モード切替機構を持つ分散環境向けJava集合ライブラリの提案

鎌田 十三郎[†] 森本 昌治[††] 二ッ森 大介[††]

最近，PC クラスタなどの分散計算環境がその高い計算能力で注目を集めているが，不規則な共有データを扱う並列プログラムの効率化には，かなりの労力を必要とする．本論文では，集合などを含んだリンクデータ構造を分散環境で共有するための計算環境を提案する．本環境上では，プログラマは配列やハッシュや木構造を含む共有データを，PE 間にまたがった形で配置し，加えて，計算局面に応じて各データのキャッシュを他の PE に配置するかどうか，オブジェクトフィールド単位で指定可能である．各 PE ではキャッシュ保持のためのプロキシが準備される．このため，共有オブジェクトイメージのもとで，本体データおよびキャッシュを利用した効率的な局所計算を行うことができる．一方で，キャッシュがない場合に備えて，プロキシはオブジェクト本体への遠隔メソッド呼び出し機構も提供する．つまり，共通のオブジェクトイメージ上で，MPI の Collective 通信のような明示的データ再配置機能と，必要に応じて遠隔アクセスを行うための機構を用いたプログラミングが可能となる．また，逐次プログラムからの移植などにおいても，オブジェクトの構造自体をほとんど変更することなく，データ再配置を実現することができる．本手法の有効性については，実アプリケーションを通して，記述面ならびに試験実装システム上での実行性能の評価を行う．

A Proposal of Java Distributed Collection Library with Multiple Execution Mode for Distributed Memory Environments

Tomio Kamada,[†] Masaharu Morimoto[††]
and Daisuke Futatsumori[††]

On distributed computing environments, parallel programs for regular or divide & conquer type computation show high performance. However, for parallel programs that treat shared and linked data structures, much effort is required to attain efficient access to shared data. This paper proposes a computing environment to share linked data structures with multiple processing elements (PEs), using library for distributed data collections such as Set, Hash, and Trees. This system allows data structures that consist of elements allocated at different PEs, and offers object caching facilities where proxy objects are allocated on all the PEs to cache the object status. At each class definition, the programmer can specify whether each field should be cached for each *calculation phase* of the program, to manage the range of data fields that each PE covers. To cope with the situation where PE irregularly wants to access data fields that are not covered with, the proxy object prepares remote method invocation mechanism which delegates calling request to the owner PE of the object. To evaluate the availability of our proposed system, we transport some sequential programs to distributed environments using our prototype system, and measures the performance of the program execution.

1. はじめに

最近，PC クラスタなどの分散計算環境がその高い計算能力で注目を集めている[5]．ただし，その計算対象は，行列などの規則計算や分割統治型の計算がほとんどであり，不規則な共有データを扱った並列プログラムが取り扱われることは少ない．

一般に，分散環境において共有データを扱う場合，対象データの性質の変化に基づいて，担当 PE（Processing Element）を決めて更新作業を行ったり，複数の PE にデータをキャッシュしたりする必要がある．このため，変数の性質に基づいたデータ再配置操作が必要となる．ただし，対象がリンク構造などの場合，再配置操作が煩雑であったり，また，データアクセスパターンが不規則だったりする場合，どのようなデータ再配置が必要か，静的に確定しない場合もある．

† 神戸大学工学部情報知能工学科
Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University
†† 神戸大学大学院自然科学研究科情報知能工学専攻
Graduate School of Science and Technology, Kobe University

現在，効率的な分散プログラムを書くのに主に利用されるのは，MPI などのメッセージ通信に基づくモデルである．プログラマはデータ構造の性質を理解したうえで，その配置を明示的に指示する．結果，PE 間の通信を抑えた効率的な処理を行うことができる．ただし，データ構造の性質を勘違いしていたり，再配置データの範囲を間違えたりすると，しばしば原因不明のバグとして悩まされることになる．

もう 1 つのプログラミング方法は分散共有メモリを利用するものである．実行時にアクセス要求に応じたデータ配置をシステムが行い，データ再配置のミスを起こす心配はないが，当然アクセス遅延の問題が生じる．また，MPI の Collective 通信のように PE 間で協調したデータ転送を実現するのは難しい．

