リンク構造を扱う共有メモリ型並列プログラムの自動最適化

鎌田，十三郎

(Degree)
博士（工学）

(Date of Degree)
2004-03-12

(Date of Publication)
2008-11-07

(Resource Type)
doctoral thesis

(Report Number)
乙2740

(URL)
https://hdl.handle.net/20.500.14094/D2002740

博 士 論 文

リンク構造を扱う共有メモリ型
並列プログラムの自動最適化

平成 16 年 1 月

神戸大学大学院自然科学研究科

鎌田　十三郎

概要

本論文は，リンクを有する共有データ構造を対象とした並列プログラムに関する，自動高速化／最適化技術を取り扱った論文である．具体的には，(A) プログラムの振る舞いの変化を捉えた変数アクセス解析手法と (B) その解析結果を踏まえた排他制御の緩和技法，加えて，(C) 共有メモリ計算機に適したオブジェクト配置法の自動メモリ管理機構への統合と，(D) 分散メモリ計算環境のためのオブジェクトキャッシュ技術を取り扱う．

並列計算は大きな計算パワーを提供できる一方で，効率的なプログラムの実現には，しばしば専門的な知識や，ユーザの詳細なコーディング能力が必要となる．リンクデータ構造などを対象とした不規則な計算においては，現状では自動並列化などによる最適化はあまり期待できない．

効率的な並列プログラムを実現するためには，プログラマは様々なことに気をつけてプログラムを記述する必要がある．まず，並列プログラムである以上，アトミックに操作すべき箇所に十分な排他制御を施した，正しい並列プログラムを記述しなくてはいけないが，一方で，排他制御区間を短くしないとプログラムがボトルネックを発生してしまう．加えて，各種計算資源をどのように配置するかが問題となる．分散環境においては，共有オブジェクトへのアクセスはノード間通信を伴うため，通信コストの削減のためにはオブジェクトのキャッシュを参照側に配置する必要がある．その際，プログラマはオブジェクトの性質を把握した上で，データ一貫性を考慮したキャッシュを実現する必要がある．一方で，共有メモリ型計算機においても状況はそれほど単純ではない．分散計算環境ほどのコストではないが，プロセッサ間でデータを共有するためには，キャッシュのコヒーレンスミスというペナルティを払う必要があり，状況によってはメモリコンテンションを引き起こすことすらある．このため，プログラムを高速化するためには，しばしば多大な実装技術と職人的コーディング技術が必要とされ，並列計算が一般用途に普及するための障害になっている．

このような現状を打開するため，本研究はプログラマには当初は正しい排他制御を行うプログラムの実現に専念してもらい，その後の高速化に関しては，多くを処理系最適化に任せるための方法を模索する．但し，不規則データを対象としたプログラムの解析は一般には難しいため，問題によってはユーザの知識をヒントとしてプログラム中に記述してもらい，あるいは対象プログラムをサンプル実行しプロファイル結果を取得することで，最適化を施していく．

排他制御の問題は，複数スレッドで共有されたデータ構造に対し変数のアクセスを解析し，データ競合が起らない範囲で排他制御区間を短縮することにある．但し，そのためには変数が更新されない区間を求める必要があるが，複数スレッドから共有される変数の状況変化を解析することは従来難しかった．本研究では，ユーザが重要だと考えるオブジェクトの状態を局面として捉え，局面毎のオブジェクトの振る舞いを解析することで，変数への更新がない区間を解析し，排他制御の緩和を実現している．

本解析結果は，効率的分散オブジェクトの実現にも応用可能である．分散環境では通信コストの削減が重要であるが，本解析結果を用いることで，ある局面の間更新を行わないと分かった変数を，参照側のプロキシの一部としてキャッシュし，遠隔メソッド呼出しの削減という積極的な最適化技法が可能となる．そのためにも，局面と変数更新に関する精度の高い解析が必要とされる．

一方で，共有メモリ型並列計算機上でのオブジェクトレイアウトの問題は，より微妙な最適化が必要である．というのは，前述のようにプロセッサ間の共有変数を介した値の伝達には，キャッシュのコヒーレンスミスを伴うが，他のキャッシュミスに比して絶対的に大きいとも言えないからである．つまり，キャッシュの無効化だけでなく，空間的局所性やキャッシュ密度を総合的に考慮したレイアウトの決定を行う必要がある．本研究では，共有メモリ並列計算機向けのオブジェクトレイアウト法を提案し，自動メモリ管理機構に統合することで，プログラマの明示的メモリ管理からの解放を目指す．本システムは，プログラムの実行プロファイルからオブジェクトの変数アクセス状況を取得し，それに基づき適切なレイアウトへの自動変換を行う．

本論文では，これら実装技術に関して実際に実アプリケーションを通した評価を行い，その有効性を示すことができた．また，対象言語モデルや開発ターゲットとして Java 言語という一般のオブジェクト指向言語を選んでおり，今後，一般的に利用可能な技術となることを期待している．

目 次

図 目 次

表 目 次

第 1 章

緒論

近年，並列プログラムを実行可能な環境は広がりを見せている．一つの動きとして，共有メモリ並列計算機が一般のサーバ目的として導入される事例が増えていることが挙げられる．もう一つの動きは，Grid[21, 20] やクラスタ [1] に代表されるような，既存計算機をネットワーク接続して高並列計算を行おうと言う動きである．汎用の PC は高性能かつ低価格化しており，高いコストパフォーマンスで大きな計算パワーを得ることができる．以上の動きに加え，最近は LSI 技術の進歩により，単一チップ内で複数スレッドを並列処理することも可能となり，一般 PC でも利用可能となっている．このため，一般利用者にとっても並列プログラムを実行するための環境は整いつつある．

並列処理を行う目的は，大規模な計算を多くの計算資源を利用することで短時間で解くことにある．その計算内容を小さい計算に分割実行できる場合は，確かに問題を短時間で解くことは難しくない．一方で，互いに密に関連した計算を実行する場合，問題を高速に処理することはしばしば難しく，その高速化には多くの労力が必要である．その原因としては，

- プログラムが，多くの排他実行区間を持つため並列度が低く，ボトルネックを発生する
- データのメモリ配置などによっては，ハードウェア上でプロセッサ間通信のコストのオーバヘッドが大きくなる（場合によっては，1CPU で計算した場合より遅くなることもある）

などの理由がある．このため，並列プログラムの高速化にあたっては，排他制御区間の削減や適切な資源配置が重要である．高速化に必要な各種実装技術については，サンプルを通して 2 章にて行うが，これらの最適化には対象プログラ

ムに関する十分な知識とともに，並列プログラムについての十分な理解，ハードウェア上の実行コスト感覚があわせて必要である．このため，複雑な問題に関しては，それが重要なプログラムやライブラリであった場合のみ，専門知識をもったプログラマが労力をかけてチューニングを行うというのが現状である．

本論文は，不規則データ構造を扱う並列プログラムを対象とした，処理系による自動最適化技術を取り扱った論文である．基本的には，オブジェクト指向言語などの構造化された，参照を有する言語を対象とする．内容は以下の通りである．

- プログラムの振る舞いの大きな変化 (局面変化) を捉えた変数アクセス解析手法
- 局面解析結果を踏まえた排他制御の緩和技法
- 分散メモリ計算環境のためのオブジェクトキャッシュ技術
- 共有メモリ計算機に適したオブジェクト配置法と，その自動メモリ管理機構への統合

本論文が目指すプログラミングスタイルは，プログラマには当初は正しい排他制御を行うプログラムの実現に専念してもらい，高速化に関しては，処理系の各種最適化支援のもと，プログラマがチューニングをおこなうものである．不規則データを対象としたプログラムの解析は一般には難しいため，全自動の最適化は難しく，プログラマの知識をヒントとして与えながら，あるいは，対象プログラムのサンプル実行を通してボトルネックを検出しながら高速化を目指すこととする．

システムが施す最適化技術の中には,「かならず成立する性質」を必要とする最適化と,「多くの場合成立する性質」が分かっていれば十分な最適化手法が存在する．例えば，ある変数には一切更新が無いと分かっていれば，変数へのアクセスを排他制御区間から外すことも可能である．一方で，ほとんど更新がないと分かっている場合は，書き込み時のペナルティを承知した上でデータ読み出しコストを重視した最適化が可能となる．本論文が，解析部を持つのは「かならず成立する性質」を取得するためであり，一方で，共有メモリ上のメモリ配置などでは「多くの場合成立する性質」が積極的に利用される．

但し，3 章で述べる解析は，必ずしも，プログラム実行中，常に成立する性質を検出するものではない．本解析の特徴は，プログラム実行を複数の局面に分割して捉え，各局面で成立する性質を解析する事にある．大きくプログラムの振る舞いが変化するポイントを，プログラマから局面変化として教えてもらうことで，各局面毎に成立する性質を検出し，それを最適化に利用することができる．従来の解析器では，常に成立する性質や局所的に成立する性質しか分からずに，共有オブジェクトの最適化に十分と言えなかったが，本解析では，プログラマの補助のもと各局面毎に安定した性質を取得できる．

4 章の排他制御緩和は，局面解析結果を利用して行われる．局面間の遷移関係と，局面毎の変数アクセス状況解析から，

- 今後，一切更新が行われない変数
- 局面内で更新が行われない変数
- 更新が行われる変数

を決定し，排他制御の緩和を行う．本手法の特徴は，一定期間定数化した変数に関してのみ，アクセス時期をずらして排他制御区間外に移動するものである．このため，言語のメモリアクセスモデルへの変更を必要としないことである．並列オブジェクト関係で行われてきた排他制御区間短縮に関する従来研究 [43, 50] が，逐次言語と異なる変数アクセス順序のモデルを採用していたのに比べ，本手法は一般プログラムに対して容易に適用可能であると言える．

5 章で取り扱うのは，局面解析に基づいた分散メモリ計算環境における効率的分散オブジェクトの実現方式である．一般に分散環境においては通信コストが大きな実行コストを占め，ネットワーク上で共有されたデータの取り扱いが重要である．オブジェクトを単一ノードに配置し，毎回遠隔アクセスを行ったのでは速度低下を招くが，一方で，参照側にオブジェクトの状態をキャッシュするためにはデータ一貫性の問題を解決する必要がある．本研究では，局面解析の結果をつかうことで解決を目指す．つまり，ある局面で安定化したデータは，その局面の間参照側ホストにキャッシュ，参照側ノードでのメソッドのローカル実行を許し，局面変化する際にキャッシュの無効化を行うというものである．但し，そのためにはキャッシュ対象データに関する参照の扱いのモデル化，また，局面とデータアクセスの関係について正確な解析が必要となる．5 章では，局面解析を用いた各アクセス命令の分類，ならびにキャッシュの一貫性を

保つためのメソッドの実行ルールを提案する．プログラマが分散環境において効率的な実行を容易に得るためには，さらに，プログラマがデータ分散を容易に記述し局所性向上を図れるための方策，遠隔アクセスに対する遅延隠蔽のためのマルチスレッド技術などについて，更なる研究が必要と考えるが，本研究で状況毎のデータアクセス傾向を解析によって保証し，それにより一貫性のあるキャッシュ機構を低実行コストで提供できる意義は大きいと考える．

6 章で取り扱うのは，共有メモリ型並列計算機上のメモリ配置の問題である．共有メモリ型並列計算機では，各プロセッサとメモリが強力なネットワークで接続されているが，資源配置をあやまると場合によっては深刻なメモリボトルネックを起しかねない．ユーザが平坦なメモリ空間を想定してプログラミングを行っていると，予想外の結果に驚くことになる．本問題の解決には，プログラマがキャッシュラインを意識したコードを準備する必要があった．但し，自動メモリ管理が普及し，プログラマが参照管理から解放された今日，一般プログラマに期待すべき内容とは言えない．本研究では，キャッシュの無効化や空間的局所性，キャッシュ密度を総合的に考慮したオブジェクトレイアウト手法を提案している．システムは，プログラムの実行プロファイルからオブジェクトの変数アクセス状況を取得し，それに基づき適切なレイアウトを決定，自動メモリ管理機構による自動配置を行うことで，問題解決を行う．

本論文は，以上の最適化技術によって並列プログラムの各種並列計算環境上の自動効率化を目指した研究である．それぞれの技術については，実際的なアプリケーションプログラムを通してその有効性の評価を評価する．また，将来実用化する上での問題点がある場合，その方策についても議論を行っていく．

第 2 章

研究背景

2.1 並列プログラムのプログラミング環境

現在の並列プログラムの実行環境には，共有メモリ並列計算環境や分散メモリ計算環境や，その混在した環境があり，また，各計算環境に対するプログラミング環境も様々である．

共有メモリ並列計算機も，小規模なものから，数十以上のプロセッシングエレメントを持つ並列計算機があり，さまざまである．また，大規模な共有メモリ計算機の実現方式も，SMP (Symmetric Multi-Processing), UMA (Uniform Memory Access) と言われるメモリに対してプロセッサが対称に配置されたアーキテクチャと，NUMA (Non-Uniform Memory Access) [32] と呼ばれるプロセッサからメモリへのアクセスが均等ではないアーキテクチャが存在する．また，分散メモリ並列計算機や Grid[21, 20] や PC クラスタ [1], NOW (Network of Workstations) に代表されるようなネットワーク接続された計算機群を用いて並列計算を行う場合もある．これらの各種計算環境では PE (Processing Element) 間の通信コストも異なり，つまり，プログラムを効率的に実行させるためのポイントも同じではない．資源の配置を考えて最適化を考えなくてはいけない．

一方，プログラミングスタイルも，大きく二つにわかれる．分散メモリ空間上でプログラミングを行う場合と，共有メモリ空間上でプログラミングを行う場合である．共有メモリプログラミングスタイルは，SMP，NUMA といった並列計算機上で行われるだけでなく，ソフトウェア分散共有メモリ上に行われることもある．

共有メモリ上でプログラミングを行った場合，データがどこに配置されるのかについて，あまり考えずにプログラミングが可能であるため,「正しい」並列

プログラムを実現するのは簡単だが，効率的に scalable なプログラムが実現できるとは限らない．効率化のためには，データ配置やスケジューリングを意識した資源配置を行う必要がある場合が多い．

一方で，当初から分散メモリを想定したプログラミングスタイルも存在する．MPI[6] などのメッセージパッシングライブラリを用いたプログラミングは，分散メモリ計算機や PC クラスタだけではなく，共有メモリ並列計算機上で利用されることもある．規則正しいデータパターンを対象とした計算などでは，プログラマが静的に適切なデータ配置を決定することで，効率的なプログラムが容易に実現できることも多い．但し，不規則データを対象としたプログラムの場合，当初から適切な資源配置を考えたアルゴリズムを考えるのは難しいケースも多く，また，資源配置の変更はプログラムの大幅な改変を伴うことが多く，プログラム再利用性の上からも課題が多い．

本研究では，不規則データを対象とした効率的な並列プログラムが簡単に実現できるようにすることを目標とする．プログラマには，

- まず，共有メモリプログラミングスタイルで「正しい」プログラムを記述してもらい，
- ボトルネック解消や資源配置などを考慮したチューニングを，処理系による自動最適化などのサポートのもと行える

ことを目指す．

以降の節では，正しい排他制御が行われただけのプログラムを高速化するためにはどのような最適化が必要であるのか，現状プログラマ自身がどのような最適化を施さなくてはならないのか，サンプルプログラムを通して簡単に紹介する．以下では，説明の順として排他制御，共有メモリ計算環境，分散メモリ計算環境の順で説明を行う（この順は，以降の章立てと若干順序が異なる）．

2.2 適切な排他制御の記述

逐次プログラムと並列プログラムが大きく異なるのは，逐次プログラムでは一意に定まるプログラムの実行順序が並列プログラムでは定まらないことである．このため，一連のデータアクセスを適切な順序で行うためには，そのトランザクション区間を排他的に実行させる必要がある．但し，排他制御区間が長い

```
class Tree2 {
    Tree2 left, right;
    int key;
    int count;

    synchronized void insert(int k) {
        if(key == k) {
            count++;
        } else if(k < key) {
            if(left == null) left = new Tree2(k);
            else left.insert(k);
        } else {
            if(right == null) right = new Tree2(k);
            else right.insert(k);
        }
    }
}
```

図 2.1: カウンタ付き 2 分木の例（単純な記述）

場合，時として実行時にボトルネックを発生し実行速度が著しく低下する．このため，排他制御区間の短縮は並列プログラムの効率化にとって重要であり，プログラマ自身によって，不要な排他制御の削除という煩雑な作業が行われ，そして，しばしばバグの原因ともなっていた．

現在も，マルチスレッドプログラムを対象とした不要な排他制御の除去を目指した研究 [18, 46, 13, 14, 39, 38] や，ロックの実行時コストを抑える研究 [11, 37, 40, 19] が盛んに行われているが，多くは基本的に共有されていないオブジェクトへの排他制御操作を軽くすることが目的である．一方で，共有オブジェクトに対する排他制御緩和を目指し，独自のメモリモデルをもった言語処理系の提案・実装 [43, 50] も行われてきた（詳しくは，4.2 節で）．但し，逐次言語とメモリモデルが異なることもあり，あまり普及していない．

図 2.1 は，Java 言語上で記述された 2 分木プログラムの例で，仮に追加操作のみが行われているとする．排他制御に関しても単純であり，`synchronzied` 宣言によって，`insert()` メソッド全体に排他制御区間が設定されている．もし，このようなプログラムを使って，複数のプロセッサが単一の木構造に対し

```
void insert(int k) {
  if (k == key) {
       synchronzied (this) count++;
  } else if (k < key) {
    /* left case */
    label:while(true) {
      if (left != null)
        left.insert(k);
      else {
        synchronzied (this) {
          if (left != null) continue label;
          left = new BinTree(k);
        }
      }
      break;
    }
  } else { /* right case */ }
}
```

図 2.2: カウンタ付き 2 分木の例（最適化例）

て，ルートからの並行挿入操作を行った場合，ボトルネックが発生する．ルートノードにおける `insert()` メソッドでは，`left.insert(val)` といった部分木に対する挿入操作も含めて排他制御区間に入っており，このため一切の挿入操作が並列に実行できないためである．

そこで，熟練したプログラマであれば，ボトルネックが起こらないように排他制御区間をより短くしようと最適化を行う．図 2.2 [1] の例では，子供の木がある場合と無い場合とによって排他制御の仕方を切り換えている．子供の木がない場合は排他制御をして子供の木の作成・登録を行い，子供の木がすでにある場合は，排他制御をせず子供の木への登録操作を起動する．このような最適化は，(1) 子供の有無によってプログラムの性質が変化するとプログラマが知っており，(2) 実際に各状況における変数更新の有無などをプログラマが把握しており，(3) その情報をもとに最適化を施す，ことで実現される．以上のような最適

[1] 但し，現在の Java のメモリモデルでは本プログラムが意図通り動作する保証はなく，動作も処理系実装による．このため，メモリモデル変更が一部で議論されている．詳しくは [2] を参照．

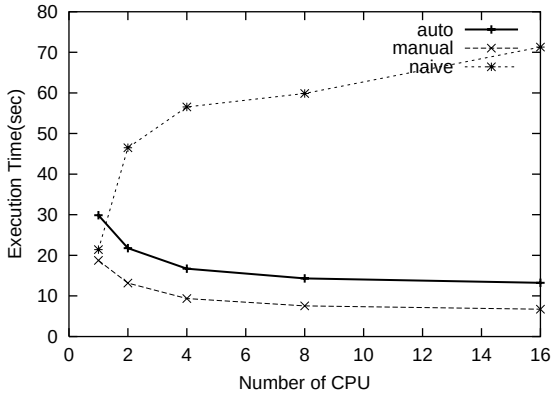

図 2.3: 排他制御緩和の性能面への影響

化は煩雑であることに加え，プログラマが状況を確実に把握しておらず最適化を誤ると，バグを引き起こすことになる．

但し，以上の最適化が性能に与える影響は絶大である．図 2.3 は，上記と同様の 2 分木プログラム（但しカウンタなし）における性能評価である．単純な排他制御をおこなったものが `naive` であり，手動最適化をおこなったものが `manual` である．`auto` は，4 章の自動的排他制御緩和を行った結果である．この結果を見て分かるように，単純な排他制御では並列プログラムは高速化するどころか，逆に速度低下を起こしている．

本研究では，上述の最適化をプログラマが容易に実現できるようにするためのアプローチとして，(1) についてのみプログラマの知識に頼ることにする．一方で，(2) 局面情報の解析, (3) 局面情報に基づく最適化については自動化を行うというアプローチをとった．本論文では，局面解析について 3 章で扱い，排他制御緩和への適用法について 4 章で述べることとする．

2.3 共有メモリ計算機上のオブジェクト配置

十分な排他制御の緩和が行われたプログラムであっても，共有メモリ並列計算機上で効率的に実行されるとは限らない．これは，主に共有メモリ計算機上のキャッシュの構成に起因する．

CPU とメモリとの間の速度差を緩和するためにキャッシュメモリを活用する現在の計算機においては，キャッシュミスを削減することがプログラムの高速

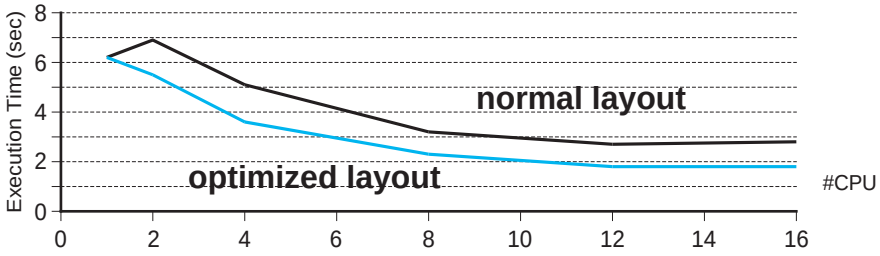

図 2.4: オブジェクトレイアウトの性能面への影響

化につながる．逐次計算環境においても，オブジェクトのメモリ配置に関する研究は盛んに行われてきた [15, 48, 47, 41, 17, 16].

但し，共有メモリ型並列計算機では，キャッシュミスが性能に与える影響は，逐次計算機に比してさらに大きい．これは，無効化によるキャッシュミス（コヒーレンスミス）の存在による．特に問題と考えられるのは，論理的にプロセッサ間共有されないデータも，無効化によるペナルティを受ける可能性がある点である．これは，キャッシュの無効化がキャッシュライン単位で行われるため起こる現象である．本来，多くの並列プログラムではデータの更新操作は普通単一プロセッサのみによって行われ，キャッシュの無効化／再読込みは更新内容を伝達するための妥当なコストと言える．但し，単一のキャッシュラインに「頻繁に読込みが行われる共有データ」と「更新を伴うデータ」が含まれる場合，本来避けることのできた無効化の問題が発生する．

図 2.1 のプログラムは，図 2.2 のように変更することで排他制御の緩和は行うことが出来た．しかし，`left`，`right` といった非常に頻繁に読込みアクセスされるフィールドと，少量ではあるが更新を伴う `count` フィールドがひとつのオブジェクトに同居しているためキャッシュの無効化の影響を受ける．つまり，プロセッサ P が `count` への更新を行った際，プロセッサ P 以外での `left,right` アクセスで，キャッシュミスが起きる事態となる．このため，更新頻度が高い場合，複数プロセッサで作業するよりプロセッサ P のみで作業した方が短時間で終わる場合すら存在する．

図 2.4 は，6 章のレイアウトの自動最適化を施して実行させた例である．このサンプル実行では，カウンタ付 2 分木に対し，1000 万データを重複度 1000 で挿入した場合である（6.8 節の評価では，もっと低い重複度で評価を行っている）．

書き込み回数としては必ずしも大きいとは言えないが，2CPU で逐次実行より速度低下がみられ，16CPU では最適化した場合に比べ 50% 近い速度差が確認される．

このため，プログラマに明示的なメモリ操作を許している C などの言語では，キャッシュを意識したデータ配置をしばしばプログラマ自身が手作業で行うこともあった．規則性をもったデータ構造の場合，熟練したプログラマにとってはアクセス傾向を意識したデータ配置は，速度向上のため当然のように行われている．但し，不規則データ構造においては明示的なメモリ管理を伴うこの種の作業は煩雑であり，バグの原因にもなりやすい．また，自動メモリ管理機構を備えた Java などの言語においては，プログラマはメモリ管理の負担から解放されたが，逆に，キャッシュを意識したオブジェクト配置を行うことができない．

上記最適化を行うためには，プログラムの性質を把握した上でのオブジェクト配置が必要であるが，従来の汎用的な自動メモリ管理機構はこの種類の情報を利用しないのが普通であった．但し，逐次計算の分野では最近プロファイル情報などを利用した積極的な最適化手法が盛んに研究され始めている [17, 16, 41]．

本論文では，共有メモリ並列計算機を対象としたオブジェクトレイアウトの自動最適化ならびに自動メモリ管理機構への統合を目指す．プログラムのサンプル実行からフィールドアクセス情報を取得し，クラスを分類・分割し，分類情報をもとに自動メモリ管理機構を用いてキャッシュを意識したオブジェクト配置を行う．詳細については，6 章において述べる．

2.4 分散オブジェクト配置

共有メモリ向けプログラムを分散環境で実行するには，

- ソフトウェア分散共有メモリを利用する [24, 49, 42]
- 単純な RPC (remote procedure call) 機構やメッセージ送受信ライブラリを利用する [7, 34, 51]
- 分散オブジェクトを利用した移植作業を行う [3, 5]

という主な選択肢がある．但し，いずれの方法であっても，オブジェクトの性質を利用してフィールド変数のキャッシュ／コピーを行わなければ実行効率を

得ることはできない.

たとえば，図 2.2 のプログラムの場合，オブジェクトには，left，right，count というフィールドがあるが，それぞれの変数について効率化への対応は異なる．left，right に関しては値が定まり次第，各プロセッサに値をキャッシュしたほうが良いが，一方で，count の更新処理については固定された単一プロセッサ上での更新作業が適している．

しかし，通常のソフトウェア分散共有メモリなどで，このようなオブジェクトの個々のフィールドの性質を捉えたような対応を行うことは難しい．一方で，現在普及している Java RMI[5] や HORB[3] といった分散オブジェクト技術に関しても，分散オブジェクトに関する処理は owner computing を行うのが一般的である．効率的な処理のためには，オブジェクトの状況に応じて，プロキシで実行可能な処理はプロキシで実行し，本体で行うべき処理を本体で実行させることである．但し，その実現のためには，現状ではプログラマ自身が分散共有メモリのキャッシュコントロールを行うか，あるいは，自身でプロキシへのデータキャッシュを実現するしか方法がない.

一方で，遠隔アクセスの削減は性能に与える影響も大きく，上記最適化の意味は大きい．最近のネットワークの高速化は大きいが，同様にプロセッサ速度も向上しており，また，遠隔アクセスのオーバヘッドはネットワークコストだけではなく，I/O 処理に関するプロセッサ処理も含まれるためである．例えば，Pentium 2.4GHz で Giga Eather を利用した場合でも HORB の遠隔メソッド呼出しは 100 μsec 程度は必要とする．他方，ローカルメソッド呼出しは 0.1 μsec 以下である．不規則計算などでデータをバースト転送しにくい場合，この差を埋めるのことは難しい.

5 章で扱う研究は，排他制御緩和でも用いた局面解析を用いることで，キャッシュを用いた効率的な分散オブジェクトの自動生成を目指したものである．プロキシがオブジェクト本体のデータキャッシュを有し，プロキシ側で実行可能な操作を局所実行することで，遠隔メソッド呼出しの削減を図ることができる．但し，安全にキャッシュを利用するためにはキャッシュデータの一貫性が重要である．本研究では，局面間遷移や局面毎のフィールドアクセスを解析することでプロキシへの一貫性を保ったデータキャッシュ法，ならびにメソッドの実行ルールを提案する．現在の所，本研究に関しては解析フレームワークが出来たところである．将来，解析ならびにキャッシュ機構の自動化を実現し，キャッ

シュ不可能な箇所についてはマルチスレッドなどの遅延隠蔽技術を併用することにより，並列プログラムの分散環境への移植／最適化を容易に実現できる環境を目指す.

第 3 章

局面解析

3.1 はじめに

並列・分散言語のプログラムの実行においては，排他制御や通信のための処理が実行時間に大きな影響を与える．このため，高速化を目指すプログラマは，その知識をもとに，排他制御規則の緩和によるボトルネック回避や，分散オブジェクトの効率的な利用といった最適化作業を手動で行ってきた．これらの最適化は性能面で大きな向上を目指すことができる一方，プログラマにとっては，プログラムの処理内容に関する知識のみならず詳細な実装記述を行う必要があり，煩雑な作業となっている．

一方，不要な排他制御除去などの最適化を，処理系で自動化する試みも行われている．但し，多くの解析系はプログラム実行全般にわたって成立する性質のみを利用するため，プログラムの実行中に変化する性質を利用した最適化を行うことができない．例として，あるプログラムがデータの構築局面と利用局面に分かれていたとする．データ利用局面においては値の更新などが行われず，読み出しのみが頻繁に行われる．つまり，利用局面においては本来一貫性保証のためには排他制御は不要であり，プログラマはこのような変化を把握して局面毎に最適化されたコードを記述することも可能である．但し，多くの解析系がプログラムを局面毎に分離して解析することはなく，これらの知識を利用した自動最適化を行うことはできない．

このため我々は，プログラムを複数の局面に分割して捉え，各局面毎のプログラムの性質を解析し，自動最適化に応用するというアプローチを取ることとした．しかし，解析機構のみで有用な局面の切り分けを行うのは困難であるため，プログラマが持つ局面に関する知識に頼ることとする．つまり，プログラ

マに自らの持つ局面に関する知識をプログラム中に記述してもらい，後は処理系が自動的に局面毎のプログラムの性質を解析し，この情報をもとに自動的に不要排他制御の除去や，局面毎のコード特化による最適化を行うことを目指す．対象とするのは，Java のような構造化された排他制御構文をもつオブジェクト指向言語である．

本章では，まず，リンクデータ構造を対象とした並列プログラムの従来解析手法について簡単にまとめる (3.2 節)．3.3 節において局面とその記述例について提案を行い，その後，局面解析アルゴリズムの概観 (3.4 節)，局所解析 (3.5 節)，大域解析 (3.6 節) について述べる．サンプルプログラムを通した評価の後 (3.7 節)，局面解析の応用や拡張や現状での問題点や将来の課題について議論を行う (3.8 節)．

3.2 関連研究

本節では，オブジェクトの参照解析や，排他制御の緩和に関する解析の研究について，概観する．

オブジェクトの参照解析の重要性は，従来にも増して大きなものである．Java 言語などの先進的な言語の多くは，動的なデータ構造生成を多用するものが多い．これらのオブジェクトは参照を介してアクセスされ，また，参照は値として他のオブジェクトのフィールドや，大域変数，メソッドの引数として伝播／格納される．一方で，オブジェクトが各所から共有されると，各種最適化への妨げとなる．以下のプログラム例は，下記にのべる関連研究でも取り上げられていた，参照解析が各種最適化に影響を与える例である．

- フィールドアクセスの結果の予測:
  例えば，図 3.1 の `foo()` メソッドにおいて，`sample` と `head` が別物であり，加えて `sample.val` に更新がないことが分かれば，`sample.val` はループ不変量として利用可能である．

- オブジェクトの利用期間の予測:
  図 3.1 のプログラムでは，`sample` は `foo()` 内でのみ参照されている．このような場合のメモリ配置に関しては，C 言語などでは Heap ではなく Stack 領域を利用した高速メモリ配置を行うが，同様の実装技術を利用したい．

```
void foo(Object0 head) {
    Object0 sample = new Object0();
    while(head.next != null) {
        head.val += sample.val;
        head = head.next;
    }
}
```

図 3.1: Aliase Analysis が有効な例

- 不要な排他制御の除去:
  Java などのマルチスレッド対応した言語では，各種汎用データ構造ライブラリは一貫性を保つための排他制御が実施されるのが一般的である．一方で，そのデータ構造が単一スレッドからのみ参照されているならば，排他制御は不要である．

参照解析に関する研究は従来から盛んに行われているが [27]，未解決の問題も多い．まずは，近年 Java を対象に盛んに行われている Escape Analysis[18, 46, 13, 14, 39, 38] の研究を簡単に概観する．

そもそも，Escape Analysis の研究は関数型言語の分野で行われていた．というのも，関数型言語においても動的なオブジェクト生成が，頻繁に行われるためである．ML などのプログラミング言語では，オブジェクトの利用期間を予測し，Stack 的データアロケーションをおこなう Region Inference [44, 23] などの研究が行われている．

一方で，Java を対象とした Escape Analysis に関する論文は，その後，同時期に多数提案されることになる [18, 46, 13, 14]．ここでは，その中でも [46] を通して簡単な解説を行う．[46] では，局所変数からの参照やオブジェクト間の参照関係を表現するために抽象的な point-to escape graph を作成する．このグラフは，プログラムの各段階でのオブジェクト参照関係を表現する flow sensitive なグラフである．グラフでは，オブジェクトと参照について，メソッド内部で生成されたものか，あるいは外部で生成されたものか識別しつつ解析を進められる．これにより，オブジェクトが Escape しているか Captured であるかが判定される．基本的には，局所解析から分かる事実をコントロールの流れにそって伝搬するタイプの解析が行われる．アルゴリズム的には，各メソッドの解析

は一度行えば良い，メソッド間解析に対応している，プログラムの一部分のみを解析可能であるという特徴を有している．成果面でいえば，この解析により排他制御のうち 24-67% の削減に成功している．但し，複数のスレッドから共有されたオブジェクトは，基本的に Escape したオブジェクトであり，排他制御緩和の対象にならない．[39] は，マルチスレッドプログラムに対して Escape Analysis を応用したものである．対象とするのは fork-join タイプの親子関係にあるスレッド間の interaction であり，親子のスレッドで共有されるオブジェクト参照の解析を目指している．但し，複数のスレッドがオブジェクトに同時にアクセスするような状況の最適化には利用できない．つまり，Escape Analysis は同時にアクセスが行われないことを解析し排他制御を除去することはできるが，現状ではボトルネック解消に利用できるわけではない．

一方で，参照に関してフローを考慮しない解析を，有効に利用した研究も存在する．[25, 26] は再帰データ構造の参照関係を対象にした研究である．[25] で提案された ADDS (Abstract Description of Data Structures) とは，再帰データ構造の参照関係の特徴についてプログラマが記述するための枠組みである．プログラマは，参照関係について

- 循環を持たない参照関係 (forward)
- forward の逆向き参照 (backward)
- オブジェクトへの唯一の forward 参照である (uniquely forward)

などの特徴づけを行い，解析によってその性質が守られている事を保証するというものである．共有されたリンクデータ構造の性質を捉えた研究として興味深い．本アプローチは，本論文においても 5 章で分散データ構造のモデリングで利用する．

共有オブジェクトへのアクセスに関する研究としては，競合検知 (race detection) の研究 [36, 35] も存在する．共有メモリモデルの場合，競合とは排他制御区間外で読込みと書込みの順序関係が定まらない場合を呼ぶ．一方で，排他制御区間外であっても，スレッドによるアクセス順序が定まっていれば，競合とは言えない．競合検知の解析も静的解析で行えることは限られており，基本的には aliase 解析以上のことは言えない．代りに，実際にプログラムを実行することで，ログ情報などからアクセス順序関係を判定し，競合の検知を行う．但し，これはあくまで，あるプログラム実行における競合を検知したことにしか

ならない．競合の存在しないプログラムであっても，ロック取得順序によって可能なプログラム実行は複数存在し，よって全ての競合の可能性を検知するのは現実的ではない．

つまり，現状の研究では共有オブジェクトへの参照をもつ複数のスレッドが存在する場合，オブジェクトへのアクセス時期やアクセススレッド数に対する正確な解析は行われていない．同時に何が行われるか明確でないため，局所的な知識か，静的な知識をもとにした最適化しか行うことができない．

一方で，オブジェクトの状態を分類し，状態毎の挙動を記述するという研究は従来から行われており [45, 31, 33]，例えば，外部からの要求に対して，状況に応じた処理内容を記述するために利用される．また応用事例も同期に関するものであり，共通点が多い．但し，アプローチに大きな違いがあり，これらの研究は，状態毎の挙動をプログラマが書き分けるために利用される．一方で，本研究ではプログラマは単に局面に関する知識だけを記述してもらい，解析によって最適化を行うアプローチをとっており，本章で提案するような局面解析に関する技術が重要となっている．

3.3 局面の導入

本節では，局面解析を用いた最適化アプローチの概観をプログラム例を通して述べる．利用するのは，2 章で紹介したような 2 分木プログラム (図 3.2) である．このプログラムを最適化し，図 3.3 のような排他制御緩和を行うためには，子供の木がある場合と無い場合とによって排他制御の仕方を切り換えると良い．子供の有無によって，`left`，`right` などのフィールドの更新の有無が変化し，排他制御の必要性も変化するからである．

我々のとるアプローチは，(1)「子供の有無によってプログラムの性質が変化する」という知識はプログラマの記述に頼り，一方で，(2) 局面情報の解析, (3) 局面情報に基づく最適化については自動化を行うというものである．本章で扱うのは (1) の記述法，ならびに (2) の局面解析についてである．(3) 局面情報を利用した最適化例については，4 章の排他制御緩和手法や，5 章の分散オブジェクト実装法が対応する．本節では，まず局面に関する記述法を紹介し，その後局面記述からどのような情報が解析されるのか，どのように応用することが可能かについて簡単に紹介する．

```
class BinTree {
    int key;
    volatile BinTree left, right;

    void insert(int k) {
        synchronized (this) {
            if (k < key) { /* left case */
               if (left != null) left.insert(k);
               else left = new BinTree(k);
            } else { /* right case */ }
        }
    }
}
```

図 3.2: 2 分木の例

```
    void insert(int k) {
        if (k < key) { /* left case */
            label:while(true) {
                if (left != null) left.insert(k);
                else {
                    synchronzied (this) {
                    if (left != null) continue label;
                        left = new BinTree(k);
                    }
                }
                break;
            }
        } else { /* right case */ }
    }
```

図 3.3: 2 分木の例 (最適化例)

3.3.1 局面記述

局面に関する知識を記述する手段として，我々は**局面変数**を導入する．局面変数は，各々のオブジェクトの状態を表すための変数である．以下では例として図 3.2 の 2 分木プログラムに対して，局面記述を行う (図 3.4).

