ハイパフォーマンス
コンピューティング 72－9
（1998．8．6）

# オブジェクト指向並列言語による N体問題の並列化とその評価

[1] 高山英道 [2] 八杉 昌宏 [3] 鎌田 十三郎 [3] 瀧 和男

**概要**

本論文では，$N$体問題をオブジェクト指向並列処理言語OPAで記述しその性能評価を行った．OPAは構造化された並列構文を提供し，並列処理を容易に記述できる．また，オブジェクトの状態に合わせてメソッドを置き換えることで，排他制御を緩和し，効率の良い処理を行うことができる．評価は，共有メモリ型並列計算機上で行った．その結果OPAでは容易に良好な並列度を得られることを確認した．また，C++を並列化したものと比べて1.2倍程度の時間で，処理が行えることを確認した．

# Parallelization and Performance Evaluation of $N$-Body Problem in an Object-Oriented Parallel Language

[1]Hidemichi TAKAYAMA [2]Masahiro YASUGI [3]Tomio KAMADA [3]Kazuo TAKI

**abstruct**

In this paper, we implement a parallel program that solves $N$-body Problem using an object-oriented parallel language, OPA, which we are developing, and evaluate the performance of the program. We confirm that OPA has two preferable properties for parallelizing sequential programs, a structured parallel construct and a dynamic method replacement mechanism. The former allows the user to parallelize the program simply, and the latter allows adaptive method replacement to eliminate mutual exclusion and enable efficient execution of the program. Through our performance evaluation on a symmetric multiprocessor, our OPA programs require only 1.2 times execution time against C++ programs in most cases, and show scalable speed up.

## 1 はじめに

近年の並列計算機の普及に伴い，従来の規則的な並列処理を行う問題だけでなく，不規則な並列処理を行う問題を扱う機会が増えている．しかし，不規則な並列処理を記述するための環境は今だ十分なものではなく，逐次プログラムを記述することに慣れた多くのユーザにとって，利用しやすいものではない．

例えば，C/C++を用いて並列処理を記述することを考える．多くの場合，ユーザは，単純なlock機構や細粒度を想定しない汎用スレッドライブラリを用いて，詳細な記述をする必要があり，プログラムが複雑になり誤りを含みやすくなる．また，プログラミングの困難さのため，単純な並列化アルゴリズム・排他制御手法を採用する場合，望んだ台数効果が得られないことも多い．効率的な並列処理の実現のためには，洗練された同期機構や低コストな細粒度スレッド機構を備えた並列処理用言語が望まれる．

我々が現在開発中であるオブジェクト指向並列処理言語OPA [1,2]は，構造化された並列構文を提供しているため，従来の並列言語に比べても並列処理の記述が容易である．また，OPAは柔軟な排他制御モデルを備えている．どのような排他制御を行うかはコンパイラが判断するためユーザによる排他制御の記述を必要とせず，プログラム中に誤りを含みにくい．さらに，実行時にメソッドを置換することができるため，状況に応じて排他制御を緩和することができる．これによって，ユーザは逐次プログラムを単純に並列化した場合においても，プログラムの持つ排他制御の緩和を容易に行うことで，実行時の並列度を上げることができる．これらの並列処理機構の実装に関しては，効率的な実装方式が提案・実現されており [3]，ユーザは低いオーバーヘッドでこれらの機構を用いることができる．

本論文では，実例として$N$体問題を挙げ，OPA上で容易に並列プログラムを記述できることを示す．また，性能面でもC++を用いた場合と同様，高効率な処理が行えたことを報告する．以下に本論文の構成を述べる．2章ではOPAの特徴について述べる．3章では今回評価に用いた$N$体問題について述べ，その逐次プログラムを並列化する過程を4章で

[1]神戸大学大学院 自然科学研究科
Graduate School of Science and Technology, Kobe University, email:takayama@chagall.cs.kobe-u.ac.jp
[2]京都大学大学院 情報学研究科
Graduate School of Informatics, Kyoto University, email: yasugi@kuis.kyoto-u.ac.jp
[3]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:{kamada, taki}@seg.kobe-u.ac.jp

