブジェクト本体に一括して書き戻される．OPA では，それ以上の局所変数の更新がない点を一括更新点とし，コンパイラによるフロー解析でこれを求める．

RO メソッド： (RO1) メソッドのルックアップ，(RO2) メソッド起動，(RO3) オブジェクトの状態をローカル変数へ一括読み込み，(RO4) メソッド本体の実行．

RW メソッド： (RW1) メソッドのルックアップ，(RW2) メソッド起動，(RW3) 一括読み込み (RW4) 一括更新までの処理，(RW5) オブジェクトを一括更新，(RW6) メソッドの残りの実行．

RO1–RO3 を RO 区間，RW5 を W 区間，RW1–RW5 を RW 区間とする．排他制御は，データを読み出し，加工して書き戻すという処理の一貫性を保つために RW 区間実行間で，一貫性のあるデータを読み出すため RO 区間実行と W 区間実行の間で必要になる．RO1，RW1 が排他制御が必要な区間に入っているのは，実行時メソッド置換を利用する場合，メソッドのルックアップから起動までと，オブジェクトの更新との間で排他制御が必要になるからである．

また，RO 区間実行間に排他制御の必要はなく，instant RO メソッドはオブジェクト内で同時に複数がブロックなしで実行可能である．

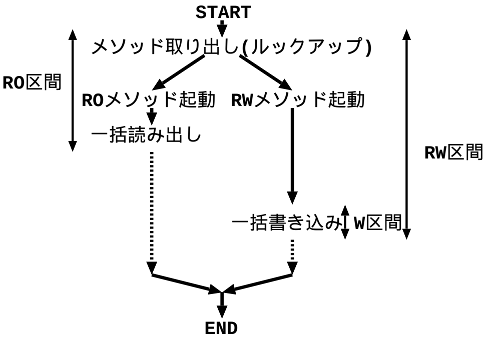

図 1: instant メソッド

2.2　free メソッド

オブジェクトの変化しない変数を読み出すのみのメソッドは，排他制御なしで実行できる．OPA では，これを free メソッドと呼ぶ．

オブジェクトの不変要素はローカルな環境にコピーでき，これによって free メソッドがローカルに実行できるため，free メソッドは分散メモリ環境での性能向上に特に有効である．

しかしながら，後述の実行時メソッド置換を採用すると，実行時に free メソッドを増やしていけるという大きな利点はあるものの，メソッドの組合せの数は最悪，指数オーダとなり，あるメソッドが free メソッドとなるための十分条件を静的に解析するのは困難となる．コンパイラの解析能力によって言語の意味論が変わるのは好ましくないため，OPA では free メソッドを言語の上では採用せず，instant RO メソッドが free メソッドとなる条件を満たした場合のみ，実行時に自動的に (排他制御や遠隔通信の有無を含む) 実行手順を切替えるものとする．これにより，完全な解析が保証できない free メソッドを，言語の意味論を崩すことなく利用できる．

3　実行時メソッド置換

オブジェクトが生成され，ある程度処理が進行すると (オブジェクトが成熟すると)，RW メソッドの中に，二度と呼ばれない，またはオブジェクトの更新を行なわないものが現れる場合がある．

OPA では，実行時メソッド置換により，更新を行なわなくなった RW メソッドを対応する RO メソッドに置換することができる．

3.1　文法

以下に OPA での実行時メソッド置換の利用法を示す．置換可能メソッド `m1` をメソッド `m2` に置換する場合，

```
setmethod(m1, m2);
```

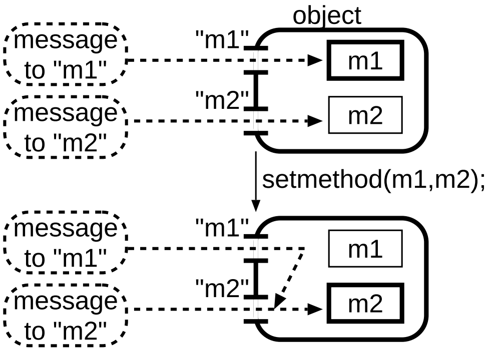

図 2: 実行時メソッド置換

のように記述する．置換後はメソッド `m1` を呼び出すとメソッド `m2` が起動する．メソッドを初期状態に戻すには置換先にキーワード `initial` を指定する．また，そのメソッドが今後呼ばれないことを明示するためには，置換先にキーワード `null` を指定する．置換結果の反映はインスタンス変数の一括更新時にまとめて行なう．また，置換可能メソッドには宣言部にキーワード `replaceable` をつけるものとする．

