プログラミング 14−10
（1997. 8. 21）

実行時メソッド置換を行なう並列言語の実装

[1]江口 重行 [2]八杉 昌宏 [2]瀧 和男

[1]神戸大学大学院自然科学研究科
[2]神戸大学工学部情報知能工学科

我々は現在，オブジェクト指向並列言語 OPA を開発中である．本論文では，その特徴的な機能である実行時メソッド置換の概念，および共有メモリ型並列計算機への実装方式について述べる．OPA では，メソッドをオブジェクトへのアクセスの性質に応じて読み書きを行なうもの，読み出しのみ行なうものに分類し，排他制御が必要な区間の縮小をはかっている．ここで，処理の進行により更新を行なわなくなったメソッドを読み出し専用メソッドに置換することで，さらに効率を改善することができる．共有メモリ型並列計算機 POWER Onyx 上で実装，評価を行なった結果，実行時メソッド置換が処理の高速化に有効であることが確認された．

A Implementation of Parallel Languages with Dynamic Method Replacement

[1]Shigeyuki EGUCHI [2]Masahiro YASUGI [2]Kazuo TAKI

[1]The Graduate School of Science and Technology, Kobe University
[2]Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University

We are designing and developing an object-oriented parallel language OPA, which features dynamic method replacement. In this paper we describe the dynamic method replacement and its implementation on shared-memory parallel computers. In OPA, methods are divided into read-write(RW) type and read-only(RO) type according to their access to the object. This makes it possible to reduce the period and frequency of mutual exclusions. The efficiency is further improved by replacing a RW method with a RO method dynamically. Such replacement is available in the case where update an object is finished through the progress of processing. The evaluation on a shared-memory parallel computer, POWER Onyx, shows the effectiveness of the dynamic method replacement.

1 始めに

近年，マルチプロセッサによって構成される計算機システムが一般化するにつれ，並列／分散環境に適したプログラミング言語の必要性が急速に高まっている．これに対し我々は，不規則な並列性を含む問題を誤りなく記述でき，分散メモリ型，共有メモリ型を問わず効率の良い並列処理が行なえることを目標として，オブジェクト指向並列言語 OPA [1,2] の設計，実装を進めている．

本論文では，OPA の特徴の一つである実行時メソッド置換について述べる．

OPA では，メソッドをその性質に応じて，RW メソッド (オブジェクトに読み書きを行なう)，RO メソッド (書き込みを行なわない)，実装上はさらに free メソッド (変化しなくなった変数のみを利用) に分類し，排他制御が必要な区間の短縮を図る．

実行時メソッド置換とは，オブジェクトには適切な更新の後変化しなくなる性質を持つものが多いことを利用し，更新を行なわなくなった RW メソッドを RO，free メソッドに置換することで処理効率の改善を図るものである．

以下，2章で OPA の言語仕様のうち，実行時メソッド置換に関係の深いものについて説明し，3章で処理系の実装方式について述べる．4章で実装した処理系の評価を行ない，5章で結論を述べる．

2 オブジェクト指向並列言語 OPA

OPA は Java ライクなシンタックスを持つオブジェクト指向言語で，構造化された並列構文を持つ．参照はオブジェクトと配列に対するものに限定され，C++ に見られるようなメモリ番地へのポインタは禁止する．

2.1 instant メソッド

OPA では，キーワード instant とともに定義されたメソッドの実行に対し独自の排他制御を行なう．メソッドをインスタンス変数へのアクセス形態により，読み出し専用メソッド (RO メソッド) と読み書き両用メソッド (RW メソッド) に分類する．また，メソッドは実行に際してオブジェクトのデータを一括読み出し (コピー) し，メソッドの実行はこのコピーを利用して行なう，コピーに書き込まれた内容については，そのメソッドでの最後の書き込みの後に一括してオブジェクト本体に書き戻す．

このとき，RO メソッドについてメソッドの起動からオブジェクトの内容を一括読み出しする部分までを RO 区間，RW メソッドについてメソッドの起動から一括更新までを RW 区間，一括更新の最中を W 区間とする[1]．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証するため RW 区間実行間の，オブジェクトから一貫性のあるデータを読み出すため，W 区間実行と RO 区間実行の間にそれぞれ必要となる．