まず局面の定義であるが，このプログラムでは子の有無が最適化にとって重要である．このため，各オブジェクトの左右の子に対し，**子を持つ局面**と**子を持たない局面**との 2 つの局面に分離する．そこで,

```
class ChildState extends
                    Phase {Empty, Full};
ChildState lstate = Empty, rstate = Empty;
```

```
class BinTree {
    /* 局面定義 */
    class ChildState extends Phase { Empty,Full };
    ChildState lstate = Empty;    /* 局面変数宣言 */
    ChildState rstate = Empty;    /* 局面変数宣言 */
    int key;
    BinTree left, right;
    /* consistent メソッド */
    consistent void insert(int k) {
        if (k < key) {  /* left case */
            if ( lstate.is(Full) ) /* 局面に関する分岐 */
                left.insert(k);
            else {
                left = new BinTree(k);
                lstate.become(Full);     /* 局面の変化 */
            }
        } else { /* right case */ }
    }
}
```

図 3.4: 局面変数を用いた 2 分木の例

のように Phase クラスを拡張する形で局面を定義する．この例では，子の局面 ChildState を Empty(子を持たない局面) と Full(子を持つ局面) に分離し，左右の子の局面をそれぞれ変数 lstate，rstate で表している．つまり，各変数は Empty，Full という局面シンボルをその値として持つ．これらの局面変数へのアクセスは，当該インスタンス内に限定される．局面の更新には以下のように become メソッドを用いる．

```
lstate.become(Full);
```

これにより，局面変数 lstate の示す局面 (左の子の局面) が Full へ変化する．解析の都合で，become の引数はリテラルで与えられた局面シンボルに限定している．一方，

```
if (lstate.is(Full)) {...} else {...}
```

のように is メソッドを用いることで，局面に関する分岐を記述することができる．上記の例では 左の子の局面が Full の時には then ブロックが，そうでない時 (Empty の時) には else ブロックが実行されることになる．

最後に，排他制御区間との関係について．上記 become 操作は，必ず対象オブジェクトの synchronized 構文，あるいは，*consistent* 構文 (4.3.1 節) 内で行

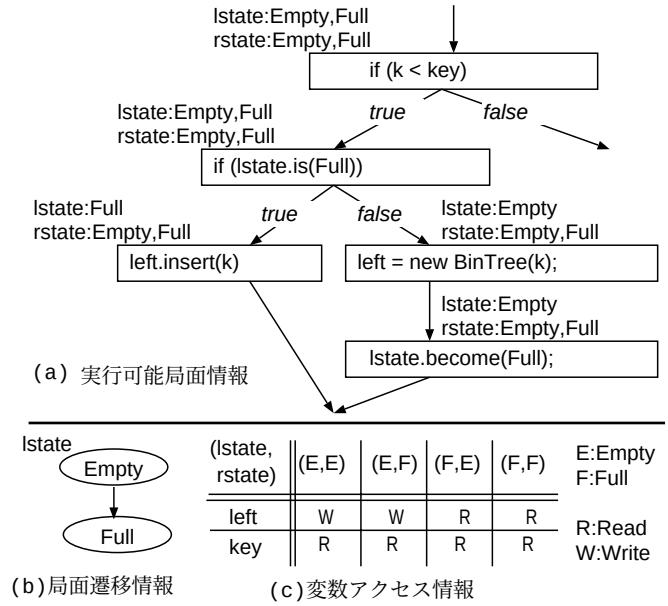

図 3.5: 局面解析情報

われることとする．consistent 構文とは，基本的には synchronized と同様の命令実行順序を守りながら，一貫性に影響のない範囲で排他制御区間の短縮を図るものである．例えば，ブロック冒頭で定数化した変数への読込みを行っている場合 (図 3.2 の `left != null` の場合の `left` へのアクセスなど)，排他制御区間から外すことを許す．いずれにせよ，become 操作はオブジェクト状態を変化させるものであり，排他的な実行が保証される．

3.3.2 局面解析によって得られる情報

ここでは局面記述されたプログラムからどのような情報が解析されるかについて述べる．局面解析から得られる情報は主に以下の 2 種類である．

- **A: 実行可能局面**
  各命令がどの局面で実行される可能性があるか
- **B: 局面遷移情報**
  どのような局面遷移が可能か

図 3.4 のプログラムを解析することにより図 3.5(a) のような実行可能局面情報 (A) が得られる．例えば，`left.insert(k);` は局面変数 `lstate` が局面 `Full`

の時のみ実行され，局面 Empty の時に実行されることはない．また left = new BinTree(k); は局面変数 lstate が局面 Empty の時に実行され，局面 Full の時は実行されないと解析できる．

また，B の局面遷移情報であるが，これは図 3.5(b) のように解析される．局面変数 lstate は初期局面が Empty であり，局面の遷移としては Empty から Full へ遷移する可能性がある．また一度 Full の局面になると別の局面へ遷移する可能性はないことが分かる．

これら二つの情報により，(a) により各々の局面におけるプログラムの性質を解析し，(b) により他のスレッドによる局面遷移の可能性に配慮した最適化が可能になる．例えば，図 3.4 のプログラムに対し排他制御緩和を行い，図 3.3 のように最適化するためには，lstate.is(Full) において left への更新が行われていないというだけでなく，他スレッドによって Full 局面から別の局面に遷移したとしても left は変化しないといった情報が必要である．一方で，本解析結果を使うと (a) から各局面でどのような変数アクセスが行われるか (図 3.5(c)) を解析可能である．加えて，(b) から Full および Full から遷移する局面 (この場合存在しない) を求め，これら局面に対し left の更新状況を確認することができる．この場合，left の更新は否定できるため，排他制御区間の短縮を達成できる．

3.4 局面解析の概観

3.4.1 アルゴリズム概略

本解析では，A: 各命令がどのような局面において実行されうるのか（実行可能局面）と，B: どのような局面間遷移がありうるのか (可能局面遷移）とを解析する．解析を行う上で注意すべき点は，排他制御ブロック外における他スレッドによる局面更新の影響である．同様に，自身のスレッドで呼出したメソッドによる局面遷移も考慮する必要がある．図 3.7 は，サンプルプログラム図 3.6 に対して，その実行可能局面の解析結果を示したものである．このプログラムでは，B3 から B4 に至るパスにおいて分岐直後の局面は Q であると考えられるが，他スレッドの become 操作 (ブロック B6) により局面が Q から R に更新される可能性があるため，最終的に B4 の実行可能局面は Q，R と判定される．一方で，その Q から R への局面遷移の可能性は，ブロック B6 の実行可能局面に Q が含ま

```
void func() {                /* B1 */
  while(true) {
    ....;                    /* B2 */
    if(!phase.is(Q)) break; /* B3 */
    synchronized(this) {
      if(...) {              /* B4 */
         m();                /* B5 */
      } else {
         phase.become(R);    /* B6 */
         .....;              /* B7 */
      }
    }                        /* B8 */
  }
  return;                    /* B9 */
}
```

図 3.6: 解析サンプルプログラム

れることに起因する．このように，A と B の解析は互いに依存しあった関係にあり，加えて，B は当該クラスの全メソッドの解析が終わらないと確定することができない．

我々は，これらの相互依存した情報の解析を高速に行うために，

1. メソッド単位で行うメソッド内解析 (3.5 節)

2. その結果を利用して行う大域解析ならびに可能局面の確定作業 (3.6 節)

の 2 部構成のアルゴリズムを提案する．大域解析においてはクラス内の局面遷移情報を取りまとめて確定を行い，必要に応じてメソッド呼出し関係を考慮した解析が可能な枠組みも提供している．

メソッド内解析は当該クラスの全メソッドを対象に行われ，メソッドをコントロールフローグラフ (以下 CFG) に分解し，フロー解析を行う．但し，局面遷移解析が終了していない段階では，具体的な局面を確定することは当然できない．第 1 段階のメソッド内解析では，準備として「もしメソッド開始時に Q が可能局面であり，他のスレッドなどによって Q から R への遷移が可能であれば，このブロックは局面 R で実行されうる」といった情報を求める．図 3.8 は，メソッド内解析結果を示したもので，B4 の実行可能局面の $R : Cq \cdot Q \rightarrow^* R$ は上の情報を意味している．メソッド中の become 操作についても，局面遷移内容とその成立条件が同様に解析される．

大域解析においては，先ほど解析された become 操作による局面遷移に関す

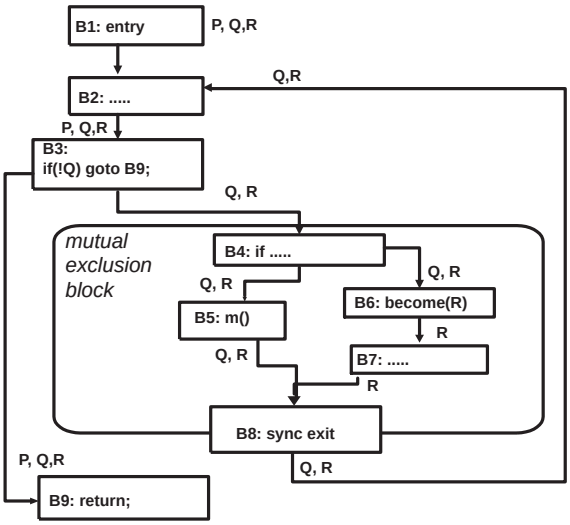

図 3.7: 可能局面解析（確定）

る情報，つまり,「局面 P から Q への局面遷移が可能であれば，局面 R から S への become の可能性がある」といった各メソッドの情報を取りまとめ，実際に可能性のある局面遷移を求めることができる．その結果，可能局面の解析結果も確定することができる．つまり，各コードブロックが具体的にどの局面において実行される可能性があるのかを求めることができる．

最後に，3.3.1 節で触れた consistent 構文との関係について．本解析では，consistent と synchronized は全く同じ扱いをうける．つまり，consistent 構文中でも，他スレッドの影響を一切無視して解析を行う．システムは本解析結果に基づきつつ，局面遷移の可能性を考慮した排他制御緩和を行う．

3.4.2 モデル

アルゴリズムについて説明をする前に，モデルと幾つかの術語の説明を行う．まず局面であるが，解析精度の問題から局面値を一般の一時変数などに格納することを禁止している．また，本論文では局面値は一般に P, Q, R などの記号を用いて表すか，添字を添えて P_i などと表す．局面値の集合は以後 $\boldsymbol{P}$ と表す．また，当初局面変数は一つしか存在しないものとして議論を行う．複数の局面変数に関する議論は 3.8.1 節にておこなう．

実行可能局面，つまり各時点においてどの局面の可能性があるかを，

$$S = \{P_0 : Cond_0, \cdots, P_n : Cond_n\}$$

で表す．$Cond_i$ は 真偽値を表す論理式であり，局面 P_i である可能性の有無を示す．また，$S(P_i) = Cond_i$ として表記する．解析終了時点では，$S(P_i)$ は `true/false` が定まり，局面 P_i であった可能性の有無が定まる．$Cond$ には，論理積演算 $\cdot$ と論理和演算 $\vee$ が存在する．加えて，実行可能局面 S の間に集合計算 $\cup, \cap$ を，以下のように定める．

$$S \cup S' = \{P_i : S(P_i) \vee S'(P_i) \mid P_i \in \boldsymbol{P}\}$$

$$S \cap S' = \{P_i : S(P_i) \cdot S'(P_i) \mid P_i \in \boldsymbol{P}\}$$

次に，`become` 操作と排他制御ブロックについて．3.3.1 章で述べたように局面変数の値更新は，`become` 操作によって排他制御区間内で実行される．なんらかの `become` 操作によって P から Q になる可能性の有無を $P \rightarrow^s Q$ で表し，真偽値 `true/false` を値としてとる．次に，$P \rightarrow^* Q$ (図中では，単に `P->Q` と表記) は，$P = P_0 \rightarrow^s \cdots \rightarrow^s P_n = Q$ （n は 0 以上）なる遷移の可能性の有無を示す．つまり局面 P から 0 回以上の `become` の遷移で局面 Q に到達しうる可能性である．$P \rightarrow^* P$ は常に成立するものとする．

我々の解析では，可能局面と可能局面遷移を以下のように定める．

- 排他制御区間外のある時点で局面 P が可能局面であり，また $(P \rightarrow^* Q) =$ `true` であれば，局面 Q も可能局面である
- ある基本ブロックの入口で局面 P が可能局面であれば，become 操作や局面の制限が行われない限りブロックの出口でも P は可能局面である
- 局面 Q への become 操作が記述されている個所で，局面 P が可能局面であれば，$(P \rightarrow^s Q) =$ `true` である．

最後に，メソッド呼出し関係解析を行うために，呼出されるメソッド $(func)$ とメソッド呼出し命令 $(call)$ に対して，局面遷移可能性 $P_i \rightarrow^{func} P_j, P_i \rightarrow^{call} P_j$ を導入する．$func$ をメソッド間解析の対象とする場合は，実行前の実行可能局面 S^{func}_{init} に対して実行後の実行可能局面 S^{func}_{ret} が以下の式を満たすように $P_i \rightarrow^{func} P_j$ を定めるものとする (for all $P_j \in \boldsymbol{P}$).

$$S^{func}_{ret}(P_j) = \vee_{P_i \in \boldsymbol{P}}(S^{func}_{init}(P_i) \cdot (P_i \rightarrow^{func} P_j))$$

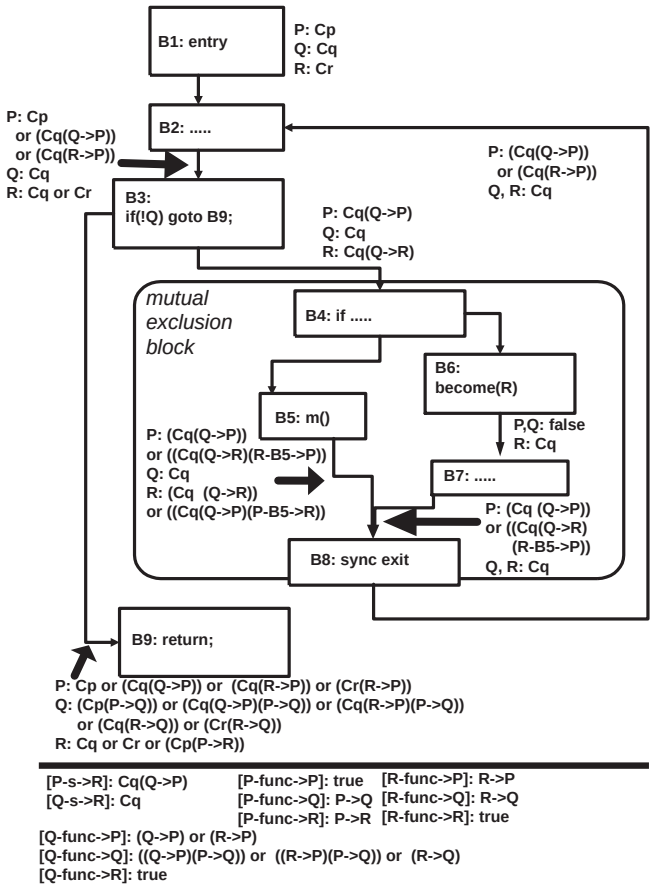

図 3.8: 可能局面解析（第 1 段階）

直観的には，$P_i \rightarrow^{func} P_j$ は，$func$ 開始時に P_i が可能局面の場合に終了時の局面が P_j となる可能性の有無を示す．当然，$(P_i \rightarrow^{func} P_j) \Rightarrow (P_i \rightarrow^{*} P_j)$ ($\Rightarrow$ は含意関係) である．$call$ に関しても同様である．呼出し関係解析を行わない場合は，保守的な解析を行い，$\rightarrow^{func}, \rightarrow^{call}$ の代りに $\rightarrow^{*}$ を利用する．

3.5 メソッド内解析

3.5.1 基本アルゴリズム

メソッド内解析の段階では可能局面遷移などが定まっておらず,「もしメソッド開始時に Q が実行局面である可能性 (C_q) が true で，他のスレッドなどによって Q から R への遷移が可能であれば，このブロックは局面 R で実行されうる」と

いった解析が行われることになる．実行可能局面 S に対して，$S(\mathtt{R}) = \mathtt{C_q}\cdot(\mathtt{Q}{\rightarrow^*}\mathtt{R})$ などと表現される．厳密には，メソッド内解析中の $S(P_i) = Cond_i$ 部は，積和標準形の正規化された論理式をとり，その原子論理式は，メソッド開始時に局面 P_i である可能性を示す論理変数 $Cond_{P_i}^{init}$，もしくは局面遷移の可能性を示す $P_i{\rightarrow^*}P_j$, あるいは $P_i{\rightarrow^{call}}P_j$ ($call$ は本メソッド中の呼出し命令に限定される) である．また，論理演算 $\cdot, \vee$ においては，論理式を合成し，正規化を行うこととする．正規化では，演算が積の場合は積和形に分配し，積の中の二つの原子論理式 A, B について $A{\Rightarrow}B$ なら B を省略し，演算が和の場合は和の中の二つの論理式 A, B について $A{\Rightarrow}B$ なら A を省く．この際，関係式 $P{\rightarrow^{call}}Q{\Rightarrow}P{\rightarrow^*}Q$ を利用するが，$\rightarrow^*$ に関する推移律関係は利用していない．

メソッド内解析では，実行可能局面 S に関するフロー解析を行う．ブロック b の前後の状態を，それぞれ $S_{in}^b, S_{out}^{b\rightarrow b'}$ で表す．但し，b' は対応する後続ブロックを表し，対応する後続ブロックが一つしかない場合は，単に S_{out}^b とも表記する．ブロック前後の状態に関しては一般のデータフロー方程式

$$S_{in}^b = \cup_{b' \in Pred(b)} S_{out}^{b'\rightarrow b}$$

が成立する．但し，$Pred(b)$ は b の先行ブロックを示す．b における可能局面は S_{in}^b によって表す．

メソッド内解析においては，まずメソッドを CFG に変換する．以下の点が一般の CFG と異なる．

- 排他制御ブロックに含まれるか否かで基本ブロックが分かれるようにし，加えて，ロック開放ポイントを独立した基本ブロックとして扱う
- 局面に関する条件分岐命令を独立した基本ブロックとして扱う
- `become` 文やメソッド呼出しを単独の基本ブロックとして取り扱う

データフロー解析を行うにあたって，基本ブロックは以下の 6 種に分かれる．各種類毎に

$$S_{out}^b = f^b(S_{in}^b)$$

なる S_{in}^b と S_{out}^b の関係を表す変換関数が定められている．以下，図 3.8 の例を通して説明する．

1. メソッド冒頭部: 現時点では，メソッド開始時の実行可能局面は確定できないため，各局面 P_i の可能性を変数 $Cond_{P_i}^{init}$ で表す.

$$S_{out}^{b} = \{P_i : Cond_{P_i}^{init} \mid P_i \in \boldsymbol{P}\}$$

また，メソッド $func$ に対して，初期可能局面を S_{init}^{func} と示す．図 3.8 においては，ブロック B1 における可能局面は，$\{\mathtt{P} : \mathtt{Cp}, \mathtt{Q} : \mathtt{Cq}, \mathtt{R} : \mathtt{Cr}\}$ となる．このように初期局面を変数で与えているのは，メソッド間解析に備えるためである．つまり，このメソッド (仮に func0) による局面遷移 $\rightarrow^{\mathtt{func0}}$ を求める必要と，メソッド開始時の局面が限定できるケースに備えるためである．一方で，このメソッドの呼出しについてメソッド間解析しない場合，$\{P_j : \mathtt{true} \mid P_j \in \boldsymbol{P}\}$ として解析を行っても構わない (詳しくは 3.6 章).

2. ロック開放ブロック: 排他制御ブロックが終了するポイントである．つまり，排他制御ブロック内においては，他のスレッドによる局面遷移の影響が無視されていたが，この時点から考慮する必要がある．もし，ロック開放ブロック内で P_k が可能局面であり，$P_k \rightarrow^* P_l$ であれば，今後は P_l も可能局面となる．つまり，S_{in}^{b} に対して，

$$S_{out}^{b}(P_j) = \vee_{P_i \in \boldsymbol{P}}(S_{in}^{b}(P_i) \cdot (P_i \rightarrow^* P_j))$$

として定めることができる.

例えば，図 3.8 においては，ブロック B8 における可能局面は，その前後で局面 P の可能性が増加する．$S_{in}^{\mathtt{B8}}(\mathtt{P}) = (\mathtt{Cq} \cdot (\mathtt{Q} \rightarrow^* \mathtt{P})) \vee (\mathtt{Cq} \cdot (\mathtt{Q} \rightarrow^* \mathtt{R}) \cdot (\mathtt{R} \rightarrow^{\mathtt{B5}} \mathtt{P}))$ に対して，$S_{out}^{\mathtt{B8}}(\mathtt{P})$ では局面 Q，R からの遷移の可能性 $\mathtt{Cq} \cdot (\mathtt{Q} \rightarrow^* \mathtt{P}), \mathtt{Cq} \cdot (\mathtt{R} \rightarrow^* \mathtt{P})$ が加わった結果，図のように定まる．$(\mathtt{Cq} \cdot (\mathtt{Q} \rightarrow^* \mathtt{R}) \cdot (\mathtt{R} \rightarrow^{\mathtt{B5}} \mathtt{P})) \Rightarrow (\mathtt{Cq} \cdot (\mathtt{R} \rightarrow^* \mathtt{P}))$ が正規化の際に利用されている.

3. 局面に関する条件分岐: 話を簡単にするため，本論文では条件分岐命令を基本ブロックとして取り扱っている．if 文の条件式として局面に関する記述を行うことによって，後続のブロックの可能局面を制限することになる．但し，条件分岐命令が排他制御ブロック内にあるか否かによって状況は異なる.

排他制御ブロック内にある場合は，単純に局面に関する条件式に基づいて可能局面を振り分ければ良い．今，条件分岐ブロック b の条件式において，分岐

枝 $e = b \to b'$ への分岐を局面 $Pset(e)$ に限定しているとする．この場合，S^b_{in} に対して，e への出力状態 S^e_{out} は以下のように定まる．

$$\begin{array}{rll} S^e_{out}(P_i) = & S^b_{in}(P_i) & \text{for } P_i \in Pset(e) \\ & \texttt{false} & \text{for } P_i \notin Pset(e) \end{array}$$

但し，条件分岐が排他制御ブロック外にある場合は，再度他のスレッドによる局面遷移を考慮する必要がある．つまり，S^e_{out} は以下のように定まる．

$$S^e_{out}(P_j) = \vee_{P_k \in Pset(e)}(S^b_{in}(P_k) \cdot (P_k \to^* P_j))$$

図 3.8 のブロック B3 は排他制御ブロック外での局面に関する条件分岐を示したものである．分岐後ブロック B4 に至る場合，分岐命令実行直後の可能局面は，$\{\texttt{P} : \texttt{false}, \texttt{Q} : \texttt{Cq}, \texttt{R} : \texttt{false}\}$ であり，$S^{\texttt{B3}\to\texttt{B4}}_{out}$ はさらに他のスレッドによる局面遷移を考慮したものとなっている．

4. become 操作: become 操作によって局面が P_k に更新される場合，当然可能局面は P_k に限定される．但し，このブロックが実行可能になるための条件式を考慮して，S^b_{in} に対して，

$$\begin{array}{rll} S^b_{out}(P_k) = & \vee_{P_i \in \boldsymbol{P}} S^b_{in}(P_i) & \\ S^b_{out}(P_j) = & \texttt{false} & \text{for } P_j \neq P_k \end{array}$$

なる S^b_{out} を出力状態としている．become 操作それ自体は排他制御ブロック内で行われる保証があるので，ここでは他のスレッドの影響を考えない．最終的に全ての $S^b_{in}(P_i) = \texttt{false}$ であれば，この基本ブロックは到達不可能であり，become はされ得ないことになる．

図 3.8 のブロック B6 では，B6 に到達するための条件式は $\texttt{Cq} \cdot (\texttt{Q} \to^* \texttt{P})$, Cq, $\texttt{Cq} \cdot (\texttt{Q} \to^* \texttt{R})$ の論理和である Cq となる．つまり，become 直後の可能局面は $\{\texttt{P} : \texttt{false}, \texttt{Q} : \texttt{false}, \texttt{R} : \texttt{Cq}\}$ と定まる．

5. メソッド呼出し: 排他制御ブロック外では，そもそも他のスレッドによる影響やメソッド呼出しの影響を考慮してあるので，メソッド呼出しによって新たに可能局面が増えることはありえない．つまり，

$$S^b_{out} = S^b_{in}$$

となる．

一方，排他制御ブロック内の場合，メソッド実行中に行われる become 操作による局面遷移を考慮する必要がある．このため，該当メソッド呼出し命令 (call) による局面遷移 ($\rightarrow^{\mathtt{call}}$) の定義に基づいて，メソッド呼出し後の局面は

$$S^b_{out}(P_j) = \vee_{P_i \in \boldsymbol{P}}(S^b_{in}(P_i) \cdot (P_i \rightarrow^{\mathtt{call}} P_j))$$

と定まる．最終的には，大域解析によって $\rightarrow^{\mathtt{call}}$ と呼出されるメソッド (func) に関する $\rightarrow^{\mathtt{func}}$ の対応関係をとり，遷移関係が決定されることになる．但し，呼出されるメソッドが特定できない場合など呼出し関係解析を行わない場合は，$\rightarrow^{\mathtt{call}}$ の代りに $\rightarrow^*$ を利用し保守的な見積りを行うことになる．一方で，呼出し関係から当該メソッド呼出し中に自オブジェクトの局面更新は行われないと分かっている場合は，局面遷移の可能性を否定して解析を行うこととなる．もし，これらの情報が事前に分かっている場合は，メソッド内解析時点でその情報を利用しても構わない．

6. その他: その他のブロックにおいては，排他制御内外に限らず

$$S^b_{out} = S^b_{in}$$

として定めることができる．なぜなら，排他制御外であっても，既に $b' \in Pred(b)$ の $S^{b'}_{out}$ の時点において，他のスレッドの影響による可能性は尽くされているためである．

第 1 段階の解析では，以上で定まるデータフロー方程式を満たす最小不動点を解として求める．各状態の値は有限であり，また単調増加するため，反復法などを用いた場合の停止性が保証される．この情報をもとに become 操作による局面遷移条件を確定することができる．become 操作ブロック b による遷移対象が P_k であったとする．この場合，$S^b_{in}(P_i) = \mathtt{true}$ であれば，$P_i \rightarrow^s P_k$ なる局面遷移が可能であると言える．

図 3.8 の例においては，$S^{\mathtt{B6}}_{in}$ は，$\{\mathtt{P} : \mathtt{Cq} \cdot (\mathtt{Q} \rightarrow^* \mathtt{P}), \mathtt{Q} : \mathtt{Cq}, \mathtt{R} : \mathtt{Cq} \cdot (\mathtt{Q} \rightarrow^* \mathtt{R})\}$ である．この可能局面において，become(R) が行われているため，$\mathtt{Cq} \cdot (\mathtt{Q} \rightarrow^* \mathtt{P}) = \mathtt{true}$ ならば $\mathtt{P} \rightarrow^s \mathtt{R} = \mathtt{true}$ といえる．$[\mathtt{P} \rightarrow^s \mathtt{R}] : \mathtt{Cq} \cdot (\mathtt{Q} \rightarrow^* \mathtt{P})$ と表記する．同様に，$[\mathtt{Q} \rightarrow^s \mathtt{R}] : \mathtt{Cq}$ も成立する．

また，このメソッド (func0()) を実行することによる局面遷移の可能性 $\rightarrow^{\mathtt{func0}}$ も定まる．図 3.8 の場合メソッドの出口は B9 のみであるため，その実行可能

局面情報 $S_{in}^{\mathtt{B9}} = S_{ret}^{\mathtt{func0}}$ と $S_{init}^{\mathtt{func0}}$ を比較して $\rightarrow^{\mathtt{func0}}$ を定めることになる．もし出口が複数ある場合は，その和が $S_{ret}^{\mathtt{func0}}$ となる．例えば，$\mathtt{P}\rightarrow^{\mathtt{func0}}\mathtt{Q}$ は，$S_{in}^{\mathtt{B9}}(\mathtt{Q})$ に対して $\mathtt{Cp} = \mathtt{true}, \mathtt{Cq} = \mathtt{false}, \mathtt{Cp} = \mathtt{false}$ を代入することで求まる．この場合，$\mathtt{P}\rightarrow^{\mathtt{func0}}\mathtt{Q}\Leftarrow\mathtt{P}\rightarrow^{*}\mathtt{Q}$ と定まる．以下，$[\mathtt{P}\rightarrow^{\mathtt{func0}}\mathtt{Q}] : \mathtt{P}\rightarrow^{*}\mathtt{Q}$ と表記する．

現実には，$\rightarrow^{func}$ は排他制御ブロック内のメソッド呼出し命令の解析のためにのみ利用される．このため，より正確な解析を行いたい場合はメソッド全体を排他制御ブロックにいれた形で $\rightarrow^{func}$ 用の解析を再度行ってもよい．

3.5.2 Java における効率的解法

前節のアルゴリズムは一般のデータフローグラフに対応したものであるが，Java のような goto 文を持たない言語の場合，より効率的な解法が存在する．局所解析の目的は，メソッド開始局面 $S_{init} = \{P_i : Cond_{P_i}^{init} \mid P_i \in \boldsymbol{P}\}$, もしくは局面遷移の可能性を示す $P_i\rightarrow^{*}P_j$, $P_i\rightarrow^{call}P_j$ を用いて，各命令の実行可能局面を表現することである．但し，データフロー解析において反復法などを用いて安定解を得るためには解析値の単調性が必要であり，そのために，メソッド内解析中の $S(P_i) = Cond_i$ は常に積和標準形の正規化された論理式として表現され，各データフロー方程式において論理式に関する演算が必要であった．一方で，Java のように CFG が可約である場合，反復法などによらない構造に基づくフロー解析が可能であり，単調性を重視する必要がない．よって，以下に述べる行列表現を用いたフロー解析を行う．