OPA では，スレッド間の通信，同期にメソッド置換を利用できる．メソッド置換において，置換先のメソッド名にキーワード suspends を指定すると，そのメソッドを呼び出そうとしたスレッドはその場でサスペンドする．他のスレッドがメソッドを通常のメソッドに置換することで，そのスレッドは動作を再開する．suspends について，本論文ではこれ以上触れない．

オブジェクト指向の立場からみると，メソッド置換はオブジェクトに届いたメッセージの名前を変更し，自オブジェクトに送り直すと定義できる．メッセージを送る側から実際に起動されるメソッドが特定できない点は通常の遅延束縛と同様である．この立場から見たメソッド置換の動作を図 2に示す．また，図 3にプログラム例として二分木辞書クラスを示す．

このプログラムにおいて，RW メソッド insert は左右の子ノードが生成された後，インスタンス変数の更新を行なわなくなる．よって，等価な動作をする RO メソッド insertF に置換することで，処理が高速化できる．RO メソッドは並列実行可能であるので，木構造の根などでもボトルネックとならない．

3.2 メソッド置換による free メソッドの実行時増加

実行時メソッド置換を行ない，RW メソッドを RO メソッドに置換することで，(1) 排他制御が必要な区間が縮小されることによるスレッド間の並列性向上，(2) 置換先の RO メソッドを記述する際に，その時点での処理に応じたメソッド内容の最適化が行なえる，の二点により処理の高速化が行なえる．

```
class BinTree{
 internal int key, value;
 internal BinTree left=null,right=null;
 public BinTree(int k, int val){
  key = k;
  value = val;
 }
 public instant int
     search(int k, int v){ … }
 public replaceable instant void
     insert(int k, int val){
  if(k < key){
   if(left != null)left.insert(k, val);
   else {
     left = new BinTree(k, val);
     if(right != null)
       setmethod(insert, insertF);
   }
  }else{
   if(right != null)right.insert(k, val);
   else {
     right = new BinTree(k, val);
     if(left != null)
       setmethod(insert, insertF);
   }
  }
 }
 internal instant void
     insertF(int k, int val){
  if(k < key)left.insert(k, val);
  else right.insert(k, val);
 }
}
```

図 3: プログラム例 (二分木辞書)

加えて，実行時メソッド置換を利用してオブジェクトの状態遷移を解析することで，更に実行時に free メソッドを増加させるという最適化が可能である．

メソッドが free であるためには，メソッドが RO メソッドでかつ，メソッドが読み出す部分が今後変化しないことが必要である．オブジェクトには必ず初期化 (コンストラクタの呼びだし) が必要なため，後者の条件は少なくとも初期化後にしか成立しない．ここで，コンストラクタはオブジェクトの初期化後呼ばれないとすると，いくつかのメソッドが free になる可能性が出てくる[1]．

オブジェクトの各インスタンス変数が更新され得るかどうかは，オブジェクトのその時点で呼ばれ得るメソッドの集合から解析できる．このメソッドの集合はメソッド置換によって変化するため，コンパイラにおけるオブジェクトの状態遷移の解析はメソッ

---

[1] Java ではコンストラクタ実行時にコンストラクタ以外のメソッドは実行されないよう工夫されている．

ド置換を利用して行なうことができる．この解析によって free であると判断できた RO メソッドは，処理系内部で自動的に free メソッドとして扱われる．ただし，メソッドがオブジェクトから読み出す情報にはそのメソッドの置換情報も含まれるため，置換可能メソッドが free メソッドになるためには，今後そのメソッドが置換されないことも条件となる．

メソッド置換を行なう場合，メソッドを取り出すまでメソッドタイプが RO，RW，free のいずれか分からず，また置換がいつ発生するかも分からないため，メソッドのルックアップ自体にも排他制御が必要になる点に注意が必要である．特に，得られたメソッドが RO のときに，ロック操作などを使わないようにする工夫が重要である．

