情報処理学会研究報告 96-PRO-8, Vol. 96, No. 82, pp. 157–162, 平成 8 年 8 月

並列処理のためのオブジェクト指向言語 OPA の設計とその実装

八杉 昌宏　　瀧 和男

神戸大学工学部情報知能工学科

〒 657 神戸市灘区六甲台町 1-1

Tel: 078-803-1189, Fax: 078-803-1218

E-mail: {yasugi,taki}@seg.kobe-u.ac.jp

本研究では，MIMD 型並列計算機における効率の良い並列処理のための，メソッドの実行時置換と構造化されたスレッドによる並列処理を特徴とするオブジェクト指向言語 OPA の設計を行い，共有メモリ型並列計算機用の実装方式について提案を行う．実装方式としては，排他制御にはバージョン番号方式，プロセッサ間通信には lock-less バッファ管理方式，スケジューリングにはスケジューラ再帰呼び出し，関数フレーム二重表現方式を提案する．共有メモリ型並列計算機 POWER Onyx 上で評価を行った結果，特にバージョン番号方式と関数フレーム二重表現方式で高速化が達成された．また，メソッドの実行時置換の効果が確かめられた．

OPA: An Object-Oriented Language for Parallel Processing — its Design and Implementation —

Masahiro YASUGI　　Kazuo TAKI

Department of Computer and Systems Engineering

Faculty of Engineering, Kobe University

1-1 Rokkodai Nada Kobe, JAPAN 657

Tel: +81-78-803-1189, Fax: +81-78-803-1218

E-mail: {yasugi,taki}@seg.kobe-u.ac.jp

An object-oriented language OPA is presented for efficient parallel processing on MIMD computers, which features dynamic method replacement and structured parallel threads. Implementation schemes on shared-memory parallel computers are proposed, including a version numbering method for mutual exclusion, a lock-less buffering method for inter-processor communication, a recursive scheduling scheme and a double function-frame representation scheme for scheduling. The evaluation on a shared-memory parallel computer, POWER Onyx, shows the effectiveness of the version numbering method and the double function-frame representation scheme. It also shows the impact of the dynamic method replacement.

1　はじめに

本研究では，MIMD 型並列計算機における効率の良い並列処理のためのオブジェクト指向言語 OPA(Object-oriented language for PArallel processing) の設計を行い，共有メモリ型並列計算機用の実装方式について提案する．また，共有メモリ型並列計算機 POWER Onyx 上での評価を行った．以下，2節で言語設計，3節で実装ついて述べ，4節で評価を行う．

2　OPA の言語設計

2.1　動機づけ

分散メモリ型，共有メモリ型を問わず効率よくプログラム実行できるような処理系の実装を可能とするとともに，不規則な並列性を含む問題の記述も容易にする言語を設計する．

並列分散環境では，変化しない変数を参照する場合，そのデータを手元においておくことで高速化できる．ところが実際には，オブジェクトの変数はオブジェクトの生成時点から変化しないというよりは，最初のうちは変化するが，適切な書き込みを行われたのちはじめて変化しないようになることのほうが多い．例えるなら，生成されて間もないオブジェクトは成長期にあってそのデータは変化しやすいが，やがて成熟期を迎えると変化しなくなる．このような場合，その時点で，メソッドを読み出し専用のものに置換すればより高速なメソッドにすることもできるし，変数が変化しなくなることをコンパイラに伝えることもできる．

また，並列分散環境では，従来，不規則な並列性を扱うには，複数の比較的独立なプロセスが，必要に応じて通信や同期を行いながら処理を進めることが多かった．この場合，既に存在しているプロセスにタスクの処理を依頼したり，タスクの処理の完了をプロセスから知らせてもらう際に，タスクの処理の完了の同期に関する誤りが容易に記述できてしまう問題点があった．また，タスクの処理の終了後もその間に fork したプロセスが生き残ってしまうことがあり，構造化がなされていなかった．もっと単純に，タスクを分割して新たに生成した複数のスレッドに実行させ，それらのスレッドの終了で，タスクの処理も終わるとすれば，このような誤りや問題が生じにくくなる．