述べる．また，5 章で並列化したプログラムの評価を述べる．6 章では，今回の評価から得られた知見に基づき議論を述べ，7 章でまとめを述べる．

## 2 オブジェクト指向並列言語 OPA

我々が現在開発しているオブジェクト指向並列処理用言語 OPA は Java [4] の言語仕様をベースとし，我々が提案・開発してきた並列構文，同期機構，排他制御モデルを組み込んだものである．

### 2.1 並列構文

OPA での並列処理は fork-join 構文によって記述され，join ブロックを入れ子構造として構造化されている．スレッドの fork はキーワード par によって行われ，fork されたスレッドは join ブロック終了地点で同期される．そのため，join の同期待ちのバグを起こすことがなく，並列処理を容易に記述できる．具体的に，次のプログラムでは join ブロックの内側でメソッド m1, m2 を実行するスレッドを fork している．ここで fork されたスレッドは，join ブロックの最後で全て join される．

```
join {...; par m1(); ...; par m2();...}
```

### 2.2 instant メソッド

instant メソッドとは，オブジェクトの変数を一括して読み出し・書き込みを行うメソッドのことである．OPA では，instant メソッドはキーワード instant をつけて宣言される．instant メソッドは，オブジェクトへの書き込みの有無によって，RO メソッド・RW メソッドにコンパイラによって自動的に分類される．以下に，RO メソッドと RW メソッドのメソッド取り出しとメソッド呼び出し手順，および排他制御を行う区間を述べる (図 1 参照)．

**RO メソッド:** (RO1) メソッドの取り出し．(RO2) メソッド起動．(RO3) オブジェクトの変数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) メソッドの残りを実行．

(RO1) から (RO3) までを RO 区間，(RW1) から (RW5) までを RW 区間，(RW5) を W 区間とする．

RW 区間の実行とその他の RW 区間の実行，W 区間の実行と RO 区間の実行は排他的に行われる．したがって，RO メソッド同士は並列実行が可能である．また，RO メソッドと RW メソッドの同時実行時には RO，W 区間が共に短いため，それらの区間の実行が衝突することはほとんどない．しかし，RW メソッドとその他の RW メソッドでは，メソッド本体の実行を含めほとんどの処理が排他的に行われるため，ボトルネックにならないよう注意が必要である．

### 2.3 実行時メソッド置換

RW メソッドの中には，プログラムの進行にしたがって，実際にはオブジェクトの更新を行わなくなるものがある．このような RW メソッドを呼び出し続けることは，不必要な排他制御を行うことになる．OPA ではオブジェクトの更新を行わなくなった RW メソッドを，対応する RO メソッドに動的に置き換えることが可能である．これを**実行時メソッド置換**と呼ぶ．状況に応じて実行時メソッド置換を行うことにより，不必要な RW 区間を削減でき，効率の良い並列処理が行うことができる．

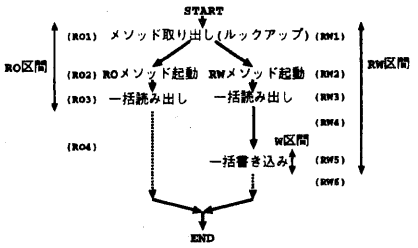


図 1: instant メソッド

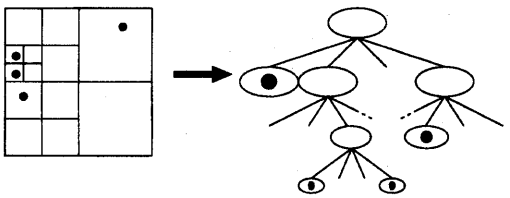

図 2: セル木構造

## 3 N体問題

N体問題とは N個の質点が互いに引力を及ぼし合いながら運動する，ニュートン力学系をシミュレートし質点の位置と速度を各タイムステップごとに求める問題である．

### 3.1 Barnes & Hut アルゴリズム

Barnes & Hut の近似アルゴリズム [5] は，十分離れている質点群を重心近似して引力を求めるもので，木の構築と力の計算を各タイムステップごとに繰り返す．各タイムステップごとの計算量は $O(N \log N)$ である．