局面遷移行列 $\boldsymbol{F}$ は，実行可能局面 S_a から S_b への遷移を表現する写像である．各要素 $\boldsymbol{F}_{i,j}$ は真偽値であり，P_j から P_i への局面遷移の可能性を示すと，以下の式が成立する．

$$S_b(P_i) = \vee_{P_j \in \boldsymbol{P}} \boldsymbol{F}_{i,j} \cdot S_a(P_j)$$

つまり，$\boldsymbol{F}$ は局面遷移関係 $\rightarrow$ を行列表現したものであり，以下では $P_j\rightarrow^{*}P_i$ に対応する局面遷移行列を $\boldsymbol{F}_{i,j}^{all}$ と, $P_j\rightarrow^{call}P_i$ に対応する行列を $\boldsymbol{F}_{i,j}^{call}$ と表記する．$\boldsymbol{F}$ 間には，以下に定義する和演算 $+$，積演算 $\cdot$ に加え 演算 $(\boldsymbol{F})^{*}$ が存在する (但し，I は単位行列である).

$$\boldsymbol{F}_{i,j}^{3} = \boldsymbol{F}_{i,j}^{1} \vee \boldsymbol{F}_{i,j}^{2} \qquad \text{for } \boldsymbol{F}^{3} = \boldsymbol{F}^{1} + \boldsymbol{F}^{2}$$

$$\boldsymbol{F}_{i,j}^{3} = \boldsymbol{F}_{i,j}^{1} \cdot \boldsymbol{F}_{i,j}^{2} \qquad \text{for } \boldsymbol{F}^{3} = \boldsymbol{F}^{1} \cdot \boldsymbol{F}^{2}$$

$$(\boldsymbol{F})^* = I + \boldsymbol{F} + \boldsymbol{F}\cdot\boldsymbol{F} + \boldsymbol{F}\cdot\boldsymbol{F}\cdot\boldsymbol{F} + \cdots$$

データフロー解析の対象は，各基本ブロック b における開始局面 S_{init} からの局面遷移行列 $\boldsymbol{F}_b$ である．つまり，各実行可能局面は $\boldsymbol{F}_b S_{init}$ で求まることになる．フロー解析において，$\boldsymbol{F}$ は，$\cdot, +, ()^*$ で結合された式として表現され，その原子式は局面遷移を表す変数 $\boldsymbol{F}^{all}, \boldsymbol{F}^{call}$，もしくは become 操作を意味する行列 $\boldsymbol{F}^{become(P_i)}$, 局面の制限を示す行列 $\boldsymbol{F}^{restrict(\boldsymbol{P})}$ である．$\boldsymbol{F}^{become(P_i)}$ と $\boldsymbol{F}^{restrict(\boldsymbol{P})}$ は，それぞれ以下のような行列で表現される．

$$\begin{aligned}
\boldsymbol{F}^{become(P_i)}_{j,k} = \quad & \texttt{true} \quad \text{for } j = i \\
& \texttt{false} \quad \text{for } j \neq i \\
\boldsymbol{F}^{restrict(\boldsymbol{P})}_{j,k} = \quad & \texttt{true} \quad \text{for } (j = k) and (P_j \in \boldsymbol{P}) \\
& \texttt{false} \quad \text{for } others
\end{aligned}$$

上記表現を用いて，前節のデータフロー方程式を以下のように自然に記述しなおすことが可能である．但し，先行ブロックの遷移行列を $\boldsymbol{F}^{prev}$ とする．

- 冒頭部: 遷移行列は単位行列 I
- ロック開放ブロック: $\boldsymbol{F}^{all} \cdot \boldsymbol{F}^{prev}$
- 局面に関する条件分岐（排他制御ブロック内）: $\boldsymbol{F}^{restrict(\boldsymbol{P})} \cdot \boldsymbol{F}^{prev}$
- 局面に関する条件分岐（排他制御ブロック内）: $\boldsymbol{F}^{all} \cdot \boldsymbol{F}^{restrict(\boldsymbol{P})} \cdot \boldsymbol{F}^{prev}$
- become 操作: $\boldsymbol{F}^{become(P_i)} \cdot \boldsymbol{F}^{prev}$
- メソッド呼出し（排他制御ブロック外）: $\boldsymbol{F}^{prev}$
- メソッド呼出し（排他制御ブロック内）: $\boldsymbol{F}^{call} \cdot \boldsymbol{F}^{prev}$

次に，循環をもつ領域の局面遷移行列についてであるが，領域内でヘッダに循環するパスの局面遷移行列を $\boldsymbol{F}$ とすると，その領域の局面遷移行列は $(\boldsymbol{F})^*$ で表すことができる．上記の手続きにより，各ブロックの局面遷移行列が $\boldsymbol{F}^{all}, \boldsymbol{F}^{call}$ に関する式として表現できたことになる．これは，つまり各ブロックの実行可能 S_b が，$P_i \rightarrow^* P_j$, $P_i \rightarrow^{call} P_j$ と開始局面 S_{init} に関する式として表現できたことを意味する．

3.6 大域解析

局所解析では，局面遷移可能性 $P_i{\rightarrow}^* P_j$, $P_i{\rightarrow}^{func} P_j$, $P_i{\rightarrow}^{call} P_j$ と，各メソッド開始時の実行可能局面 S_{init}^{func} 間の制約条件が求まっている．また，各メソッド呼出し命令における実行可能局面 S_{in}^{call} も求まる．大域解析では，メソッド呼出し関係を考慮しつつ，以上の制約を満たす最小の解を求める．

局所解析から，

- $P_i{\rightarrow}^s P_j$
- $P_i{\rightarrow}^{func} P_j$
- $S_{in}^{\texttt{call}}$

に関する制約が，基本アルゴリズムの場合，項の積和形として定まる．例えば，$[\texttt{P}{\rightarrow}^s\texttt{R}] : \texttt{Cq} \cdot (\texttt{Q}{\rightarrow}^*\texttt{P})$ であれば，$(\texttt{P}{\rightarrow}^s\texttt{R}){\Leftarrow}(\texttt{Cq} \cdot (\texttt{Q}{\rightarrow}^*\texttt{P}))$ を意味する．このため

- $P_i{\rightarrow}^* P_j$
- $P_i{\rightarrow}^{call} P_j$
- S_{init}^{func}

に関する制約を同様に定めれば，後はこの制約を満たす解を greedy に求めることができる．つまり，初期状態ではすべての値を `false` とし，制約に応じて必要な項を `true` に変更するという操作を制約を充足するまで繰り返す．制約を否定項などを含まない純粋な積和形ですべて表すことができれば，アルゴリズムの単調性と停止性が保証できる．

$P_i{\rightarrow}^* P_j$ は，P_i から P_j に $\rightarrow^s$ の 0 回以上の繰り返しで到達できるかどうかを示すものであり，単調性も明らかである．

$P_i{\rightarrow}^{call} P_j$ と S_{init}^{func} に関する制約は，メソッド呼出し関係の解析から定まる．呼出されるメソッドの開始時の実行可能局面 S_{init}^{func} は，呼出し命令の実行可能局面の総和で求めることができる．もし直接あるいは間接的に当該メソッドを呼出している自クラス中の呼出し命令 (`call`) がすべて特定できている場合は，$S_{in}^{\texttt{call}}$ の論理和をとる．一方で，メソッドの呼出し元が特定できない場合などは，初期局面 P_0 から $\rightarrow^*$ 到達可能な全局面の可能性がある．つまり保守的に $S_{init}^{func}(P_0) = \texttt{true}, S_{init}^{func}(P_i) = P_0{\rightarrow}^* P_i$ として解析を行うことになる．

メソッド呼出し命令 call による局面遷移に関しては，call によって直接あるいは間接的に自オブジェクトのメソッドが呼出される場合，その可能性を考慮して $P_i{\rightarrow}^{\mathtt{call}}P_j$ を定める必要がある．例えば，自オブジェクトへのメソッド func 呼出しが内部で 1 回以上行われているか，あるいは行われていないかもしれない場合は，以下のように定める．

$$\begin{aligned} P_i{\rightarrow}^{\mathtt{call}}P_i &\Leftarrow \mathtt{true} \\ P_i{\rightarrow}^{\mathtt{call}}P_j &\Leftarrow \vee_{k\in\boldsymbol{P}}\,(P_i{\rightarrow}^{\mathtt{call}}P_k)\cdot(P_k{\rightarrow}^{\mathtt{func}}P_j) \end{aligned}$$

一方で，もし呼出されるメソッドが特定できない場合は，$P_i{\rightarrow}^{call}P_j = P_i{\rightarrow}^{*}P_j$ として保守的な見積りを行うことになる．

図 3.8 の例の場合，メソッド内解析からは

$$\begin{aligned} \mathtt{P}{\rightarrow}^{s}\mathtt{R} &\Leftarrow \mathtt{Cq}\cdot(\mathtt{Q}{\rightarrow}^{*}\mathtt{P}) \\ \mathtt{Q}{\rightarrow}^{s}\mathtt{R} &\Leftarrow \mathtt{Cq} \end{aligned}$$

と $\rightarrow^{func}$ に関する式が定まっている．仮に，他にメソッドが存在せず，メソッド呼出しについては保守的な見積りをする場合は，$\rightarrow^{s}$ と $\rightarrow^{*}$ に関する関係式に加えて，以下の関係が成立する．

$$\begin{aligned} \mathtt{Cp} &\Leftarrow \mathtt{true} \\ \mathtt{Cq} &\Leftarrow \mathtt{P}{\rightarrow}^{*}\mathtt{Q} \\ \mathtt{Cr} &\Leftarrow \mathtt{P}{\rightarrow}^{*}\mathtt{R} \end{aligned}$$

もし，他に条件が無い場合，一切局面遷移なしという解になる．一方，別のメソッドのメソッド内解析から

$$\mathtt{P}{\rightarrow}^{s}\mathtt{Q} \Leftarrow \mathtt{true}$$

である場合は，$\{\mathtt{P}{\rightarrow}^{s}\mathtt{Q}, \mathtt{Q}{\rightarrow}^{s}\mathtt{R}\}$ なる局面遷移が起ることになる．また，この条件のもと，図 3.8 の例について可能局面を確定すると，図 3.7 のように確定する．

以上と同等の議論は，3.5.2 節の表現においても成立する．局所解析により，メソッド開始局面 S_{in}^{init} と $\boldsymbol{F}^{all}, \boldsymbol{F}^{call}$ を用いて，メソッド呼出し局面 $S_{in}^{\mathtt{call}}$ と $\rightarrow^{func}$ に対応する遷移行列 $\boldsymbol{F}^{func}$ が求まり，また，become 操作の実行可能局面から個々の局面遷移の可能性についても求まっている．これらの行列式は，和演算 $+$，積演算 $\cdot$ に加え 演算 $(\boldsymbol{F})^{*}$ だけで表されており，同様に greedy に大域解析を行うことで，条件を満たす最小解を求めることができる．

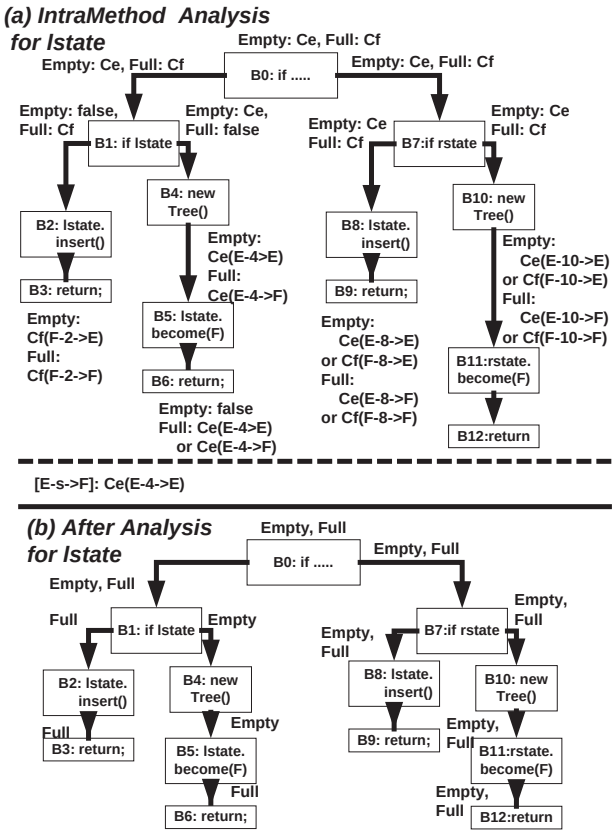

図 3.9: 2 分木プログラムの解析

3.7 解析手法の評価

本章では，サンプルプログラムを通して本解析手法の有効性を議論する．評価するのは 2 分木プログラムと N 体問題プログラムである．これらのプログラムは，4 章において排他制御緩和の効果についても評価を行う．

まずは，3.3 節で利用した 2 分木プログラムについて．このプログラムの場合，局面変数は二つ存在するが今回は別々の解析を行う．`insert()` のメソッド内解析の結果 (図 3.9(a))，局面遷移の可能性としては，$[\texttt{Empty}\rightarrow^{s}\texttt{Full}] : \texttt{Ce} \cdot (\texttt{Empty}\rightarrow^{\texttt{B4}}\texttt{Empty})$ が検出される．また，コンストラクタの解析に関しては，内部でメソッド呼出しを行わない場合は特別に排他制御ブロック内と同じ扱いで解析を行い，局面遷移なし (内部は常に初期局面のまま) として処理している．

大域解析においては，コンストラクタ呼出し (B4,B10) についてのみ呼出し関係解析を行うと想定して話を進める．本来，コンストラクタと呼出し側では

```
class Node {
    class MyState extends Phase {CLeaf, CPartial, CFull, Mass, Acc};
    internal MyState  mystate  = CLeaf;

    /* 内部用途 */
    consistent Node children(int i) { ... }
    consistent void setChildren(int, Node) { ... }
    void makeChild(...) { ...; setChildren(...); }

    /* 外部公開されたメソッド */
    Node(...) {  /* field 初期化のみ */ }

    consistent void insert(.....) { // 構築局面
        if(mystate.is(CFull)) {
            ...;
        } else if(mystate.is(CPartial)) {
            if(...) mystate.become(CFull);

        } else if(mystate.is(CLeaf)) {
            mystate.become(CPartial); ...;

        } else { throw new Error(); }
    }
    consistent void getCenterMass(CenterInfo) {
        // 構築局面のはず
        assert(!(mystate.is(Acc)||mystate.is(Mass)));
        mystate.become(Mass); // 重心局面へ.
        ...;
        mystate.become(Acc);  // 計算局面へ.
    }
    consistent void calAcc(Particle) { // 主計算部
        assert(mystate.is(Acc));   // 計算局面のはず
        ....;
    }
}
```

図 3.10: N 体問題プログラム (Node.java)

this に相当するインスタンスが異なるが，これについては解析不能だったとする．この場合でも，コンストラクタ内で局面更新は行われないと判定でき，$\rightarrow^{B4}$, $\rightarrow^{B10}$ は局面変化なしと判断する．一方で，insert() メソッド呼出しは呼出し関係解析を行わなかったため，初期局面は，Ce$\Leftarrow$true, Cf$\Leftarrow$(Empty$\rightarrow^{*}$Full) となる．また，$\rightarrow^{B2}, \rightarrow^{B8}$ の代りに $\rightarrow^{*}$ を使うこととなる．これらの情報を取りまとめると，結果的に $\rightarrow^{*}$ については Empty$\rightarrow^{*}$Full が確定し，insert() メソッド中の各ブロックも妥当な実行可能局面に定まる (図 3.9(b))．

次に，N 体問題プログラムについて．このプログラム (図 3.10，詳細は [8] にて公開) は J.Barnes & P.Hut アルゴリズム [12] を採用しており，8 分木構造に対応する Node クラスが計算の主体となる．図 3.10 は，メソッド構成と局面遷

移部を簡単に書下したものである．本プログラムでは，計算の段階に応じて変数アクセス状況が異なるため，Node オブジェクトを木の構築局面 (子供の数で 3 種に分類: CLeaf，CPartial，CFull) と，重心計算局面 (Mass)，加速度計算局面 (Acc) に分離した．メソッドは，他オブジェクトからの呼出されるメソッド (コンストラクタと他 3 種) 以外に，内部用途の 3 種のメソッドがある．局面記述は，外部公開されたメソッドについて，その開始局面に関する制約をユーザの知識として記述している．

処理内容が再帰的であるため，外部公開メソッドについては呼出し関係解析不能とする．他方，内部処理用メソッドや他クラスのメソッド呼出しについては，呼出し段数も高々 2 段程度のため，呼出し関係の解析を行うこととする．以上の条件のもと解析を行った結果，CLeaf$\rightarrow^s$CPartial, CPartial$\rightarrow^s$CFull, CLeaf/CPartial/CFull$\rightarrow^s$Mass, Mass$\rightarrow^s$Acc への一方向の局面遷移が確認できる．また, 内部メソッドについても実行可能局面が解析されており，setChildren() メソッドの実行局面が Cpartial 局面だけに制限できる．一方で，もし呼出し関係解析を一切行わなかった場合，内部メソッドの実行可能局面はユーザの明示的指示を省略すると正しく解析できなかった．また，局面遷移関係に関しても，注意深くプログラムを書かないと内部メソッド呼出しの影響で，解析結果が甘くなるケースがあった．

以上のプログラムでは，再帰呼出しなどについて呼出し関係解析を行わず，インライン可能な程度のメソッドについてしか呼出し関係解析を行っていない．以上の呼出し関係解析を行っただけでも，外部公開されたメソッド中の局面記述の結果，十分な解析成果が得られている．一方で，内部処理的なメソッドについては，呼出し関係解析により詳細な局面記述が不要となっている．今後の局面の利用法として，一般のユーティリティクラスを，局面情報を利用した同期部でラップして利用する場合を考えると，単純な呼出し関係解析でも効果的に機能するのではないかと期待している．

一方で，呼出し関係解析をより強化する場合，インスタンスの種別を行うことが重要である．上記プログラムの再帰呼出しは下降的であり，実際には自オブジェクトの局面更新を引き起こすことはない．より精密な解析を目指すには，再帰や繰り返しを考慮した上で自オブジェクトの参照がどの範囲に伝搬しているかを解析する必要がある．このような解析を実現するためには，ADDS[25] などのモデル化及び解析技術は有効だと考えられる．本論文でも，5 章において，

集合データ構造のモデル化で利用している．

最後に，メソッド呼出しの影響を考慮した局所解析を行うと，3.5.1 節のアルゴリズムでは実行可能局面を表す式が大きく膨らむ傾向にある．特に，逐次的な関数呼出しが行われた場合，積和表現のままでは組合わせが大きくなる恐れがある．このため，今後も 3.5.2 節で紹介したような，より効率的な解析手法の開発が重要であると考える．

3.8 局面毎の情報解析

3.8.1 複数の局面変数への対応

前節まで，ある局面変数に着目して解析の議論を進めてきた．実際には，オブジェクトは複数の局面変数を持ちうる．本節では，複数の局面変数に対応するための，モデル及び解析の拡張を行う．以下では，局面変数を L, M, N などの記号，もしくは添字を添えて L_i などと表す．局面変数の集合は $\boldsymbol{L}$ で示す．また，ある局面変数 L が局面値 P を取ることを L/P と表記し，ある局面変数の取りうる局面値の集合は $\boldsymbol{P}_L$ と表記する．各時点における局面可能性 S や，局面遷移可能性 $P{\rightarrow}^s Q, P{\rightarrow}^* Q$ についても，局面変数を明示する必要がある場合は，L/S や $L/P{\rightarrow}^s Q, L/P{\rightarrow}^* Q$ などと表記する．あと，簡単のため，L が全ての局面を実行可能局面とする場合， L/ALL とも記載する．

単一局面の解析によって得られた情報は，以下の二つである．

- 各命令 $inst$ に対する，各局面変数 L_i の実行可能局面 S_{inst}
- 各種局面遷移 $L_i/P{\rightarrow}^s Q$ 命令に対する，各局面変数 L_j の実行可能局面 S

つまり，実行可能局面に関する解析から，例えばある書込み命令 $inst$ が実行されうるのは，局面変数 L_1 が可能局面 S_1 の時であり，加えて L_2/P_2 でもある，といった情報が得られる．このように，$inst$ の実行可能局面は全局面変数を考慮した実行可能局面

$$\boldsymbol{S}_{inst} = \{L_1/S_1, \cdots, L_n/S_n\}$$

と表すことができる．以下では，$\boldsymbol{S}$ の各局面 L_i の実行可能局面を $\boldsymbol{S}(L_i)$ と表記する．また，$\boldsymbol{S}$ 間には集合積演算 $\cap$ が，

$$\boldsymbol{S}_1 \cap \boldsymbol{S}_2 = \{L_1/\boldsymbol{S}_1(L_1) \cap \boldsymbol{S}_2(L_1), \cdots\}$$

```
class Sample {
    class MyState extends Phase { P1, P2};
    internal MyState  L1 = P1;
    internal MyState  L2 = P1;

    void func1() {
        if(L1.is(P1) && L2.is(P1)) L2.become(P1);
        if(L1.is(P1) && L2.is(P2)) L1.become(P2);
        if(L1.is(P2) && L2.is(P2)) L1.become(P1);
    }
}
```

図 3.11: 組合わせ的局面遷移の例

として定まる.

局面遷移の可能性についても，同様に若干の変更が加えられる．つまり，ある局面遷移命令に対する実行可能局面も $\boldsymbol{S}_{become}$ として表現されるため，局面遷移の条件も

$$[\boldsymbol{S}_{become}] : L_i/P_m \rightarrow^s P_n$$

などと定義されることになる．このため，ある時点での実行可能局面が $\boldsymbol{S}_{inst}$ であるとき，他のスレッドの影響 $\rightarrow^*$ の影響を考慮すると，$\boldsymbol{S}_{inst} \cap \boldsymbol{S}_{become} \neq \phi$ であれば 局面遷移 $L_i/P_m \rightarrow^s P_n$ が可能といえ，加えて L_i/P_m が $\boldsymbol{S}_{inst}$ の実行可能局面に含まれるならば L_i/P_n も遷移可能局面として加えて良いことになる．$\boldsymbol{S}_{inst}$ からの全ての遷移可能性を考慮した局面 $\boldsymbol{S}_{fix}$ を求めるためには，以上の操作を初期局面から順に繰り返していき，新たな遷移可能局面がなくなるまで繰り返すことで求めることができる.

但し，以上の手続きは局面に関する全組合わせを考察しているわけではなく，単に，個々の局面変数に対して従来通りの解析を行い，その結果の応用法を考えているに過ぎない．つまり，図3.11の局面遷移のように局面が (L1/P1，L2/P1)，(L1/P1，L2/P2)，(L1/P2，L2/P2) の間で遷移している場合でも，そのようなことは解析できない．一方で，例えば，(L1/P1，L2/P2) から遷移しうる局面は (L1/{P1，P2}，L2/{P1，P2}) であることや，(L1/P2，L2/P1) から遷移しうる局面が (L1/{P2}，L2/{P1，P2}) に限定されることは解析される.

3.8.2 局面毎の変数アクセス情報

上記局面解析により実行可能局面が求まれば，各局面におけるそれぞれのフィールドへのアクセス内容を調べることは簡単である．つまり，あるフィールド $field$ に対して書込みが行われる局面は，$field$ への書き込み命令の集合 $WI(field)$ に対して，

$$Write(field) = \bigcup_{inst \in WI(field)} \boldsymbol{S}_{inst}$$

として求めることができる．同様に読み込み命令の集合 $RI(field)$ に対しても，以下のように定めることができる．

$$Read(field) = \bigcup_{inst \in RI(field)} \boldsymbol{S}_{inst}$$

一方で，あるアクセス命令が局面 $\boldsymbol{S}_{read}$ で $field$ からの読出しを行っていたとする．この場合，$Write(field)$ と $\boldsymbol{S}_{read}$ の間に交わりが存在すれば，その局面では当該アクセス命令と並行して実行される書き込みが存在すると言え，十分な排他制御が必要であると言える．逆に交わりが存在しないと解析できた場合，当該実行局面にいる間は，読出しへの排他制御は不要であると言える．

これらの応用面については，4 章の排他制御緩和や，5 章のキャッシュ付プロキシの実現で詳しく紹介する．

3.8.3 Code Versioning 応用

局面情報をさらに積極的にメソッドの Code Versioning などに利用することもできる．

例えば，3.7 節の N 体問題プログラムにおいて，`getCenter()` という `center` の位置を返すだけのメソッドがあったとする．ただ，`center` 自身は `FCalc` では更新されないが，`CMass` では更新される．つまり，`getCenter()` は `FCalc` では最適化可能であるが，`CMass` では通常通りの実行が必要となる．対処策として，例えばメソッド開始時の局面に応じた別々の最適化コードを準備すれば，状況に応じた効率化が可能となる．

上記の解析を行うためには，各メソッドの開始局面を制限した場合の解析情報が必要となる．つまり，局面 `FCalc` で開始したときのメソッドの性質，ならびに `CMass` で開始したときのメソッドの性質をそれぞれ解析できれば良い．加

えて，開始時の局面をどのように制限すれば最適化ケースを分離できるかが分かると，無駄な code versioning を避けることができる．

このような要求に対しても，3.5 節中の実行可能局面のように，開始局面 S_{init} を論理変数として扱っている場合，簡単に答えることができる．3.5 節では，実行可能局面は論理変数 S_{init} と $P_i \rightarrow^* P_j$, $P_i \rightarrow^{call} P_j$ を用いて表現され，後の大域解析 3.6 において，上記変数値が確定される．つまり，$P_i \rightarrow^* P_j$, $P_i \rightarrow^{call} P_j$ のみ確定値を利用することで，実行可能局面は S_{init} に関する論理式として表現可能である．例えば，先ほどの例であれば，$Write(\texttt{center})$ であるのは `CMass` である．一方で，`getCenter()` 内で `center` にアクセスする際の実行可能局面 S_{inst} については，$S_{inst}(\texttt{CMass}) = S_{init}(\texttt{CMass})$ として求めることができる．つまり，初期局面において $S_{init}(\texttt{CMass})$ を分離すれば，それ以外の局面では最適化が可能といえる．具体的な応用例については、5.4.6 節で述べる．

3.8.4 記述法

3.2 節で述べたように，オブジェクトの状態を分類し状態毎の挙動を記述するという研究は従来から行われている [45, 31, 33]．これらの研究では同期記述に関する継承時の問題について議論されており，同様の対策が本記述手法にも必要であると考える．

また，本研究は当初から局面を意識したプログラミングを行うのではなく，一般プログラムに局面記述を挿入していくという使い方を意識している．このため，標準 API などに対しても局面記述による最適化を行えるべきである．例えば Java Grande Benchmark[4] の例でも，スレッド間通信は明示的な同期を利用せず，Vector などの汎用の同期データ構造を介して行う事が多く，無駄な排他制御も行われることになる．このような状況で並列プログラムのチューニングを行うには，新たに専用ライブラリを記述するべきではないと考え，図 3.12 のように単に各メソッドの利用条件を局面記述の形で差分的に記述するだけで，利用状況に応じた最適化が可能になるよう計画中である．差分記述を可能にすることで局面記述によるコードの複雑化を回避できる．

記述面の改善としては，プログラマは局面遷移を記述するのではなく，重要局面が成立するための条件式 (2 分木の例であれば，`(left == Empty)`) だけを記述する方法も考えうる．但し，現時点では局面遷移がアトミックに行われる保証が解析精度の都合上必要であり，この性質を強いるために局面記法が現在

```
class VectorCustomA extends Vector {
  class PhaseA extends Phase {P0, P1};
  PhaseA phase = P0;

  synchronized void put(Object obj) {
    assert(phase.is(P0));
    super.put(obj);
  }
  synchronized void becomeStable() {
    phase.become(P1);
  }
}
```

図 3.12: 局面の差分記述

のようになっている.

3.9 まとめ

本章においては，局面変数を利用した適応的なオブジェクトに関して，その局面情報の解析手法を提案した．本解析の結果，プログラムの実行可能局面と局面遷移が解析され，プログラムの局面毎の性質などを解析することができる．これによって，処理系は局面毎のプログラムの性質を利用した適応的な最適化が可能となる.

本解析アルゴリズムは，各コードブロックがどの局面において実行されうるのか (実行可能局面) と，どのような局面間遷移がありうるのか (可能局面遷移) を解析するが，この 2 種類の情報は本来互いに依存しており，その確定には大域的な解析を必要とする．本論文では，解析の効率化のため (1) メソッド単位に実行可能局面の解析と，(2) その結果を利用して行う可能局面遷移の確定との 2 段階に分離したアルゴリズムを提案している．また，大域解析においては，メソッド呼出し関係が特定できる場合にはメソッド呼出しを考慮した解析が行えるような枠組みを提供している.

本解析手法は，複数の局面変数が存在する場合にも利用することができ，また，応用例として，局面によるメソッドの code versioning などに利用することもできる.