2.2　概略

OPA の言語仕様の概略を述べる．図 1のような C++(JAVA) ライクなシンタックスを用いたオブジェクト指向言語である．メモリ番地としてのポインタは許さず，関数呼び出しの引数は値渡しであるが，オブジェクトや配列への参照が値として利用できる．継承

```
class bin_tree(int key, int value){
 bin_tree left = null, right = null;
 void regist1(int k, int val){
  if(k < key)left.regist(k, val);
  else right.regist(k, val);
 }
public:
 int search(int k, int v){
   if(k == key)return value;
   else if(k < key){
    if(left != null)return left.search(k, v);
    else return v;
   }else{
    if(right != null)return right.search(k, v);
    else return v;
   }
 }
 void regist(int k, int val){
  if(left != null && right != null)
   setmethod(regist, regist1);
  if(k < key){
   if(left != null)left.regist(k, val);
   else left = new bin_tree(k, val);
  }else{
   if(right != null)right.regist(k, val);
   else right = new bin_tree(k, val);
  }
 }
}
```

図 1: Example of Dynamic Method Replacement (Binary Tree Dictionary)

やインタフェースに関しては JAVA[2] のそれを少し改良したものを用いる．また，自動ゴミ集めを用いる．

2.3　オブジェクトと排他制御

Actor[1] ベースの並行オブジェクトモデルでは，オブジェクトはデータ保存場所としてだけでなくプロセスとしての役割を果たしている．このモデルはオブジェクトへの並列アクセスをオブジェクトのスレッドを用いて逐次化することで，オブジェクトの一貫性を容易に保証できたが，オブジェクトとスレッドが分離しておらず，スレッドに対して構造を入れることが難しかった．

これに対し，Monitor のようにオブジェクトとスレッドは分離しおり，オブジェクトはその並列アクセスを排他制御して，一貫性のある状態を保つデータ保存場所とする見方もある．OPA では後で述べるようなスレッドを用いた並列処理を行うため Monitor ベースのモデルを用いる．

2.4　メソッド分類

オブジェクトのメソッドを分類して，それぞれに適した手順で呼び出すことで，メソッド呼び出しを高速化することができる．例えば，オブジェクトのもはや変化しなくなった変数を読み出すだけのメソッドを，free なメソッドと呼ぶことにすると，free なメソッド

のアクセスする変数をコピーしておくことで，(特に分散環境では) 高速なアクセスが可能になる．また，free なメソッドは全くの排他制御なしで起動できる．

メソッドを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) に分けることも意味がある．RO メソッドは一貫性のあるオブジェクトの状態を読み出すようにだけ排他制御してやればよく，読みだした後の排他制御をなくすことができる．

RW メソッドについても，[4] のようにインスタンス変数の更新をメソッドの中の特定の点で一括して行うことで，RW メソッドの排他制御の区間を短くできる．OPA では，コンパイラがフロー解析を用いて書き込みが最後であると判断した点で一括更新を行う．

次の一括更新までの間，RO メソッドは最後に更新した状態が読み出せるものとすると，RO メソッドは必ずデッドロックなしで実行できることになり，RO メソッドと free なメソッドは性能上の違いはあれ，計算モデル上の違いはなくなる．free なメソッドは実装では用いるが，言語としては以下では取り上げない．

RO メソッドと RW メソッドにつき，メソッド起動手順と排他制御する区間を整理すると次のようになる．

RO メソッド： (RO1) メソッドを取り出す．(RO2) メソッドを起動する．(RO3) 一括してオブジェクトの必要なデータを読み出す．(RO4) メソッドの続きを実行する．

RW メソッド： (RW1) メソッドを取り出す．(RW2) メソッドを起動する．(RW3) 一括更新までのメソッドの処理を行う．(RW4) 一括更新を行う，(RW5) 一括更新後のメソッドの処理を行う．

以下では，RW1 から RW4 までの処理の区間を RW 区間，RO1 から RO3 までを RO 区間，RW4 を W 区間と呼ぶ．必要な排他制御は，RW 区間とその他の RW 区間の間の排他制御，W 区間と RO 区間の間の排他制御である．ここで，RO 区間とその他の RO 区間の間の排他制御はしない．また，W 区間は RW 区間に含まれるため W 区間と W 区間の間の排他処理は RW 区間のそれに含まれる．

2.5　実行時メソッド置換

さらに高速化するには，実行時にメソッド置換を行うことで，排他制御を減らしたり，変化しない変数を増やすことができる．図 1の二分木辞書の例では，左右の子ノードを生成した後は，登録時にインスタンス変数の更新を行わなくなるため，`setmethod(regist, regist1)` の部分でメソッド `regist` を RO メソッドの `regist1` に置換している．