本論文が目指すのは，

- プログラマには共有メモリイメージに近いプログラミングモデルを提供し，各種集合データ構造もシステムが提供する，
- プログラマは，計算局面に応じた各フィールドや集合データの性質を記述し，それをもとにシステムがキャッシュ配置やデータ移動を実現する，

という計算環境である．つまり，プログラマが各データ構造に関して局面ごとのアクセス方針を指定することで，システムが MPI の Collective 通信と同様のデータ移動を実現する．また，一般のリンクデータ構造を対象としたデザインを行う．

以下では，まず，2 章で従来の分散プログラミング環境について簡単にまとめ，その後，3 章で本提案手法について概説する．4 章で基本的なプログラミングモデルについて，5 章で集合データ構造について紹介する．その後，本方式に基づいて逐次プログラムを分散環境に移植する作業を，6 章では記述面について紹介し，7 章では試験的実装に基づく実行性能評価を行う．8 章で本手法の課題や問題点を述べたうえで，9 章でまとめを行う．

2. 既存のプログラミング手法

2.1 MPI の利用

ある配列データ構造を複数の PE 間で共有して計算を進める場合，MPI[3] の Collective 通信は有効な手段である．たとえば，配列 `a[]` を共有する場合，単純には，各 PE 上に相当の領域を確保し，計算の進行状況に応じて Collective 通信によるデータ再配置を行えばよい．`a[]` の要素が並列に read される場合は，全 PE にデータをコピーして配置する．一方，PE 間で分担箇所を決めて要素の更新を行う場合は，たとえば Scatter/Gather 系の通信を利用可能である．また，ある部分について複数 PE で計算を行った場合は，結果を集計する目的で Reduce 系通信を利用できる．

また，一般の MPI library はプリミティブデータ配列を対象としたものであるが，mpiJava[4),6)] などの Java 上の MPI 処理系では，Serialize 機能を併用することで，配列要素がオブジェクトの場合であっても，データ再配置が可能である．

では，集合などの共有データ構造を扱う Java のプログラムを，簡単に mpiJava などで並列化できるかというと，必ずしもそうとは限らない．配列要素がオブジェクトの場合，各フィールドのアクセス傾向が異なることも多い．たとえば，Java Grande Benchmark[1)] の逐次版および MPI 版 Moldyn プログラムにおいて，粒子は位置（`coord`），速度（`velocity`），力（`force`）を備えたデータ構造である．ただし，力の計算段階終了時に送受信したいデータは，`force` だけである．このため，MPI 版プログラムでは，`force` を配列に詰めなおしたうえで通信が行われている．

データアクセスが不規則な場合も問題が生じる可能性がある．たとえば，PE 間でデータ担当場所を分担したが，稀に別 PE の担当領域にアクセスする必要が生じたとする．この場合，別 PE にデータ要求もしくは計算依頼を行うために，メッセージ送受信を行うことになる．ただし，MPI では遠隔問合せ機構が整備されておらず，受信側で定期的にメッセージを polling するようなプログラム記述が必要となる．加えて，ハッシュや木構造といった不規則データ構造を，ホスト間にまたがった形で共有する枠組みも提供されていない．

2.2 分散共有メモリ

共有データを扱う並列プログラムを実現するもう 1 つの方法は，分散共有メモリを利用する方法である．

まず，プログラム記述面の検討を行う．分散共有メモリ機構の中には，純粋な共有メモリ空間を提供するもの以外に，共有メモリ空間とローカルオブジェクトの 2 種類のデータ構造から構成されるタイプがある．前者の純粋な共有メモリの実現には，ページ単位のデータ管理を行うことが多い[14)]．一方で，後者の例である JDSM[13)] や Jini[2)] の JavaSpaces では，共有空間相当のオブジェクトに，一般のローカルオブジェクトを格納して利用する．このタイプの処理系では，オブジェクトのコピーを転送することで PE 間のデータ共有を実現することが多い．たとえば JavaSpaces の場合，共有空間へのデータ格納・取得には，オブジェクトのコピーを用いると意味付けされている．このため，参照の分散透明性が確保されていない．