本解析手法をアプリケーションを通して評価を行った結果，適切に局面記述された場合，正確な局面解析を行うことができた．一方で，本アルゴリズムは

メソッド呼出し関係の解析方法については，特に言及をしていない．将来的に，一般アプリケーションに対して局面記述を利用したチューニングを効果的なものにするためには，正確なメソッド呼出し関係解析との連携法について，方策を考える必要がある．

第 4 章

排他制御緩和

4.1 はじめに

本章では，3 章の結果をもとに並列プログラムの排他制御緩和を行う．基本的なアプローチは，局面毎に変数の更新の有無を解析し，変化しない変数へのアクセスは排他制御せずに行うというものである．局面遷移を考慮した上で変数へのアクセスを 3 種類に分類し，それぞれに応じた排他制御規則を適用する．分類は，アクセス対象の変数が今後一切更新されないか，あるいは現在の局面においては更新されないか，あるいは更新を伴うかである．一切の更新がない場合は排他制御は不要であり，現在の局面において更新されない変数へのアクセスは，局面更新操作と排他的に実行されれば十分であるためである．

本章では，まず 4.2 節で排他制御区間の短縮に関わる関連研究について述べたあと，4.3 節で排他制御緩和に向けたアプローチについて述べる．4.4 節で排他制御の緩和法について，4.5 節で実行時コード技術について述べた後に，4.6 節で評価を述べた後，4.7 節で議論を行う．

4.2 関連研究

近年 Java を対象におこなわれている排他制御削減の研究としては，3.2 節でも紹介した Escape Analysis を利用したものが多い．解析によって Thread Local であると判断されたオブジェクトに対する排他制御を削減を行う．[46] においては，Java プログラム中の排他制御を 24%-67% まで削減できたとの報告もある．一方で，これらの研究が意図しているのは，Thread Local である，つまり元来ボトルネックなど起こり得なかった場所の排他制御を削減し，ロックのオーバヘッドを削減したに過ぎない．Java においては，ロックの高速化に関

する研究も盛んに行われており [11, 37, 40, 19]，衝突が起こらない場合のロックの実行コストを下げる事に成功している．一方で，複数のスレッドが同時に単一オブジェクトにアクセスするような状況に対しては，あまり対策が行われていない．

一方で，並列プログラムにおける排他制御区間の短縮を目指した研究として，Schematic[43] や OPA[50] が知られている．これらのプログラミング言語では一般の手続き型言語とは若干異なるメモリモデルを採用しており，メソッド内の全フィールド読み出しをメソッド開始時に一括実行し，最終書き込み点でオブジェクトの状態更新とロック開放を行うという意味を与えることで，メソッド内解析のみで排他制御区間の緩和を行う．一方で，排他制御メソッド `foo()` が更新を伴う `bar()` を呼出した場合であっても，`bar()` 内の更新内容が呼出し側の `foo()` には反映されないといったことが起る．仮に，メモリモデルを一般の手続き型言語に近づけようとして，内部呼出しされたメソッド中のアクセスを含めた一貫性保証を行おうとした場合，`bar()` などのメソッド中のフィールドアクセスを全て解析した上で一括読み出しや最終書き込み点の解析を行う必要がある．よって，解析の問題が解決しない限り，本技術をそのまま一般的なメモリモデルへ応用することは難しい．一方で，[50] では free method という定数化した変数を利用した最適化アプローチも提案されている．本章で紹介する排他制御緩和手法は，free method の手法 (本論文では immutable に相当) を応用，局面を用いて体系化したものとも言える．

4.3 排他制御緩和へのアプローチ

4.3.1 提案する排他制御構文

Java の排他制御緩和は排他制御区間内に変数アクセスが行われることを保証したものであり，排他制御区間を勝手に短縮して良いものではない．そこで我々は，Java の synchronized 構文に代わる新たな consistent 構文を提案する[1]．Java の synchronized 構文は，ブロック入口と出口にそれぞれ lock，unlock 操作を行うことで，ブロック内で実行される全メモリアクセス命令が他ブロックと排他的に実行されることを保証する (Java の排他制御モデルは [22] を参照)．それに対し，consistent 構文では，Java の命令実行順序は守るが，排他制御区

[1] **consistent{...}** などと記述し，method 修飾詞としても使用可能

```
class A {
    private int val = 0;
    consistent void foo(ValFolder folder) {
        int tmp = folder.ctxt;   //  E0
        if (val < 10) {          //  E1
            inc(tmp);              //  E2'
            return val;            //  E3
        } else
            return val;            //  E4
    }
    private void inc(int arg) {
        if (arg > 0) val += arg; //  E2
    }
}
```

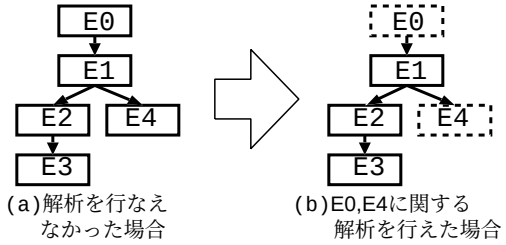

図 4.1: 排他制御区間

間 (lock, unlock の場所) のみが synchronized と異なる．consistent 構文では，ブロック内で行われる全アクセス命令列のうち，以下に指定する命令群についてのみ排他制御区間での実行を保証する．

- 自身オブジェクトのインスタンス変数への書き込み (書き込み内容がメモリに書き出されることを保証)
- 自身オブジェクトのインスタンス変数からの読み出しのうち，排他制御区間から外れた場合，ブロック全体を排他制御区間とした場合に比して，その結果に変化があるもの．つまり，排他制御を行わなければ他スレッドによって結果が変化しうるもの

つまり consistent 構文は，対象でないオブジェクトへのアクセスや，排他制御区間から出してもアクセス内容に変化がない読み出し命令を排他制御区間に入れないことで，対象オブジェクトのデータの一貫性を保証しつつ，かつプログラムの並列度向上を目指す．

例えば，図 4.1 のプログラム例を考える．この例では，メソッド `foo()` に対して consistent 構文が使われている．synchronized 構文と異なり，consistent の

対象オブジェクトは常に `this` である．この例では，インスタンス変数へのアクセスとして，E0, E1, E2, E3, E4 が存在する (ここでは説明の簡単化のため，各アクセスを複数行に分離した記述を行っている)．

まず E0 は異なるクラスへのインスタンス変数アクセスであり，自身のオブジェクトへのアクセスではありえない．このように，自身オブジェクトへの読み書きを含まないと解析できたコード片は，consistent 構文では排他制御区間から外すことが可能である．

次に E4 についてであるが，そのアクセス結果はブロック入口のメモリ状態のみに依存するといえる (もし，入口での値が 10 未満の場合は，E4 に到達せず，10 以上の場合は値が更新されない)．そのため，このことが解析できた場合は，E1 $\leftrightarrow$ E3 だけを排他制御区間として静的に決めることが可能となる (図 4.1(b))．

ここで，従来研究 (Schematic[43] や OPA の `instant` メソッド [50]) との比較を簡単に述べる．これらの言語では，オブジェクトの意味的なシングルスレッドネスを保証しつつ, 単一オブジェクト上の複数スレッドの並列実行を許す．但し，命令実行順序のモデルが Java とは大きく異なり，図 4.1 の例では，デッドロックを引き起こすか，もしくは E2 における更新結果が E3 に反映されない．このため，Java の synchronized ブロックをこれらの一貫性保証構文に単純に置き換えただけでは，プログラムは従来通りの動作をしない．

4.3.2 ケーススタディ

4.3.1 節で述べた consistent を実現するためには,「E0 は他のオブジェクトへのアクセスである」，また「E4 は val $\geq$ 10 の時にしか実行されず，その時は val の値が更新されることはなく常に同じ結果になる」，という情報を解析する必要がある．前者の解析は本論文では取り上げず，4.6.1 節で示すようにオブジェクト外部からのアクセスを禁止した変数を導入することで対処した．一方で，後者に関しても解析機構のみで対応するのは非常に困難である．

そこで我々は 3 章で提案した**局面**を利用する．プログラマには**局面変数**を用いて，プログラムの振舞いが変化するポイントを記述してもらうという手法をとる．プログラマは局面変数を用いて，プログラムが取り得る局面，局面の変化，局面による挙動の変化等を記述することができる．

例えば図 4.1 のプログラムにおいては，オブジェクトの状態を **val $<$ 10 の局**

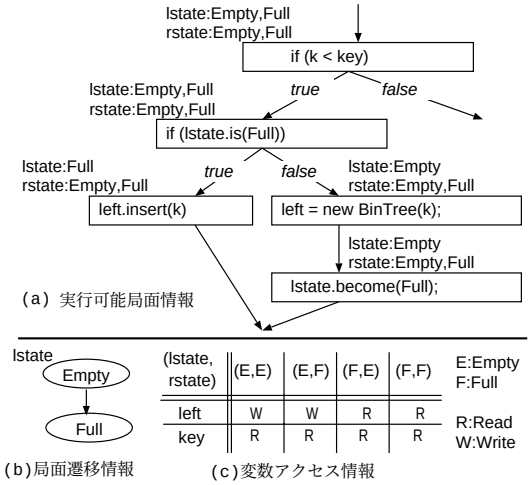

図 4.2: 局面解析情報

面 と **val** $\geq$ **10 の局面**に分離することができる．このような局面記述がされたプログラムを解析することで，変数 val は val ≥ 10 の局面では値が変更されない，また E4 は val ≥ 10 の局面でしか実行されない，といった情報を得ることができ，この解析情報を利用して排他制御区間の緩和を行うことが可能になる．

4.3.3 局面解析によって得られる情報

ここでは局面記述されたプログラムを解析 (3 章) することにより，得られる情報について述べる．

局面解析から得られる情報は以下の 3 つである．

1. **実行可能局面**： 各命令がどの局面で実行される可能性があるか
2. **局面遷移情報**： どのような局面遷移が可能か
3. **変数アクセス情報**： 各変数が，どの局面の時に，どのようなアクセスが行われるか

以下では (1)〜(3) を，例を用いて説明する．

図 3.4 を解析することにより図 4.2(a) のような実行可能局面情報が得られる．つまり `left.insert(k);` は局面変数 `lstate` が局面 `Full` の時のみ実行され，局面 `Empty` の時に実行されることはない，また `left = new BinTree(k);` は局面

変数 lstate が局面 Empty の時に実行され，局面 Full の時は実行されないなどといった情報が得られる．

次に局面遷移情報であるが，これは図 4.2(b) のように，局面変数 lstate は初期局面が Empty であり，局面の遷移としては Empty から Full へ遷移する可能性がある．また一度 Full の局面になると別の局面へ遷移する可能性はない，などといった情報が得られる．

最後に変数アクセス情報であるが，これは例えば変数 left は局面変数 lstate が局面 Empty の時には write アクセスされ，局面 Full の時には read アクセスされるといった情報が得られる．但し，局面変数が複数存在する場合は図 4.2(c) のように局面変数の組合せで結果が出されることになる．

4.4 排他制御緩和の実現法

4.4.1 概要

4.3.1 節で述べた通り，consistent の基本方針は，定数化した変数へのアクセスを排他制御区間に含めないことと，可能な限りプログラム実行の並列度を上げることである．そのため，consistent ブロック内の変数アクセス命令を次のように分類する．

- **Iimmutable**：定数化した変数 (ある局面，またその局面から遷移可能な全ての局面で値が変化しない変数) へのアクセス
- **Istable**：ある局面で変化しない変数へのアクセス
- **Imutable**：ある局面で変化する変数へのアクセス
- **Ibecome**：局面更新操作

Iimmutable はアクセスする変数が定数化しているので，排他制御は不要である．また Istable は局面が更新されない限り値が変化しないので，局面更新操作，すなわち Ibecome とのみ排他実行すれば，他の命令とは並列実行可能となる．Imutable は更新する変数へのアクセスなので，Imutable 同士は排他実行しなければならない．また局面更新によって変数のアクセス状況が変化する可能性があるので，Ibecome とも排他実行しなければならない．

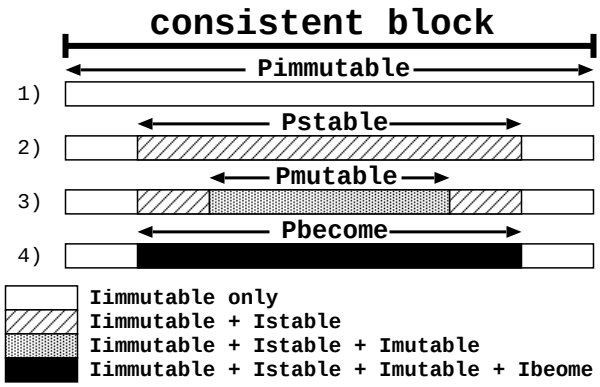

図 4.3: consistent ブロックの実行種別

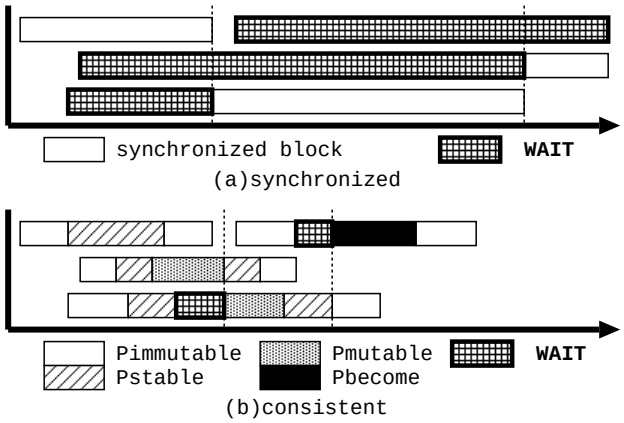

図 4.4: 並列実行イメージ

上記の分類を用いて，consistent ブロック内における変数アクセスの一貫性を保持するために，consistent ブロックの実行区間を，図 4.3 の 1)∼ 4) に分類する．各区間の排他制御は Iimmutable ∼ Ibecome の関係からも，Pstable/Pbecome 間，Pmutable/Pmutable 間，Pmutable/Pbecome 間について行えばよい．また，図 4.3 において Pmutable は Pstable に包含されているものとしているので Pmutable/Pbecome の排他制御は暗黙的に行われる．synchronized ブロックの実行イメージを図 4.4(a) に，consistent の実行区間を分類することによる並列実行イメージを図 4.4(b) に示す．図 4.4 より， consistent ブロックを分類することでプログラムの並列度を向上させることが可能となり，WAIT 区間を短くできることがわかる．

consistent ブロックを図 4.3：1) ～ 4) の区間に分類するために，局面記述されたプログラムから CFG(*controll flow graph*) を作成し，CFG の各ブロックについて変数へのアクセス状況に応じた分類を行い，適切な排他制御区間を決定する．各プロセスについては以降の節で詳しく説明する．

4.4.2 局面分岐コード

ここでは排他制御区間を決定する前に，局面分岐の取り扱いについて述べる．プログラマは局面に関する条件を記述することで，各コード片の実行局面を限定している．一方で，対象となる局面変数には複数の局面とその遷移の可能性があるため，そのアクセスを Istable として排他制御区間に入れる必要が生じる．つまり，consistent のほとんどが排他制御区間に残ることになる．

このような状況を回避し排他実行区間を短縮するために，ちょうど図 3.3 の `if (left != null) continue label;` の再チェックに相当する技法を導入する．つまり，条件分岐 `if (lstate.is(Full))` をあらかじめ排他制御区間外で行い，次の else 節の冒頭で排他実行区間に入った後，lock 獲得までの間に更新されたかも知れない局面を再チェックする．一方で then 節 (`lstate:Full`) の場合，局面遷移は単調性でありこの局面からの遷移はあり得ないと解析できている．このため，`lstate` への再チェックアクセスは Iimmutable と認識することができ，再チェックの必要や排他区間にいれる必要はなくなる．

4.4.3 ブロックの分類

排他制御緩和に関しても，3.5 節と同様に基本ブロックをもつ CFG に変換し，その後各ブロックを変数アクセス状況により分類する．簡単のため，局面変数は一つとして話をすすめ，最後で複数局面の対応について述べる．

4.4.1 節で紹介したように Iimmutable ～ Imutable は次のようにして定める．

- **Iimmutable**：命令の対象変数が，当該命令の実行可能局面，またはそこから遷移可能なすべての局面において更新がない
- **Istable**：命令の対象変数が，当該命令の実行可能局面全てにおいて更新がない
- **Imutable**：命令の対象変数が，当該命令の実行可能局面で更新の可能性がある

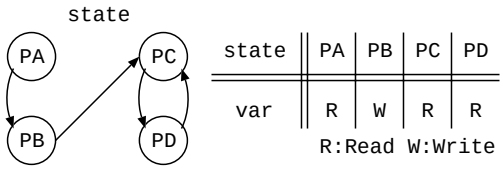

図 4.5: 局面解析情報例

インスタンス変数へアクセスを行うブロックの分類の例として局面 PA，PB，PC，PD をとり得る局面変数 state が存在し，図 4.5 のような局面解析情報が得られた場合を考える．

変数 var へアクセスを行うブロックの実行可能局面が PC，PD であった場合を考えると，局面 PC，PD から遷移可能な全ての局面 (PC，PD) で var は read アクセスしかされない．従ってそのブロックは Iimmutable として扱われる．同様に変数 var へアクセスを行うブロックの実行可能局面が PA，PC，PD であった場合を考えると，局面 PA，PC，PD では var は read アクセスしかされないが，局面 PA から遷移可能な局面 PB では write アクセスされる．したがってそのブロックは Istable となる．また変数 var へアクセスを行うブロックの実行可能局面が PB，PC，PD であった場合を考えると，局面 PB では var へ write アクセスされるので，そのブロックは Imutable と分類されることになる．

また **if** 分岐に関する再チェックブロックは Istable とし，条件分岐自身は Iimmutable 扱いとする．局面の更新 (become 命令) を行うブロックは Ibecome とし，局所変数へのアクセスや算術演算のみを行うブロックは Iimmutable 扱いとする．問題となるのはメソッド呼出しに関してであり，メソッド内の副作用を含めて解析を行う必要がある．残念ながら，本章のプロトタイプシステムにおいては，かなり保守的な見積りが行われている．現在はメソッド呼出し関係の解析を行っていないため，メソッド呼出しは，その呼出し局面で起こりうる，すべてのフィールドアクセスの可能性を持つとして扱われる．例えば，現在の実行局面 P で Ibecome 命令が実行される可能性があれば，メソッド呼出し命令を Ibecome と判定するといった具合に行う．これについては，4.7.2 節で議論を行う．

以上のように CFG の各ブロックを分類した後，図 4.3 に示すように，排他

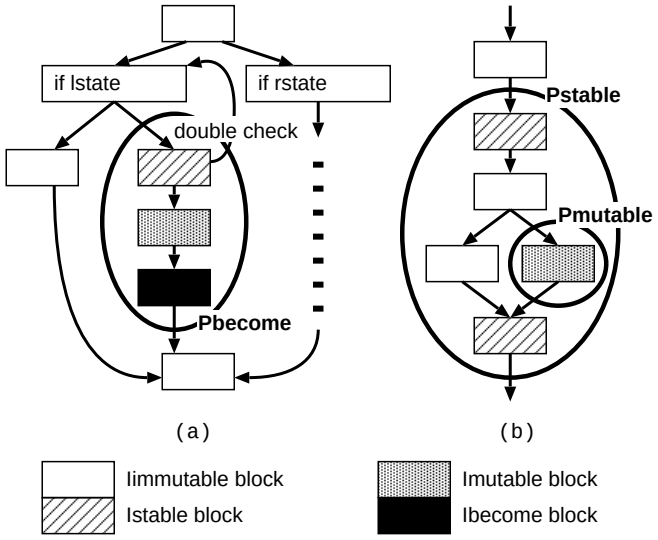

図 4.6: 排他制御区間解析

制御区間を決定する．基本的には，Iimmutable, Istable，Imutable，Ibecome と Pimmutable, Pstable, Pmutable, Pbecome の包含関係を守りながら，かつ各排他制御区間を最小化するよう決定する (アルゴリズムは省略)．各区間の決定の例を図 4.6 に示す．図 4.6 の (a) に示すように Ibecome を含む区間は Pbecome 区間となり，(b) に示すように Pstable は Pmutable を含むことができる．

最後に，複数の局面がある場合についてであるが，実行可能局面を 3.8.1 節のように考えれば，そのまま同じ議論が成立する．遷移可能性と更新の有無についても，それぞれ 3.8.1 節，3.8.2 節の通りである．但し，本プロトタイプの実装においては，簡単のため $\boldsymbol{S}_b$ からの遷移可能な局面 $\boldsymbol{S}_b^*$ は，各局面に関する実行可能局面 $\boldsymbol{S}_b(L_i)$ から $L_i/\rightarrow^*$ によって遷移可能な実行可能局面を求め，それを $\boldsymbol{S}_b^*(L_i)$ としている．

4.5 実行時コード

本プロトタイプでは，consistent の実現にあたっては，実装の portability を持たせるため直接処理系に改編を行なわず，consistent 構文や局面記述など，拡張記述された Java プログラムを，同様の処理を行う標準のプログラムに source to source コード変換を行うことで実現している．本節では実際に consistent を実現するため (4.4.1 節で述べた排他制御規則を実現するため)，どのようなコー

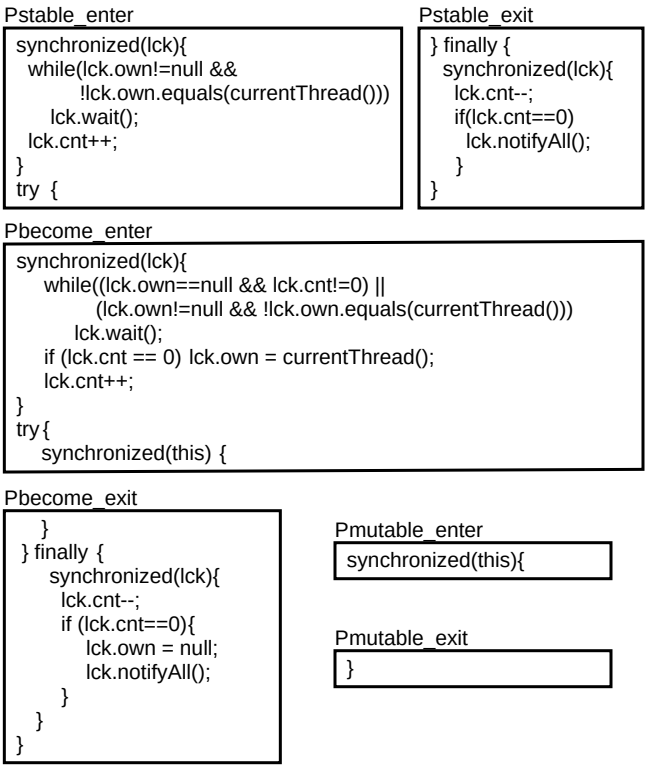

図 4.7: コード変換

ドに変換すればよいかについて説明する．

Pmutable/Pmutable，Pmutable/Pbecome，Pbecome/Pbecome については，実行区間を synchronized(this) で囲むことによって実現する．

Pstable/Pbecome については，メソッド呼出しなどによる入れ子状の consistent に対応するため，スレッドの owner 情報 (`lck.own`)，ネスト回数 (`lck.cnt`) 情報を利用し，ネストされた Pstable，Pbecome を実行する時に，デッドロックが起きないようにする．

以上より，コード変換は，実行区間分類で決定される各区間の入口，出口に対して，図 4.7 に示す変換を行う．但し，全てのインスタンス変数へのアクセスが consistent で囲まれている場合，単一のメソッド呼出し命令で構成される Pstable，Pmutable，Pbecome 区間については排他制御処理を行わない．これは，呼び出されたメソッド側で必要な consistent 処理が行われるためである．このため，図 3.4 の `left.insert(k)` の場合については，解析上は Pbecome 扱いされるが，実際の排他制御を回避することが可能となる．

4.6 評価

4.6.1 評価環境

解析機構ならびに排他制御緩和機構をプロトタイプ実装し，その評価を行った．また，コード変換には EPP[29] を使用した．評価環境として SunEnterprise6500(20CPU，JDK-1.3.1) を用いた．

但し，現在の解析機構では他のオブジェクトによる write アクセスの可能性を解析できていない．つまり，このままでは全局面において write アクセスの可能性が存在してしまう．今回は，オブジェクト外部からのアクセスを禁止した変数 (`internal` 変数) を準備し，consistent 構文が一貫性保証する対象を `internal` 変数に限定することで対応している．この制限により，今回の実装では間接参照などの影響を無視して解析を行えている．`internal` 変数を宣言する際は，

```
internal int key;
```

のように修飾子を付加する．

4.6.2 2 分木プログラム

2 分木に 100 万個のノードを追加するプログラムについて評価を行った．評価対象としては図 3.4 のプログラムにおいて，`consistent` を単に `synchronized` としたもの (**naive**) ，手動で最適化を行ったもの (**manual**) ，本最適化機構を用い自動最適化を行ったもの (**auto**) について比較を行った．その実行結果を図 4.8 に示す．

図 4.8 より，自動最適化を行ったものは手動最適化に比べ 2 倍近く実行時間がかかっていることがわかる．これは `Pbecome_enter`，`Pbecome_exit` のコストが原因であると考えられる．しかし，naive で生じていたボトルネックについては，自動最適化により除去できたと考える．

4.6.3 N 体問題

このプログラム (図 3.10，詳細は [8] にて公開] は J.Barnes & P.Hut アルゴリズム [12] を採用しており，木構造に対応する Node クラスが計算の主体となる．本プログラムでは，計算の段階に応じて変数アクセス状況が異なるため，Node オブジェクトを木の構築局面 (`CLeaf`，`CPartial`，`CFull`) と，重心計算局

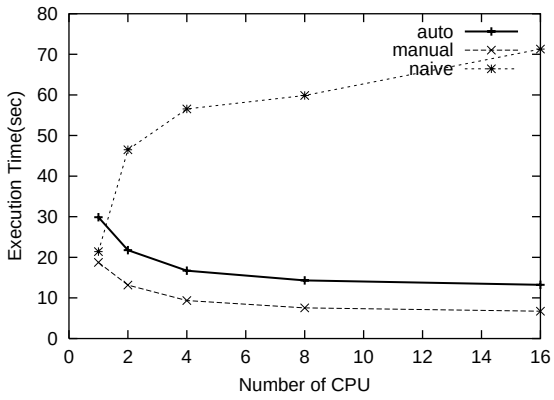

図 4.8: 評価：2 分木の実行時間

面 (Mass)，加速度計算局面 (Acc) に分離した．構築局面中の 3 局面は，Node が葉 (CLeaf) であるか，子がそろった枝 (CFull) であるか，それ以外の枝 (CPartial) であるかを示す．Node クラスの全メソッドは単に consistent メソッドとして記述し，その実行局面について限定できる場合は `assert(!phase.is(Acc));` のように表記している．その結果，計算時間のほとんどを占める Acc 局面における `calAcc()` が `Pimmutable` 扱いとなり，排他制御が一切不要となった．これは，Acc 局面においては局面遷移と変数更新が否定できたためである．一方で，木の構築局面については変数更新のない CFull 状態を分離することで，ボトルネック解消を目指している．

図 4.9 は自動生成したコード (**auto**) と，手動最適化によるコード (**manual**) の比較である．Acc 局面のみの時間と合計時間 (Total) を表示している．評価時には GC の影響を受けないように，初期メモリを大きくとり，また局面遷移時に明示的に GC を行った (GC は実行時間に含めていない)．Acc 局面においては，排他制御を完全除去できた結果，手動に匹敵する速度が達成された．但し，合計時間においては手動のような顕著な台数効果は現われていない．これは，CFull 状態を分離しただけでは十分でなく，2 分木で行ったように各々の子に対して Empty/Full 局面を持たせるべきだったといえる．

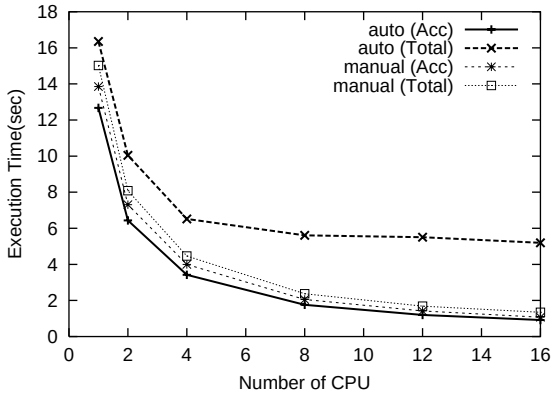

図 4.9: 評価：N 体問題の実行時間

4.7 議論

4.7.1 複合オブジェクトへの対応

この節では，例えば `Vector` や `Hashtable` といった，一般に利用されているライブラリに consistent 構文を導入する際の，問題点について議論する．

Java 上の並列アプリケーションの場合，4.6 節のように新たにプログラムを書き下す場合もあるが，一方で，プログラマは明示的な排他制御を記述せず，`Vector` などの同期機構つき共有オブジェクトを介して，スレッド間データ交換を行う場合も多い [4]．このため，これらのライブラリに局面を導入し，局面に応じた排他制御規則を適用する利点は大きいと考える．

一方で，現在の consistent の枠組みと解析系では，上記ライブラリへの応用は現実的ではない．一つの問題点は，外部から直接参照されるオブジェクトのインスタンス変数だけに着目しても，一貫性保証という観点で意味がない点である．実際に，上記クラスの実装を調べても，外部に見えているオブジェクトは窓口に過ぎず，要素データは，配列や inner class といった，別オブジェクトに格納されている．つまり，これら内部実装に利用される一連のオブジェクト群を，一つのオブジェクトの内部とみなす枠組みが必要である．加えて，これらの内部オブジェクトへの参照が，外部に漏れていないことも保証する必要がある．今後の検討課題である．

4.7.2 メソッド呼出し解析

現在のプロトタイプ実装においては，アクセスに関してメソッド呼出し解析ができていない．このため Iimmutable ～ Ibecome の分類にあたっては，メソッド呼出し命令には保守的な見積もりを行っている．つまり，その呼出し局面で起こりうる，すべてのフィールドアクセスが可能であるとして，解析を行う．このため，図 3.4 の `left.insert(k)` 呼出しも，解析上は become の可能性ありとみなされている (但し，この例では 4.5 節で紹介した実行時技術により，実際には排他制御は不要となっている)．実際には `left.insert(k)` 呼出し実行中に，このオブジェクトに対する `become()` は起こり得ないのであるが，今回は解析できていない．

但し，現在，分散向けのアクセス解析として，5.5.4 節のようなメソッド間解析の枠組みを提案しており，同様の方式は排他制御緩和においても利用可能である．但し，5.5.4 節ではオブジェクトの参照関係解析を簡単にするため，データ構造に ADDS (3.2 節参考) に似た構造を入れて対応している．共有メモリ上のプログラムで解析精度をあげるためには，オブジェクトの内外を分離した解析などを導入する必要があると考える．

4.8 まとめ

本章では，局面解析結果を用いた並列プログラムの排他制御緩和の自動化手法を提案，プロトタイプシステムを用いた評価を行った．提案しているのは，局面毎の性質に基づいた緩い排他制御規則を適用することのできる，一貫性保証構文 `consistent` である．プログラマに，オブジェクトの局面に関する宣言的な記述を行ってもらうことにより，システムはその局面変化や各局面毎の変数アクセス状況を解析し，状況に応じて排他制御区間の短縮を行う．ある局面で更新がないと保証された変数へのアクセスは，当該局面においては排他制御なしで実行され，さらに，今後一切更新が無いと保証できた場合は，一切の排他制御を行わない．

プロトタイプシステム上で，いくつかのアプリケーション例を通して評価を行った結果，ユーザが単純に一貫性保証区間 `consistent` を宣言した場合であっても，局面記述次第ではボトルネック解消に成功していることが分かった（２分木，N 体問題）．Iimmutable と解析できた場合については，実行速度面でも手

動最適化を行ったプログラムに匹敵しており，良好な結果であると言える．同様のプログラムを consistent ではなく synchronized をつかって書いた場合，完全なボトルネックを生じていたことを考えると大きな成果と言える．

一方で，本手法をより一般に適用するためには，複合的なオブジェクトの扱いが重要である．というのも，オブジェクトの中には複数のオブジェクトを利用することで，一つの意味的なオブジェクトを表現しているケースも多いためである．この場合，一貫性もそのオブジェクト群単位で考える必要がある．今後，参照解析や一貫性の扱いなどについて更なる考察が必要と言える．

第 5 章

効率的分散オブジェクトの実現

5.1 はじめに

最近，分散メモリ計算環境は High Performance Computing の世界においても重要視されており，Grid, PC クラスタに代表されるようなネットワーク接続された計算機群が高い計算性能を示している [9]．但し，これらの環境におけるプログラムの主流は分割統治スタイルであり，リンク構造を持った共有データを扱う並列プログラムを実現し，効率化していく作業はかなりの労力を必要とする．

現在，広く普及した RMI, HORB といった分散オブジェクト技術を利用することで，遠隔メソッド呼出しは容易に実現可能となった．但し，これらの技術は，あるホスト上のオブジェクトに遠隔アクセスするためのものであり，複数のホストから同時にアクセスされる状況ではボトルネックが生じる．分散メモリ環境でデータを共有する場合，あるノードのみに当該データが存在したのでは，毎回ノード間通信が発生する．一方で，各ノードにキャッシュとしてデータのコピーを配置した場合，読み出しアクセスは高速化されるが，一方で書き込み操作はキャッシュ更新のための通信を必要とし，コストが高い．元来，更新作業は排他的に行われるものであり，更新が多く行われるような状況では本体のみにデータを配置し，集中処理したほうが効率的である．逆に，更新があまり行われず，読み出しアクセスが並行に行われると分かっていれば，キャッシュを積極的に利用するべきである．

本章の目標は，共有メモリプログラムを容易に分散環境に移植し，処理系のサポートの下，容易にチューニングを行えるような環境の実現である．そのために，以下のようなアプローチを提案する．

- 共有データ格納のための集合系ライブラリの提供
- 要素セルの実現には，キャッシュ付プロキシを含めた効率的実装を利用

つまり，プログラムの移植に辺り，まず共有メモリ上のデータ構造を集合系ライブラリに変換する．但し，データを分散させただけでは高速化は果たせないため，要素セルの実現にはキャッシュ付プロキシを利用するというものである．

また，キャッシュ付プロキシの実現は局面解析に基づいた自動生成を目指す．前述したように効率的なキャッシュの実現には，しばらく変わらないデータと，更新を伴うデータをあらかじめ判別することが重要である．そのための方策として，要素セルの性質が変化するポイントを局面変化としてプログラマが記述し，その情報をもとに処理系が各局面毎の変数の性質を解析し，キャッシュの実現に応用する．これにより，局面内で更新が行われない変数はキャッシュとして利用することができ，局面変化の時にだけ，無効化処理を行えば良いことになる．

本章では，関連研究について述べた上で (5.2 節)，集合系ライブラリの提案 (5.3 節) と，要素セルのキャッシュプロキシを用いた効率的な実装方法 (5.4 節)，また，そのための局面解析およびアクセス解析の詳細 (5.5 節) について述べる．本研究は，システムの提案ならびに解析手法の提案にとどまっており，実システムを用いた評価はできない．このため評価に関しては，5.6 節で述べるように，アプリケーション例を通しての解析手法などの評価にとどめる．

5.2 関連研究

現在，分散環境向け Java プログラミング環境として，HORB[3] や Java RMI[5] といった分散オブジェクト技術が広く普及している．これらの技術は，遠隔ノードにあるオブジェクトに対して，一般のメソッド呼出しのようにアクセスすることを許している．呼出しに際しては，引数がオブジェクトの場合，その再帰的なコピーが渡されることになる．HORB においては，さらに遠隔オブジェクトの動的生成や非同期メソッド呼出しの機能が提供されており，プログラマは分散に関してはあまり意識せずプログラミングを行うことができる．一方で，これらの技術は基本的には単一ノードにオブジェクトを配置し，それに対し遠隔メソッド呼出しを行うものである．広く共有されたオブジェクトへのアクセス

は多くの遠隔メッセージを引き起こすことになり，非同期呼出しによる遅延隠蔽技術だけではボトルネックの発生を抑えるには十分とは言えない．

一方で，分散環境向けに分散共有メモリを提供するアプローチも存在する．遠隔メソッド呼出しの代りに，データがアクセス側ホストに移動し，計算はアクセス側で行われる．複数のホストが並列に読み出しアクセスをする場合も，データのコピーを分配することが可能であり，一方で，あるホストがロックを取得して書き込みを行う場合，まず所有権が当該ホストに移動し，ロックの解放をもって書き込みが行われる．Java を対象とした研究としても Java/DSM[49], JDSM[42] などが存在する．このうち，Java/DSM は汎用の分散共有メモリ機構と同じく，ページ単位のメモリ管理を行っている．これは，意味的に関連がないオブジェクトへの書き込みの影響を受けるという事である．影響を抑えるには，明示的に関連するデータをまとめて配置する必要がある．一方で，JDSM ではオブジェクト単位でのコピーの取得／無効化を行う．但し，共有メモリプールに登録されたオブジェクトから参照されたオブジェクトについては，一貫性の議論が行われていない．また，分散共有メモリ的アプローチの最大の問題は，ロック操作のコストである．このため，JDSM ではコピーの取得とロックの取得をプログラマが使い分け，プログラマ自身が一貫性を意識するというプログラミングスタイルが取られる．このような手法は性能を引き出すためには重要であるが，一方でバグの原因ともなりやすい．

また，規則計算の分野では MPI[6]，特に Collective 通信を利用したプログラミングが多い．Collective 通信とは，配列データの分配・再配置をホスト間で行うための，高度で強力な通信機能である．データ転送時は，通信に参加するホスト間で協調動作が取られる．規則計算では，各時点でデータがどのように利用されているかプログラマが把握していることが多く，このため，明示的データ再配置が広く利用されている．

5.3 分散コレクションライブラリ

5.3.1 基本モデル

分散コレクションは，ホスト間にまたがった分散した存在として実現される．また，一種の分散オブジェクトとして各ホストからのアクセスが可能である．具体的なデータ構造としては，集合やハッシュ，それから後述する N 分木のよう

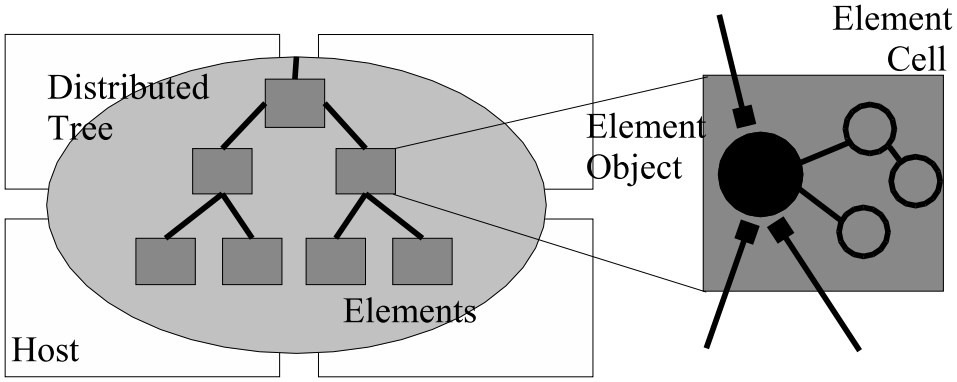

図 5.1: 分散コレクションのイメージ (左：集合，右：Tree)