メソッド置換ではメソッドとインスタンス変数は合わせて一括更新するものとする．また，排他制御も前節で定義した区間について行う．注意が必要な点は，メソッド置換の影響により，実行しようとしているメソッドが RO メソッドであるか RW メソッドであるかは実際にメソッドを取り出すときまでわからないにもかかわらず，メソッドを取り出す操作自身も排他制御が必要な点である．

2.6　スレッドを用いた並列処理

スレッドを用いた並列処理は，fork-join 型並列処理とパイプライン型並列処理に分けることができる．このうち fork-join 型並列処理は，入れ子構造として構造化を行うことができる．

並列処理をできるだけ fork-join 型並列処理に構造化するため，スレッドを fork したときはその join 先は join ブロックの入れ子のもっとも内側のものと定める．例えば，次のプログラムでは関数 `f1`，関数 `f2` のスレッドを fork しているが，それらの終了は join ブロックの最後で join される．

```
join{ par f1(...); ... par f2(...); }
```

また以下のように `f2` の実行中に `f3` のスレッドを fork する場合，`f3` のスレッドも上と同じ場所に join する．さらにいえば条件付きで fork ができるので実際に join するスレッドの数は実行時に定まる．

```
void f2(....){  ...if(...) par f3(...); ...}
```

一方，パイプライン型並列処理の例には，I-structure やチャネルを通じた通信，バリア同期などがある．これらには複数のスレッドが通信/同期するような並列性の記述が必要となるが，ここでは詳細は述べない．

2.7　プロセッサ間通信

オブジェクト生成時やスレッドの fork 時には注釈を用いてプロセッサの指定が行える．ただし，ユーザは性能の問題を除き，リモート/ローカルの違いを意識する必要はない．

2.8　スケジューリング

処理するタスクの量が有限であるならば，スケジューリングは実行時間に影響を与えるが処理そのものは必ず停止する．そこで，OPA では，スレッドのスケジューリングの fairness は保証しない．処理系はワーキングセットの大きくなりやすい round robin を用いる必要はない．ただし，特定のスレッドが長時間プロセッサを占有すると，クリティカルパスが長くなる可能性があるため，ユーザは yield を陽に指定できる．

3　実装

本節で述べるスレッドのサスペンドや実行再開，他プロセッサへの移動は，すべて言語のランタイムで管理する．また実装は，共有メモリ型並列計算機 POWER Onyx を用いた．実行コードは C のコードを介してコンパイルされる．

3.1　共有メモリ型並列計算機 POWER Onyx のアーキテクチャ

POWER Onyx はバス接続された共有メモリ型の並列計算機である．POWERpath-2 バスはプロセッサ，メモリ，及び I/O 間で高帯域幅なキャッシュ・コヒーレント通信を提供する分割トランザクション・バスであり，1.2GB/秒の実効転送速度を持つ．MIPS R8000 プロセッサは，64 ビット・スーパースカラ・RISC プロセッサであり，クロック周波数 90MHz で 360MFLOPS 及び 360MIPS のピーク性能を持つ．実装に用いた POWER Onyx は R8000 を 12 個，メモリは 2GB 搭載している．

SVR4 ベースの 64 ビット OS，IRIX6.1 では，仮想メモリ空間を共有したプロセスを生成することで，共有メモリプログラミングが可能である．

3.2　排他制御の実装方式

今回は free なメソッドの実装は行っていない．

RW メソッドでは RW 区間–RW 区間の排他制御，RO メソッドでは RO 区間–W 区間の排他制御が必要である．実際にはメソッド置換の可能性があるため，RO メソッドか RW メソッドかわからない場合は，まずは RO メソッドとして処理を開始する．

3.2.1　RW 区間の排他制御

RW 区間の排他制御には，次の二つのアトミック処理と，オブジェクトの使用中フラグとキューを用いる．

RW 区間開始時のアトミック処理: メソッドタイプをチェックして RW メソッドであることを確認する．オブジェクトの使用中フラグのチェックして，使用中ならサスペンドしてオブジェクトのキューで待ち，使用外なら使用中フラグを立てる．