いずれの方式であっても，各 PE が共有空間上のオブジェクトにアクセスすると，必要に応じて共有空間から対応データの取得が行われ，更新を行った場合は最後に書き戻す操作が必要となる．ただし，分散環境では排他制御は PE 間通信をともなう遅い操作であり，ボトルネックを生じやすい．このため，`synchronized` 構文などを用いる代わりに，共有空間からのデータ読み出しや書き込みに関して，write の有無を意識した明示的キャッシュ操作をプログラマ自身に行わせるのが，分散共有メモリ機構の API として一般的である[8]．

次に速度面の特徴について概観する．最初の問題点としてあげられるのは，データアクセス要求に応じてデータ移動を行うため，アクセス遅延が生じる可能性が高い点である．ただし，これに関しては事前にデータアクセスパターンを測定するための Inspector を実行させることで，予測に基づいたキャッシュ配送による効率化も検討されている[10),12)]．

もう 1 つの問題点は，高速化にむけて，しばしばデータ構造の変更が必要とされる点である．MPI 同様，オブジェクトのフィールドすべてが同じアクセスパターンであるとは限らない．このため，データ転送をオブジェクト単位やページ単位で行うと，無駄に大きなデータ転送が必要となる場合がある．さらに大きな問題は false share の可能性である．この問題は，特にページ単位でデータ管理する場合に深刻で，たとえば複数 PE が別のアドレスに書き込んでいたとしても，そのアドレスが同一ページに存在するだけでページ所有権を争うことになる．このような問題を避けるためには，データ構造の変更，ならびにオブジェクトレイアウトのチューニングが必要となる．

一方で，オブジェクト単位で共有メモリへの登録を行った場合，上述したように共有メモリ空間の透明性は得られない．加えて，JavaSpaces のようにコピーによる意味付けを行った場合，仮に同じ PE に存在したとしても，共有メモリ空間内のオブジェクトへのアクセスにはコピーのオーバヘッドが発生する．

2.3 比　　較

分散環境上で共有データを扱うための計算環境として MPI と分散共有メモリについて概観してきたが，効率的なプログラムを実現するうえで共通するのは

- データへの読み書きの有無を把握したうえで，データの整合性をプログラマ自身が保証し，
- 同様にアクセス傾向を把握したうえで，効率化のためのデータ移動もしくはレイアウト変更が必要，

という点である．一方で異なるのは，

- MPI：PE へのデータ配置をプログラマが明示的に指定．ただし，計算中に他の PE 上に配置されたデータを必要とする場合の記述は煩雑．また，配列以外のデータ構造への対応が充実していない．
- 分散共有メモリ：アクセス要求に応じてキャッシュを取得するため，不規則データへの対応は楽である．ただし，アクセス遅延への対応が必要．また，データ配置はキャッシュとして利用され，MPI の Reduce のように積極的に別データとして利用することは少ない．

という点である．

3. アプローチ

本論文では，PC クラスタなどの分散計算環境において，集合などの共有データを扱うための Java 計算環境を提案する．より具体的な目標は，様々な集合データを扱う Java の逐次プログラムを分散環境に移植する方法として，MPI よりも安全かつ容易で，しかも MPI 並みの実行速度を達成できる計算環境を実現することである．

本計算環境でも共有メモリイメージの提供を目指すが，既存の方法同様に，

- データの整合性についてはプログラマが保証する，
- データ配置の指針は，プログラマの知識に頼る，

ことにする．ただし，既存研究の問題を解消するために，以下の方針をとる．

- プログラマは明示的にデータ移動などを指示せず，代わりにデータ配置やキャッシュの方針を指示する．
- 分散環境向けの各種集合データ構造の提供を行う．

提案する計算環境の基本的なイメージは，以下のとおりである（**図 1**）．プログラム中には共有データを保持するためのオブジェクトが各種存在する．これらのオブジェクトは，一般の分散オブジェクト同様の遠隔呼び出し機構に加え，各 PE 上にキャッシュ付きプロキシを配置することができる．このキャッシュは，プログラムの進行に従って全オブジェクトでいっせいに有効化/無効化可能である．つまり，MPI の Collective 通信同様のデータ移動を実現し，各 PE ではキャッシュされたデータを用いた効率的な計算を進めることができる．一方で，キャッシュが利用できない場合は，対応オブジェクトへの遠隔呼び出しを用いることになる．