なデータ構造を予定している．

分散コレクションがオブジェクトを要素として登録 (**要素オブジェクト**という) する場合，コレクション内部では**要素セル**と言われる空間が作られ，要素オブジェクトは，そのセルの**代表オブジェクト**になる．分散コレクションの特徴の一つは，要素セルへの参照が外部に対して隠蔽され，また，要素セル内で参照関係が基本的に閉じていることにある．このため，必要に応じて要素セルの移動なども行うことができる．

外部から要素オブジェクトへのアクセスは，以下のように必ず分散コレクションを通して行われる．

```
DistCol<Elem> col = new DistCol<Elem>();
col.d_foo(id, args);    /* id 相当の Element への foo(args)  */
col.dall_bar(args);     /* 全要素に対し bar(args)  */
```

また，メソッド呼出しもリモートメソッド呼出しとして意味付けされる．つまり，引数や返り値としてオブジェクトが渡される場合，オブジェクト自身ではなく，その（再帰的）コピーが渡される．以下のような操作は要素セルデータの更新を意味せず，単に，取得データをローカルに更新したことになる．

```
ElemData data = col.d_getData(index);
data.field++;
```

各要素セル内に存在するのは，代表オブジェクトとそこから参照されるオブジェクト群である．外部から引数として渡されたオブジェクトや，内部で新規

生成されたオブジェクトも存在する．一方で，要素内外の参照を遮断するため，要素内では以下のような制限がある．

- static field へのアクセスは許されない．また，標準 API についても，要素内で利用可能なものを制限する．これは，`static` や `native` を介した参照漏れを防ぐためである．一方で，`java.util` パッケージなど，問題がないと認められたライブラリは，自由に利用することができる．
- 分散オブジェクトへの参照は，許さない．
- 代表オブジェクトは特別なクラスとして扱われ，セル内で新規生成したり，参照をセル外に渡すことは許されない．
- 隣接要素セル間のアクセス手段は，分散コレクションが提供する

これらの保証は，コンパイルやランタイムエラーによって実現予定である．

このように要素セルは一つの計算ホストのように振る舞う．`synchronized` 構文についても同様で，排他的アクセスが保証されるのは要素セル内のメモリアクセスのみとする．つまり，メモリアクセスの一貫性は各要素セル毎に保証することとする．一方で，あるセル間呼出しが自セルに戻ってきた場合，それは同一スレッドの動作として見なされる．これは，一連の操作が複数のスレッドの動作と見なされ，デッドロックが生じるのを防ぐための処置である．

5.3.2 再帰データ構造

ライブラリの中には，2 分木データ構造などの再帰データ構造も提供される．但し，要素間参照を直接扱すことはできず，ライブラリの提供する API を用いて隣接ノード間の呼出しのみ許される．また，要素間の参照関係は順序関係がつけられており，現状では ADDS[25]（3.2 節参照）における 1 direction, uniquely forward/backward な参照のみが許されるものとする．

2 分木クラス `Tree` の場合，`Left`，`Right`，`Parent` でラベルづけされた要素間参照が存在する．但し，`Left`，`Right` は一定方向の参照であり，`Parent` はその逆向きリンクと特徴づけされている．これにより，`Left/Right` 方向へデータ構造をたどっても，自身にたどり着くことはなく，また，別のパスを介して同じ要素にたどり着くこともない．逆向きリンクは，必ずしも存在しなくても良い．加えて，木構造の更新も，上記の性質を守る操作のみが提供される．

```
class MyNode extends DBinTreeNode{
    int val;
    void init(int val) { this.val = val; }
    synchronized void insert(int val) {
      if(val < this.val) {
         if(has(Left))  /* Left への insert(val) 呼出し */
             d_insert(Left, val);
         else           /* Left 節生成 + init(val) による初期化 */
             make(Left, val);
      } else { ...}
    }
    ...
    /* 以下は，処理系による自動生成 */
    void make(FowardLabel label, int val); /* init() 相当 */
    void d_insert(Label label ,int val);  /* insert() 相当 */
    ...
----
利用者コード
    DBinTree<MyNode> tree = DBinTree<MyNode>.make(val);
    tree.d_insert(val); ....
```

図 5.2: Tree の利用例 (イメージ)

```
class MyNode2 extends DBinTreeNode{
    class Mode extends Phase { Removable, AddOnly, NoChange }
    Mode mode = AddOnly;
    void insert() {
        assert(!mode.is(NoChange));
        /* 先ほどと同様のプログラム */
    }
    void remove() {
        assert(mode.is(Removable));
        /* 先ほどと同様のプログラム */
    }
}
```

図 5.3: Tree への局面記述追加 (イメージ)

2 分木クラスを利用したプログラムサンプルは，図5.2の通りである．`d_insert(Left, args)` などが隣接要素セルへのメソッド呼出しで，`make(Left, args)` などが隣接要素セルの作成である．また，`this.has(Left)` が隣接要素セルの有無を示す．

5.3.3 要素セルの効率的実装に向けて

ある要素オブジェクトが複数のホストから頻繁に呼出される場合，データを単一ホストに配置していたのでは遠隔メッセージ呼出しが頻発する．このため，1) プログラマは要素セルの時間的性質の変化を局面を利用して記述し，2) 局面解析により局面毎のセル内のフィールドアクセスを解析，一貫性のあるキャッ

シュの実現を行う．本節では，1) について紹介する．

図 5.2 に局面記述を付加されたプログラムのイメージが図 5.3 である．この際，局面と見なされるのは

- Left，Right の有無: つまり make(Left)，detach(Left) などで自動的に局面が変化する．
- mode の導入: 例えば， remove() や insert() の呼ばれる時期が特定されている場合，プログラマは局面を導入，各局面で呼ばれるメソッドを制限することができる．

このように，オブジェクトの重要な性質を局面として切出してもらうことで，処理系は局面毎のオブジェクトの性質を解析し，局面毎に特化した最適化を施すことができる．例えば，AddOnly，NoChange といった局面では Left，Right への参照は変化しないといえ，これに基づいて Left，Right への参照や，子供オブジェクト自身のキャッシュを検討することができる．

5.4 キャッシュ付プロキシの実現法

5.4.1 はじめに

キャッシュの実現は，基本的には 4 章の排他制御緩和と通じる話題が多い．つまり，局面毎にアクセス傾向を調べ，変化しないものについて最適化を行う．これは，4.7 節で述べたのと同様の問題点が存在する事を意味する．問題点として存在するのは，

- 複合的オブジェクトの扱い
- 参照解析の精度の問題
- アクセスに関するメソッド呼出し解析の必要性

である．特に，ノード間通信は大きなペナルティを引き起こすため，解析精度を高める必要がある．

上の二つの問題への対応として，5.3 節の分散コレクションモデルが存在する．つまり，要素セルが複合的オブジェクトに相当し，要素セル間の直接参照が遮断される．これにより，要素セル単位でのキャッシュが可能となる．加え

て，ライブラリによって提供される要素セル間の参照は，順序関係に関する制約が導入されることで，インスタンスを意識した解析を行いやすくなっている．

最後の問題については，5.5 節のアクセス解析手法によって解決を行う．

5.4.2 概略

プログラマの負担を軽くするためにも，プロキシの持つキャッシュは，正しい値を保持するよう管理されなくてはいけない．プロキシ側でキャッシュを利用して計算を行う際の基本的な方針は以下の通りである．

- 更新 (局面更新を含む) を行う計算は，本体で行う．
- キャッシュは各局面内で更新されない変数やオブジェクトを保持する．
- プロキシでは，キャッシュへのアクセスで済む計算のローカル実行を許す．加えて ロック操作も更新と見なさず，ロック操作を伴うメソッドのプロキシでの実行も許可する．
  一方で，キャッシュ利用中は，利用中のキャッシュが無効になるような局面更新はブロックする．
- オブジェクト本体は，局面更新操作を行う場合，各プロキシに古いキャッシュの廃棄を指示，その完了を待って更新操作を完了する．当該局面を利用した計算がプロキシで行われている場合は，その終了を待ってキャッシュの無効化を行う．

本ルールは，`synchronized` ブロック内の計算が，オブジェクト本体とプロキシとで並行して実行することを許している．本来，Java のメモリモデル [22] におけるロックの基本ルールでは，排他制御区間内のメモリアクセスについては，ロック取得後に主メモリからの読み込みを行い，ロック開放前に主メモリに書き出しを行うというものである．上記実行はこれに対し，読み込みに関しては解析によって保証された値を，事前にキャッシュとして読み込むことを許すというものである．このため，等価な Java 実行の存在を容易に保証することができる．

一方で，上記ルールに基づいてプロキシでの実行の余地を増やすためには，1) 更新を伴わない変数だけを利用した計算区間を多く見つけることであり，加え

て，2) その計算を行うためにはどの局面更新をブロックする必要があるかを割り出す必要がある．

以降の小節では，まず局面解析ならびにアクセス解析から得られる情報のモデル化を行い (5.4.3 節)，その後，プロキシ利用のためのキャッシュ利用ルール (5.4.4 節) やアクセス傾向による命令の分類法 (5.4.5 節) について解説する．最後に，Code Versioning を利用したキャッシュ利用区間の拡大法などについて紹介を行う (5.4.6 節)．局面解析やアクセス解析手法の具体的なアルゴリズムなどについては，5.5 節にて解説する．

5.4.3 モデル

本解析で扱う局面は，セル代表オブジェクトの局面だけである．但し，代表オブジェクトの局面は単一とは限らず，複数の局面変数を保持しうる．局面に関するモデル化は，3.4.2 節ならびに 3.8.1 節に従う．

キャッシュの利用法を検討するためには，まずフィールドアクセスをモデル化し，また，メソッド呼出しの影響を考慮した解析が行われる必要がある．

まず，フィールドアクセスのモデル化であるが，今回解析対象とするのはセル内のフィールドアクセスのみで，セル間呼出しに伴うセル外でのアクセスは解析対象外とする．これは，5.3.1 節で述べたように，本分散モデルでは `synchronized` は要素セル内部のアクセスの一貫性のみを保証するからである．また，現在は，セル内解析ではインスタンスを区別せず，クラス単位の解析を行っている．つまり，`A` クラス `x` フィールドへの読み出しなどと取りまとめて解析を行う．セル内には代表オブジェクトは一つしか存在しないため，その局面変数の混同も起きない．

各命令によるフィールドアクセスのモデル化は，以下の 3 要素を重視する．

- Become を行う可能性
- 各フィールドへの読み出し／書き込み可能性
- セル外へのメソッド呼出し可能性

また，上記解析はメソッド呼出し命令についても行われる必要がある．特に 5.4.4 節で述べるように，排他制御ブロック内でのフィールドアクセスについては正確な解析が望まれる．アクセス情報に関するメソッド間解析は 5.5.4 節の

アルゴリズムによって求められる．解析は，メソッドの開始局面を意識して行い，どの局面でメソッド $func$ を呼出すと以下の副作用がおきる可能性があるか，$\cup\boldsymbol{S}$ の形で求める（$\boldsymbol{S}$ の定義は 3.8.1 節参照）．

- Become を起こす可能性のある，開始局面の集合: FB_{func}
- 書き込みを起こす可能性のある，開始局面の集合: FW_{func}
- フィールド $field$ への読み出しを起こす可能性のある，開始局面の集合: $FR_{func}(field)$
- セル外へのメソッド呼出しを起こす可能性のある開始局面の集合: FO_{func}

各局面毎の性質を求めているのは，呼出し局面によってフィールドアクセスの可能性が変化する場合に対応するためである．但し，$FB_{func} \neq \phi$ の状況では他の 3 種の情報は利用されないため，解析は行わない．

5.4.4 キャッシュ利用ルール

本節では，セル内計算に対するプロキシでの計算実行の基本ルール，及びオブジェクト本体とプロキシの連携方式を決定する．セル内の計算は全てセル代表オブジェクトのメソッドを通して実行される．一方で，局面解析ならびアクセス解析により，各メソッド中の命令 (メソッド呼出しを含む) の実行局面，フィールドアクセス内容が解析されている．本節では，上記情報をもとに，セル代表オブジェクトのメソッドのキャッシュ利用区間を決定する．

まず排他制御区間外でのフィールドアクセスへの対応について述べる．排他制御区間外で必要となるのは，メモリの書き込みが正しく本体オブジェクトに反映されることである．一方で，メモリの読み出しに関しては，特に指定がない限り[1] 従来読み込んだ値をつかっても良いと考えられる．このため，排他制御区間については，書き込み命令は必ず本体オブジェクトで行い，読み出し命令のみの間はプロキシ実行を許すこととする．但し，メソッド途中などでプロキシと本体間で計算を中断するのが面倒な場合は，そのメソッド $func$ の解析結果 FW_{func}, FB_{func} と可能なメソッド開始局面 $\boldsymbol{S}_{init}^{func}$ との積集合を調べ，局

[1]C 言語などでは volatile 宣言された変数へのアクセスは，主メモリアクセスが要請されると考えられうことも多い．但し，近年の多くの計算機では memory barrier 命令やその他同期命令が行われるまで，主メモリとの同期が保証されないアーキテクチャも多い．また，Java の volatile は，これとはさらに別の意味を持ち，[2] のような議論も起きている．

面遷移や書き込みがあると判断されたら本体メソッドを呼出すという選択をしてもよい.

一方で，排他制御区間内では読み書きの一貫性をとる必要があり，プロキシとオブジェクト本体での計算を協調されるためには課題も多い．議論に先立ち，まずメソッド内の各命令を主にセル内のフィールドアクセスから，以下の 4(5) 種類に分類する (分類法は，5.4.5 節).

- **Iimmutable**：セル内のフィールドアクセスを行わない，もしくは，定数化した変数 (ある局面，またその局面から遷移可能な全ての局面で値が変化しない変数) へのアクセスを行う.
- **Istable**：ある局面で変化しない変数にのみアクセス.
  他のセルへの影響から，さらに以下の 2 種類に分類される.
  - **Istable-IN**: セル間呼出しを伴わない
  - **Istable-OUT**: セル間呼出しを伴う
- **Imutable**：ある局面で変化する変数へのアクセスを伴う
- **Ibecome**：局面更新操作

排他制御緩和の時 (4 章) と異なるのは，Istable が 2 種類に別れていることである．Iimmutable はセル内のフィールドアクセスに関して排他制御は一切不要であり，キャッシュの利用に制限はない．一方，Istable はキャッシュを利用可能であるが，その際，他の局面への更新をブロックする必要がある．Istable が IN/OUT に分かれているのは，再試行可能な計算かどうか示すためである．Istable はセル内のフィールド更新を伴わないため，セル内で計算している限りは，いつでも中断・再試行が可能である．但し，Istable-OUT はセル間呼出しを含むため再試行は許されない．Imutable は，更新を伴う変数へのアクセスのため本体オブジェクトで実行され，また実行順序を変化させることも難しい．Ibecome は局面更新であり，実行中の Istable との排他実行が必要となる.

オブジェクト本体とプロキシでどのように排他制御を行うかを決めるのに先立ち，まず簡単なケーススタディを行う．図 5.4 の `foo()` をプロキシで実行しようとしたとする．この計算は，運良く then 節に分岐した場合，プロキシでの実行が可能である．一方で，else 節に分岐した場合は，本体での実行が必要とな

```
synchronzied void foo() {
    int val = ISTABLE0;
    if(IIMMUTABLE < val) {
        ISTABLE1;
    } else {
        IBECOME;
    }
}
```

図 5.4: セル内の実行モデルのためのケーススタディ

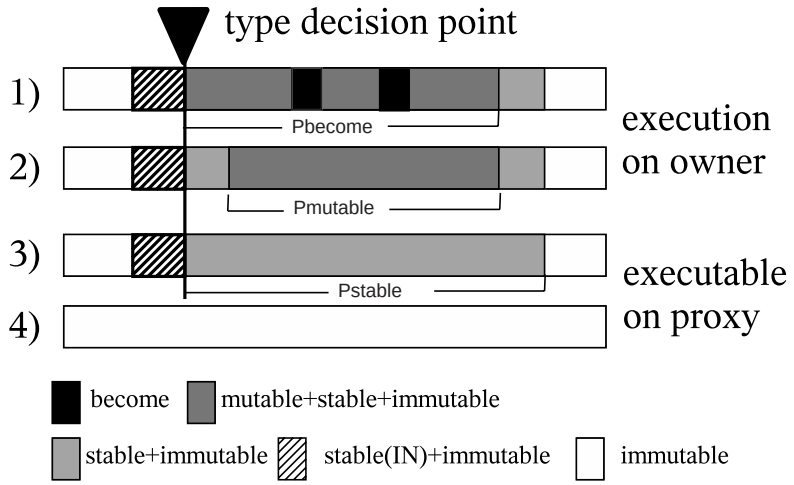

図 5.5: 排他制御区間の分類

る．今，複数のプロキシで同時に foo() の実行が行われており，分岐地点までプロキシで実行し，両方 else 節に分岐したとする．この場合，いずれを先に実行しようとしても，既に実行済みの ISTABLE0 が問題となる．局面遷移によって，ISTABLE0 で読み出した値が無効となるためである．この際，ISTABLE が Istable-IN であれば該当部分を再実行という選択肢もあるが，隣接セル間呼出し (Istable-OUT) の場合はそういうわけにもいかない．つまり，ISTABLE0 がセル間呼出しの場合は，この時点から他の foo() と排他的に実行する必要があったといえる．

以上の議論に基づき，実行区間を図 5.5 のように分類することとする．図 5.6 がオーバラップ実行例である．図中の各区間の意味は以下の通りである．

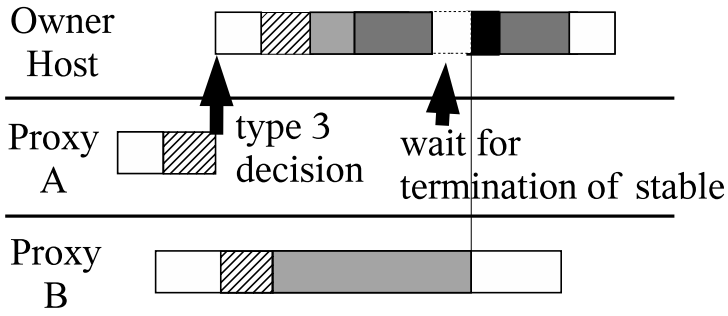

図 5.6: 並列実行イメージ

- type 0: Ibecome を含む可能性がある計算．本体実行．
- type 1: Imutable を含むが，Ibecome を含まない計算．本体実行．
- type 2: Istable は含むが，Imutable, Ibecome は含まない計算．プロキシ実行可．
- type 3: Iimmutable しか含まない計算．プロキシ実行可．

Pbecome, Pmutable, Pstable, Pimmutable 区間の意味も，図中に示した通り，含まれる Ibecome ∼ Iimmutable 命令により決まる．また，type 0-2 の分類は，排他制御区間開始の後，最初の Ibecome, Imutable, Istable-OUT があった時点で決定されていなければならない．

また，各計算の排他制御ルールは，以下のとおりである．複数の局面に関する議論は，後程詳しくおこなう．

- 排他制御区間は Pmutable 及び Pbecome 区間同士で排他的に実行される．
- type 2 の計算で stable 区間実行中は，想定する局面からの遷移をブロックすべくフラグを立てる．
  type 決定ポイントまでは，計算に応じて順次フラグを増やしてもよいが，途中で become があったと分かった時点で再実行が必要となる．一方で，type 決定ポイントにおいては，後続する全ての Istable 計算で想定する局面について，遷移ブロックフラグを立てなくてはいけない．
- become 命令では，局面変更によって無効化されるべきプロキシキャッシュを無効化し，当該局面を仮定した stable 区間の実行終了を待つ，もしく

は，再実行を促した上で更新処理を完了する．

- プロキシからのキャッシュ要求については，局面更新中を除き随時実行可能．

この結果，あるメソッドが Iimmutable からだけなる計算や Istable-IN を実行している間は，完全にプロキシ実行が可能となる．

メソッド内を各実行区間に分類するにあたり，メソッドをまず CFG(Control Flow Graph) に分割し，その後，Iimmutable, Istable-IN/OUT, Imutable, Ibecome の 5 種類に分類された各命令 (分類法は 5.4.5 節) を，排他制御区間が最小になるように求めていく．基本的な分割方針は以下の通りである．

1. type 判断ポイントの決定：メソッドの初期ブロックから，他の Istable-OUT, Imutable, Ibecome を経ずに到達可能な，Istable-OUT, Imutable の集合を作成する．そのうち，Ibecome 自身，もしくは後続する Ibecome が存在する命令は，type 0 の判断ポイントである．それ以外の命令のうち，Imutable 命令，もしくは後続する Imutable 命令がある場合，type 1 の判断ポイントである．残ったものが type2 の判断ポイントとなる．
   type2 の決定ポイント $inst$ に後続する `Istable` 命令の集合を，$IS(inst)$ として以後表記する．type2 には，その決定ポイントがない (全て Istable-IN から構成される) 場合も存在する．

2. Pbecome 区間: type0 の判断ポイントに後続し，それ以降に Ibecome, Imutable がないものが Pbecome 区間の終端．

3. Pmutable 区間の決定:
   type 1 の判断ポイントに後続し，それ以降に Imutable がない命令が Pmutable 区間の終端

5.4.5 局面に基づいたアクセス分類

本節では，まず局面ならびにアクセス解析結果をもとにした各命令分類の方法を紹介し，その後 stable 区間の想定する局面について議論する．

まず IBecome 命令の分類であるが，対象命令が become 命令である場合は単純に IBecome と判断する．対象がメソッド呼出し命令であった場合は，実行可

能局面 $\boldsymbol{S}_{call}$ とアクセス解析結果 FB_{func} を利用する．つまり， $\boldsymbol{S}_{call}$ と FB_{func} の積集合が空でなければ Become の可能性あり (IBecome) と判断する．

次に，IBecome でないものに対して Imutable の判定を行うが，この際，当該命令が書き込みを行う場合は当然 Imutable と判定される．書き込み命令そのものに加え，メソッド $func$ の呼出しの中で $FW_{func} \cap \boldsymbol{S}_{call} \neq \phi$ のものは Imutable と判定する．一方で，書き込みを行わなくても，読み出し対象フィールドに同時期に他からの書き込み可能性があれば Imutable である．まず，読み出し対象フィールドを確定する必要があるが，メソッド呼出しの場合は $FR(field) \cap \boldsymbol{S}_{call} \neq \phi$ となる $field$ を見つけることになる．その上で，実行可能局面 $\boldsymbol{S}_{inst}$ と $field$ への書き込み局面の集合 $Write(field) = \bigcup_{inst \in WI(field)} \boldsymbol{S}_{inst}$ とに交わりが存在するか確認を行うことになる．つまり，もし空でない積集合が存在すれば，その局面において読み出しと書き込みが並行して実行される可能性があり，Imutable と判定される．

最後に残ったのは，Istable と Iimmutable の判定である．この判定を行うためには，現在局面から遷移可能な局面 $\boldsymbol{S}_{fix}$ を求める必要がある．3.8.1 節に基づいて $\boldsymbol{S}_{fix}$ を求め，Istable と同様の手続きを $\boldsymbol{S}_{inst}$ の代りに $\boldsymbol{S}_{fix}$ に対して行うことで，Iimmutable の判定を行うことができる．この判定法は，各 $\boldsymbol{S}_{inst}$ に対して $\boldsymbol{S}_{fix}$ を求める計算が必要となるが，連続した Pstable 区間においては実行可能局面 $\boldsymbol{S}_{inst}$ は共通であるため，この計算は Ibecome 前後でのみ計算すれば十分なものである．

最後になるが，type 2 の stable 区間実施の際に指定する局面遷移ブロック操作について．単純に考えて，stable 区間実行中は，開始時の局面を全て固定すれば安全な実行を行うことができる．つまり，実行中は一切の局面遷移をブロックすれば良いのである．但し，それでは当該 stable 区間とは関係ない become 操作までブロックされるし，無駄にキャッシュ要求を行うことになりかねない．例えば，メソッド `int foo() { return x + y;}` の例を考える．局面変数は `L1`，`L2` があり，`L1/P1`，`L2/ALL` で `x`，`y` は更新がなく，`L1/ALL`，`L2/P2` で `z`，`w` に更新がなかったとする．仮に，オブジェクトの局面が `L1/P1`，`L2/P2` であったとしても，`foo()` の実行のために， L2 の遷移をブロックするのは無駄である．つまり，当該 Pstable 区間が利用するフィールドが更新されないよう局面が設定できれば十分といえる．このような局面を求めるには，Pstable 区間で読み込まれる全 $field$ の $R(field)$ を含み，$W(field)$ を含まないような実行局面を

```
synchronzied int foo() {
    return this.val;
}
synchronzied void setVal(int val) {
    assert(state1.is(SetupMode));
    this.val = val;
}
```

図 5.7: ケーススタディ (Code Versioning)

求めれば十分であり，これは don't care を含む論理式の簡単化とおなじ手続きで求めることができる．

実際のプロキシの運用に当っては，キャッシュの戦略はいろんな方針が考えられる．これについては，5.6.2 節で議論する．

5.4.6 Code Versioning

5.4.4 節においては，メソッド中の命令をアクセス傾向に応じて分類し，type 2, 3 に対してプロキシ実行を許した．本節では，局面をより積極的に利用することで type 2, 3 実行を増やす方策について紹介する．つまり，メソッド開始時の局面に応じたメソッドの Code Versioning である．

例えば，図5.7のようなメソッド `foo()` があったとする．本メソッドは，`state` という局面変数を持ち，`state/SetupMode` の時のみ `val` の更新がある．つまり，`foo()` メソッドは，`DebugMode`，`SetupMode` であれば本体実行を必要とするが，両方該当しない場合はプロキシにおけるローカル実行が可能である．しかしながら，5.4.4 節の分類に基づけば，`foo()` メソッド内の `val` アクセスは実行可能局面が `state/ALL` と判断される以上 Imutable と判断せざるを得ない．但し，`foo()` をメソッド開始時の局面によって Code Versioning を行い，`SetupMode` 以外用のメソッドを準備できれば，当該メソッドでは stable 実行が可能となる．但し，全ての初期局面でプログラムを versioning していれば無駄にコードが増えるだけである．

本節で紹介する方法は，メソッド実行中 Ibecome や Imutable を引き起こすような開始局面を判定し，分離するというものである．幸い，局面更新ならびに書き込みを起こす可能性があるメソッドの開始局面は，既に $FB_{func} = \cup \boldsymbol{S}$ の形

で求められている (FW_{func} も同様)．Imutable を否定するためには，書き込みがないだけでなく読み出し対象への更新も否定する必要がある．これについても，メソッド内で $field$ に対し読み込む可能性のある局面の集合 $FR_{func}(field)$ と，$field$ への更新が行われる局面の集合 $FW(field)$ が求まっており，この二つの交わる局面の集合を否定すれば良い．以上で求められた局面の集合と，それ以外の局面を分離することで，type 2, 3 のメソッドを増加させることができる．

Code Versioning によって特化されたメソッドの実行局面は，必ずしも再計算する必要はない．つまり，3 章のメソッド内解析では実行可能局面は局面 P_i においてメソッドが開始される可能性 (真偽値) $S^{func}_{init}(P_i)$ と各メソッド呼出しによる局面遷移の可能性 (真偽値) $P_i \rightarrow^{func} P_j$ を論理変数とした論理式の形で表現されており，局面解析の結果求まった $P_i \rightarrow^{func} P_j$ とバージョニングされた開始局面を代入することで各開始局面の確定は可能である．後は 5.4.4 節の解析手順に従って，特化された各メソッドの命令の分類と区間解析を行う．

5.5 解析アルゴリズム

5.5.1 概要

局面解析ならびに，アクセス解析は基本的に以下のステップで行われる．

1. 各メソッド毎の局所解析 (局面解析，メソッド呼出し関係グラフの準備)
2. 大域解析に向けてのメソッド呼出し関係グラフ作成
3. 局面の大域解析 (実行可能局面の確定，可能局面遷移の確定)
4. 各メソッドのアクセス解析
5. 各メソッドの実行ブロック分割ならびに，Code Versioning 実行

以下では，まずメソッド呼出し関係グラフの作成について述べ，その後，局面解析ならびにメソッド内アクセス情報の導出，アクセス情報に関するメソッド間解析の順で解説を行う．

```
class LinkedCell {
   // Forward Direction = { Next};
   // Backward = Prev;

   void funcA() { /* EffectA */
      objX.funcB();
      d_funcC(Next);
   }
   void funcBar() { /* EffectB */ }

   void funcC() { /* EffectC */
      d_funcD(Prev);
   }
   void funcD() { /* EffectD */ }
}
```

図 5.8: セル間呼出し関係 (例：イメージ)

5.5.2 メソッド呼出し関係グラフ

本分散モデルにおいては，メソッド呼出しは 2 種類存在する．一つはセル内のメソッド呼出しであり，もう一つはセル間呼出しである．セル内の解析においてはインスタンスの差異は解析せず，クラス単位の解析が行われる．一方で，セル間解析においては，参照関係がモデル化された再帰データ構造が解析対象となる．現在のところ，利用可能なデータ構造は線形リストや Tree 構造と言った順序関係の分かりやすいデータ構造に限られ，3 次元格子など合流のあるデータ構造は許していない．但し，リンクに対して逆向きリンクも存在するため，循環のあるデータ構造は存在する．本節では，メソッド呼出し関係，特にセル間呼出しの関係の解析について説明する．

さて，メソッド呼出し関係グラフを作成するのは，あるメソッド `foo()` を実行した際，`foo()` 内で呼出される `bar()` などのメソッドの影響を求めるためである．まず，簡単なサンプルプログラム (図 5.8) をもとに，問題の把握を行う．

本解析では，セル内メソッド呼出しの場合，インスタンスを意識せずに解析を行う．つまり，`objX.funcB()` 呼出しについては，単に `LinkedCell#funcB()` の呼出しとして解析が行われる．一方で，`d_funcC(Next)` のようなセル間呼出しの場合，EffectC は隣接セル上の副作用として解析することで，自セル内のフィールドアクセスの正確な解析を目指す．但し，呼び戻された `d_funcD(Prev)` によって自セル内に EffectD が起こる可能性もある．本モデルでは，`funcD()` は `funcA()` のメソッド呼出しの中で起きたと考えるため，`funcA()` のプロキシ実行には `funcD()` 内の影響を含めて解析する必要がある．つまり，メソッド呼出し関係解析の目的は，セル内については再帰的に呼出す可能性のあるメソッ

ドを見つけ出し，セル間呼出しについては，コールバックのあるメソッドを探し出すことにある．

解析の手順は以下の通りである．

- Step1: セル代表オブジェクトの public メソッドに対し，自身およびそこからの再帰的セル内メソッド呼出しによって，セル間メソッド呼出しが行われるか探し，見つけたセル間呼出しについて対象メソッドとラベル (`Forward/Backward`) を登録する．

- Step2: セル間呼出しされているメソッド (仮にラベル`L0`，メソッド `func0`) に対して，`func0` の呼出すセル間メソッドの中に逆向きラベルのものがないか調べ，もしあれば (仮に `func1` とする)，`d_func1(L0)` 呼出しは自セルの `func1` 呼出しを含むと見なす．

- Step3: 追加された情報を追加して，さらに Step1 及び 2 を繰り返し，見つからなくなるまで繰り返す．

局面解析の大域解析部においては，局面を持つオブジェクト (ここではセル代表オブジェクト) が自身のメソッドを呼出すかどうか判定する必要があるが，これに関しても上記解析をつかうことで求めることができる．というのも，セル間メソッド呼出しを介して自身のオブジェクトを呼び戻したのが上記に求めた例だからである．もう一方の，セル内メソッド呼出しのみによって自セル呼出しを行うものについても求めることは簡単である．つまり，セル内に代表オブジェクトと同一クラスのインスタンスは他に存在しないため，セル内メソッド呼出しを再帰的に調べることで自クラスへの呼出しを見つければよい．

5.5.3 局面解析ならび取得情報

局面遷移解析は，基本的に 3 章と同じである．一つの局面変数に対する解析をそれぞれの局面変数に対して行うだけである．つまり，各局面変数について，各命令の実行可能局面，可能な局面遷移を求めることになる．解析対象となるのは，局面変数にアクセスしうる代表オブジェクトのメソッドである．また，大域解析の際に利用される自オブジェクトへの呼び戻し可能性については，前小節で求めた情報が利用される．

一方で，局面解析の結果取得される情報は様々である．

```
synchronized void foo() {
   if(state.is(P0) || state.is(P1)) {
       this.var++;
       bar();
   }
   ....
}
```

図 5.9: 初期局面と命令の実行可能局面の関係 (プログラム例)

- Become 操作について,
  - どの局面変数が，どの局面からどの局面に遷移したか？
  - 当該命令は，各局面変数のどの局面において実行されうるか？

  以上情報がまとめて，5.4.3 節の局面遷移情報となる．
- セル内の各命令について,
  - 各局面変数がどの局面の時に実行されうるか？
  - メソッド開始時の局面がどの局面の時に実行されうるか？

  フィールドアクセスに関する実行可能局面が，5.4.3 節の Read/Write 情報としてまとめられる．また，開始局面に基づいたアクセス情報が，後続のアクセス解析で用いられることになる．

局面解析自体は，代表オブジェクトのメソッドに対してしか行っていないが，他の要素セル内のオブジェクトのメソッド実行局面も，その呼出し局面の総和として簡単に求めることができる．上記解析情報のうち，Become 操作ならびに命令の実行局面については，5.4.3 節で述べた通りである．基本的には，それぞれの局面変数について各命令の実行可能局面を求めただけのものである．よって，以降ではメソッド開始局面による各命令の実行可能性に関して説明を行う．