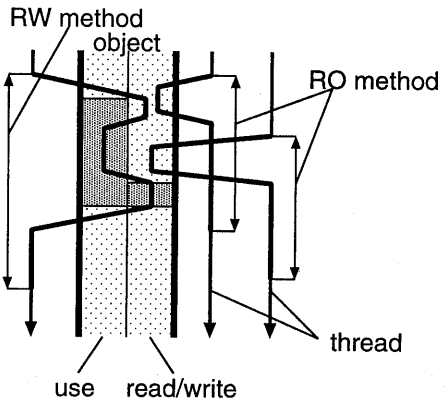

図 1: instant method

これにより，メソッド実行全体を排他制御する場合に比べ，排他制御が必要な区間が短縮され，単一オブジェクトについて同時に複数のスレッドが動作することが可能になる．instant メソッドの動作を図 1に示す．

2.2 実行時メソッド置換

オブジェクトには適切な更新の後，変化しなくなるという特性を持つものが多く見られる．また，これはプログラマに把握されていることが多い．これを利用し，更新を行なわなくなった RW メソッドを RO メソッドに置換することで排他制御が必要な区間を縮小し，並列性を向上させることができる．これは，メソッド内部で処理を分岐させることでは得られない効果である．

メソッド置換の結果，更新されなくなったインスタンス変数は各プロセッサのローカルな環境にコピー可能である．これは分散環境の場合に特に処理効率の改善に有効であると思われる．また，使用するインスタンス変数が全て変化しなくなったメソッドは free なメソッドに置換可能であり，どのメソッドが free になるかははメソッド置換によるオブジェクトの状態の遷移として解析できる．よって，一つ

[1]実際には，後述するように実行時メソッド置換を行なう場合，RO，RW 区間にはメソッドルックアップも含まれる．

のメソッドを置換する際にこれらのメソッドをまとめて置換することで，さらに処理効率の改善が期待できる．ただし，この二次的なメソッド置換については今回実装を行なっていない．

このようなプログラミングを可能にするため，OPAではメソッドの実行時置換を許す．具体的には置換可能メソッド m1 をメソッド m2 に置換する場合，

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

のように記述する．また，置換可能なメソッドには，宣言部にキーワード `replaceable` をつけるものとする．

多重定義されているメソッドに関して setmethod を行なった場合，置換元メソッド，置換先メソッドでシグニチャが一致する組み合わせをメソッド置換の対象とし，これらを一括して置換する．ただし，これは setmethod を実行するメソッドから見えるメソッドのみを対象とし，継承時に多重定義として追加されたメソッドは対象外である．また，継承時のメソッド再定義は置換可能メソッド，置換先メソッドとも通常のメソッドと同様に行なうことができる．

OPA では，スレッド間での通信，同期のため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したスレッドはその場でサスペンドする．他のスレッドによって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．

オブジェクト指向の立場からメソッド置換をみると，これはオブジェクトに送られたメッセージの届き先をスイッチするという考え方ができる．メッセージを送る側から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に捉えることができる．この立場から見たメソッド置換の動作を図2に示す．

ただし，メソッド置換を行う場合，メソッドを取り出すまでそれが RO メソッドか RW メソッドか分からず，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．

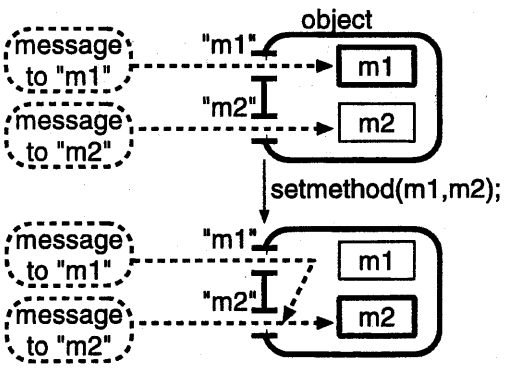

図 2: 実行時メソッド置換

3 実装

SGI 社製共有メモリ型並列計算機 POWER Onyx (MIPS R10000 プロセッサ 12 個，メモリ 2GB 搭載) に対して処理系の実装を行なった．OPA のコンパイラは，OPA のソースコードを C のコードに変換するものである．また OPA のランタイムは，OS のシステムコールにより各プロセッサに一つ仮想メモリ空間を共有するプロセスをを生成し，OPA のスレッドは全てこのランタイム上で管理される．

3.1 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．

RO 区間と W 区間の排他制御，すなわちインスタンスから矛盾のない値を読み出すための排他制御には，バージョン番号方式を用いた．これは，各インスタンスにバージョン番号 (初期値 1) を持たせ，インスタンスを更新するスレッドは，書き込みを行なう際これに 0 をセットし，更新終了後に元の値に 2 を加えたものを書き戻す．読み出す側は，読み出しの前後でのバージョン番号を比較することにより，適切な値が読めたかを判別することができる．この