RW 区間終了時のアトミック処理: メソッドの置換を含めた一括更新をし，RO メソッドへのメソッド置換があった場合は全スレッドを，そのようなメソッド置換がなかった場合はスレッド一つを，キューから取り出し ready にする．

ready になったスレッドは，最初のメソッドタイプのチェックに戻って実行を再開する．これは，RW 区間を実行する順番が回ってくるまでに，対応するメソッドが RO メソッドに変わり，RW 区間を実行する必要がなくなっている場合があるからである．RW 区間終了時に RO メソッドへの置換があったときに，全スレッドを ready にするのもこのためである．

ここで，開始時の処理でメソッドタイプが RO に変っていた場合は，処理を中止し，RO メソッド用の処理に切り替える．

上記のアトミック処理は，他のプロセッサが行うアトミック処理と排他制御する．その排他制御を実現するには，ロックを用いるか，常にスレッドをオブジェクトが属するプロセッサへ移動させて実行すればよい．

3.2.2　RO 区間と W 区間の排他制御とバージョン番号方式

RO 区間と W 区間の排他制御としてバージョン番号方式を提案する．提案方式では，オブジェクトにバージョン番号 (初期値 1) を付けて以下の処理を行う．

W 区間: 最初にオブジェクトにバージョン番号を読み出し，代わりに 0 を書き込んでおく．つづけて一括更新を行った後，最初に読み出したバージョン番号に 2 を加えて書き戻す．

RO 区間: 最初にオブジェクトにバージョン番号を読み出し，次にメソッドタイプをチェックして RO メソッドならそのメソッドを起動する．RO メソッドでは，一括で変数の読み出しを行った後，もう一度バージョン番号を読み出す．もし，読み出したバージョン番号が 0 であったり，二回読んだ番号が等しくなければ，リトライする．

ここで，RO 区間の処理でメソッドタイプが RW に変っていた場合は，処理を中止し，RW メソッド用の処理に切り替える．

RO 区間と W 区間の排他制御でも，バージョン番号方式以外に，ロックを用いた方式や，スレッド移動方式を使うことができる．

3.3　join の実現

効率良く join の完了を判定するには，join 構造体に対して重み付き参照カウンタを設け，重み付き参照カウンタが 0 になっているかどうかを調べれば良い．実行時に新たなスレッドの fork を行うときには，そのスレッドはその親スレッドから join 先の参照を重み付きで引き継ぐ．こうすることで join するスレッドの数が実行時にしか分からなくても効率的に join 操作ができる．

3.4　プロセッサ間通信

スレッドを他のプロセッサに生成したり，移動させたりするための通信バッファ(スケジューリングキューを

兼ねる) の管理には，送信元・受信先のペア毎にキューを設けてロックを排した lock-less 方式 [5] を用いる．本研究では，[5] の方式を改良し，送信側が必要なバッファをリンクしていくとともに，すでに受信されたバッファの回収も行う．また，バッファ内のどこまでを通信に使用したのかを示す変数をバッファと同じ構造体に置くことで，書き込みの局所性が高められた．

3.5 スケジューリング

スケジューラがスレッドをどのようにスケジュールするか，また，各スレッドの実行においてその関数フレームをどのように管理するかについて述べる．

3.5.1 スレッドのスケジューリング

基本的なスレッドのスケジューリングは，スケジューラが各スレッドのコードを呼び出すことで行う (実際には後述の関数フレームスケジューラを動かす)．この際，各スレッドのコードはスケジューラの C のスタックを引き継いで利用する．つまり，[3] と同様，各プロセッサのスケジューラは，その一つのスタックを複数のスレッドのために用いる．

実行中のスレッドが，同期待ちなどでサスペンドしなくてはならない場合は，まず「スタックを使用中でかつ ready になっているスレッドの数」を保持するカウンタをデクリメントし，スケジューラを再帰呼び出しする．スケジューラは，そのカウンタが正にならない限り制御を戻さない．スケジューラから制御を戻された時点で ready になっている場合はそのまま実行を続ける．一方，ready となっていない場合には実際にサスペンドして使用していたスタック領域を解放して制御を返す．

3.5.2 関数フレームの管理

スレッドの関数フレームには [3] のように二種類の表現を用いる．一つは通常のように C のスタックの上にとる．もう一つはフリーリストから割り当て，ヒープ上で子の関数フレームから親フレームへのリスト表現となるようにする．OPA ではそれぞれの表現に合わせて二つのバージョンのコードを生成する．OPA の関数の返り値は，C の関数の返り値ではなくプロセッサの変数上に置く．