局面解析では，局所メソッド解析の際，局面 P_i においてメソッドが開始される可能性 (真偽値) $S_{init}^{func}(P_i)$ と各メソッド呼出しによる局面遷移の可能性 (真偽値) $P_i \rightarrow^{func} P_j$ を仮定し，上記の値を論理変数としたまま解析が行われる．大域解析により上記変数の値が定まることになるが，そのうち $P_i \rightarrow^{func} P_j$ だけを求まった値で固定すれば，初期局面と各命令の実行可能局面の関係式となる．つまり，3.8.3 節の応用例である．

例えば，図 5.9 のプログラムの例を考える．今，ある局面変数 L に対して P_i が初期局面である可能性（真偽値）を，$C_i = S^{func}_{init}(L/P_i)$ で表す．このとき，var++実行時の局面可能性 S は，$S = \{P_0/C_0, P_1/C_1, P_2/false, \cdots\}$ となる．つまり，何らかの局面で var++ が実行されるのは，初期局面が $L/\{P_0, P_1\}$ の時と言える．また，同様に bar() の呼出し局面も同じ形で求められる．これらの情報は，3.5 節の局所解析結果に対し，確定した $\rightarrow^*, \rightarrow^{func}$ 情報を埋め込むことで求めることができる．

同様の情報を別の局面変数に対しても求めることで，foo() を呼出したときに this.var++が実行されるような初期局面を，実行可能局面

$$\boldsymbol{S} = \{L_1/S_1, \cdots, L_n/S_n\}$$

と同じ形で求めることができる．つまり，メソッド内のフィールドアクセス命令に関して上記情報を集めることで，foo() メソッド (呼出したメソッドは除く) 内でフィールドの読み書き，局面更新，セル間メソッド呼出しが行われるためのメソッド開始局面を $\cup\boldsymbol{S}$ の形で求めることが可能である．それぞれを，$FR^{local}_{\texttt{foo}}, FW^{local}_{\texttt{foo}}, FB^{local}_{\texttt{foo}}, FO^{local}_{\texttt{foo}}$ の形で表記する．

一方で，メソッド呼出しに関しては，当該メソッドが排他制御区間内で，局面遷移も起こらない，要は局面分岐／制限しかない条件下では，以下の性質が成り立つ．つまり，foo() 中で bar() を呼べるための局面の制限を S^{bar}_{foo} で表記すると，foo() が初期局面 S で呼ばれたときのメモリアクセスは，bar() が初期局面 $S \cap S^{\texttt{bar}}_{\texttt{foo}}$ で呼ばれた時のメモリアクセスを含むという性質である．ここでは，$S^{bar}_{foo} = \{P_0 : true, P_1 : true, P_2 : false, \cdots\}$ である．実は，$\boldsymbol{S}^{\texttt{bar}}_{\texttt{foo}}$ とは局面変数 L に対する bar() 呼出しの実行可能局面の事である．同様のことが各局面変数についていえ，つまり，foo() が初期局面 $\boldsymbol{S}$ で呼ばれた時の影響は，bar() が初期局面 $\boldsymbol{S} \cap \boldsymbol{S}^{\texttt{bar}}_{\texttt{foo}}$ で呼ばれた時の影響を含むといえる．

5.5.4 メソッド間アクセス解析

本節の目的は，前節の局面解析情報やフィールドアクセスに関する議論をもとに，アクセス解析，特にメソッド呼出しに伴うフィールドアクセスの解析を行うことである．つまり，5.4.3 節の $FR_{func}, FW_{func}, FB_{func}, FO_{func}$ を求めることである．

アクセス解析手法は，排他制御区間内と排他制御区間外で若干異なる．本節

では，5.4.4 節の分類において重要度の高い排他制御区間内の解析について，まず解説する．

今，任意の局面 $\boldsymbol{S}_{init}$ で排他制御区間内のメソッド $func$ を実行した場合の Become の可能性を $FB_{func}(\boldsymbol{S}_{init})$ と仮に表す．その際，前小節の議論に基づくと，

$$\begin{aligned} FB_f(\boldsymbol{S}_{init}) \ = (\ & FB_f^{local} \cap \boldsymbol{S}_{init} = \phi) \vee \\ & \bigvee_{g_i} FB_{g_i}(\boldsymbol{S}_f^{g_i} \cap \boldsymbol{S}_{init}) \end{aligned}$$

である．但し，g_i は f によって呼出されるメソッドであり，FB_f^{local} は f 内で Become を行うための初期局面の集合，$\boldsymbol{S}_f^{g_i}$ は f 内で g_i がよばれる際の局面制限の事である．前節で説明した通り，$FB_f^{local}, \boldsymbol{S}_f^{g_i}$ は局面解析結果から求めることができる．上の式の主張は，もし当該メソッド内で局面遷移が存在するなら $FB_{foo}(\boldsymbol{S}_{init}) = true$ であるし，当該メソッド内で局面遷移がないなら，$func_i$ の開始局面は $\boldsymbol{S}_f^{g_i} \cap \boldsymbol{S}_{init}$ であり，Become の可能性は $FB_{g_i}(\boldsymbol{S}_f^{g_i} \cap \boldsymbol{S}_{init})$ で求めることができるということである．ところで，$FB_{g_i}(\boldsymbol{S}_{call_i} \cap \boldsymbol{S}_{init})$ をさらに展開する場合，上記式に当てはめると，

$$\begin{aligned} FB_{g_i}(\boldsymbol{S}_{call_i} \cap \boldsymbol{S}_{init}) \ = (\ & FB_{g_i}^{local} \cap \boldsymbol{S}_f^{g_i} \cap \boldsymbol{S}_{init} = \phi) \vee \\ & \bigvee_{h_j} FB_{h_j}(\boldsymbol{S}_{g_i}^{h_j} \cap \boldsymbol{S}_f^{g_i} \cap \boldsymbol{S}_{init}) \end{aligned}$$

ということになる．ここで，$FB_{g_i}^{local} \cap \boldsymbol{S}_f^{g_i}$ ならびに $\boldsymbol{S}_{g_i}^{h_j} \cap \boldsymbol{S}_f^{g_i}$ は変数項は含まれない．つまり，メソッド呼出しの影響を解析するには，単に局面制限情報を付加しながら再帰的にメソッドの影響を加えていけば良い．加えて，$FB_{func}(\boldsymbol{S}_1) \Rightarrow FB_{func}(\boldsymbol{S}_1 \cap \boldsymbol{S}_2)$ であるため，自己再帰的に呼ばれるメソッドの解析は省略することができる．

結果，再帰的に呼出されるメソッド $func_i$ への局面制限を $\boldsymbol{S}_{func_i}^*$ で表すと，以下のような関係式が求まる．

$$\begin{aligned} FB_f(\boldsymbol{S}_{init}) &= \bigvee_{func_i}(FB_{func_i}^{local} \cap \boldsymbol{S}_{func_i}^* \cap \boldsymbol{S}_{init} = \phi) \\ &= ((\bigcup_{func_i}(FB_{func_i}^{local} \cap \boldsymbol{S}_{func_i}^*)) \cap \boldsymbol{S}_{init} = \phi) \end{aligned}$$

つまり，$FB_{func} = \bigcup_{func_i}(FB_{func_i}^{local} \cap \boldsymbol{S}_{func_i}^*)$ となり，5.4.3 節で述べたように $\bigcup \boldsymbol{S}$ の形で求まったといえる．同様の手続きで FB_{func} に含まれない (つまり局面遷移が否定された) 局面の集合に対しては，$FW_{func}, FR_{func}(field), FO_{func}$ を求めることができる．

一方で，排他制御区間外ではすこし話が面倒である．つまり，f が g_i を呼出す局面を $\boldsymbol{S}_f^{g_i} \cap \boldsymbol{S}_{init}$ などと表記できず，常に他のメソッドによる局面遷移の可

能性 $\rightarrow^*$ を考慮して解析を行う必要がある．一方で，排他制御区間外では解析精度への要求も高くないため，以下のような近似を行うこととする．

$$FB_f(\boldsymbol{S}_{init}) = (\quad FB_f^{local} \cap \boldsymbol{S}_{init} = \phi) \vee \bigcup_{g_i} FB_{g_i}(\boldsymbol{S}_f^{g_i})$$

$\boldsymbol{S}_f^{g_i}$ は，f 内で g_i を呼ぶ際の実行可能局面である．つまり，メソッド呼出しについては $\boldsymbol{S}_{init}$ の影響を加味せず，全ての実行可能局面での解析結果を使うことにする．これによって，各初期局面に対し $\rightarrow^*$ の影響を再計算することを避けることができる．解析の停止性についても，問題はない．

5.6 議論

5.6.1 局面解析ならびにアクセス解析の精度評価

本節では，実アプリケーション例に対して局面解析ならびにアクセス解析について，その有効性や問題点の検証を行う．また，今回のキャッシュ利用方式について，運用面も含めた課題の検討を次節にて行う．アプリケーション例としては，N 体問題プログラム (図 5.10) の例を通し，各種バリエーションを加えながら問題点の検討を行う．なお，現在，複数局面やメソッド間アクセス解析を実装したシステムは完成していないため，以下の議論は手動解析に基づくものである．

本プログラムでは計算の主体となる `Node` は 8 分木データ構造であり，その主なメソッドは，木の構築局面 `Const` で使われる `insert()`，重心計算局面 `Mass` で使われる `getCenter()`，主要な局面である力の計算局面で利用される `calAcc()` が存在する．以上のメソッドに加えて，`make(`*Direction*`)`（コンストラクタ相当）が `public` メソッドとして存在する．

局面変数には，他にも `Leaf` であるかどうか，8 分木の各子供の有無が存在する．上記プログラムでは，8 分木の各ラベルは変数 `LX` として表記しているが，本来のモデルでは，ラベルは即値としてしか用いることはできず，上記コードは単純に約 8 倍に増加する．今後の検討が必要である．

セル間呼出し解析: 本プログラムでは，`insert()`，`getCenter()`，`calAcc()` といったメソッドが，各局面においてセル間呼出しに利用されている．但し，本プログラムの例では `Parent` 呼出しを利用したケースは存在しないため，セル間呼出しは副作用なしとして解析を行うことができる．3.7 節の評価では，参照

```
class Node {
   // Forward Direction =  L000, ..., L111; 以下, 表記上 LX と記述
   // Backward = Parent
    class Mode extends Phase { Const, Mass, Calc }
    Mode mode = Const;
    class Leaf extends Phase { True, False }
    Leaf leaf = True;
    synchronzied void insert(int i, double myX, myY, myZ, double myMass) {
        assert(mode.is(Const));
        if(leaf.is(True)) {
            leaf.becomes(False);
            insert2(cx, cy, cz, mass);
        }
        insert2(cx, cy, cz, mass);
    }
    synchronized void insert2(...) {
        Label LX = /* L000 - L111 のどれか*/;
        if(has(LX)) d_insert(LX, ...)
        else make(LX, ....);
    }
    synchronzied CenterInfo getCenter() {
        assert(mode.is(Const)); mode.become(Mass);
        if(leaf.is(False)) {
             for(/* 全ラベル: LX について*/)
                 if(hasChild(LX)) { CenterInfo tmp = d_getCenter(LX)); }
        }
        mode.become(Calc); return new CenterInfo(..);
    }
    synchronized AccInfo calAcc(Particle target) {
         assert(mode.is(Calc));
         if(/* 十分近かったら*/) return new AccInfo(....);
         else {
             for(/* 全ラベル: LX について*/)
                 if(hasChild(LX)) { AccInfo tmp = d_calAcc(LX, ..); }
             return new AccInfo(...);
         }
    }
}
```

図 5.10: N 体問題プログラム (イメージ)

関係の順序関係が仮定できないため，呼出し先での become が自身への become かもしれないとして解析するしかなかったので，参照関係のモデルを導入した効果は大きいと言える．

さらに，仮に calAcc() 中で return new AccInfo() の代りに，親へのコールバックである d_addAcc(Parent, ...) というメソッドを呼出していたとする．この場合，d_calAcc(LX, ..) の呼出しは addAcc(..) を伴うと解析でき，局面遷移解析やアクセス解析に影響を与えると解析できる．この場合も，副作用の内容が明確であり，解析精度の点で問題はない．

局面遷移解析: 局面遷移解析では，局面変数 mode ならびに各子供の有無

が解析対象となる．まず，mode の局面遷移についてであるが，assert() によって局面制限がかかっており，mode/Const→^{s}Mass や mode/Mass→^{s}Calc が解析できる．次に LX，leaf との相関であるが，特になにもないため，結果，[mode/Const→^{s}Mass] : {mode/Const, leaf/ALL, LX_i/ALL} といった解析を得る．このように本来相関がない局面間では，L/ALL という結果が多く見られる．

次に，LX の解析について．各 LX の解析では，Empty→^{s}Full が解析され，加えて，mode/Const 局面においてのみ実行されることが分かる．同様の情報は，leaf/True→^{s}False についても解析できる．話を戻すが，８分木の子供を扱うためには，将来的に記述面の向上のためにも，ラベル値を LX のような変数で扱えるようにすべきであると考える．但し，そのためには，LX のスコープ単位で解析を行うなどの対策が必要となる．

実行可能局面: insert() メソッドでは仕事の大部分を insert2() メソッドで行っている．局面解析においては，insert2() の開始局面は insert() の呼出し局面と同じと解析されるため，insert2() における各種副作用は，Const 局面におけるものとして解析される．

アクセスに関するメソッド間解析: 上記プログラム例では，5.5.4 節で紹介したアクセス解析は利用の機会がない．というのも，実行可能局面解析の結果，複数の局面で利用されるメソッドがほとんど存在しないためである．一方で，仮に calc() 冒頭で以下の様なメソッドを呼んでいた場合には効果が大きい．

```
void becomeCalcIfNot() { if(!mode.is(Calc)) mode.become(Calc); }
```

この場合，このメソッドを Calc 局面で実行した場合は become がなく，呼出し側のアクセス解析において，その事実を利用することができる．

要素セル内オブジェクトの解析: 実は，このプログラムでは，CenterInfo，AccInfo というデータ構造は，全てのフィールドは final，つまり更新がないように作られている．つまり，全てキャッシュ可能という当たり前の結果が出る．一方で，例えば，クラスが HashMap などのデータ構造を有しており，Const 局面でのみ更新を行うような場合，その他の局面では HashMap をキャッシュすることができると解析可能である．

一方で，問題点も存在する．もし，calAcc() 内で一時データ格納用の AccInfo のインスタンスを作成し，その更新を行ったとする．この場合，現在の解析では AccInfo は更新を伴うとして，AccInfo へのアクセスが Imutable と判定される．この種のフィールドは，メソッド内に限定されたオブジェクトの利用は

本来 Escape Analysis などで解析可能であり，今後の課題として検討が必要である．

5.6.2 区間分類ならびにキャッシュ運用の考察

本節では，前節の解析結果に基づいたキャッシュ利用プロキシの作成を通じて，その有効性や問題点について考察を行う．

図 5.10 の様に複数の局面変数がある場合，プロキシは常に局面変数すべてキャッシュするわけでない．一方で，ある局面変数をキャッシュする際は，その局面値で更新されない変数をキャッシュを行う．

まず，Const 局面で利用される insert() について考察する．insert() でよく実行されるのは mode.is(Const)，has(LX)，leaf.is(True) である．局面はキャッシュ可能であり，同様に，子供木への参照もキャッシュ可能である．一方で，局面更新があれば，上記キャッシュ内容は当然無効化される．また，キャッシュ利用区間についても，本メソッドでは，become/make 操作以外は全て stable/immutable である．また，その間，セル間呼出しも存在しない．このため，leaf/True への分岐，もしくは insert2() 中の LX/hasNot への分岐がない限り，プロキシでの実行が可能である．但し，問題として残るのは，いつ stable な局面変数のキャッシュをいつ行うかである．

本メソッドに関しては，キャッシュ方針は様々考えられる．例えば，最初にオブジェクトにアクセスする時点で，全局面変数を想定したキャッシュを取得することもできる．但し，この方法は必ずしも得策とは言えない．というのも，leaf/True はすぐに False に代り，プロキシのキャッシュの無効化が必要となるからである．だからといって，immutable だけをキャッシュしていたのでは，mode/{Const，Mass} をキャッシュできず，insert() が全て本体実行することになってしまう．このため，どの局面変数を重視してキャッシュを行うか，良く検討が必要である．例えば，初期呼出しはオブジェクト本体呼出しを行い，その際，利用した stable な局面変数を immutable な局面変数と一緒に持ち帰るとか，あるいは，プロファイル情報などから問い合わせの多いメソッドや局面を探して積極的にキャッシュするなどの方策が考えられる，今後，各種アプリケーションを通して実評価を行うことで判断したいと考える．

一方で，主計算局面で実行される calAcc() はとても良い性質を持っている．というのも，アクセス対象が全て immutable と判断できるためである．例え

ば，局面変数についても，calAcc() の実行可能局面が，mode/Calc，leaf/ALL，Li/ALL であるが，この局面では leaf，Li に関する局面遷移も否定でき，calc において stable であった局面変数も immutable と判断できる．

5.7 まとめ

本節では，共有メモリ型プログラムの分散環境への移植の効率化，ならびにその後の自動最適化技術を目指して，分散コレクションライブラリと，その要素セルのキャッシュ手法を提案した．

効率化にとって問題なのは，共有メモリ型プログラムにおいて広く共有されていたオブジェクトを，分散環境に移植する場合である．そのようなオブジェクトの計算を単一ホストのみで実行した場合，容易にボトルネックが想像される．理想的には，データが並行して読み出される場合はキャッシュを配置し，逆に更新を伴った計算が行われる場合は本体実行を行うのが良い．

本研究では，局面変数をもちいることで，しばらく変化しない変数を解析し，キャッシュとして利用することとした．将来的には，プログラマが局面記述したプログラムから，自動的にキャッシュを利用したプロキシが生成されることを意図している．本研究においては，キャッシュ実現のために適した環境として分散コレクションライブラリを利用した計算環境を提案し，また，その上での局面解析結果を利用したキャッシュ付プロキシの実現方式を提案した．加えて，アクセス情報に関するメソッド呼出し解析手法による解析精度の向上，最適化領域を増加させるための Code Versioning 手法を提案した．

上記解析手法の有効性は，実際のアプリケーションを通して行い，最適化のための情報の取得に成功したと考えている．一方で，実際に解析情報を利用した最適化を行ううえでの，運用面の問題点も検討した．残念ながら，本研究は，まだ提案段階であり，解析系を含めたシステムの完成はまだであるが，解析面での理論的足掛かりはできたと考えている．今後，実システム上での評価を通して，本手法の有効性を議論したいと考えている．

第 6 章

共有メモリ計算機向けオブジェクトレイアウト

6.1 はじめに

CPU とメモリとの間の速度差を緩和するためにキャッシュメモリを活用する現在の計算機においては，キャッシュはプログラムの実行性能に主要な役割を果たしており，共有メモリ型並列計算機における，無効化によるキャッシュミス（コヒーレンスミス）が性能に与える影響は大きい．このため，十分な排他制御が行われたプログラムであっても，メモリコンテンションのため複数プロセッサをつかった実行が単体プロセッサ利用時より低速になることすらある．このため，プログラマに明示的なメモリ操作を許している C などの言語では，キャッシュを意識したオブジェクト[1] 配置をしばしばプログラマ自身が手作業で行うこともある．しかし，明示的なメモリ管理を伴うこの作業は煩雑であり，バグの原因にもなりやすい．一方，自動メモリ管理機構を備えた Java などの言語においては，プログラマはメモリ管理の負担から解放されるが，逆に，キャッシュを意識したオブジェクト配置を行うことができない．

キャッシュミスを抑えるアプローチとして，本研究では頻繁にアクセスされる変数群を近くにまとめて配置する一方で，頻繁に書き込みが行われる変数に対しては別の領域に配置することとした．これにより capacity miss と coherence miss をともに抑え，実行効率の向上を目指す．本研究では，まずオブジェクト内のレイアウト方法として，固定レイアウト法とレイアウト切替法を提案している．固定レイアウト法では，各クラスに対して固定されたレイアウトを利用するのに対し，レイアウト切替法ではプログラム局面毎に特化されたレイアウトを準備し，実行時にレイアウト変更を行う．いずれも変数アクセス情報はプ

[1]本章では，C の構造体変数や Java のインスタンスなど，1 つ以上の変数（フィールド）の集まりをオブジェクトと表記.

ロファイラなどによる取得を想定する.

加えて，オブジェクト間の配置に関しても最新の自動メモリ管理機構を考慮した配置方法を考察，評価を行った．対象として利用したのは既存の Java 処理系である Sun HotSpot VM[28] の自動メモリ管理機構である．深さ優先コピー方式やアクセス傾向別領域を導入したプロトタイプ上で，クラスを分類・分割配置の効果を評価した.

本章では，まず関連研究について 6.2 節で述べた後，オブジェクト内レイアウト法について固定レイアウト法 (6.3 節) とレイアウト切替法法 (6.4 節) を検討，評価 (6.5 節) を行う．加えて，オブジェクト間レイアウトならびに自動メモリ管理機構との統合について検討 (6.6 節) し，6.7 節で Sun HotSpot VM[28] 上での実装法についてのべ，6.8 節にて評価を行う.

6.2 関連研究

キャッシュミスは，以下の 4 種類に分類される.

- **cold miss**: 当該メモリブロックに対する最初のアクセスの際に発生するミス
- **conflict miss**: 同じキャッシュラインを利用する多くのメモリブロックが同時期にアクセスされた場合に発生.
- **capacity miss**: キャッシュメモリ容量に起因するミス
- **coherence miss**: キャッシュ無効化に起因するミス

このうち，cold miss は回避不能であり，conflict miss はかなり規則的なオブジェクト配置を行ったときのみ問題となる．逐次計算機においては capacity miss が主要な問題であり，共有メモリ並列計算機においては，場合によって coherence miss が深刻な問題を引き起こす.

リンク構造を扱うプログラムの場合，キャッシュを有効利用するためには，メモリ上でのオブジェクト間の配置を考慮する必要があり，特に次の 3 点が重要であると考える.

空間的局所性の向上　例えば，参照関係にあるオブジェクト同士をメモリ上で近くに配置すると，参照元オブジェクトにアクセスした際に参照先オブジェクトもキャッシュに取り込まれる可能性が高くなる．

キャッシュ密度の向上　頻繁にアクセスされるオブジェクト同士を隣接配置してキャッシュラインを埋めることで，必要となるキャッシュ容量を抑えることが可能となりキャッシュの利用効率が向上する．

無効化の影響の回避　キャッシュの無効化はキャッシュラインを単位として生じるため，頻繁に更新されるオブジェクトを他のオブジェクトとは別ラインに配置することで無効化の影響を回避できる．

6.2.1　空間的局所性の向上

自動メモリ管理機構による空間的局所性の向上を目指す研究には，互いに参照関係にあるオブジェクト同士や，時間的に同時期にアクセスされるオブジェクト同士を近くに配置することで行うものがある．

まず，GC 時のコピー順序を変更することで空間的局所性を高める研究がある．通常 copying GC においては幅優先順にオブジェクトのコピー処理を行う [15]．そのため，例えば木構造の場合，木の兄弟節点同士はメモリ上で近くに配置されるが，木の深さが増すほど親節点と子節点とが離れるため局所性が悪くなる．一方，幅優先順に対して深さ優先順に処理を行う方法 [48] や，参照元オブジェクトと参照先オブジェクトとを 1 つのグループとして扱い，各グループを階層的にコピーする方法が提案されている [47, 30]．これらは，直接参照関係にあるオブジェクト同士をメモリ上で近くに配置することで空間的局所性の向上を実現する．

また，GC 時のコピー順序を変更する方法以外にも，メモリ割当て時から局所性の向上を目指した研究も行われている．メモリを順に割当てる Coying GC はもともと局所性は高いが，例えば文献 [41] では，頻繁にオブジェクトが生成されるクラスを prolific types として区別し，参照関係にある prolific types のオブジェクト同士を 1 つのグループとしてメモリを割当てることで局所性の向上を行っている．

一方これらに対して，オブジェクトのアクセス時期・頻度を考慮することで

局所性を向上させる研究も行われている．例えば文献 [17] では，実行時にアクセスしたオブジェクトのアドレス値を取得し，GC の際に，取得したアドレス値から時間的な親和性グラフを作成し，同時期にアクセスされるオブジェクト同士を近くに配置することで空間的局所性を高める方法を提案している．この方法により直接参照関係にないオブジェクト同士の局所性を向上させることが可能となる．

6.2.2 キャッシュ密度の向上

キャッシュを意識した研究には，上記のオブジェクト間の配置を考慮する方法以外にも，オブジェクト内のレイアウトに注目した方法も存在する．

逐次計算機向けのオブジェクトレイアウトの研究として，文献 [16] は structure splitting と呼ばれる方法を提案しており，1 つのクラスを頻繁に参照される変数を格納するクラスとあまり参照されない変数を格納するクラスとに 2 分割することで，キャパシティミスを削減し，キャッシュ密度を向上させている．オブジェクト配置は，前述の自動メモリ管理機構 [17] で行っている．

一方，並列環境においてキャッシュ密度の向上を図るには，スレッド毎の局所性も考慮する必要がある．つまり，ある特定のスレッドからのみアクセスされるスレッドローカルオブジェクトは，他のスレッドにとっては無駄な領域であり，その扱いが重要となる．例えば，今回利用する HotSpot Server VM のように，各スレッド毎にメモリ割当て用の領域を用意する場合，メモリ割当ての高速化に加えてスレッド毎の局所性の向上も期待できる．また，スレッドローカルオブジェクトは他の共有オブジェクトからは参照されていないため，前述した深さ優先方式の GC により，複数のスレッド間でのスレッドローカルオブジェクトの混在を抑制できると考える．

6.2.3 無効化の影響の回避

同時期または頻繁にアクセスされるオブジェクト同士を隣接配置することで空間的局所性やキャッシュ密度を向上させる上記の配置は，逐次計算機においては有効である．しかし，共有メモリ型並列計算機では，隣接配置されたオブジェクトの更新に伴うキャッシュの無効化も考慮する必要がある．無効化の影響を回避する 1 つの方法は，スレッドローカルオブジェクトを他のオブジェクトと分離することであるが，オブジェクトがスレッドローカルであるかどうか

を静的に判定するのは難しい.

一方で，共有オブジェクトの更新による無効化の影響の回避については，コーディング技術として，頻繁に書き込まれるフィールドの前後にパディングをする方法が知られている．しかし，該当インスタンスが多い場合，メモリ使用量などの面で問題がある．また，各種コーディング技術としての存在であり，自動化を目指した研究は知られていなかった.

6.3 固定レイアウト法

6.3.1 アプローチ

議論に先立って，複数の CPU に共有されているインスタンス変数を，そのアクセス状況に応じて以下のように分類する.

(1) W 変数：頻繁に write される変数

(2) R 変数：(1) 以外の頻繁に read される変数

(3) N 変数：(1)，(2) 以外の変数

我々はレイアウト法を決めるにあたり，キャッシュコンテンションを最も回避すべき問題と考えた．W 変数と R 変数が同じキャッシュラインに配置された場合，R 変数への read access においても，W 変数の無効化の影響を被る状況が発生し，coherence miss が頻発する．この状況をキャッシュコンテンションと呼ぶ．コンテンションによる R 変数へのアクセス速度の低下を回避するため，我々は W 変数と R 変数を別々のキャッシュラインに配置する．その方法として，6.2.2 節で紹介した [16] の structure splitting を応用する．coherence miss 削減のため，[16] と違い,

- プロファイルを行う際， read / write を区別して各インスタンス変数のアクセス回数を計測する
- クラスの分割の際，W 変数は R 変数とは別の場所に格納する

また R 変数については，集中配置することで従来通り capacity miss を削減する.

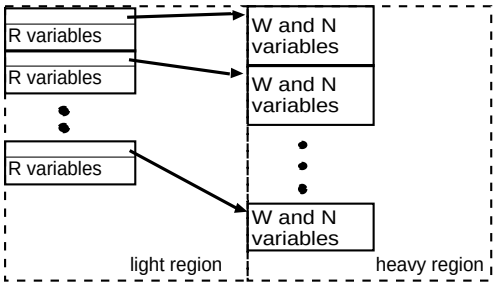

図 6.1: 固定レイアウト法のメモリ配置 (イメージ)

6.3.2 クラスの分割

本研究で提案するレイアウト法は，[16] の structure splitting と同じくクラスを 2 つに分割する．ただし，変数アクセス頻度情報は読み書きを区別して集計し，変数を R 変数，W 変数，N 変数の 3 種類に分類する．なお，分割されたクラスを light クラス，heavy クラスと呼ぶ．

light, heavy クラスはそれぞれ，

- light クラス：R 変数配置用領域
- heavy クラス：W，N 変数配置用領域

として利用される．つまり，[16] の hot/cold 部 (図 6.3) がそれぞれ light/heavy 部に相当する (図 6.1)．heavy クラスのインスタンス変数へのアクセスも，Chilimbi の cold クラスと同じく間接参照によってなされる．オブジェクトのアロケーションの際は， light 用の領域と heavy 用の領域が別個確保され，light 部を light 用領域に，heavy 部を heavy 用領域に配置する (図 6.1)．この方式により，heavy 領域への書き込みでは light 部に coherence miss は発生しない．また，light 領域に R 変数が集中的に配置されているため，capacity miss の削減が望める．

この方式では，各変数へのアクセスコストは以下のようになる．

- R 変数:無効化の影響は抑制され，また，密に配置されるためキャッシュに残る確率が高くなる．
- W 変数:間接参照が必要．1 つの W 変数が複数 CPU によって読み書きされる場合，無効化の影響は CPU 間通信に必要なコストと言える．一方，単一 CPU によってのみ利用された場合，他の W 変数が当該キャッシュラ

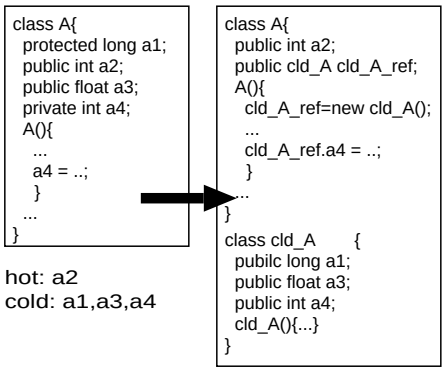

図 6.2: プログラム変換例

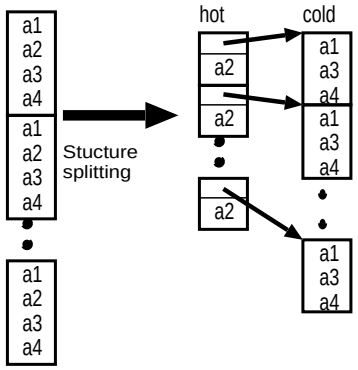

図 6.3: キャッシュを意識したメモリ配置 (イメージ)

インに存在しない限り無効化の影響を受けない．反対に，キャッシュライン上に複数の W 変数が存在し，同時期に複数の CPU により書き込みが行われた場合は，無効化が頻発する (false sharing)．

- N 変数:間接参照が必要．また W 変数の無効化の影響を受ける．ただしアクセス頻度は少ない．

本方式は，クラス階層に関して実装上の課題が存在するが，これは，6.9 節で議論する．

6.4 レイアウト切替え法

本節では，6.3 節で提案した固定レイアウト法が有効に機能する状況と機能しない可能性のある状況について考察し，その対応策として複数のレイアウトを切替える方式の提案を行う．

6.4.1 レイアウトの動的変更

固定レイアウト法では，プログラム実行全体を通してのアクセス頻度情報をもとにインスタンス変数を R/W/N 変数に分類し，レイアウトを決定した．この手法は変数アクセス傾向が実行中に大きく変化しない場合有効に機能する．しかし，計算が進行していくにつれ変数のアクセス傾向が大きく変化していく場合，固定レイアウト法は適切なレイアウトを提供できない可能性がある．これを例を用いて説明する．あるプログラムが局面[2] A，B の 2 つの局面から構成され，当初の局面 A においては頻繁に更新が行われていたインスタンス変数 x が，次の局面 B において頻繁に読み込まれ，書き込みはほとんどないとする．この場合，固定レイアウト法では局面 A，B を通しての x の総 write アクセス回数によってその配置を決定する．その結果，x が R 変数と決定され light 部に配置された場合は，局面 A において他の R 変数にコンテンションの影響をおよぼすこととなる．一方，W 変数と決定され heavy 部に配置された場合は，局面 B において他の W 変数からコンテンションの影響を受けることとなる．つまり，この変数は light 部に配置しても heavy 部に配置しても問題を生じることとなる．

我々はこの問題を回避するため，局面毎に動的にレイアウトを切替えることを考える．これにより，例えば前述の変数 x は 局面 A では heavy 部に配置し，局面 B への遷移時に light 部に位置を変更することで上記の問題を回避することが出来る．この局面毎に適したレイアウトに切替える方法のためには，まず局面毎にレイアウトを用意する必要がある．また，実装上の種々の問題がありこれを解決しなければならない．そこで，6.4.2 節で例を用いて局面毎にレイアウトを決定する方法について述べ，そして 6.4.3 節で実装上の問題と解決策を説明する．

[2] ここでの局面は，解析と関連しているわけではない．

6.4.2 局面毎のレイアウト決定手順

局面毎に適したレイアウトに動的に設定することを，プログラム例を用いて説明する．例えば，木の構築を行い，木が完成した後はその木のデータを使って計算するプログラムの場合を考える．この場合，木の構築局面と，計算局面にそれぞれ適したレイアウトを決定し，計算局面への遷移時にレイアウトを切替える．