また，現在の Java の文法では，最も強いアクセス制限を示す修飾子 `private` でも，同じクラスの他オブジェクトからの直接参照を禁止できないため，常に全ての変数が外部から書きかえられる可能性を持つ．OPA では，アクセス制限を示す修飾子 `internal` を導入する．これは，対象となるインスタンス変数 / メソッドに対する，一切の外部からのアクセスを禁止する．これにより，更新されない変数を増加させることができる．

実行時のオブジェクトの状態遷移を考慮して free メソッドの増加を図るためには，実行時メソッド置換以外に変数の最終書き込みを明示するという手段も考えられる．しかし，この方法では最終書き込みを行うメソッドは free メソッドにならず，また誤った最終書き込みの静的な解析は困難であり，実行時のチェックが必要になるが，変数への書き込み毎にこれを行なうのはオーバーヘッドが大きいという問題がある．

4 free メソッドの検出アルゴリズム

4.1 アルゴリズム

最初にアルゴリズムの概略を述べる．まず，オブジェクトのその時点で呼ばれ得るメソッドの組合せ (メソッド状態) の，メソッド置換による遷移を表した有向グラフを考える．ループを構成するノードをまとめて不可逆な状態遷移に関する情報を得るため，グラフの強連結成分を抽出し，この成分ごとに free メソッドの検出を行なう．以下にアルゴリズムの詳細を述べる．

メソッドのルックアップは，メッセージ（a）をメソッド（m）に写す関数とみなせる．また，メソッドテーブル（T）は基本的にクラス毎に用意されるが，free メソッドを使用する際は，クラスに複数用意される．

T はメッセージをテーブルエントリ（e）に写す．

$$T = \{a_1 \mapsto e_1, \ldots, a_n \mapsto e_n\}$$

各置換可能メソッドの状態を記憶するための場所は，その置換可能メソッドを持つオブジェクト毎に用意される．その状態が記憶される位置 x はサブクラスのおいても共通である．テーブルエントリ e はメソッドのタイプ（$\mathrm{RO}, \mathrm{RW}, \mathrm{or}, \mathrm{free}$）が指定されたメソッド m か，位置 x である．

$$e ::= \mathrm{RO}\, m \mid \mathrm{RW}\, m \mid \mathrm{Free}\, m \mid \mathrm{Replaceable}\, x$$

メソッド状態は，$M = (T, S, H, R)$ で表される．R は変名規則であり，各場所で対応するメッセージを保持する．

$$R = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$$

$S(m)$ はメソッド m の実行により発生し得るメソッド置換（これも R で表す）の集合を与える．

$$S = \{m_1 \mapsto \{R_{11}, \ldots, R_{1l_1}\}, \ldots, \\ m_n \mapsto \{R_{n1}, \ldots, R_{nl_n}\}\}$$

メッセージ (a) はメソッドの数だけ用意される．置換可能メソッドのエントリの内容はメソッド本体でなく位置 x であるため，別に置換可能メソッドの初期メソッドへのエントリが必要である．このため，隠れたメッセージ $H = \{a_1, \ldots, a_n\}$ を導入する．メソッド置換 R によって，変名規則 R_1 は以下のように関係する場所を上書きすることで，R_2（$R_1 R$ と表記）に変更される．

$$\mathrm{Dom}(R_2) = \mathrm{Dom}(R_1) \cup \mathrm{Dom}(R)$$
and
$$R_2(x) = \begin{cases} R_1(x) & \text{if } x \in \mathrm{Dom}(R_1) - \mathrm{Dom}(R) \\ R(x) & \text{if } x \in \mathrm{Dom}(R) \end{cases}$$

メソッドルックアップ関数 $lookup$ は以下のように定義される．

$$lookup(a, T, R) = rename(T(a), T, R)$$

where，

$$rename(\mathrm{RO}\, m, T, R) = m$$
$$rename(\mathrm{RW}\, m, T, R) = m$$
$$rename(\mathrm{Free}\, m, T, R) = m$$
$$rename(\mathrm{Replaceable}\, x, T, R) = lookup(R(x), T, R)$$