スタックを用いているときは，子の関数は C の返り値として 0 または，その関数フレームを返す．関数フレームが返されるのはサスペンドしなくてはならない場合なので，子の関数フレームへリンクした関数フレームを作ってリスト表現へ変換する．

リスト表現の場合，次のような単純な関数フレームスケジューラを用いてスレッドを実行する．子の関数は C の返り値として，スケジュールしてほしい親の関数フレームを返す．

```
while(fr){ fr = (fr->f)(pr, fr); }
```

実際には，リスト表現での実行中に新たな関数呼び出しがあった場合はスタックを用いるので，一つのスレッドが，子のほうはスタック表現，親のほうはリスト表現となり得る．

4 評価

評価結果を表 1に示す．評価は，図 1の二分木辞書の regist メソッドを並列に呼び出して，3 万個の木のノードを生成することで行った．1, 2, 4, 8, 12 台のプロセッサを用い，各プロセッサにはスレッドを 1 個または遅延隠蔽のため 4 個生成した．登録する key の値はランダムである (same)．また，key の値の範囲を各プロセッサに分配することで局所性を高めた場合についても調べた (dist)．数値は実行時間 (ms) であり，20 回実行させて平均と標準偏差 (括弧内) を求めた．

4.1 排他制御の実装方式の比較

バージョン番号方式 (vn)，スレッド移動方式 (mv)，ロックを使用した方式 (lock) を組み合わせた四つの方式を表の一番上のブロックに示す．ここで vn.mv は，RO 区間-W 区間の排他制御に vn 方式，RW 区間-RW 区間の排他制御に mv 方式を用いていることを表す．

RW 区間の排他制御については，same についてみると，1PE のときは mv 方式，2PE 以上では lock 方式の性能が良かった．dist についてみると，4PE までは mv 方式，8PE 以上では lock 方式が良かった．これは，mv 方式では，スレッドが移動した結果，負荷が不均等になるためと考えられる．

RO 区間-W 区間の排他制御については，vn 方式が他の方式と比べて良い性能を示した．

4.2 その他実装方式の比較

プロセッサ間通信実装方式については，表 1の vn.mv-lk エントリに，vn.mv 方式においてプロセッサ間通信の部分を通常のようにロックを用いて実装した場合の値を示す．same についてみると，lock-less 方式を用いた vn.mv のほうが常に優れているが，dist についてみるとロックを用いた方式の性能が良かった．これは dist ではプロセッサ間通信があまり起こらないため，lock-less 方式のキューの多さがオーバーヘッドとなったものと考えられる．

スケジューリング方式の比較としては，vn.hpmv エントリに，vn.mv 方式をリスト表現のみを用いてス

same:

		1PE	2PE	4PE	8PE	12PE
vn.mv	1	354.5 (14.8)	779.2 (4.4)	660.9 (11.5)	439.1 (8.3)	347.2 (9.1)
	4	355.4 (12.3)	686.3 (15.0)	510.9 (5.3)	332.9 (8.9)	260.6 (6.4)
mv.mv	1	353.5 (2.5)	2199.9 (20.2)	2991.8(126.5)	2177.0 (54.9)	2067.6 (92.1)
	4	354.0 (2.3)	2260.7(102.2)	2207.6 (69.9)	1710.1 (93.4)	1691.2(195.2)
vn.lock	1	582.4 (12.9)	345.7 (0.7)	206.7 (2.9)	116.8 (3.1)	82.3 (1.9)
	4	559.0 (2.5)	346.2 (0.5)	209.0 (3.9)	115.8 (3.1)	82.5 (2.8)
lock.lock	1	750.4 (99.3)	668.6 (31.1)	457.6 (22.1)	275.9 (13.3)	212.2 (13.5)
	4	726.8 (95.9)	716.7 (34.2)	459.9 (19.7)	271.7 (9.1)	208.9 (9.3)
vn.mv-lk	1	367.8 (2.5)	828.0 (5.2)	718.2 (14.0)	478.6 (9.6)	343.2 (7.5)
	4	366.1 (3.1)	709.2 (14.4)	557.1 (8.1)	360.1 (5.4)	264.8 (5.5)
vn.hpmv	1	537.6 (14.7)	1091.9 (11.4)	842.2 (30.9)	519.5 (9.5)	388.4 (9.4)
	4	531.4 (4.6)	845.0 (17.6)	592.7 (24.7)	351.8 (6.1)	256.8 (5.6)
r0vn.mv	1	385.2 (2.7)	2414.1(324.1)	3240.1(128.0)	2430.4 (73.8)	2135.6(111.4)
	4	401.6 (2.8)	2229.3(177.6)	2484.9 (91.2)	1922.2(114.3)	1671.1(226.6)
r0vn.lock	1	798.2 (2.9)	826.4 (13.9)	470.4 (12.4)	287.1 (6.7)	224.3 (11.1)
	4	829.5 (3.4)	784.9 (17.7)	480.8 (11.0)	288.3 (5.8)	267.9 (75.3)

dist:

		1PE	2PE	4PE	8PE	12PE
vn.mv	1	356.3 (14.1)	188.7 (2.5)	119.3 (15.9)	82.9 (39.5)	124.5 (82.2)
	4	357.4 (12.6)	193.3 (1.9)	142.2 (32.8)	103.2 (27.3)	97.7 (11.8)
mv.mv	1	356.0 (2.7)	745.3 (4.4)	766.8 (53.7)	772.6 (62.4)	814.7 (38.0)
	4	358.4 (12.2)	571.4 (4.9)	637.2 (17.2)	645.2 (15.6)	794.4 (24.3)
vn.lock	1	585.7 (15.6)	280.6 (1.7)	154.7 (7.1)	91.3 (8.9)	68.4 (5.1)
	4	562.7 (3.5)	281.7 (3.6)	157.9 (10.6)	85.9 (5.6)	68.1 (5.8)
lock.lock	1	756.3(100.6)	358.6 (32.8)	241.6 (17.6)	195.2 (29.4)	169.2 (17.8)
	4	727.9 (95.0)	368.6 (33.3)	250.7 (18.1)	191.0 (26.7)	167.5 (22.9)
vn.mv-lk	1	368.4 (2.7)	170.4 (1.8)	122.1 (16.0)	78.4 (11.1)	114.2(109.6)
	4	356.4 (11.8)	172.8 (0.6)	102.5 (6.1)	94.0 (49.3)	89.4 (57.9)
vn.hpmv	1	538.1 (12.8)	332.9 (3.4)	334.0 (59.4)	266.6 (78.1)	227.6(101.1)
	4	534.1 (3.3)	290.8 (9.4)	300.0 (91.5)	244.9(127.8)	177.5 (43.3)
r0vn.mv	1	386.4 (2.4)	860.3 (1.7)	1208.2(143.5)	866.2 (50.7)	856.1 (42.7)
	4	403.3 (2.4)	686.6 (5.5)	744.6 (28.3)	718.3 (16.6)	810.2 (36.7)
r0vn.lock	1	808.6 (16.6)	444.2 (4.1)	304.7 (27.0)	207.8 (23.9)	196.6 (20.7)
	4	835.2 (2.1)	471.6 (8.3)	270.3 (12.2)	212.1 (12.2)	251.2 (22.4)

表 1: Elapsed Time [ms] for Various Implementation Schemes (Average and Standard Deviation)

レッド実行させたときの値を示す．vn.mv 方式のほうが優れていることが分かる．

実行時メソッド置換の効果については，r0vn.mv エントリと r0vn.lock エントリに，メソッド置換を使わないプログラムをそれぞれ vn.mv 方式と vn.lock 方式で実行した場合の値を示す．メソッド置換をしない場合排他制御を減らすことができず，実行時間がかなり大きくなることがわかる．

5 結論と今後の課題

メソッドの実行時置換と構造化された並列処理を特徴とするオブジェクト指向言語 OPA の言語設計について述べた．また，実装方式について提案し，共有メモリ型並列計算機 POWER Onyx を用いて評価を行った．評価の結果，特にバージョン番号方式と関数フレーム二重表現方式については，条件によらず高速化が達成できた．また，実行時メソッド置換えの効果が確かめられた．

今後は，処理系の実装を進めるとともに，オブジェクトのマイグレーションによる負荷の均等化などについて検討していく予定である．

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. *The Java Programming Language*. Addison-Wesley Publishing Company, 1996.

[3] K. Taura, S. Matsuoka, and A. Yonezawa. *StackThreads*: An abstract machine for scheduling fine-grain threads on stock CPUs. *JSPP'94*, pp. 25–32, 1994.

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

[5] 八杉, 瀧. 分散共有メモリ型並列計算機 KSR 上の細粒度並列処理用実行方式の評価. 信学技報 CPSY-95-59(SWoPP'95), 95(210):71–78, 1995.