そのためには，まず局面毎に適したレイアウトを決定しなければならない．インスタンス内部のレイアウトは，6.3 節の固定レイアウト法と基本的に変化はない．つまり，オブジェクトは light/heavy の 2 部に分割され，インスタンス変数はその局面のアクセス状況に応じて適切な場所に配置する．但し，レイアウトの変更時のデータ移動のコストならびに その際の light 部への書き込み (による隣接したオブジェクトへの影響) を抑える，つまりレイアウト変更時に移動するデータ量を小さくするためにレイアウト決定は以下の手順で行う．

- 各局面 p における R/W/N 変数の数 (R_p, W_p, N_p) を求める．
- オブジェクトの light 部のサイズを $LIGHT = max(R_i)$ とし，heavy 部のサイズを max(max(W_i), ObjSize - LIGHT) とする
- 各局面において R, W 変数をそれぞれ light/heavy 部に必ず配置し，残った領域に N 変数を配置する．但し，レイアウト切替え時にあまり変数の位置が変化しないように N 変数の位置を決定する (局面遷移後に W 変数になるものを heavy 部に，R 変数になるものを light 部に優先して配置する)

レイアウトの異なった局面間で遷移が行われる場合，変数の位置を動的に変更するだけでは問題がある．次の節では，その問題と解決策を述べる．

6.4.3 実装上の問題と解決策

前節で述べた問題とは，位置が変わる変数へアクセスするコードが，レイアウト変更後，変更前の位置にアクセスしてしまい目的の変数に対して操作できなくなることである．これを回避する簡単な方法は，アクセスの度に分岐や間接参照を行うようにアクセスコードを変更することである．しかし，これでは性能向上を得ることができない．

このため実装上の方針として，

- レイアウトを変更するクラスのメソッドが，位置が変わる変数にアクセスする場合，適切にアクセスできるようにレイアウト毎にあらかじめ特化したメソッドを用意する．

- また，レイアウト変更で位置が変わるのは，外からアクセスされないと分かっている変数に限定する．(Java で private 宣言されている変数)

- レイアウトを変更するクラス内の，特化メソッドを呼び出すメソッドについても，レイアウト毎に特化したものを準備する．

- 他クラスからの特化メソッド呼び出しは，仮想化により適切なレイアウト用のメソッド呼び出しを実現する

なお，仮想化の詳細は後で説明する．

但し，この実装方針ではメソッド起動のタイミングでのみ，現在の局面値による実行コードの切り替えを行っているため，特化メソッドの実行中に局面が切り替わった場合問題が生じる．つまり，目的変数の位置が移動したにもかかわらず，以前の位置にアクセスしてしまう．

このため，我々は，局面とレイアウトが変化する際に特化メソッドが実行されていないことが保証できた場合のみ，利用を許可することとした．例えば対象プログラムを fork-join により構造化されている場合や，3 章の局面解析系などにより，実行局面が保証された場合が相当する．例を用いて説明する．局面 A と B により構成されているプログラム中に，レイアウトを変更したいクラス Node がある場合を考える．この Node クラスは，局面 A 用のレイアウト a と，局面 B 用のレイアウト b を用意し，局面遷移時にレイアウトを変更する操作が行われる．局面 A が fork-join により構造化されていれば，A 局面で呼び出されるレイアウト a 用の特化メソッドは局面 A の終わりで終了することが保証され，局面 B で実行される心配はない．

最後にメソッドの仮想化について説明する．仮想化を行うための操作は以下の 3 つである．

- それぞれの局面に一意に番号をつけ，

- 特化メソッド毎に，局面数分要素がある配列を用意し，そこに，局面番号と対応したメソッド番地を入れておき，

表 6.1: アクセス情報と変数分類 (カウンタつき 2 分木)

変数	アクセス数 (x10^7)	write の割合	Chilimbi 分類	固定分類
val	5.92	0%	hot	R:light
left	5.92	0.00%	hot	R:light
right	5.92	0.00%	hot	R:light
obj_lck	0.00	Lock	cold	N:heavy
count	0.20	50%	hot	W:heavy
count_lck	0.20	Lock	hot	W:heavy

- メソッド呼び出しは，上記の配列と局面番号をもとに行う．

つまり，`insert();` という呼び出しが，仮想化により `insert[現在の局面番号]();` となる．

6.5 オブジェクト内レイアウト法の評価

提案した 2 種類のレイアウト法の有効性を確かめるために，他のレイアウト法との比較実験を行う．比較対象は，通常のレイアウト (original) と，オリジナルの structure splitting[16] 法 (Chilimbi) である．これらを ccNUMA アーキテクチャの SGI Origin2000 で評価した．計測対象の計算機のキャッシュ構成は 1 次命令キャッシュ，1 次データキャッシュと 2 次キャッシュから成り立っている．この 2 次キャッシュのサイズは 4Mbyte で，キャッシュラインは 128byte である．また，総メモリ量は 4Gbyte，CPU は MIPS R10000(195MHz) の 28 台構成となっている．なお評価プログラムは C 言語を用い，並列化は Pthread ライブラリにより実現した．

評価を行うにあたり変数アクセス情報が必要となるが，現在手動でプロファイル用コードを作成し，アクセス情報を入手している．また，レイアウト切替用のプログラムの局面切り分け，並びに局面毎のコード特化についても手動で行っている．

次に，今回用いた不規則計算を行う 2 つのプログラムと，その実験結果を説明する．

表 6.2: 実行結果 (カウンタつき 2 分木)

	実行時間		キャッシュミス数	
original	1.23sec	100%	4.034×10^6	100%
Chilimbi	1.11sec	90%	3.696×10^6	92%
固定	0.89sec	72%	2.012×10^6	50%

表 6.3: 変数アクセス情報 (N 体問題)

変数	CnstP	ApprP	CalcP	Total
x,y,z	7.81×10^5 (0%)	0 (0%)	0 (0%)	7.81×10^5 (0%)
region	3.90×10^5 (0%)	0 (0%)	2.18×10^7 (0%)	2.22×10^7 (0%)
cx,cy,cz	9.84×10^4 (0%)	1.97×10^5 (24%)	2.19×10^7 (0%)	2.22×10^7 (0.2%)
mass	4.93×10^4 (0%)	5.44×10^5 (36%)	1.86×10^7 (0%)	1.92×10^7 (1.0%)
child[] (total)	1.18×10^6 (12%)	5.38×10^5 (0%)	2.58×10^7 (0%)	2.75×10^7 (0.5%)
leaf_f	8.35×10^5 (6%)	1.49×10^5 (0%)	3.51×10^6 (0%)	4.49×10^6 (1.1%)
lock	3.50×10^5	0	0	3.50×10^5
合計	5.89×10^6	2.81×10^6	1.67×10^8	1.75×10^8

6.5.1 カウンタ付き 2 分木

プログラム概略：コンテンションが発生する状況において，固定レイアウト法が有効に働くか確認する．評価プログラムはカウンタ付き 2 分木である．これは，通常の 2 分木とは違い各節点がカウンタを持つ．木の構築時，節点の値と同じ値が挿入された場合，その節点のカウンタがインクリメントされる．この 2 分木プログラムに，無効化が発生するデータとして，0-2047 の整数値 100 万個のデータを挿入する．この挿入データでは，節点が 2048 個作られ，各節点のカウンタが平均約 500 回インクリメントされる．

各節点オブジェクトは表 6.1 に示される変数を持つ．このカウンタ付き 2 分木に並列にデータ挿入を行うため，節点オブジェクトは全 CPU で共有され，インスタンス変数は複数の CPU にアクセスされる．一貫性保持のため，各節点

表 6.4: 変数分類 (N 体問題)

変数	固定レイアウト分類	レイアウト切替え法分類		
		CnstP	ApprP	CalcP
x,y,z	N:heavy	R:light	N:heavy	N:heavy
region	R:light	R:light	N:light	R:light
cx,cy,cz	R:light	N:heavy	N:light	R:light
mass	R:light	N:light	N:light	R:light
child[]	R:light	R:light	N:light	R:light
leaf_f	R:light	R:light	N:light	R:light
lock	N:heavy	W:heavy	N:heavy	N:heavy

に 2 つのロック (カウンタ用のロック (count_lck)，子供へのポインタ用のロック (obj_lck)) を持たせた．各 CPU が節点に子供をつくる場合，まずその節点の obj_lck を獲得し，子供を作った後 obj_lck を解放する．また，カウンタをインクリメントする場合，count_lck を獲得し，インクリメントした後，count_lck を解放する．

プロファイル結果: 上記のデータを挿入した場合の変数アクセス回数をプロファイルした．結果を表 6.1 に示す．但し，オブジェクト生成時の変数の初期化に関してはアクセス回数にカウントせず，また，lock へのアクセスは書き込みと見なしている．なお，このプロファイル結果は逐次実行時のアクセス情報である．このアクセス情報はプログラムの性質上，並列時のアクセス情報と劇的に違うことはないと考えられる．

レイアウトの決定: 次に，各レイアウトを決定する．表 6.1 の各変数のアクセス情報から，Chilimbi, 固定レイアウトでの変数分類は同表のように決定される．変数 val,left,right は頻繁に読み込まれ，count は頻繁な書き込みがある．そのため，変数 val,left,right を，count 変数と同じキャッシュラインに配置した場合，val,left,right への read アクセスは，無効化の影響を受けることになる．固定レイアウトは変数 val,left,right 変数を count 変数とは別のキャッシュラインに配置しているため，キャッシュミス数を削減することが期待できる．

評価：決定したレイアウトを用いて，評価プログラムを 8CPU で並列実行した．その実行時間並びに 2 次キャッシュのキャッシュミス数を表 6.2 に示す．固定レイアウトは，original のレイアウトと比べて 28%，Chilimbi のレイアウト法と比べても 20%の速度向上を実現している．これは，キャッシュミス数が original

表 6.5: 実行結果 (N 体問題)

レイアウト	実行時間 (sec)						キャッシュミス数	
	CnstP	layout 変更	ApprP	CalcP	ALL			
original	0.16		0.09	3.74	4.01	100%	8.210×10^6	100%
固定	0.28		0.07	3.24	3.64	91%	5.653×10^6	69%
切替え	0.19	0.15	0.07	3.28	3.71	93%	5.514×10^6	67%

と比べて 50%，Chilimbi と比べて 54%となっているためである．

original, Chilimbi, 固定レイアウトの cold miss は同じ程度あると考えられるが，capacity, coherence miss 数はレイアウトにより異なる．固定レイアウトが最もキャッシュミス数が少ないのは capacity miss と，coherence miss をともに抑制できているためと考えられる．capacity miss が抑制できている理由は，全体のアクセス数の 97%を占める変数 val,left,right 同士を近くにまとめて配置しているからである．また，coherence miss を抑制できている理由は，全体のアクセス数の約 2%を占める obj_lck,count,count_lck への書き込みによる無効化の影響を，頻繁に read access される変数 val,left,right に及ぼさなかったためである．

以上の結果から，キャッシュコンテンションが発生するプログラムに対して，固定レイアウト法が有効であると言える．また，ccNUMA アーキテクチャー上で capacity miss と coherence miss を抑制することが実行速度向上のために重要であるといえる．

6.5.2 N 体問題

プログラム概略：次に，レイアウト切り替え法の有効性について評価を行う．利用したのは，Barnes-Hut 法を利用した天体のシュミレーションを行う N 体問題プログラムである．レイアウト対象である節点クラスのインスタンス変数を，表 6.3 に示す．

このプログラムは 3 つの局面があり，実行にともなって，木の構築局面 (CnstP)，重心計算局面 (ApprP)，加速度計算局面 (CalcP) と変化する．CnstP では，天体のデータをもとに木が並列に構築される．この際，節点クラスのインスタンスは全て共有され，一貫性保持のため lock が利用される．次の ApprP では，インスタンスは単一 CPU からのみアクセスされる．最後の CalcP では，イ

ンスタンスは再び複数の CPU によりアクセスされるが，write アクセスはなく lock も使用されない．

プロファイル結果：天体のデータ数が 10 万個のときの N 体問題プログラムを実行し，各局面の変数の総アクセス回数及びアクセスの種類をプロファイルした．その結果を表 6.3 に示す．なお，オブジェクト生成時の変数の初期化に関してはアクセス回数にカウントしていない．括弧内は総アクセスに占める write の割合である．また，このプロファイル結果は逐次実行のアクセス情報である．

表 6.3 より，局面の遷移に伴いインスタンス変数のアクセス傾向も変化していることがわかる．

レイアウトの決定：表 6.3 のアクセス情報から，固定レイアウト及び，レイアウト切替え法の各局面のレイアウトを決定した．各レイアウトを表 6.4 に示す．CnstP では全ての変数がアクセスされるが，ApprP では変数 x,y,z,region,lock が一度もアクセスされない．次の CalcP でも，x,y,z, と lock がアクセスされていない．ApprP と CalcP においてアクセスされない変数を，アクセスされる変数とは別のキャッシュラインに配置することで，capacity miss の抑制が期待できる．このため，レイアウト切替え法では，CnstP に対するレイアウトと，ApprP，CalcP に対する共通レイアウトの計 2 つのレイアウトを準備し，CnstP から，ApprP への遷移時にレイアウトの切替えを行う．なお，ApprP,CalcP に対する共通レイアウトでは，変数 x,y,z,lock を他の変数とは別の領域に配置している．

次に固定レイアウト法のレイアウトについて述べる．固定レイアウト法では，プログラム全体のアクセス傾向をもとにレイアウトを決定する．そのため，全体を通してアクセス数の少ない変数 x,y,z,lock が heavy クラスに配置される．このレイアウトは，CalcP に適したものとなっている．なぜなら，CalcP が全体のアクセスの 9 割以上を占めているため，全体のアクセス数をもとにレイアウトを決定すると，CalcP に適したものとなるからである．

評価：決定した各レイアウトで，評価プログラムを 8 台 CPU で実行した．CnstP, ApprP, CalcP をそれぞれ 1 回実行した場合の実行時間と 2 次キャッシュのキャッシュミス数を表 6.5 に示す．

固定レイアウト法，レイアウト切替え法はともに original のレイアウトと比べ，大きくキャッシュミスを抑制できている．これは，各レイアウトでの cold miss は同程度だと考えられるが，capacity, coherence miss が抑制されているためで

ある．固定レイアウト，レイアウト切替え法はともに，ApprP,CalcP でアクセスがない変数 x,y,z,lock を heavy クラスに配置している．このために，capacity miss を削減でき，ApprP で約 23%，CalcP で約 14%の実行時間の短縮が実現できていると考えられる．

次に，CnstP での固定レイアウト法とレイアウト切替え法のレイアウトの違いを考える．両者のレイアウトの違いは，変数 x,y,z 及び cx,cy,cz の位置が light 部にあるか，heavy 部にあるかだけである．レイアウト切替え法では，CnstP 内でアクセス数の約 40%を占める変数 x,y,z を light 部に配置し，書き込みの約 64%を占める lock とは別の領域に置くことで capacity miss と coherence miss をともに抑制している．このために，固定レイアウトに比べて約 33%もの実行時間短縮を実現している．

しかし，全体の実行時間は固定レイアウトが一番短い．これはレイアウト切替え法のレイアウト変更のオーバヘッドのためである．このため固定レイアウトが相対的に速くなっている．

N 体問題は，CalcP において総アクセスの約 95%を占めている (表 6.4)．このため，レイアウトが固定である固定レイアウト法でも，特定の局面に特化することで十分キャッシュミスを抑制できた．また，このプログラムは元来他の局面においてキャッシュコンテンションによる大幅な性能低下などを起こしていたわけでもなく，レイアウト変更による性能向上は限られたものであった．しかしもし，変数へのアクセスが複数の局面に分散しており，アクセス傾向が大きく変化するのであれば，レイアウト変更のメリットがコストよりも大きくなる可能性がある．

カウンタつき 2 分木や，N 体問題の実行結果をまとめると，状況によって固定レイアウト法が有効な時も，またレイアウト切替え法が有効な場合もある．固定レイアウト法が有効になる場合は，多くの局面で変数へのアクセス傾向が同じ場合や，ただ 1 つの局面にアクセスが集中している場合である．レイアウト変更のコストを払ってでもレイアウト切替え法を用いるべき状況は，固定レイアウト法とは反対の状況である．つまり，複数の局面にアクセスが均等に分散されており，アクセス傾向が大きく変化する場合である．

6.6 オブジェクト配置方式

6.6.1 アプローチ

前節までで，キャッシュミスを削減したオブジェクト内レイアウト法について議論を行ってきた．一方で，実際のメモリ管理機構に統合してレイアウトの自動化を実現するには，実システム上でどのようにアクセス傾向別領域を設けるか，どのようにスレッド局所性を向上するかなど，検討課題も多い．

本研究では，キャッシュを有効利用するにあたって，自動メモリ管理機構（copying GC）上で

1. 頻繁に読み出されるオブジェクト同士をメモリ上にまとめて配置することでキャッシュ密度の向上と，キャッシュの無効化の影響を回避する．

2. 深さ優先コピー方式により空間的局所性とスレッド局所性の向上を目指す

という配置を行う．これにより，6.2 節で紹介したキャッシュ効率に関する 3 つの項目に対し，一通りの対応を講じることが出来る．（1）については 6.5 節の評価の結果，固定レイアウト法 (6.3 節) によるクラス分割やアクセス傾向別領域の導入により解決を目指し，（2）に関しては今回はスタックを利用する単純な深さ優先コピー方式の実装を行っている．

また，本提案は 6.7 節において，先進的な Java 処理系の一つである SUN HotSpot Server VM 上に実装され，6.8 節において実アプリケーションを通しての統合評価を行うことになる．

本システムは，最終的に以下の 2 つの機構から構成される．

- **プロファイラ機構:**対象プログラムのプロファイリングにより，フィールドアクセス数を取得し，アクセス傾向による各クラスの分類・分割を行う．

- **メモリ管理・配置機構:**自動メモリ管理機構にアクセス傾向別領域を導入し，クラス分類情報に基づいてメモリ割当て・GC を行う．また，深さ優先コピー方式を導入する．

以下では，クラス分類方法ならびに両機構の連携について説明する．対象処理系である HotSpot VM 内の自動メモリ管理機構に本メモリ管理・配置機構を組み込む方法については 6.7 節で述べる．

6.6.2 クラスの分類

プロファイラ機構は，各クラスのインスタンスフィールドに対する読み出し・書き込みアクセス数を収集し，クラスを以下の 2 種類に分類する．

- FreqRead（Frequently Read）クラス：書き込みは少なく，かつ頻繁に読み出されるクラス．
- Other クラス：頻繁に書き込まれる，またはあまりアクセスされないクラス．

ただし，クラスの中には頻繁に読み出し・書き込みの行われるフィールドがそれぞれ存在し，フィールド毎にアクセス傾向が異なる場合もある．そのようなクラスに対しては固定レイアウト法により light, heavy にクラス分割を行った後，それぞれ FreqRead, Other に分類することとする．変数分類基準に関しては，6.8 節で評価において示す．

6.6.3 プロファイラ機構との連携

プログラムのアクセス傾向を取得する方法としては，対象プログラムの事前実行による取得や，文献 [17] のようなプログラムの実行時プロファイリングによる取得などが考えられるが，Java VM の改良範囲を最小限に抑えるため，また実行時に加わるオーバヘッドを避けるために本研究では前者の方法を用いる．

従って，Java VM の外部のプロファイラ機構から内部のメモリ管理・配置機構にクラス分類情報を伝える必要がある．そのため方法として，Java の標準機能である interface を利用する．FreqRead クラスにおいては，マーカとして以下の宣言を行う．

- `implements FreqReadObj`
- `implements FreqReadAry`

`FreqReadObj` はそのクラスのオブジェクトが FreqRead であることを示す．加えて，そのクラスのオブジェクト配列が FreqRead であることを示す `FreqReadAry` マーカも存在する．これらの interface の中身は空で，その名前のみが意味を持つ．Java VM 側では，クラスロード時に interface 名を調べることでクラス分

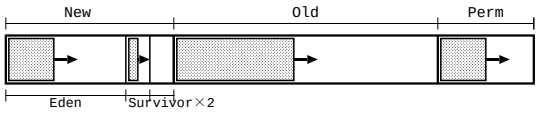

図 6.4: ヒープ構造 (SUN HotSpot Server VM)

類情報を取得する．クラス分類情報は，Java のクラスオブジェクトを表す Java VM 内でのデータ構造に分類情報を表す変数を追加し，そこに格納する．

なお，プロファイル結果に基づくクラスの分類・分割ならびにマーカ interface の付加は，バイトコード変換により自動化する予定である．

6.7 メモリ管理・配置機構の実装

本節は，SUN HotSpot Server VM への，アクセス傾向別領域の導入ならびに深さ優先 GC の実装に述べたものである．本節の内容については，共同研究者の松田聡氏が行ったものである．ここでは，6.8 節の性能評価の上でも必要な知識と考え，その処理系実装について紹介する．

6.7.1 Sun HotSpot Server VM

本節では，HotSpot Server VM のメモリ管理部分について説明し，次節で今回の改良について述べる．

図 6.4 に HotSpot VM のヒープ構造を示す．HotSpot VM では世代別 GC が採用されており，ヒープは新・旧 2 つの世代に分かれている [10]．図の Old と Perm とが旧世代であり，新世代用空間（New）はさらに Eden 空間と 2 つの Survivor 空間とから成る．Java のインスタンスは Eden 空間に生成され，2 つある Survivor 空間は，GC の際に一方は From 空間として，もう一方は To 空間として使われる（どちらかは常に空）．なお，Java の通常のインスタンスが配置されるのは，図の New, Old の空間であり，Perm にはクラスオブジェクトなどが配置される．

メモリ割当ては並列化されており，各スレッドは Eden 空間から Thread-Local Eden（TLE）と呼ばれる小領域を確保して使用することで，TLE 領域内に空きがある間は他のスレッドと非同期にメモリ割当てが可能になっている（TLE を確保する際は同期が必要）．

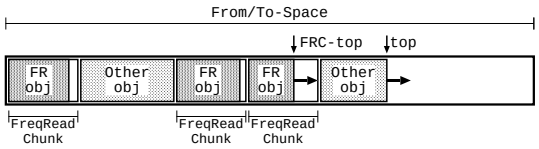

図 6.5: FreqRead Chunk

新世代用の GC として幅優先方式の copying GC が採用されており，GC の際の From 空間となる Eden 空間・Survivor 空間の一方から，To 空間となる Survivor 空間のもう一方，または Old 空間へと生きているオブジェクトがコピーされる．また，全世代用の GC として Mark-Compact GC が採用されており，Eden 空間内のオブジェクトを格納するだけの充分な領域が Old 空間に無い場合に起動される．なお，GC の並列化はされていない．

6.7.2 アクセス傾向別領域の導入

以下では，メモリ割当て用の空間（Eden）を From，GC 時に生きているオブジェクトをコピーするための空間（Survivor・Old）を To と表記する．

FreqRead オブジェクトとその他のオブジェクトとを離して配置するための方法として，TLE と同様の方法をとる．つまり，From/To 空間から FreqRead 専用の小領域 (FreqRead Chunk: FRC) を確保し，FreqRead オブジェクトは FRC からメモリを割当てる．From 空間内の FRC サイズは TLE と同様である．一方，To 空間の FRC のサイズは TLE のサイズに現在のスレッド数を乗じたものとしている．従って，図 6.5 のように，From/To 空間内に FRC がいくつか存在することになる．

次に，FRC 導入に伴うメモリ割当て・GC の変更について説明する．メモリ割当てにおいては，まず割当てるオブジェクトのクラスについて FreqRead かどうかを分類情報から判断し，FreqRead オブジェクトであれば FRC から，その他であれば通常の TLE から割当てる．そのため，インタプリタおよび JIT コンパイラのメモリ割当て部分にクラス分類判定のための変更を加えている．FRC のメモリ割当て方式は TLE と全く同様であり，各スレッドは各々の FRC を持ち，他のスレッドとは非同期に割当て可能である．FRC 領域が尽きた場合は，From 空間から再び FRC を確保する．また，FRC サイズ以上のメモリを

要求された場合は通常の From 空間から割当てる．

GC 時も，コピーの際にオブジェクトが FreqRead かどうかの判定を行い，FreqRead オブジェクトの場合は FRC へコピーし，他のオブジェクトであれば一般の To 空間内の領域にコピーする．FRC は必要になった時点で To 空間から確保されるため，図 6.5 に示すように空間内でオブジェクトが連続して配置されるとは限らない．つまり，To 空間をキューとして利用する従来の幅優先方式をそのまま用いることはできない．しかし，今回は空間的局所性を向上させるために深さ優先方式を導入するため問題はない．

6.7.3 深さ優先コピー方式の導入

GC 時においては，オブジェクトの分類判定後，FreqRead/Other ともに深さ優先方式でコピー処理を行う．深さ優先コピー方式の実装方法としては，スタックを利用し，コピーしたオブジェクト内の参照のアドレスを格納していくという単純な方法を用いている．スタック用の領域は Mark-Compact GC のマーキング時に使われるスタック用の領域・データ構造を利用することとした．このデータ構造は足りなくなるとそのサイズを伸ばすことが可能となっているため，スタックサイズは固定していない．

クラス分割を適用してクラスを FreqRead/Other に 2 分割した場合，単に深さ優先方式で処理すると，FreqRead から Other への参照の存在により FreqRead オブジェクトと Other オブジェクトとが隣接配置されてしまうが，前述のアクセス傾向別領域によりこの問題は解決される．

なお，現在の実装では Mark-Compact GC の変更は行っていない．これは，FreqRead オブジェクトは固まって配置されており，また，Compaction 操作も生きているオブジェクトの並びはそのままであるため，FreqRead オブジェクトと他のオブジェクトとが混在することはないからである．

6.8 評価

6.8.1 評価環境

プロトタイプ実装の有効性を確かめるために共有メモリ型並列計算機である Sun Enterprise 6500 上と Sun Fire 12K 上とで評価を行う（表 6.6）．評価プログラムとしてカウンタ付き 2 分木と Nbody，MolDyn を用い，次の 3 つの Java

表 6.6: 実行環境

Machine	Sun Enterprise 6500	Sun Fire 12K
CPU	UltraSPARC-II 336 MHz (20 CPUs)	UltraSPARC-III 1050 MHz (20 CPUs)
L2Cache/CPU (Cacheline)	4 MB (64B)	8 MB (64B)
Memory	8 GB	40 GB
OS	Solaris 2.6	Solaris 8
Java	Sun JDK1.3.1(-Xboundthreads)	

VM を使用した場合の実行時間を比較する.

- breadth: 標準の Server VM（GC は幅優先コピー方式）.
- depth: 標準の Server VM の GC を深さ優先方式に変更したもの.
- FRdepth: 本プロトタイプ. FreqRead/Other を分けて配置する. GC は深さ優先方式利用.

標準 Java API 等はどの VM でも共通であり，Sun JDK1.3.1 のものを利用している. また，実行時には -Xboundthreads オプションを指定して Java のスレッドを Solaris の Light Weight Process（LWP）に張り付けることで，スレッドスケジューリングの影響を抑制している. なお，アクセス情報の取得やクラスの分類・分割は手動で実現している.

6.8.2 変数分類基準

また，本論文の評価環境においては,

- 書き込みアクセス総数が，全アクセス数の 0.5 %以上ないと分割対象としない
- W 変数: 対象クラスへの全書き込みアクセス数のうち，20%以上書き込みアクセスされる変数.

表 6.7: 節点のアクセス情報（カウンタ付き 2 分木）

Fields(type)	Read (M:×10^6)	Write (M:×10^6)
val(R)	78.5M [T:45%]	0.3M [W: 8%]
left(R)	46.9M [T:27%]	0.5M [W:12%]
right(R)	41.6M [T:24%]	0.5M [W:12%]
found(W)	1.0M [T:1%]	1.3M [W:34%]
counter(W)	0.7M [T:0%]	1.0M [W:26%]
lock(N)	0.7M [T:0%]	0.3M [W: 8%]
total(R)	169.4M [T:98%]	3.9M [W:100%]

- R 変数: W 変数以外の変数で，対象クラスへの全アクセス数のうち，1%以上読み出しアクセスされる変数.

- N 変数: W, R 以外の変数.

と定める．その上で，Light/Heavy クラスをそれぞれ FreqRead/Other クラスとして配置を行う．この判断基準は，ある程度アクセス頻度が高い変数は R 変数と見なすが，他変数に比して書き込み回数が高いものは W 変数として除外する方針による．なお，上記の分類基準となる数値は，各種計算機における無効化コストの差に応じて調整すべき数値であり，本評価環境では実験の結果から有効だと判断している．

例として，6.8 節の評価で使用しているカウンタ付き 2 分木の節点（図 6.6 左）のクラス分割を考える．表 6.7 にアクセス情報を示す．表の Read・Write はそれぞれ各フィールドの読み出し・書き込みアクセス数を表す．また，Read 数の T はそのクラスの全アクセス数（読み出し + 書き込み）に対する各フィールドの読み出しアクセス数の割合を表す．一方，Write 数の W はそのクラスの全書き込みアクセス数に対する各フィールドの書き込みアクセス数の割合を表す．そして，Fields の括弧内の記号は，それぞれ上記の分類を表す．表より，W 変数である `found`，`counter` と N 変数である `lock` とを Heavy クラス（`HvyNode`）とし，R 変数である `val`，`left`，`right` と Heavy クラスへの参照を加えて Light クラス（`Node`）とする（図 6.6 右）.

```
// Original class
class Node {
  int val;
  Node left, right;
  int counter, found;
  Object lock;
}
```

```
// Light class
class Node {
  HvyNode ref;
  int val;
  Node left, right;
}
// Heavy class
class HvyNode {
  int counter, found;
  Object lock;
}
```

図 6.6: 節点のデータ構造 (カウンタ付き 2 分木)

6.8.3 カウンタ付き 2 分木

まず，単純な並列プログラムであるカウンタ付き 2 分木を用いて本プロトタイプの有効性を評価する．加えて，クラス分割やアクセス傾向別領域の導入により加わるコストの評価を行う．

プログラム概要：入力データを各スレッドで分担して並列に木への挿入を行い，木の完成後，自スレッドが挿入した値を探索するプログラムである．節点は通常の 2 分木の節点にカウンタを加えたもので，挿入時に節点と同じ値のデータが来た場合は，その値の出現回数を数えるカウンタ（counter）のインクリメントを行う．また，探索の際に見つかった回数を記録するためのカウンタ（found）も持つ（図 6.6）．従って，複数のスレッドが並列に挿入・探索する場合，これらのカウンタへの書き込みによるキャッシュの無効化が発生する．入力データは，重複した値が 3〜4 つ含まれるランダムな整数 100 万データを用いる．生成節点数は約 31.6 万であり，挿入・探索時に 1 つの節点あたり 3〜4 回カウンタがインクリメントされる．

節点のアクセス数とクラス分割については，3 節で説明済みであるため省略する．節点のサイズは 32 バイトであり，分割により FreqRead: 24 バイト，Other: 24 バイトとなり，L2 キャッシュラインの半分以下のサイズである．

実行においては，Java VM のヒープサイズを 32 MB に固定した．これは各 VM の GC 回数の違いによる測定結果の差を回避するためで，各 VM ともに挿入時に 1 回 GC が生じるサイズである．探索時に GC は生じない．今回の様に，新・旧空間ならびに TLE のサイズを明示指定しない場合，HotSpot Server VM では，New: 10.625 MB, Old: 21.375 MB となる．また，TLE サイズは初

表 6.8: カウンタ付き 2 分木の実行結果

上段 insert(gc)[sec], 下段 search [sec])

#cpus		1	2	4	8	16
Enter-prise	breadth(no-split)	6.11(0.90)	4.13(0.90)	2.94(0.89)	2.26(0.88)	1.94(0.85)
	depth(no-split)	5.50(0.89)	3.78(0.90)	2.78(0.89)	2.16(0.88)	1.85(0.85)
	FRdepth(split)	5.79(0.97)	3.88(0.95)	2.83(0.94)	2.24(0.90)	2.01(0.92)
Fire	breadth(no-split)	3.27(0.52)	2.30(0.52)	1.60(0.51)	1.32(0.49)	1.11(0.48)
	depth(no-split)	2.74(0.46)	1.97(0.46)	1.44(0.46)	1.14(0.45)	1.09(0.44)
	FRdepth(split)	3.30(0.50)	2.17(0.47)	1.52(0.48)	1.19(0.45)	1.07(0.44)

Enter-prise	breadth(no-split)	5.36	3.08	1.77	1.05	0.66
	depth(no-split)	4.02	2.28	1.29	0.78	0.50
	FRdepth(split)	4.40	2.28	1.24	0.74	0.45
Fire	breadth(no-split)	3.36	1.94	1.12	0.67	0.47
	depth(no-split)	2.38	1.36	0.80	0.45	0.35
	FRdepth(split)	2.71	1.45	0.78	0.45	0.32

期値 (8KB) と最大値 (Eden サイズから，この例では 20.75 KB に定まる) の間で自動調節されており，頻繁に TLE 要求するスレッドにはサイズを大きく設定する工夫が行われている．

評価：実行結果を表 6.8 に示す．表の上段が Enterprise での，下段が Fire での結果である．表の各値は順に挿入時間，その内の GC 時間，探索時間を表す (10 回実行の平均．単位は秒)．no-split はクラス分割を行っていない場合，split は分割した場合である．

breadth(no-split)，depth(no-split)，FRdepth(split) の実行時間を比較すると，全体の傾向として，挿入時間は depth が一番良く，探索時間は，無効化の生じない 1CPU では depth が速いが，CPU 数が増加すると，FRdepth の方が速くなっている．各 CPU 台数時における breadth と比較すると，Enterprise においては，

- depth(no-split) では，挿入時間が 5〜10%，探索時間が 25〜27%向上している．

- FRdepth(split) では，16 CPU を除いて，挿入時間が 0〜6%，探索時間が 18〜32%向上している．

一方，Fire においては，

表 6.9: カウンタ付き 2 分木の実行結果：オーバヘッド評価

上段 insert(gc)[sec], 下段 search [sec]