キャッシュ方針を記述するにあたり，本提案環境では局面を用いた宣言的な記述法を導入する．局面はプログラムの進行状況を表すために用いられ，プログラマは，「初期化局面では，このフィールドは更新をともなうので，キャッシュしない」あるいは，「計算局面で

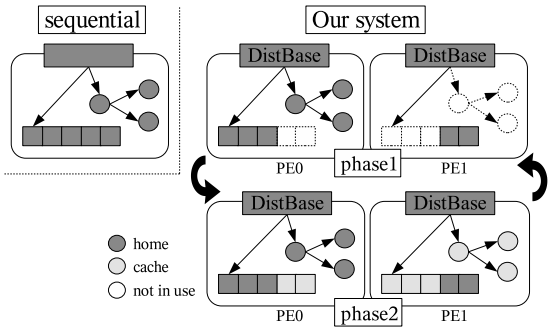


図 1 データ構造イメージ
Fig. 1 Data structure image.

は，このフィールドは更新がないから，キャッシュしても大丈夫」といった指示を各フィールドに対して行う．これにより，データ構造の改変をともなうことなく，フィールド単位のキャッシュ方針を指示することができる．一方で，プログラマがフィールドアクセスに関する知識を記述するということは，プログラマの誤解を検出することにも役立てることができる．つまり,「更新がない」と思っていた局面での更新を，将来的にシステムが検出することができれば，データ整合性に関するバグの発見に有効といえる．

逐次プログラムを分散環境に移植する際は，まずPE 間にまたがる共有メモリ空間を作成し，その中に上記キャッシュ機能を有したオブジェクトを生成する．対象データ構造が集合などの大きなデータ構造であった場合は，本計算環境の提供する配列やハッシュや木構造といった分散環境向けの集合データ構造を利用することができる．各要素データの本体を PE 上に分散して配置することや，キャッシュデータを全 PE に配置することが可能である．プログラマは，各 PE 上のワーカースレッドを生成し，上記分散配置されたデータを対象に並列計算を行うことができる．

4. 実行モデル

4.1 基本データ構造

データ分散に関して，オブジェクトは以下の 4 種類に分かれる．言語記述は，JDK1.5 を基本としている．

- `DistBase< Phase >`：PE 間の共有空間を表す．`Phase` 型の**局面**を有する．次の `DistNode` の一種でもある．
- `DistNode`：PE 間で共通の ID を有する分散オブジェクトで，遠隔参照可能である．各 PE にキャッシュ機能付きプロキシオブジェクトを配置し，フィールド（およびそこから参照されるオブジェクト群）をキャッシュ可能である．また，一部メソッドについてはプロキシ上で実行せず，本体オブジェクトに委譲する機能も持つ．
- `DistCollection`：`DistNode` の一種で，PE にまたがって集合などを保持するデータ構造．データ配置を指定可能（5 章）．
- ローカルオブジェクト：一般のオブジェクト．他の PE への移動はコピーベースで行われる．

```
enum MyPhase implements Phase {
  Init, Others, Calc,
}
class Moldyn extends DistBase<MyPhase> {
  // Partilce の分散配列. 後述
  DistArray<Particle, DistArrayManager> one;
}
class Particle  implements Seasonal<MyPhase> {
  @Dist({@PConf(``Init, Others'', ``HomeOnly''),
         @PConf(``Calc'', ``Cache'')}
    double xcoord, ycoord, zcoord;

  @Dist(/* 同上 */)
    double xvelocity, yvelocity, zvelocity;

  @Dist({@PConf(``Init, Others'', ``HomeOnly''),
         @PConf(``Calc'', ``Reduce(Sum)'')}
    double xforce, yforce, zforce;
}
```

図 2 局面記述例（Moldyn）
Fig. 2 Phase description (sample: Moldyn).