また，メソッド状態の可能な遷移 $M_1 \to M_2$ は以下の定義により定まる．

$$M_1 \to M_2$$
$$\text{iff} \quad ((\exists m \in MS(T_1, H_1, R_1) . \exists R \in S(m) . R_2 \\ = R_1 R) \wedge T_1 = T_2 \wedge S_1 = S_2 \wedge H_1 = H_2)$$
where
$$M_1 = (T_1, S_1, H_1, R_1) \text{ and } M_2 = (T_2, S_2, H_2, R_2)$$

ただし，$MS(T,\ H,\ R)$ は，現在のメソッド状態において，呼び出され得るメソッドセットを表しており，以下のように定義される．

$$MS(T,\ H,\ R) = \\ \{m \mid \exists a \in (\mathrm{Dom}(T) - H)\,.\,m = \mathit{lookup}(a,\ T,\ R)\}$$

以上の定義を元に free メソッド検出アルゴリズムの概要を以下に示す．また，これを図 4に示す．

free メソッド検出アルゴリズム

入力:初期メソッド状態 $M_0 = (T_0, S_0, H_0, R_0)$

出力:遷移条件付き DAG(a Directed Acyclic Graph) $N_{out} = (G_{out}, c_{out})$

G_{out}：頂点をメソッドテーブル（T）とする DAG

c_{out}：G_{out} の辺に対して，遷移条件 $\{R_1, \ldots, R_n\}$ を対応させる関数

遷移条件 $\{R_1, \ldots, R_n\}$：メソッド状態の変名規則が $R_1, \ldots, R_n$ のいずれかであること

出力の意味：遷移条件が成立した時，メソッドテーブルを G_{out} に沿って切替えられる．

１．初期状態をセットする．

$$G_0 = (V_0, E_0) = (\{M_0\}, \{\})$$

２．$G = G_{n+1} = G_n$ になるまで以下の手順を繰り返し，有向グラフ G を生成する．(図 4-(1))

$$\begin{aligned} G_{n+1} &= (V_{n+1}, E_{n+1}) \\ &= (V_n \cup \{M' \mid \exists M \in V_n\,.\,M \to M'\}, \\ &\qquad E_n \cup \{(M, M') \mid \exists M \in V_n\,.\,M \to M'\}) \end{aligned}$$

３．G 上すべての強連結成分を検出する．（強連結成分を頂点とするグラフ G_{sc}(図 4-(2)) は DAG の形を成す．)

４．各強連結成分毎に free メソッドを検出する．この際，G_{sc} 上末端側の頂点から行なうことで計算量が減らせる．(a, m) が free メソッドである条件は，(1) メソッドの型が RO である (2) 現在以降の点で，a における lookup の結果が常に m となる (3) 現在以降の点での MS に含まれるメソッドが，m が読み出す変数の更新を行なわない．

５．各強連結成分毎に検出される free メソッドにつき T を更新した T' を準備する．

$$T' = T\{a_1 \mapsto \mathrm{Free}\ \ m_1, \ldots\}$$

６．G_{out} として T' を頂点とする G_{sc} と同形（ただし，T' が同一の頂点は一つにする）の DAG を構成すると共に，その辺に c_{out} の与える遷移条件を $\{R \,|\, (T,\ S,\ H,\ R)$ が G_{sc} の辺に対応する G 上のいずれかの辺の終点である $\}$ とする．(図 4-(3))