#cpus		1	2	4	8	16
Enter-prise	depth(no-split)	5.50(0.89)	3.78(0.90)	2.78(0.89)	2.16(0.88)	1.85(0.85)
	depth(split)	5.98(0.91)	4.06(0.90)	2.95(0.91)	2.33(0.90)	2.05(0.88)
	FRdepth(no-split)	5.75(0.95)	4.01(0.94)	2.90(0.94)	2.25(0.93)	1.87(0.83)
Fire	depth(no-split)	2.74(0.46)	1.97(0.46)	1.44(0.46)	1.14(0.45)	1.09(0.44)
	depth(split)	3.59(0.45)	2.37(0.45)	1.62(0.45)	1.23(0.44)	1.14(0.44)
	FRdepth(no-split)	3.00(0.51)	2.18(0.51)	1.57(0.51)	1.24(0.49)	1.02(0.41)
Enter-prise	depth(no-split)	4.02	2.28	1.29	0.78	0.50
	depth(split)	4.44	2.51	1.42	0.85	0.54
	FRdepth(no-split)	4.26	2.46	1.43	0.85	0.55
Fire	depth(no-split)	2.38	1.36	0.80	0.45	0.35
	depth(split)	3.03	1.65	0.90	0.55	0.39
	FRdepth(no-split)	2.73	1.81	1.06	0.53	0.41

- depth(no-split) では，挿入時間が 2〜16%，探索時間が 26〜33%向上している．

- FRdepth(split) では，1CPU を除いて，挿入時間が 4〜10%，探索時間が 19〜33%向上している．

depth と FRdepth との挿入時間の差は，以下で示すオーバヘッドが加わっているためと考える．また，CPU 数が多い時には探索において depth より FRdepth の方が高速であり，無効化の影響を回避した効果と言える．

まとめると，2 分木構造においては深さ優先方式による配置は有効である．また，固定レイアウト法を適用した場合，Other オブジェクトの生成・コピーのコストが加わる．従って，2 分木の構築と探索に分かれている場合，構築時に加わるコスト以上の性能向上を探索時に得ることができれば有効である．

オーバヘッド評価：クラス分割の有無やアクセス傾向別配置の有無による差を比較するため，depth(no-split) と depth(split) や FRdepth(no-split) の比較を行う（表 6.9）．まず，クラス分割によりメモリ割当て・GC 時に加わる Other オブジェクトの生成・コピーのコストを，depth 上でのクラス分割の有無による実行結果の差を見ることで行う．同 CPU 台数において depth(no-split) と depth(split) とを比較すると，depth(split) は，Enterprise では挿入時間は 6〜11%，探索時間は 9〜10% 悪化しており，Fire では挿入時間は 5〜31%，探索時

```
// Original class
class Node {
  Node c0, c1, c2, c3,
       c4, c5, c6, c7;
  boolean leaf;
  double x, y, z;
  double region;
  double cx, cy,cz;
  double mass;
}
```

```
// Heavy class
class HvyNode {
  double x, y, z;
}
// Light class
class Node {
  HvyNode ref;
  Node c0, c1, c2, c3,
       c4, c5, c6, c7;
  boolean leaf;
  double region;
  double cx, cy,cz;
  double mass;
}
```

図 6.7: class Node (Nbody Program)

間は 11〜27%悪化している．この差はそのままクラス分割により加わったコストである．

次に，FRdepth の実行時の FreqRead 判定・配置のコストを見る．同じ深さ優先方式を採用している depth(no-split) と FRdepth(no-split) とを比較すると，Enterprise では挿入時間は 1〜6%，探索時間は 6〜11%の差があり，Fire では挿入時間は 5〜31%，探索時間は 11〜27%の差がある．FRdepth(no-split) の実行においては，分割していない節点クラスを FreqRead としている．従って，各スレッドは通常の TLE に加えて FRC を確保してメモリ割当てを行うため，From 空間が尽きる時期が若干早くなる．今回の測定においてはヒープサイズを 32MB に固定しているため，TLE のみを使用する depth と比べて GC が起動されるまでの時間が短い．つまり，depth と比べて生成節点数が少ない段階で GC が生じるため，深さ優先順に並んでいる節点も少なくなる．この違いが，挿入・探索時間の差の主な理由と考える．

6.8.4 Nbody

プログラム概要：質点の運動をシミュレーションする N 体問題プログラムである．アルゴリズムとして Barnes-Hut 法 [12] を利用しており，まず 8 分木を構築し，この木を用いてその後の計算（重心・加速度計算）を行う．木の構築においては節点に対する書き込みが生じるが，計算時におけるアクセスはほとんどが読み出しである．なお，並列化は各スレッド毎に担当する質点を決めて処理することでなされている．

表 6.10: アクセス情報（Nbody: Node）

Fields(type)	Read (M:×10^6)	Write (M:×10^6)
mass(R)	56.3M [T:28%]	0.2M [W:11%]
cx,cy,cz(R)	66.1M [T:33%]	0.6M [W:34%]
region(R)	21.9M [T:11%]	0.1M [W: 9%]
c0,···,c7(R)	48.7M [T:24%]	0.1M [W: 9%]
leaf(R)	4.3M [T: 2%]	0.2M [W:11%]
x,y,z(N)	2.3M [T: 1%]	0.4M [W:26%]
total(R)	199.7M [T:99%]	1.7M [W:100%]

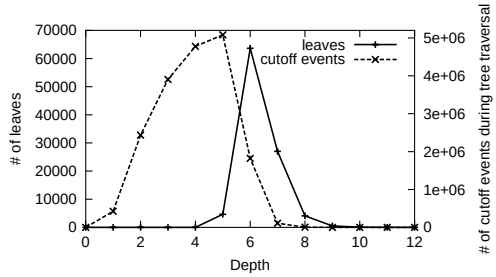

図 6.8: 葉節点の深さと加速度計算時にたどる木の深さ

主要なデータ構造である 8 分木の節点（表 6.7 左）のアクセス数を表 6.10 に示す．1 つの枠に複数のフィールド名が書かれている部分は，それらのアクセス数の合計である．例えば，cx, cy, cz の各アクセス数は表の値の 1/3 である．アクセス数より，`x`，`y`，`z` は N 変数，その他のフィールドは R 変数となり，分割によりそれぞれ Other/FreqRead クラスとなる（図 6.7 右）．節点のサイズは 112 バイトであり，分割により FreqRead: 88 バイト，Other: 32 バイトとなる．従って，分割後も節点のサイズはキャッシュラインよりも大きい．

入力として，x 座標でソート済みの質点 10 万データを使用した．生成節点数は約 14.9 万である．実行においては，Java VM のヒープサイズを 48 MB に固定した．これは，木の構築時に GC が 1 回起動されるサイズである．計算時に GC は起きない．各世代のサイズは New: 16 MB, Old: 32 MB となり，TLE サイズの最大値は 31.25 KB である．

評価：実行結果を表 6.11 に示す．全体の傾向として，木の構築時においては，FRdepth はクラス分割による生成コストが加わっていることもあり一番悪い．

表 6.11: Nbody の実行結果

上段 insert(gc)[sec], 下段 calc [sec])

#cpus		1	2	4	8	16
Enter-prise	breadth(no-split)	4.24(0.59)	2.91(0.61)	1.97(0.61)	1.42(0.63)	1.15(0.63)
	depth(no-split)	4.14(0.55)	2.92(0.56)	1.97(0.57)	1.40(0.58)	1.12(0.59)
	FRdepth(split)	4.40(0.72)	3.09(0.74)	2.14(0.73)	1.55(0.72)	1.22(0.68)
Fire	breadth(no-split)	1.82(0.41)	1.76(0.44)	1.38(0.42)	1.21(0.43)	0.56(0.40)
	depth(no-split)	1.75(0.36)	1.62(0.35)	1.28(0.36)	1.05(0.35)	0.55(0.35)
	FRdepth(split)	1.88(0.43)	1.73(0.44)	1.39(0.44)	1.21(0.44)	0.62(0.41)
Enter-prise	breadth(no-split)	69.87	36.65	21.35	12.27	7.42
	depth(no-split)	72.44	38.10	21.02	12.18	7.53
	FRdepth(split)	71.77	36.79	21.04	12.36	7.55
Fire	breadth(no-split)	24.57	12.91	7.64	4.63	3.35
	depth(no-split)	25.04	13.12	7.61	4.64	3.37
	FRdepth(split)	24.91	13.09	7.14	4.37	3.49

Enterprise では，breadth と depth との構築時の差はあまりないが，若干 depth の方が速い場合がある．また，計算時においても全体的にあまり変わらないが．breadth の方が若干速い場合がある．実際に同 CPU 数 の breadth と比べると，

- depth(no-split) では，構築時間は 0〜3%向上しているが，計算時間は 0〜4%悪化している．
- FRdepth(split) では，構築時間は 4〜9%，計算時間は 0〜3%悪化している．

一方，Fire では，構築時は depth が一番速い．計算時は，逆に breath や FRdepth が速い場合があるが，ばらついている．同 CPU 数の breadth と比して，

- depth(no-split) では，構築時間は 2〜13%向上しているが，計算時間は 0〜2%悪化している．
- FRdepth(split) では，構築時間は 1〜11%，計算時間は 1〜4%悪化している．

つまり，クラス分割の効果があまり出ていない．また，幅優先方式と比較して深さ優先方式は，構築時間はわずかに速度向上しているが，計算時間では速度低下している．つまり，多少のばらつきはあるが，全体として幅優先方式の方が良い結果を示している．以下ではこれらの原因を考察する．

まず，クラスの分割の効果が見られないのは，木の構築時にはオブジェクトの識別・生成コストが加わることに加えて，分割してもキャッシュラインサイズより大きく，FreqRead オブジェクトを詰めて配置する効果が少ないためと考える．なお，6.5 節の評価において効果が見られたのは，実行環境が cc-NUMA 型の並列計算機上であることに加え，サイズの大きな pthread ロック領域をクラス分割により移動できたためと考える．一方，本環境では明示的なロック領域は存在せず，クラス分割の効果は少ない．

次に幅優先方式と深さ優先方式との差を考察する．Barnes-Hut 法では近似計算によりたどる必要のある節点数を減らしており，根付近のアクセス頻度が高い．図 6.8 は葉節点の深さと加速度計算の関係を示している．図の x 軸は根節点を 0 とした深さを表す．実線（左 y 軸）は木の各深さにおける葉節点の数を表し，点線（右 y 軸）は加速度計算時に近似可能として計算を打ち切った（cutoff）回数を表す．図より，深さ 6〜7 に葉節点が多いが，実際に計算時にたどった深さは 3〜5 までが多いことが分かる．つまり，葉節点までたどる前に近似計算可能となることが多く，計算時にたどる必要がある節点は木の上側（根の方）である．幅優先方式の場合，木の上部から下部へと深さ順に節点が並ぶのに対して，深さ優先方式の場合は，親から子へと節点をコピーするため，頻繁にたどる木の根付近の節点とあまりアクセスしない葉付近の節点とがメモリ上に混在した配置となる．そのため，深さ優先方式の場合，プログラムのワーキングセットが増加する配置となり，キャッシュの利用効率が低下していると考える．従って，今回は実装していないが，文献 [47, 30] のような木構造を階層的にコピーする方法が有効である可能性が高い．

6.8.5 MolDyn

プログラム概要:逐次 Java ベンチマークである Java Grande Forum Sequential Benchmarks[4] の MolDyn を，粒子配列をブロック分割することで並列化したものである．粒子配列は全スレッドで共有しており，各スレッドが計算した力を粒子に足し込む際には排他制御を行う．シミュレーションは，8788 個の粒子（SizeB）に対して 50 ステップ実行される．

主なデータ構造である粒子（図 6.9 左）のアクセス情報を表 6.12 に示す（実際には各フィールドは 3 次元であり，表の値は合計アクセス数である）．アクセス数より，`coord` は R 変数，`force`，`velocity` は W, N 変数となり，それ

```
// Original class
class particle {
  double xcoord,
    ycoord, zcoord;
  double xvelocity,
    yvelocity, zvelocity;
  double xforce,
    yforce, zforce;
}
```

```
// Heavy class
class HvyParticle {
  double xvelocity,
    yvelocity, zvelocity;
  double xforce,
    yforce, zforce;
}
// Light class
class particle {
HvyParticle ref;
  double xcoord,
    ycoord, zcoord;
}
```

図 6.9: class Particle (MolDyn Program)

表 6.12: アクセス情報（MolDyn: Particle）

Fields(type)	Read (M:×10^6)	Write (M:×10^6)
coord(R)	11587.0M [T:98%]	1.3M [W: 2%]
force(W)	87.4M [T: 1%]	84.8M [W:95%]
velocity(N)	9.3M [T: 0%]	2.7M [W: 3%]
total(R)	11683.7M [T:99%]	88.9M [W:100%]

ぞれを FreqRead/Other クラスに分割する（図 6.9 右）．粒子のサイズは 80 バイトであり，クラス分割により FreqRead: 40 バイトと Other: 56 バイトとになる．従って，分割前はキャッシュラインサイズよりも大きいが，分割後は 1 つのラインに収まるサイズである．

粒子数は 8788 個（約 0.7 MB）であるため GC は生じず，粒子生成時のメモリ配置が実行時間を決める．なお，粒子の作成は時間測定前に行われ，実行時間には含まれない．

評価：実行結果を表 6.13 に示す．GC は生じないため breadth/depth は共通の結果となる．また，MolDyn においては，粒子（FreqRead）を引数として synchronized ブロックによる排他制御を行っているが，この排他制御を行う際のオブジェクトを Other オブジェクトに変更して実行した結果を表の FRsplit(Olock) に示す．これは，HotSpot VM では文献 [11, 19] と同様にオブジェクトヘッダを利用した排他制御方法を採用しており，排他制御の際にヘッダ対する書き込みを行う．本プログラムでは，force と同数の排他制御操作が行われており，結

表 6.13: MolDyn の実行結果 [sec]

#cpus	1	2	4	8	16
Enterprise					
no-split	414.8	228.0	127.9	75.7	49.9
FRsplit	429.6	233.4	127.3	69.5	42.0
FRsplit(Olock)	434.5	229.9	121.3	62.7	35.0
Fire 12K					
no-split	223.5	138.6	81.49	50.8	34.6
FRsplit	212.7	125.9	76.03	45.4	28.6
FRsplit(Olock)	209.7	115.3	65.08	38.6	22.4

果として無効化が生じるためである．

Enterprise においては，同 CPU 数 の no-split と比べると，1, 2 CPU 台数時を除いて FRsplit は，

- FRsplit では，1〜16%速度向上している．
- FRsplit(Olock) では，5〜30%速度向上している．

一方 Fire においては，

- FRsplit では，5〜17%速度向上している．
- FRsplit(Olock) では，6〜35%速度向上している．

表 6.12 より，読み出しアクセス数が 99%以上もあり，書き込みアクセス数は少ないが，FRsplit においては，CPU 数の増加につれて速度向上率が増加しており，メモリ割当て時に FreqRead/Other オブジェクトを別々の領域に配置することにより無効化の影響を回避していると言える．

FRsplit と FRsplit(Olock) とを比べると，Enterprise においては，無効化の影響のない 1CPU 実行以外では，Olock の方が 1〜17% ，Fire においては 1〜21%高速になっている．このことから，排他制御に FreqRead オブジェクトを用いると FreqRead/Other を分けて配置する効果が失われてしまう結果となる．従って，排他制御もそのオブジェクトに対する書き込みとして扱うなどの対応が必要である．

6.8.6 評価の総括

以上の評価結果をまとめる．

全般的な傾向：オブジェクトサイズがキャッシュラインサイズより大きい場合は効果は小さい．

クラス分割：排他制御操作も含めて，書き込みが多いオブジェクトに対して効果がある．Moldyn においては最大 30%の速度向上を得た（16CPU 時）．

深さ優先方式：オブジェクトサイズがキャッシュラインサイズより小さくプリフェッチ効果が期待できるものは効果がある．ただし，Nbody など深さによってアクセス頻度が大きく変わる場合，単純な深さ優先方式では逆にワーキングセットの増加を招くこともあり得る．

また，正確な測定を行っていないため経験則になるが，オブジェクトの生存期間が短い場合も効果は薄いと考えられる．これは，GC 時の再配置による効果がなく，オブジェクトが共有されることも少ないためである．

次にクラス分割指針をまとめると，クラス分割を適用する際はクラス全体の読み出し・書き込みアクセス比率だけではなく，分割後のオブジェクトサイズに着目する必要がある．今回評価に用いたアプリケーションでは書き込み比率はいずれも 1% 前後であり，加えて分割後 FreqRead サイズがキャッシュサイズに比べて小さい場合はクラス分割が有利に働いていた．但し，以上パラメータは計算環境に依存するものであり，ベンチマークプログラムなどを利用した各計算環境に応じたパラメータ取得方式の検討が今後必要である．また，オブジェクトサイズに関する調査は，例えば 文献 [16] で Java プログラムを対象に行われているが，並列プログラムにおける傾向の調査が必要である．

6.9 議論

本レイアウト法は，対象オブジェクトがクラス階層の一部としての側面をもつ場合に問題がある．一般に親クラスの実行コードを再利用するためには親クラスと子クラスは同形をしている必要があるが，一方，変数アクセス状況は各階層で異なり適切なレイアウトも異なる．レイアウトを親子で同形にしない場合は，各メソッド，変数アクセス部について仮想化をおこなう必要が生じる．この問題は，[16] でも指摘されている．

例えば，Java 言語にこれらの技術を導入する場合，対象クラスと Class Object

(全クラスの親クラス) にも親子関係が存在する．全クラスのレイアウトを同形にしながら今回のレイアウトの利点を享受することは困難であり，Object クラスの機能を対象クラスで再定義することも親クラス部における final method 呼び出しを仮想化することを意味し，問題が多い．

但し，実際には，今回の対象プログラムを Java で記述した場合も対象クラスのインスタンスは親クラスのメソッド内で利用されたことはない．つまり，対象クラスのインスタンスへの参照が外部に漏れていないことが保証できれば，問題は解決されることとなる．

6.10 まとめ

本研究では，並列プログラムのオブジェクトレイアウトの自動最適化を実現するため，オブジェクト内レイアウト法の検討，ならびに自動メモリ管理機構への統合，評価を行った．レイアウト変換は，共有メモリ型並列計算機で問題となるコヒーレンスミス，とりわけ，論理的には関連のないフィールド間のコヒーレンスミスを削減するためのものである．固定レイアウトを選択するものと，レイアウト切替法を提案し，その実システム上での評価を行った．結果，固定レイアウト法は良好な結果をえられたが，一方で切替法が切替時のコスト（キャッシュ無効化を含む）を上回るケースが限られることも確認した．

固定レイアウト法については，既存の Java の処理系である Sun HotSpot VM の自動メモリ管理機構に組み込まれ，複数のアプリケーションを通して評価を行い，その有効性と特性を検討した．同時に，深さ優先方式との併用の効果も測定した．結果，オブジェクトサイズがキャッシュラインサイズより小さく，書き込み比率が多いプログラムに対して高い効果を発揮し，Moldyn においては最大 30%の速度向上を得た（16CPU 時）．今後，各アーキテクチャに応じた R/W 判断基準の自動検出などを通して，システムとしての完成度を向上を目指したいと考える．

第 7 章

結論

本論文では，リンク構造を扱う共有メモリ型並列プログラムの自動最適化を目標とした研究である．リンク構造などの不規則データ構造を共有するようなプログラムにおいては，並列性の抽出や適切な資源管理は大変な仕事であり，プログラムの性質と各種アーキテクチャ特性を把握した熟練プログラマが，知識と労力に頼ったプログラミングを行っていた．そして，アーキテクチャの変遷にしたがって，しばしば再プログラミングが必要とされる．

本研究が目指すのは，まずはプログラマが正しいプログラムを記述してもらい，高速化に関しては処理系の自動最適化サポートのもと，プログラマの知識や実行プロファイルを使ったチューニングを行うというプログラミングスタイルである．

具体的な本研究の内容は以下の通りである．

- 変化するオブジェクトの性質を捉えるため，局面をもちいたフレームワークの提案ならびに解析手法の確立
- 局面解析に基づく排他制御緩和機構の提案ならびに実装評価
- 局面解析により一貫性保証されたキャッシュを用いた分散オブジェクト実装法の提案
- 共有メモリ型並列計算機のためのオブジェクト配置自動化

局面とは，オブジェクトの大きな状態変化をとらえるための枠組みである．従来の解析においては，局所的な内容や静的な情報を取得することはできるが，プログラムの大きな振舞いの変化を捉えることは出来なかった．このため，プログラマ自身が状況の変化に応じた最適化技法を明示的に施す必要があった．本

研究は，プログラマには振舞いの変化するポイントや，状況に関する知識をプログラム上に局面として追記してもらうことで，局面変化や局面毎の性質を処理系が取得し，自動最適化に利用できるようにするためのものである．

局面解析がおこなうのは，各コードブロックに対する実行可能局面の解析と局面遷移の可能性の解析である．この 2 種類の情報は本来互いに依存したものであり，その確定には大域的解析を必要するものである．本研究では解析の高速化のため，メソッド局所解析と情報確定のための大域解析の 2 段階に分離されたアルゴリズムを提案した．

局面解析の応用例の一つは，共有メモリ型プログラムの排他制御緩和への応用である．局面毎の性質に基づいた緩い排他制御規則を適用することのできる一貫性保証構文 `consistent` を提案し，プロトタイプシステムをとおしての評価をおこなった．本方式の基本的アプローチは，ある局面で更新がないと保証された変数へのアクセスは当該局面内では排他制御なしで実行し，今後一切更新が無いと保証できた場合は一切の排他制御を削除するというものである．局面解析情報ならびに局面毎のアクセス情報を利用することで，排他制御ブロックを 4 種類に分類し，緩和された排他制御ルールを適用している．

プロトタイプシステム上で，いくつかのアプリケーション例を評価した結果，プログラムが単純な一貫性保証の宣言を行った場合であっても，ユーザの局面記述の内容によっては排他制御緩和，ボトルネック解消が可能であった．特に，Iimmutable と解析され一切の排他制御の削減を行えた場合は手動最適化を行ったプログラムに匹敵する実行速度を達成している．一方で，本アプローチを広く利用するためには，複合的なオブジェクトの扱いや，詳細な参照解析が必要であるという知見を得た．

局面解析のもう一つの応用例は，共有メモリ型プログラムの分散環境への移植に向けた効率的分散オブジェクト実装である．共有メモリ型プログラムを分散環境で効率的に実行するためには，複数のホストで共有されたオブジェクトの実現方法が重要である．最近良く利用される分散オブジェクト技術のようにデータを単一ホストだけに配置したのでは十分ではなく，キャッシュを利用した効率的分散オブジェクトの実現が望ましい．

本研究は，上記効率的分散オブジェクトの自動生成を目的として，分散コレクションライブラリの提案と局面解析を利用したキャッシュ付プロキシの実現法を提案している．基本的アプローチは，ある局面で更新がないと分かった変数

をプロキシに配置して利用するというものである．加えて，メソッド呼出し解析に対応したアクセス解析手法の提案，局面によるメソッドの Code Versioning 手法の提案を行うことで，解析精度の向上やプロキシ利用機会の拡大を図っている．本研究については残念ながら，モデルならびに解析手法の提案で終わっているが，プログラムサンプルに対するケーススタディでは，各種最適化のための情報の取得に成功している．今後，システムの完成に向けて重要な理論的足掛かりが出来たと言える．

以上の一連の局面解析を用いた研究は，従来の解析技術では得られなかった情報取得し，排他制御緩和や一貫性のあるキャッシュの実現と言った自動最適化への利用を可能にしたものである．今後，さらなる解析精度の向上を図り，実システムへの適用を行っていきたい．

最後の研究は，共有メモリ型並列計算機上でのオブジェクト配置の自動化である．共有メモリ型並列計算機であっても，資源配置はしばしば性能に大きな影響を与え，特に，キャッシュのコヒーレンスミスは大きな問題となっていた．このため，しばしばプログラマが性能向上のための明示的メモリ配置をおこっていた．本研究は，プログラムの傾向を実行プロファイルから取得することで，適切なレイアウト配置に自動変換するという研究である．

本研究では，共有メモリ型並列計算機むけのオブジェクトレイアウト，ならびに，自動メモリ管理機構への統合を行っている．レイアウト変換における基本アプローチは，頻繁に読み出されるフィールドは集中配置し，よく書き込まれるフィールドとは分離して配置するというものである．そのために，必要に応じてオブジェクト分割によるフィールドの分離配置を行っている．また，本レイアウト方式は実際の Java 処理系上への統合を行っており，SUN HotSpot VM 上にアクセス傾向別領域を導入することでその実現を行った．

複数のアプリケーションを通しての評価では，書き込み比率が高々 1 % 程度であっても速度向上が観測され，Moldyn ベンチマークでは 16CPU 時に最大 30%の速度向上を得た．また，これらの評価によりオブジェクトレイアウト指針に関する多くの知見を得ることができた．これらの評価は実システム上で通常アプリケーションを対象に行ったものであり，実用性の面からも意義深いと考える．

本研究に関する発表論文

1. 鎌田十三郎, 八杉 昌宏. 適応的オブジェクトのための局面解析手法. 情報処理学会論文誌：プログラミング Vol. 44, No. SIG2(PRO16), pp 13-24, Feb., 2003

2. 安永雅典, 鎌田十三郎, 八杉昌宏, 瀧和男. 局面解析を利用した排他制御緩和機構. Proc. of JSPP 2002, pp 245-252, 2002

3. 前田昌樹, 鎌田十三郎, 瀧 和男. 共有メモリ型並列計算機におけるキャッシュを意識したオブジェクト内レイアウト法. JSPP2001, pp 149-156, 2001

本研究に関する参考論文

1. 松田聡, 鎌田 十三郎. 並列計算機におけるキャッシュを意識した自動メモリ管理機構. 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 11(ACS 3), pp 126-136, Aug. 2003,

2. 松田聡, 鎌田 十三郎. 並列計算機におけるキャッシュを意識した自動メモリ管理機構. In Proc. of SACSIS, pp 57-64, May, 2003

参考文献

[1] Beowulf project. http://www.beowulf.org/.

[2] *The "Double-Checked Locking is Broken" Declaration*. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[3] Horb. http://horb.a02.aist.go.jp/horb-j/.

[4] *The Java Grande Forum Benchmark Suite*. http://www.epcc.ed.ac.uk/computing/research_activities /java_grande/index_1.html.

[5] Java remote method invocation - distributed computing for java. White Paper, http://java.sun.com/marketing/collateral/javarmi.html.

[6] The message passing interface (mpi) standard.

[7] Ninf project. http://ninf.apgrid.org/.

[8] *Seasonal Sync*. http://www.cs26.scitec.kobe-u.ac.jp/~pl/seasonal/sample.html#nbody.

[9] Top 500 supercomputer site. http://www.top500.org/.

[10] *Tuning Garbage Collection with the 1.3.1 Java^{TM} Virtual Machine*. http://java.sun.com/docs/hotspot/gc/index.html.

[11] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In *OOPSLA*, pp. 207–222, Oct 1999.

[12] Josh Barnes and Piet Hut. A Hierarchical $O(N \log N)$ Force-Calculation Algorithm. *Nature*, Vol. 324, pp. 446–449, 1986.

[13] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *Proc. of OOPSLA '99*, pp. 20–34, 1999.

[14] Jeff Bogda and Urs Holzle. Removing unnecessary synchronization in java. In *Proc. of OOPSLA '99*, pp. 35–46, 1999.

[15] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, Vol. 13, No. 11, pp. 677–678, Nov 1970.

[16] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *PLDI*, pp. 13–24, May 1999.

[17] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ISMM*, pp. 37–48, Oct 1998.

[18] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proc. of OOPSLA '99*, pp. 1–19, 1999.

[19] David Dice. Implementing fast java^{TM} monitors with relaxed-locks. In *the Java^{TM} Virtual Machine Research and Technology Symposium(JVM'01)*, Apr 2001.

[20] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. In *Proc. of Workshop on Environments and Tools*, 1996.

[21] Ian Foster and Carl Kesseleman. *The Grid: Blueprint for a New Computing Infrastracture*. Morgan Kaufmann, 1998.

[22] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification - Chapter17 Threads and Locks*. http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html.

[23] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proc. of PLDI*, pp. 141–152, 2002.

[24] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sugimoto, and T. Takahashi. Dynamic home node reallocation on software distributed shared memory. In *Proc. of HPC Asia 2000*, pp. 158–163, 2000.

[25] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proc. of PLDI*, pp. 249–260, 1992.

[26] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. pp. 35–47, 1990.

[27] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. of Workshop on Program Analysis for Software Tools and Engineering(PASTE)*, 2001.

[28] *JAVA HOTSPOT VIRTUAL MACHINE[tm] Sun Community Source Licensing*. http://wwws.sun.com/software/communitysource/hotspot/download.html.

[29] Yuuji Ichisugi. *EPP: An Extensible Java Pre-Processor Kit*. http://staff.aist.go.jp/y-ichisugi/epp/.

[30] 伊藤智一, 八杉昌宏, 小宮常康, 湯淺太一. 局所性を高める階層的コピー gc 方式. 日本ソフトウェア科学会第 19 回大会論文集, 2002.

[31] Dennis G. Kafura and Keung Hae Lee. Inheritance in actor based concurrent object-oriented languages. In *Proc. of ECOOP '89*, pp. 131–145, 1989.

[32] James Laudon and Daniel Lenoski. The sgi origin: A ccnuma highly scalable server. pp. 241–251, 1997.

[33] Satoshi Matsuoka, Kenjiro Taura, and Akinori Yonezawa. Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages. In *Proc. of OOPSLA '93*, pp. 109–126, 1993.

[34] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. Gridrpc: A remote procedure call api for grid computing.

GWD-I, APM Research Group, 2002. http://www.eece.unm.edu/~apm/docs/APM_GridRPC_0702.pdf.

[35] R. B. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 1, March 1992.

[36] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *Proc. of PPOPP*, pp. 133–144, 1991.

[37] Tamiya Onodera and Kiyokuni Kawachiya. A study of locking objects with bimodal fields. In *Proc. of OOPSLA*.

[38] Incrementalized Pointer and Escape Analysis. Frederic vivien and martin rinard. In *Proc. of PLDI*, pp. 35–46, 2001.

[39] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Proc. of ACM SIGPLAN symposium on Principles and practices of parallel programming*, pp. 12–23, 2001.

[40] Michael L. Scott and William N. Scherer. Scalable queue-based spin locks with timeout. In *Proc. of ACM SIGPLAN symposium on Principles and practices of parallel programming*, pp. 44 – 52, 2001.

[41] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of java applications at allocation and garbage collection times. In *OOPSLA*, pp. 13–25, Nov 2002.

[42] Yukihiko Sohda, Hidemoto Nakada, Satoshi Matsuoka, , and Hirotaka Ogawa. Implementation of a portable software dsm in java. In *Proc. of ACM JavaGrande/ISCOPE 2001 Conference*, 2001.

[43] Kenjiro Taura and Akinori Yonezawa. Schematic: A concurrent object-oriented extension to scheme. Technical report, University of Tokyo, 1996.

[44] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proc. of In Symposium on Principles of Programming Languages(POPL)*, pp. 188–201, 1994.

[45] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *Proc. of OOPSLA '89*, pp. 103–112, 1989.

[46] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proc. of OOPSLA '99*, pp. 187–206, 1999.

[47] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *PLDI*, pp. 177–191, 1991.

[48] 八杉昌宏, 伊藤智一, 小宮常康, 湯淺太一. 少量のスタックで大部分を深さ優先順にコピーするゴミ集め方式. In *SPA*, 2000.

[49] W. Yu and A. Cox. Java/dsm: a platform for heterogeneous computing. In *Proc. of ACM 1997 Workshop on Java for Science and Engineering Computation*, Vol. 43.2, pp. 65–78, 1997.

[50] 江口 重行, 八杉 昌宏, 鎌田 十三郎, 瀧 和男. 適応的オブジェクトによる排他制御の実行時緩和. 情報処理学会論文誌, Vol. 40, No. 5, pp. 2084–2092, May 1999.

[51] 佐藤三久, 朴泰祐, 高橋大介. Omnirpc: グリッド環境での並列プログラミングのための grid rpc システム. In *SACSIS 2003*, pp. 105–112, 2003.

謝辞

まずは、本論文の審査にあたっていただいております神戸大学大学院自然科学研究科 金田悠紀夫教授、田村直之教授、増田澄男教授に感謝致します。また、神戸大学助手に就任して以来、自由な研究の場を与えて頂きました神戸大学工学部瀧和男教授（現エイ・アイ・エル株式会社代表取締役）に感謝致します。また、引き続き研究の場を提供して頂きました神戸大学工学部 永田真助教授に感謝致します。ともすれば仕事を先延ししがちな著者に対し、お二人は論文提出を心配し、ご指導して頂きました。大変感謝しております。

また、著者が研究活動を開始する場を与えて頂きました東京大学 米澤明憲教授には大変感謝しております。本来、博士課程終了後すぐにでも論文提出を行う予定でしたが、途中で研究テーマを変更し、今に至っております。ご迷惑をお掛け致しました。米澤研究室における研究活動の活気が、当時プログラミングをはじめて 1-2 年の著者を研究者の道に導いたと考えています。また、著者のプログラミング言語研究の基礎となるのも、米澤研究室時代の経験だと思っています。当時、助手として指導して頂きました松岡聡先生（現東京工業大学教授）、先輩として指導して頂きました小林直樹様、八杉昌宏様、田浦健次朗様、増原英彦様にも感謝致します。特に、現京都大学助教授の八杉様には、著者が神戸大学に来てからも共同研究などを通して、暖かい指導や助言を頂きました。また、同期として学生生活を過ごさせて頂きました関口龍郎様、高橋俊行様、細部博史様、今野和浩様に感謝致します。

最後になりますが、神戸大学に来てから一緒に研究させて頂きました神戸大学工学部情報知能工学科 26 講座の方々に感謝したいと思います。本研究内容は、主に著者が神戸大学に来てからのものです。共著者にもなっている前田昌樹様、安永雅典様、松田聡様には特に感謝したいと思います。著者の思いつきに振り回されながらも、良く一緒に研究活動を続けてくれました。ありがとうございました。