共有データ構造を作成したい場合，プログラマは `DistBase` を継承するクラスを定義し，参加する PE を指定して共有空間を作成する（**図 2**）．これにより `DistBase` のプロキシオブジェクトが各 PE に生成される．`DistBase` の共有空間とは，`DistBase` から参照によってたどることのできる範囲を指す．`DistBase` に格納されたデータについては，プログラマの指示によって各 PE 上のプロキシにキャッシュすることが可能であり，参照先オブジェクトについても再帰的なキャッシュ操作が可能である．

一方で，局面は `enum` データとして定義される．

4.2 適応的データ配置

所属する共有空間 `DistBase` の局面 `Phase` に応じてデータ構造の振舞いを変化させたい場合，その対象クラス（`DistNode` に限らない）に対して，`Seasonal<` `Phase` `>` インタフェースを付加し，フィールドやメソッドの振舞いをメタ修飾の形で規定する．

各局面に応じたフィールドやメソッドの性質は，`@Dist(...)` の形で記述される．図 2 は，2.1 節でも紹介した Moldyn プログラムの例である．フィールドの場合は，その変数のキャッシュ方針（*Mode*）を指定することができ，個々の局面（*Phase*）について `@PConf(`*Phase*`,`*Mode*`)` の形で列挙する．指定されなかった場合は，標準ルールが適用される．

変数に対して現在準備しているモードは，以下の 5

種類である．

- HomeOnly：キャッシュを行わない．標準ルールである．当該局面終了時，プロキシ上のデータ（以下，キャッシュ）は無効として扱う．将来，プロキシ上のアクセスを禁止・検出する機能を導入する予定．
- Cache：当該局面に遷移した時点で，キャッシュを行う．参照フィールドの場合は，再帰的にキャッシュ操作を行う．ただし，前局面でキャッシュ済みの場合は何もしなくてよい．将来，書き込みを禁止・検出する機構を導入予定．
- CacheToDiscard：基本は Cache と同じであるが，別の局面に遷移する際に，キャッシュを無効として扱う．つまり，続く局面で Cache が要求された場合に，再コピーを行う．
- CacheOnDemand：`DistNode` オブジェクトへの参照フィールドについて指定可能．アクセス要求があるまで，参照先 `DistNode` のキャッシュを行わない．
- Reduce(*Ope*)：たとえば，各 PE ごとに値の合計を求め，最後に全 PE の総計を行う場合に利用．振舞いは操作（*Ope*）内容に応じて異なり，`Sum` の場合は，局面遷移時に `0` で初期化し，局面終了時に本体にデータを集計し reduce 操作を行う．終了後のキャッシュは無効として扱われる．

プログラマは，各オブジェクトの各フィールドがどのように扱われるかを把握し，局面設定を行うことになる．たとえば，Moldyn の例では，座標や速度の更新を行う `Others` 局面では，各種変数は `HomeOnly` でよいが，計算局面 `Calc` においては，座標（`coord`）と速度（`velocity`）をもとに力（`force`）を求める必要があるため，`coord`，`velocity` を `Cache` し，全 PE の合計値を求める `force` は `Reduce(Sum)` に設定する．以上のデータの移動は，`DistNode` の `void become(MyPhase)` により局面更新されたタイミングで行われる．たとえば，`Calc` 局面になった時点で `coord`，`velocity` のキャッシュが行われ，`Calc` 局面が終了する時点で `force` に対する `Reduce(Sum)` 操作が行われる．

このように，局面に応じてフィールドの値が有効であるかどうかが変化するので，プロキシにおいてメソッドを実行すべきか否かも同時に変化する．このため，メソッドにも同様のメタ修飾を指定できるが，これについては，4.3 節で解説する．

オブジェクトキャッシュの定義：前述したように，PE 間で共通の ID を有するのは `DistNode` である．よって，データキャッシュ機能も `DistNode` 単位に行われる．また，`CacheOnDemand` は一般のオブジェクトには許されない．キャッシュを行う際は，起点となる `DistNode a` に対して，以下の手順でデータキャッシュ操作を繰り返す．

- 対象オブジェクトが，`Seasonal` であるなら，現在の局面に関し `Cache`，`CacheToDiscard` 指定されているフィールドに対して，以下のフィールドキャッシュ手続きを行う．対象オブジェクトが，`Seasonal` でない場合，`transient` でないすべての field に対して，フィールドキャッシュ手続きを行う．
- フィールドキャッシュ手続き：
  int，double などのプリミティブデータの場合，値をキャッシュ．オブジェクトへの参照である場合，対象がローカルオブジェクトであり，まだキャッシュしていないなら，`DistNode a` のキャッシュとして，再帰的にキャッシュ操作を行う．ただし，参照先が別の `DistNode b` の場合，現在の `a` のキャッシュとは独立に，一連の手続きを行う．