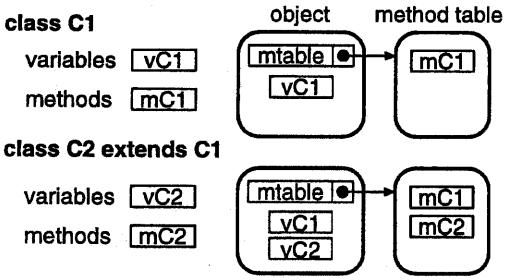

図 3: オブジェクトの実装

方式の利点は，ロックを一切必要としないこと，同時に複数のスレッドが値を読み出せることで，低コストで高い並列性を得ることができる．オブジェクトにバージョン番号をつける方式は [3] にも見られるが，これが二つのカウンタを必要とするのに対し，ここで説明した方法はカウンタを一つで済ましているのが特徴である．

3.2 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，クラス毎に用意される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新しいメンバとしてその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． メソッドの置換情報は各オブジェクトに固有のものであり，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 3 に示す．

メソッドテーブルには，そのクラスの持つメソッドに関する情報 (メソッドタイプ，メソッド本体への参照) が格納される．各メソッドに関して，その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．置換メソッドである場合は，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って，置換情報を保持しているインスタンス変数テーブル内のオフセットが格納される．

置換情報は，置換先メソッドのメソッド情報を保持しているメソッドテーブル内のオフセットからなる．ただし，置換先メソッド名に 2.2節で述べたキーワード suspends が指定されている場合，その旨を示す情報と，このメソッドを呼び出してサスペンドしているスレッドのキューの情報からなる．

3.3 置換メソッドの起動シーケンス

以下に疑似コードによる o.massage(args) に対応するメソッド起動手順を示す．

```
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];
  if (msg != Suspends) goto LOOKUP;
  if (vn) {
    lock(o);
    if (vn != o.vn){vn=0; goto RETRY;}
  }
  suspends(o+i,START);
case RO:
  if (vn && vn != o.vn) goto ROfail;
  status = invoke(method, vn, 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;}
  }
```

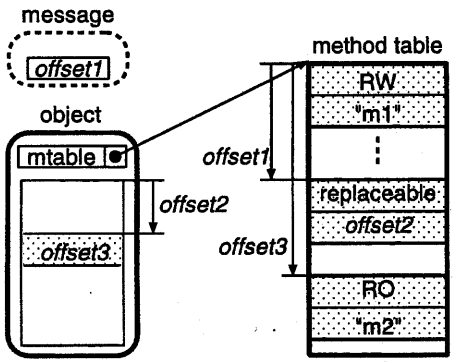

図 4: 置換メソッドのルックアップ

```
  if (o.use) suspends(o,START);
  status = invoke(method, args);
  break;
}
```

メソッドを呼ぶスレッドは，まずオブジェクトが更新中でないことを確認し，その時点でのバージョン番号を保存する．

次にメソッドテーブルを取り出し，メソッドのルックアップを行なう．これはメソッドテーブルからオブジェクトに送られたメッセージで指定された項目を索くことで行なわれ，対象メソッドが非置換メソッドであれば，この時点でメソッドタイプとメソッド関数への参照が得られる．置換メソッドの場合，ここで得られるのはインスタンス変数テーブル上のその置換メソッドに関する情報への参照であり，これに従って再びメソッドテーブルを検索することになる．メソッド置換先が置換メソッドであった場合はその都度メソッドテーブルの検索を行なうことになるが，置換の仕方によっては参照が循環して無限ループとなる可能性もある．これはプログラマの責任とする．置換メソッドのルップアップの様子を図4に示す．

ルックアップ終了後，得られたメソッドタイプに応じて起動を行なう．instant RO メソッドの場合，メソッドのルックアップから，メソッドの起動後，オブジェクトの内容の一括読み出しが終了するまでの間に対象オブジェクトの更新がなければ一切のロックを用いずにメソッドを起動できるが，更新があった場合，メソッドの再置換の可能性により，メソッドのルックアップから処理をやり直す必要がある．このため，一括読み出し後に現在のバージョン番号と保存しておいたバージョン番号を比較し，更新があれば今度は他のスレッドが起動プロセスに割り込まないためのロックをオブジェクトにかけた後にリトライを行なう．このロックは一括読み出し後に解放される．(注：vn=0 でロック後リトライを表す)