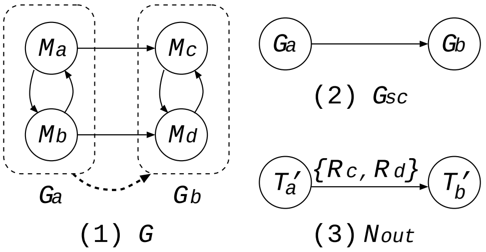

図 4: free メソッド検出アルゴリズム

4.2 解析例

２分木辞書クラスにおける G は以下のようになる．ただし，`search` メソッドは考慮に入れてない．

$$G = (\{M_0, M_1\}, \{(M_0, M_1)\})$$

where,

$$\begin{aligned} &M_0 = (T_0,\ S_0,\ H_0,\ R_0)\ \ M_1 = (T_1,\ S_1,\ H_1,\ R_1) \\ &T_0 = T_1 = \{\texttt{insert} \mapsto \mathrm{Replaceable}\, x_{\mathrm{insert}}, \\ &\qquad\qquad \texttt{insertF} \mapsto \mathrm{RO}\, m_{\mathrm{insertF}}, \\ &\qquad\qquad a_{\mathrm{insert}} \mapsto \mathrm{RW}\, m_{\mathrm{insert}}\} \\ &S_0 = S_1 = \{m_{\mathrm{insert}} \mapsto \{\{x_{\mathrm{insert}} \mapsto \texttt{insertF}\}\}, \\ &\qquad\qquad m_{\mathrm{insertF}} \mapsto \{\}\} \\ &H_0 = H_1 = \{a_{\mathrm{insert}}\} \\ &R_0 = \{x_{\mathrm{insert}} \mapsto a_{\mathrm{insert}}\}\ R_1 = \{x_{\mathrm{insert}} \mapsto \texttt{insertF}\} \end{aligned}$$

この例において M_0 ,M_1 は，それぞれ強連結成分である．結局得られる結果は以下の通りである．

$$N_{out} = ((\{T_0, T_1'\}, \{(T_0, T_1')\}), \{(T_0, T_1') \mapsto \{R_1\}\})$$

where,

$$\begin{aligned} T_1' = \{&\texttt{insert} \mapsto \mathrm{Free}\ \ m_{\mathrm{insertF}}, \\ &\texttt{insertF} \mapsto \mathrm{Free}\ \ m_{\mathrm{insertF}}, \\ &a_{\mathrm{insert}} \mapsto \mathrm{RW}\, m_{\mathrm{insert}}\} \end{aligned}$$

ここで，$a_{insert} \in H_1$ なので，RW m_{insert} が呼び出されることは無い．

5 共有メモリ型並列計算機への実装

SGI 社製共有メモリ型並列計算機 POWER Onyx (MIPS R10000(195MHz) × 12 個，メモリ 2GB) に対して処理系を実装した．OPA のコンパイラは，OPA のソースプログラムを C コードに変換する．ランタイムは，仮想メモリ空間を共有する OS のプロセスを各プロセッサに一つ生成し，OPA のスレッドは全てこのランタイムで管理される．

5.1 排他制御の実装

2.1節で述べた通り，instant メソッドの実装に必要な排他制御は RW 区間実行間と，RO 区間実行と W 区間実行の間のもののみである．

RW 区間同士の排他制御については，オブジェクトに使用中フラグと使用待ちスレッドキューを用意することで行なう．使用中フラグとキューの更新についての排他制御にはロックを用いた．

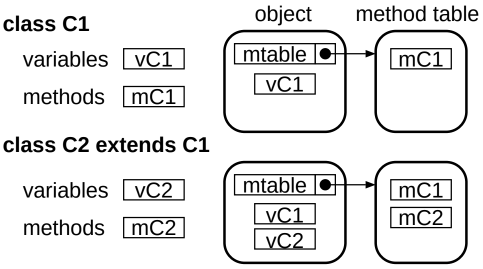

図 5: オブジェクトの実装

RO 区間と W 区間の排他制御には，バージョン番号方式 [6] を用いた．これは [5] を改良したもので，各オブジェクトに番号 (初期値 1，更新中 0，更新後 更新前の値+2) をつけてオブジェクトを管理するものである．この方式はロック操作を一切用いずに複数のスレッドが同時に値を読み出せ，低コストで高い並列性を実現できることが利点である．

5.2 オブジェクトの実装

オブジェクトは，C 言語上では構造体で表現され，排他制御のための管理情報，メソッドテーブルへの参照，インスタンス変数テーブルなどで構成される．子クラスのオブジェクトを親クラスのオブジェクトとして利用できる部分型の機能を低コストで実現するため，オブジェクト構造体は親クラスの構造体に子クラスで定義されたインスタンス変数テーブルを付加した形で定義される．また，メソッドの置換情報もインスタンス変数テーブルに記憶される． オブジェクトの構成を図 5に示す．