まず，全ての質点を含む立方体のルートセルを考え，それを半分の大きさの 8 個の立方体セルに分割する．このセルの分割を，セルに含まれる質点の数が 1 以下になるまで再帰的に行う．このように分割された空間は，セル木構造で表現される (図 2 参照)．各セルは，セルに含まれる質点群の重心の情報を持つ．単一質点を含むセルでは，その質点そのものの情報を持つ．ある質点 $x$ に及ぼされる力の総和を求める際は，ルートセルから始めて，そのセルが質点 $x$ にとって近似可能な距離にあるかどうかを調べる．近似できない場合には，その子セルについてこれを再帰的に行い，質点 $x$ にかかる力の総和を求める．

### 3.2 並列化する上で問題点

N体問題を解く場合，その計算のほとんどは力の計算に費される．ただし，力の計算では各質点をプ

```
public instant void insert(particle){
  if(初期状態である){
    index0 = 対応インデックス(初期質点);
    child0 = セルを作る;
    this.child[index0] = child0;
  }
  index0 = 対応インデックス(particle);
  if(対応する子セルなし) {
    child0 = セルを作る;
    this.child[index0] = child0;
  }
  else children[index0].insert (particle);
}
```

図 3: 木の構築部分の OPA 上での逐次コード

ロセッサに適切に割り当てるだけで容易に並列化することができ，台数効果も大きく引き出せる．一方，木の構築部分の計算量は少ないが，逐次アルゴリズムを単純に並列化すると，排他制御のために木の根付近でボトルネックが生じ台数効果がほとんどでないことが多い．これは，数倍程度の高速化では問題にならないが，数十倍の高速化を行う場合には無視できなくなる．

# 4 OPA 上での並列化

OPA では，前節で述べたようなボトルネックの原因である排他制御を，実行時メソッド置換を用いることなどで容易に緩めることができ，高い並列度を得ることができる．本章においては，どのようにして OPA 上で排他制御を緩めていくことができるかを，具体的に木の構築部分のプログラムを並列化していく過程を通して説明する．

## 4.1 実行時メソッド置換を用いたボトルネックへの対応

まず，木の構築部分の逐次プログラムを図 3に示す．木の構築は，木構造のルートセルの `insert` メソッドを呼び出すことで行われる．各セルオブジェクトは質点の情報を 1 つだけ持つ状態で生成される．新たに質点を登録する場合は，適切な子セルに与えられた質点を登録し，必要であれば新たな子セルを生成する (セルの分割)．これを再帰的に行うことで木を構築する．

このアルゴリズムを OPA 上で並列化するには，ユーザは単純に木に質点を並列に登録すれば良い．しかし，このままでは insert メソッドが排他制御区間の長い RW メソッドであるために，木の根本付近の計算が逐次的に実行され，台数効果がほとんどでない．したがって，台数効果を出すためには，プログラムから定まる排他制御を緩めることが必要である．まず，実行時メソッド置換を用いて RW メソッドを RO メソッドに置き換えるために，セルの状態を初期状態を含む 3 つの状態に分け，その状態ごとに新たな質点が登録された時の処理を考える

**状態 1 :** 子セルはなく，質点情報を 1 つのみ持つ状態 (初期状態)．
**処理 :** 初期質点と与えられた質点を含む子セルを生成，登録，状態 2 へ．
**状態 2 :** 子セルはあるが，すべては揃っていない状態．
**処理 :** 与えられた質点を含む子セルを生成．既にある場合は，そのセルに登録．子セルが全て揃うと状態 3 へ．
**状態 3 :** 全ての子セルが存在する状態．
**処理 :** 与えられた質点を含む子セルに登録．

各状態に対応したこれらの処理をそれぞれ別のメソッドとする．さらに，実行時メソッド置換を用いて，これらのメソッドを状態に合わせて随時置き換えて用いる．状態 1, 2 での処理は子セルの生成を行うため，いずれも RW メソッドである．一方，状態 3 での処理は子セルに登録を行うだけなので，RO メソッドである．したがって，状態 2 から状態 3 へ遷移した後は，排他制御区間の長い RW メソッドが呼び出されることがなくなり，高速な処理が可能になる．