また，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オブジェクトの待ちキューにスレッド自身を登録，サスペンドする．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．

これにより，メソッドを取り出すまでメソッドタイプの分からない置換メソッドを安全に起動することができる．

4 評価

実装した処理系について評価を行なった．サンプルプログラムとして二分探索木のクラスを作成した．

これは木に対する探索，登録メソッドを持つクラスで，登録メソッドに関しては，一度左右に子ノードが出来た後は新たな子ノードを追加する (オブジェクトを更新する) ことがないため，RO メソッドに置換が可能である．

各プロセッサにスレッドを 1 または 4 本生成し，合計で 3 万個のノードを登録するのに要した時間を測定した．比較したのは，メソッド置換を用いたもの (do_replace)，メソッド置換を用いないもの (normal)，メソッド置換は用いないが，登録メソッドを置換可能としたもの (replaceable) の 3 種類である．各プログラムを 20 回実行し，実行時間の平均，標準偏差 (表中括弧内に表示) を求めた．測定結果を表 1に，これをグラフ化したものを図 5に示す．

結果，あらゆる場合でメソッド置換を用いたプログラムが高速に動作した．メソッド置換を用いないプログラムと比較すると，プロセッサ数を増加させ

		1PE	2PE	4PE	8PE	12PE
do_replace	1	311.3(4.0)	239.8 (4.8)	153.2 (3.9)	98.4 (4.0)	80.6 (3.1)
	4	311.2(3.5)	242.6 (4.0)	153.3 (4.4)	97.5 (3.4)	78.9 (2.5)
normal	1	472.7(2.6)	698.7(19.3)	510.6 (8.8)	317.8(12.2)	251.1(20.9)
	4	474.7(7.0)	703.9(16.9)	518.7(20.8)	313.6(12.9)	260.2(32.6)
replaceable	1	520.1(2.8)	713.8 (9.3)	514.0(11.9)	313.0 (9.9)	255.2(17.0)
	4	517.7(2.7)	730.6(17.6)	511.4(15.6)	317.9(11.7)	258.2(20.3)

表 1: 実行時メソッド置換の評価 (ms)

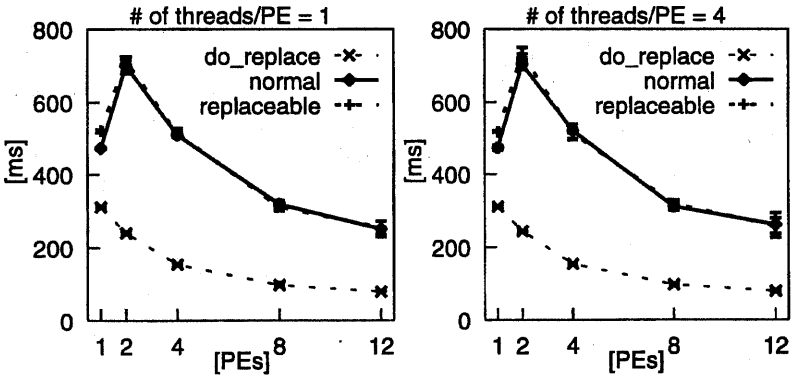

図 5: 実行時メソッド置換と他の方式の比較

たときに特に性能向上が顕著であり，これはメソッド置換によりスレッドの並列性が向上したことを示している．

また，1PE の場合スレッドのサスペンドが発生しないため，純粋なメソッドの処理時間を比較することができる．do_replace と normal の比較により，処理の軽いメソッドに置換することによる処理速度の向上が確認できた．さらに，normal と replaceable の比較により，メソッドを置換可能にした場合のオーバーヘッドは約 10 %であることがわかった．多プロセッサ下でこの差が目立たないのは，全実行時間 (表の実行時間×プロセッサ数) では，スレッドのサスペンドのコストなどがより重要になるためである．

5 結論と今後の課題

オブジェクト指向並列言語 OPA の実行時メソッド置換について述べた．また，共有メモリ型並列計算機 POWER Onyx 上で実装を行ない，メソッド置換が処理の高速化に有効であることを確認した．今後は 2.2節で述べた二次的なメソッド置換による free メソッドの増加について検討，実装を行なう予定である．

参考文献

[1] 八杉昌宏, 瀧和男. 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, Vol. 96, No. 82, pp. 157–162, 1996.

[2] 八杉昌宏, 瀧和男. 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.

[3] Leslie Lamport. Concurrent reading and writing. *Communication of the ACM*, Vol. 20, No. 11, pp. 806–811, 1977.