メソッドテーブルは，オブジェクトのその時点で呼ばれ得るメソッドに関する情報を保持する．各メソッドに関して，メソッドが置換可能でなければ，メソッドタイプとメソッドを実現している C の関数への参照が格納される．置換メソッドの場合メソッドタイプは replaceable となり，メソッド本体への参照に代わり，インスタンス変数テーブル内のそのメソッドの置換情報へのオフセットが格納される．置換情報は，メソッドテーブル内の置換先メソッドのメソッド情報へのオフセットからなる．これらの情報は図 6のように表現できる．

5.3 置換メソッドの起動シーケンス

図 7に，疑似コードによる `o.message(args)` に対応するメソッド起動手順を示す．

これにより，4節で定義した関数 *lookup* に従ってルックアップが行なわれることがわかる．実装技術として注意すべき点は，RO メソッドにバージョン番号 2 を渡すことでメソッドが free であることを通知している点である．

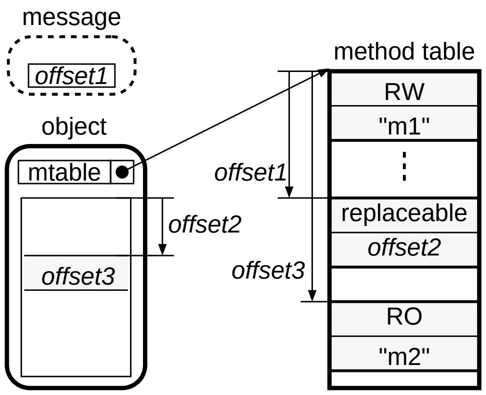

図 6: 置換メソッドのルックアップ

メソッド置換先が置換メソッドであった場合，置換の仕方により参照が循環して無限ループとなる可能性があるが，これはプログラマの責任とする．

ルックアップ後，メソッドタイプに応じて起動を行なう．RO メソッドは本来ロックを用いずに起動できるメソッドであるが，置換メソッドとして呼ばれる場合でも，メソッドのルックアップから，起動後，オブジェクトの一括読み出しが終了するまでの間にオブジェクトの更新がなければは，一切ロックを用いずにメソッドを起動できる．しかし，更新があった場合，メソッドが置換されている可能性があるため，今度はオブジェクトをロックしてルックアップから処理をやり直す．(vn=0 でリトライ中を表す)

RW メソッドの場合はオブジェクトをロックし，オブジェクトの使用中フラグにより，他の実行中 RW 区間の存在を調べる．オブジェクトが使用中であれば実行をサスペンドし，オブジェクトのキューで実行再開を待つ．使用中でなければメソッドを起動し，使用中フラグをセットしたのち，ロックを解放する．

以上により，メソッドを取り出すまでメソッドタイプが不明な置換メソッドを安全に起動できる．

6 分散メモリ型並列計算機への実装

Sun Microsystems 製 WS SPARCstation5 (microSPARCII 110MHz)4 台を完全結合した WS クラスタを用いて，OPA 処理系の分散メモリ環境への実装を行なった．WS 間の結合には高速シリアル通信インターフェース STAFF-link [7] を用いた．

ランタイムは各ワークステーションに一つプロセスを生成する．共有メモリ用の実装と同じく，これらが OPA のスレッドを管理する．

分散メモリ環境では他ノードのメモリを直接参照できないため，排他制御は共有メモリの場合より単純に実装できる．リモートオブジェクトへのアクセスは，自ノード上にオブジェクトごとに生成したコ

```
START:
do { vn=o.vn; } while(vn==0);
RETRY:
msg = message;
LOOKUP:
e = lookup(o, msg); method = e.method;
switch(e.mtype) {
case Replaceable:
  i = method; msg = o[i]; goto LOOKUP;
case RO:
  if (vn && vn != o.vn) goto ROfail;
  status = invoke(method, vn, o, args);
  if (status == ROfail) {
ROfail:
    lock(o); vn=0; goto RETRY;
  }
  break;
case RW:
  if (vn) {
    lock(o);
    if (vn != o.vn){vn=0; goto RETRY;}
  }
  if (o.use) {suspends(o); goto START;};
  status = invoke(method, o, args);
  break;
case Free:
  if (vn == 0) unlock(o);
  status = invoke(method, 2, o, args);
  break;
}
```