## 4.2 各メソッド内での RW 区間の短縮

この節では，各 RW メソッドの排他制御区間を短く抑えるための改良点を述べる．ある RW メソッドの中で排他制御が行われるのは，メソッド開始時からオブジェクトへの**最後の書き込み**が行われるまでである (RW 区間)．したがって，各処理を実装するに当たり，OPA 上では以下のような注意を払うことで RW 区間の短縮を図ることができる．

(1) オブジェクトへの書き込みをできるだけ**早い**段階で行う．(特にメソッド呼び出しとの前後関係に注意を払う)
(2) 必ず行われるとは限らない書き込み操作などは，独立した RW メソッドに切り分ける．これによって，元のメソッドの RW 区間を削減，もしくは元のメソッドを RO メソッドとして扱うことができる．(但し，分離されたメソッドが見るオブジェクトの状態は，元のメソッド起動時以降のものである可能性があることに注意)

以上のことを，OPA 上で実装した場合のソースコードを図 4 に示す．このコードでは，子セルオブジェクトの生成部分を独立したメソッド (`divide` メソッド) に切り分けている．このことで，`insertF1` メソッドが RO メソッドになる．また，`insert` メソッドでの最後の書き込み点であるメソッド置換 `setmethod` を，子セルを生成する前に行うことで，RW 区間を短縮している．

# 5 性能評価

## 5.1 評価環境

評価環境について述べる．使用した計算機は，SGI Cray 社製共有メモリ型並列計算機 (SMP)
POWER Onyx (MIPS R10000 (195 MHz) × 12 台，メモリ 2GB ) である．また，テストデータとして一様分布な 10 万個の質点データを使用した．

## 5.2 全体の評価

この節では，$N$ 体問題を通して，OPA 処理系の全般的な評価を行う．具体的には，前述の OPA プログラムの実行時間を，C++ で記述された逐次版，並列版それぞれのプログラムと比較し，その性能比，ならびに台数効果を評価する．

```
public instant replaceable void insert(particle){
  ....
  setmethod(insert, insertF1);
  index0=対応インデックス(初期質点);
  if(対応する子セルなし) divide(particle);
  else children[index0].insert(particle);
  index0=対応インデックス(particle);
  if(対応する子セルなし) divide(particle);
  else children[index0].insert (particle);
}
public instant void insertF1(particle){
  index0 = 対応インデックス(particle);
  if(対応する子セルなし) divide(particle);
  else children[index0].insert (particle);
}
public instant void insertF2(particle){
  index0 = 対応インデックス(particle);
  children[index0].insert (particle);
}
```

図 4: 木の構築部分の OPA 上での並列コード

まず，並列実行時の実行時間ならびに台数効果を評価する．木の構築部分，力の計算部分の各々について，使用するプロセッサ数を変化させながら処理時間を計測した．測定結果として，木の構築部分 (maketree) と力の計算部分 (calforce)，及び C++ を用いた場合の力の計算部分 (calforce(C++)) の処理時間と台数効果を図 5 と図 6 にそれぞれ示す．対象となる C++ プログラムは OPA のものと同様の処理を行っている．但し，力の計算部分ではオブジェクトへの書き込みが力の総和の書き込みだけであるため，一切の排他制御は行っていない．また，OPA，C++ ともに，スレッドはプロセッサ数と同数だけ生成している．木の構築部分では台数効果が 4 台以降は全く出ていない．このことに関しては，我々の排他制御の緩和が成功していたかも含めて，次節で詳しく評価する．一方，力の計算部分では OPA，C++ のどちらを用いたものも，ともに良好な台数効果が得られた．また，この部分での OPA の処理時間は C++ の 1.2 倍程度であり，高速な処理が行えている．力の計算部分において，C++ と比較して OPA の処理系のオーバーヘッドとして考えられるのは，処理の一貫性保証のための排他制御と，スレッド管理のためのコストである．特に排他制御やスレッド管理のためのメモリ領域の使用のために，OPA のオブジェクトの大きさが C++ のものよりも大きい．そのためにキャッシュミスが増加していたことが原因と考えられる．これは今後さらなる改善を行うべき点である．

次に，逐次実行時の実行時間の比較を行う．表 1 に逐次実行時の OPA と C++ の実行時間を示す．対象となった C++ プログラムは逐次用に書かれたものである．また，オブジェクト生成時の `malloc` のコストを削減するため，`malloc` はプログラムの先頭で一回だけ実行し，オブジェクトの生成時にはそのメモリ領域を切り分けて利用している．計測の結果，力の計算部分 (calforce) では OPA は C++ の約 1.2 倍遅い程度であるが，木の構築部分 (maketree) では約 2 倍遅い．これは，OPA のオブジェクト生成のコストが高いことが原因の一つだと考えられる．このことは，次節で用いたテストプログラムでの結果からも裏付けられている．また，OPA プログラムには逐次実行用の変更は加えていないため，木の構築部分において，RW メソッド実行時の排他制御や実行時メソッド置換が行われている．これらの処理のコストも処理速度の低下の原因だと考えられる．

表 1: 逐次実行時の OPA と C++ の比較

| | OPA [sec] | C++ [sec] | OPA/C++ |
|---|---|---|---|
| maketree | 1.205 | 0.591 | 2.04 |
| calforce | 25.47 | 21.80 | 1.17 |
| all | 28.73 | 22.65 | 1.27 |

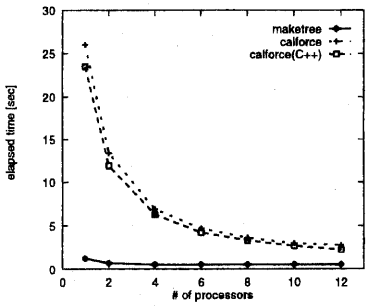


図 5: 各部分の処理時間

全体の処理時間については，C++ と比較しても遜色のない結果 (C++ の 1.27 倍) が得られており，OPA の行っている排他制御やスレッド管理のためのコストなどを考えれば，十分な処理速度だと言える．

### 5.3 木の構築部分の評価

図 6 では，木の構築部分の台数効果が 4 台以降では出ていなかった．本節では，台数効果が出ていなかった原因を探るとともに，我々の行った排他制御の緩和によって，十分な並列度が出ていたのかを調べる．

ボトルネックの発生によって十分な並列度が得られていなかったのかどうかを確かめるために，プロセッサのビジー時間とアイドル時間の内訳を調べた．図 7 に，全プロセッサのビジー時間とアイドル時間の総和を示す．ここでいうアイドル時間とは，主にプロセッサが全ての仕事を終え何もすることがない