図 7: 置換メソッドの起動シーケンス

ピー (プロキシ) を介して，リモートメソッド呼び出しを行なうことで実現する．このとき無視できない遅延が発生するが，これはオブジェクトの不変部分をプロキシにコピーし，free メソッドをローカルに実行可能とすることで解消できる．また，各プロセッサに複数の OPA スレッドを生成し，常にプロセッサが処理できるタスクを確保することで遅延を隠蔽できる．

6.1 プロキシの実装

プロキシは，各ノードでそのオブジェクトへの参照が最初に発生したときに生成され，以降そのオブジェクトへの参照はプロキシに対する参照に変換される．

プロキシのメソッドが呼ばれた場合，メソッドが free でなければプロキシはオブジェクトが存在するノードのサーバに依頼し，リモートメソッド呼び出しを行なう．メソッドの実行後，呼ばれたメソッドが free になっていれば，サーバは戻り値に加え，オブジェクトのコピーをメッセージとして返す．これを受けとったプロキシは，以後同じメソッドの呼びだしを全てローカルに処理することが出来る．

プロキシは，そのノード上のどのオブジェクトからも参照されなくなった時点でガーベジコレクタに回収される．

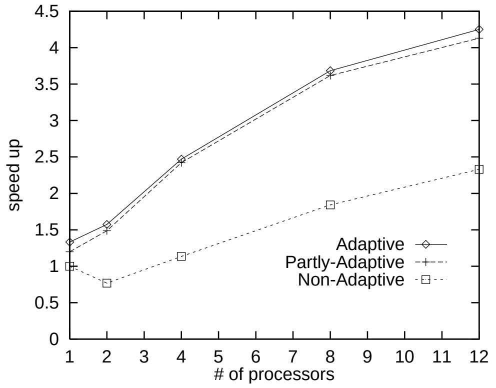

図 8: 共有メモリ環境における性能比較

7 評価

実装した処理系の性能評価を行なった．サンプルとして二分探索木クラスを作成した．これはオブジェクトを要素として木を構成するクラスで，探索，挿入メソッドを持つ．挿入メソッドは，一度左右に子を作った後はオブジェクトの更新を行なわないため，その時点で RO メソッドに置換可能であり，このとき探索，挿入両メソッドが free メソッドとなる．

このクラスを用いて，ランダムに生成したキーによって単一の木構造に 30,000 個の要素を登録するのに要した時間を評価基準とした．各プログラムは 4 回実行し，2 番目に速い結果を採用した．

共有メモリ用処理系の評価に，以下の 3 種類のプログラムを用いた．

- **Adaptive** 実行時メソッド置換，free メソッドを利用するもの
- **Partly-Adaptive** 実行時メソッド置換のみ利用するもの (free メソッドは利用しない)
- **Non-Adaptive** 実行時メソッド置換を用いないもの

Non-Adaptive，PE 数 1 の場合 (0.458s) を 1 とした，各プログラムの性能向上を図 8に示す．

全体的に，ほぼ Adaptive，Partly-Adaptive，Non-Adaptive の順に良い性能が出ている．並列性のない PE 数 1 の場合より，実行時メソッド置換によってメソッドの処理が軽くなり，free メソッドによってそれがさらに改善されたことがわかる[2]．複数 PE 下では実行時メソッド置換による処理の並列性向上により，Non-Adaptive との性能差がより顕著になっている．また，共有メモリ環境では RO メソッドと free メソッドの間に大きな性能差はないことが確認された．これは，分散メモリ環境では free メソッド