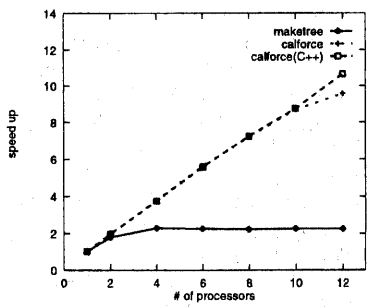


図 6: 各部分の台数効果

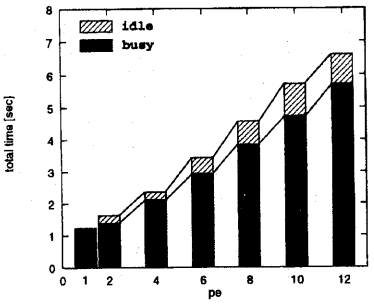


図 7: 処理時間の内訳

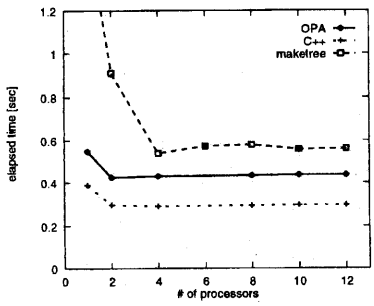


図 8: テストプログラムの処理時間

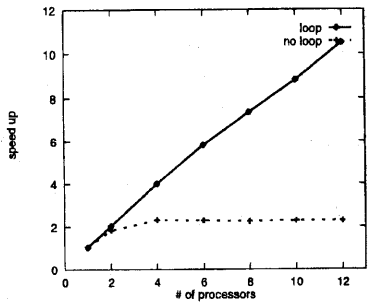


図 9: 空ループ付加時と付加していない時の台数効果

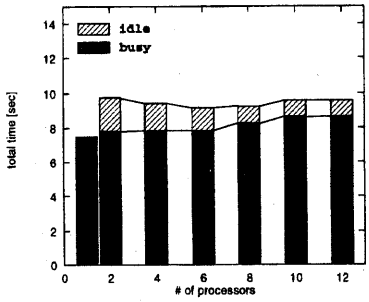


図 10: 空ループ付加時の処理時間の内訳

状態である時間と，RW 区間実行間衝突のためにプロセッサに実行可能な仕事がなかった時間のことである．RO 区間と W 区間の衝突は 12 プロセッサ使用時で高々1 回であったため，ここでは無視する．

測定の結果，プロセッサ数が増加してもアイドル時間はあまり増えてないことがわかる．しかし，ビジー時間は大きく増加している．これらのことから，台数効果が出ていない主な原因は，RW 区間同士の衝突によるアイドル時間の増加ではないということがわかる．ビジー時間の増加の原因としては，

(1) OPA の RW 区間での排他制御のためのコスト．
(2) 頻繁なキャッシュの無効化による速度低下．
(3) アクセスの多さによるメモリ反応時間の低下．

などが挙げられる．

結論から述べると，原因は複数のプロセッサからのメモリへの書き込みが多発し，メモリバスが飽和していたことであった．このことを確認するために以下に述べる実験を行った．C++ と OPA のそれぞれで，今回用いたセルオブジェクトとコード上同程度のサイズのオブジェクトを並列に生成するだけのテストプログラムを用意し，計測を行った．また，逐次実行での評価の時と同様に `malloc` はプログラムの先頭で一回だけ実行する．生成したオブジェクトの個数は 10 万個である．計測した結果を図 8 に示す．図 8 の maketree は，先程の OPA での木の構築時間である．C++，OPA のいずれも，ほとんど台数効果が得られていないことがわかる．これは，メモリ領域へ書き込みを短時間に数多く行ったために，メモリバスが飽和しているためである．木の構築時には，このテストプログラムより多くのセルオブジェクトが生成されている．また，木の構築時間と単にセルオブジェクトを生成するだけの時間に大きな差がない．これらのことから，メモリバスが飽和しているために，木の構築時間のほとんどがオブジェクトの生成に費されていることがわかる．そのため，台数効果が出なかったのである．OPA が C++ の 1.4 倍程度遅い原因は，OPA にはプログラム上に現われない，並列処理に必要なメモリ領域を，確保しているためだと考えられる．

次に，メモリバスが飽和しない状況で，どの程度の台数効果が出るかを調べた．各メソッド起動時に適当な回数の空ループをいれることで，メソッドの実行速度を一律に低下させ，メモリバスへの負荷を軽減させた．ただし，空ループは各メソッドの排他制御区間内にいれ，排他的に処理を行う区間も延長している．この時の木の構築部分の台数効果を図 9 に示す．比較のため，空ループをいれない時の台数効果も同時に示す．また，図 10 には空ループを含む場合の全プロセッサのビジー時間とアイドル時間の総和を示す．空ループを含む場合では，台数効果が大きく出ているのがわかる．また，アイドル時間もほとんど増えていないことから，効率良く排他制御が行われ，並列度が上がっていると言える．

これらの結果から，メモリバスのバンド幅が処理速度に対応し切れなかったために，空ループを含まない場合の木の構築で台数効果が出なかったと言える．また，空ループを含まない木の構築でも，排他制御が十分に緩和され，並列度は向上していたことがわかる．

## 6 議論

### 6.1 並列プログラミング環境

この節においては，実際に逐次プログラムの並列化の過程で，プログラムの実行性能の向上を図るには，言語処理系がどのようなツール群を提供すべきかを議論する．これは，本研究の性能評価の過程で得られた知見に基づくものである．我々の実験は，対称型共有メモリ計算機を対象としたものだが，同様の議論は他の計算環境においても成り立つと考える．

我々はプログラムの速度低下の原因を探るにあたり，(1) 排他処理の衝突によるプロセッサアイドル時間の増加，(2) 複数プロセッサによる同一メモリ領域へのアクセスから起こるキャッシュの無効化による速度低下，(3) メモリバス等のプロセッサ以外の計算資源の飽和による速度低下のいずれかであると仮定して，仮説を個々に検証している．しかしながら，これらの調査を一般ユーザが行うための環境が整っているとは必ずしも言えない．

まず，(1) の調査を容易に行えるためには，処理系の挙動についてユーザが知るのことのできる枠組が提供されていなくてはいけない．今回の調査のために，OPA 処理系にはビジー/アイドル時間が切り替わる箇所などにフックを準備した．これにより，ユーザはボトルネックの発生を容易に知り，対処することができると考える．また，(2)，(3) の調査については，ユーザにはメモリ使用に関する一般的なプロファイラしか提供されていないのが現状である．このため，プログラムの実行状況に応じて，メモリ使用状況を示すようなツールが望まれる．これについては，プログラムの再演機構等と組み合わせることで実現可能ではないかと考えられる．

### 6.2 実行時メソッド置換の有用性

本論文で我々の主張してきた実行時メソッド置換の意義をまとめると，以下の通りである．(1) 各メソッドの実行に必要な排他制御が，オブジェクトの状況にしたがって変化していることを，ユーザに意識させ，(2) 状況に応じて使用するメソッドを置換することで必要に応じた排他制御を行うことができる．

今回の $N$ 体問題では，結果的に (2) の意味においては，実行時メソッド置換は必ずしも必要ではない．これは 4 章での改良にさらに改良を加えると `insert`, `insertF1`, `insertF2` メソッドが全て RO メソッドになるためである．つまり，これらのメソッドは全て RO メソッドとなり，実行時メソッド置換は排他制御の緩和には影響せず，オブジェクトの状態に対する条件分岐としてのみ働くことになる．

但し，注意しておきたいのは 4 章の改良は必ずしも全ての状況で行えるわけではない．先の例の場合，`divide` メソッドによるオブジェクトの状態の更新は，呼び出し元の `insert` メソッドには反映されていない．このため，複数の状態の変化を何度も行うようなメソッドの場合には不都合が生じる可能性がある．

加えて，実行時メソッド置換の有効な利用法を，排他制御の緩和以外にもう一つ紹介しておきたい．OPA では，`free` メソッド [3] と呼ばれる，変化しないオブジェクトの変数のみを用いるメソッドを解析することができる．`free` メソッド の特筆すべき性質は，分散メモリ環境の実装において，定数化したオブジェクトの変数を呼び出し側ノードにコピーしておくことが可能であり，このため，大幅な効率化を図ることができる．つまり，実行時メソッド置換によって，排他制御の緩いメソッドに切り替えるだけではなく，遠隔メモリアクセスのコストを抑えた `free` メソッドへの切り替えを行うことも可能である．但し，今回の実験においては，SMP を対象とした実験であったため，`free` メソッド については評価していない．

## 7 まとめ

今回，$N$ 体問題の逐次プログラムを OPA を用いて並列化し，その性能を評価した．まず，OPA の柔軟な排他制御モデルを用いることで逐次プログラムを容易に並列化が行えることを示した．性能面では，木の構築部分で今回の評価に用いた計算機のメモリバスが飽和し，処理系の評価が十分に行えたとは言い難い．しかし，メモリバスが飽和しないような仮想環境では，良好な台数効果が得られたことから，排他制御を効率良く行えたと言える．また，力の計算部分では C++ の 1.2 倍程度の時間で処理を行うことができ，十分な台数効果も得られることがわかった．

今後の課題としては，排他制御機構やスレッド管理機構の効率化が挙げられる．特にメモリ使用量が，実行効率に影響を与えることを考慮し，メモリ使用量の削減を図る必要がある．

**謝辞**

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] 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] 江口重行, 可児亜祐美, 八杉昌宏, 瀧和男. 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. 並列処理シンポジウム JSPP'98, pp. 111–118, Jun 1998.

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

[5] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, Vol. 324, pp. 446–449, 1986.