---

[2] 挿入メソッドは約 52 万回呼ばれ，80 %以上を RO または free として処理できている．

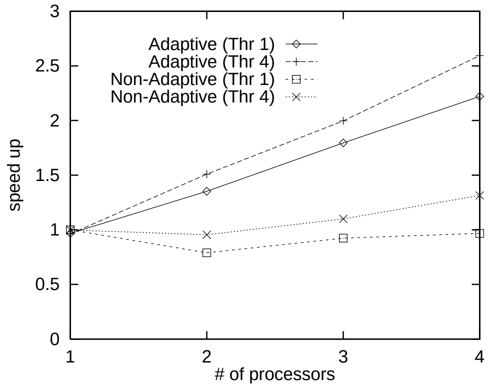

図 9: 分散メモリ環境における性能比較

のときのみソフトウェアで行なわれている不変部分の複製が，ハードウェアキャッシュで自動的に行なわれているためである．

分散メモリ用処理系の評価には，Adaptive，Non-Adaptiveについて，プロセッサあたりのスレッド数を1または遅延隠蔽のため4とした，計4種類のプログラムを用いて測定を行なった．Non-Adaptive，PE数1，Thr数1の場合(2.80s)を1とした，各プログラムの性能向上を図9に示す．

図より，PE数2～4ではAdaptiveがNon-Adaptiveと比較してほぼ2倍速くなっている．また，同じプログラムのPEあたりのスレッド数が1と4の場合を比較して，分散環境ではスレッドの多重化による遅延隠蔽が有効であることが示された．

8 関連研究

Actors [1]にはbecomeコンストラクトがあり，これは実行時にオブジェクトのクラス全体を入れ換えるものに相当する．OPAでは，このような操作はシステムが担当すべき低レベルな処理だと考え，ユーザにはメソッド置換のみを提供し，メソッドテーブルの切替えはシステムが行なう．

Concurrent Aggregates [3]は，並行アクセス時のボトルネックを解消できるaggregateという概念をもつ．これは，representativeと呼ばれるオブジェクトの集合である．aggregateに届いたメッセージは，無作為に選ばれたrepresentativeに渡される．representativeは必要であれば他のオブジェクトへメッセージを送る．OPAでは同種の機能をよりシステム化された方法で提供している．

[4]では，部分計算を利用し，定数化した変数に適応したメソッドのコードを実行時に生成する．freeメソッドは定数化したインスタンス変数のみを利用するため，実行時生成されたコードは無効化されることが無い．実行時メソッド置換はfreeメソッドを実行時に増加させてゆく技術であり，これと実行時コード生成を組み合わせることにより，処理系の性能をより向上させることが出来ると考えられる．

9 結論と今後の課題

本論文では，並列処理のボトルネックを緩和するため，並列アクセスに適応するオブジェクトとその記述手段として実行時メソッド置換を提案した．また，提案方式の共有メモリ型，分散メモリ型並列計算機への実装と，処理系の性能評価結果について述べた．この結果，提案方式の有効性が確認された．

今後はより総合的な性能評価，実行時コード生成機能の処理系への組み込みの検討を行なう．

10 謝辞

STAFF-Linkを使用したWSクラスタの使用に便宜を図っていただいた神戸大学の中條拓伯博士に感謝いたします．本研究の一部は文部省科学研究費（奨励(A)09780278）による．

参考文献

[1] G. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1987.

[2] K. Arnold and J. Gosling eds. The Java Programming Language. Addison-Wesley Publishing Company, 1996.

[3] A. A. Chien. *Concurrent Aggregates.* The MIT Press, 1993.

[4] N. Fujinami. Automatic Run-Time Code Generation in C++. In *Proc. of ISCOPE'97*, No. 1343 in LNCS. Sprnger-Verlag, 1997.

[5] L. Lamport. Concurrent reading and writing. *CACM*, 20(11):806–811, 1977.

[6] 八杉, 瀧. 並列処理のためのオブジェクト指向言語OPAの設計と実装. 情報処理学会研究報告, 96(82):157–162, 1996.

[7] H. Nakajo, A. Ichikawa, and Y. Kaneda. An Implementation and Evaluation of a Distributed Shared-Memory System on Workstation Clusters Using Fast Serial Links. *Proc. of Int.Symp. on High Performance Computing(ISHPC)*, pp. 143–158, 1997.

[8] K. Taura and A. Yonezawa. Schematic: A concurrent object-oriented extension to scheme. Technical report, Dept. of Information Science, University of Tokyo, 1995.

[9] 八杉, 瀧. 実用的な並列処理のためのオブジェクト指向言語OPAの設計. 第13回オブジェクト指向計算ワークショップ (WOOC'97), Mar 1997.