このようなキャッシュ操作が，`DistBase` を起点に `DistNode` 単位に再帰的に行われ，他の PE 上の `DistNode` 対応プロキシに展開される．このため，複数の `DistNode a`，`b` からローカルに参照されるオブジェクト `o` については，キャッシュ作成の際は `a`，`b` それぞれのキャッシュ用に別々のコピーが作成され，送信されることになる．

最後に，現時点では，`Serializable` でないオブジェクトや `private` なフィールドの取扱いについては，まだ決定していない．

4.3 プログラム実行

各 PE で並列に計算を行う際は，たとえば **図 3** のように，`DistBase` の `ParallelExec` 修飾されたメソッドを実行すればよい．これにより，各 PE 上で並列にメソッドが実行され，すべての終了をもってメソッド実行の完了と見なされる．現在，返り値の型は `void` に限定している．これ以外にも，将来的には，他の PE 上で Thread を実行する機構や，`DistCollection` の各要素に対して計算を行う `foreach()` などのメソッド提供を予定している．

また，局面遷移に関しては `become()` メソッドを利用して行う．初期局面は，指定局面（図 3 では，`MyPhase`）の最初に記載された局面値（図 2 の `Init`）となる．

このように各 PE 上で並列に計算が開始され，各 PE 上の本体オブジェクトやプロキシを利用して計算を進めることになる．我々の考える計算の方法は，

- ロック操作や，書き込みのあるフィールドへのア

```
class Moldyn extends DistBase<MyPhase> {
  public void run() {
    .../* Init */
    for(...) {
      become(MyPhase.Othres);
      ..
      become(MyPhase.Calc);
      calc(); // 並列実行
    }
  }
  @Dist({@PConf(‘‘Calc’’, ‘‘ParallelExec’’), ...})
  void calc() {
    int id = hostID();
    int processors = hostNum();
    for(int i = id; i < size; i+= processors)
      for(int j = 0; j < size; j++)
        force(one[i],one[j]);
  }
}
```

図 3 プログラム実行（Moldyn）
Fig. 3 Program description (sample: Moldyn).

クセスは本体オブジェクト上で行う，

- しばらく更新されないフィールドへのアクセスは，プロキシを利用する，

というものである．このため，基本的にメソッドを以下の 3 種類に分類して計算を行う．

- HomeExec：本体実行させるためのモード可能．プロキシ上で呼び出された場合，`DistNode` は本体オブジェクトへ委譲する．一方，ローカルオブジェクトの場合，`TryHomeCallException` が発生することで，本体データの存在する PE 上での再実行を依頼する．
- CacheExec：キャッシュを利用したプロキシ上の実行を許す．
- CacheExecWithRetry：`DistNode` に対してのみ指定可能．プロキシ上での実行を許すが，内部で `TryHomeCallException` が起こった場合，その例外を捕捉したうえで，本体オブジェクトへの遠隔メソッド呼び出しを行う．

プログラマは，キャッシュデータを用いて計算が行えないと思った場合は，`HomeExec` モードとする．一方，キャッシュデータを積極的に利用したい場合は `CacheExec` を実行する．ただし，計算を進めているうちに本体オブジェクトで実行する必要が生じた場合は，その時点で計算を打ち切り，本体呼び出しの形で当該メソッドを再実行してもかまわない．プロキシで実行するのは基本的にはフィールド更新をともなわない計算だからである．**図 4** のプログラムで，`Phase1` では，`DistNode0` の `foo()` メソッドは，ほとんど `then` 節に分岐し，また，アクセス対象もキャッシュ利用可能であったとする．この場合，`foo()` は `CacheExecWithRetry` 指定すべきである．これにより，`else` 節が実行され，`HomeExec` モードにある `LocalObject` の `bar()` を呼

```
class LocalObject implements Seasonal<Phase> {
  // counter が，Phase1 で更新の恐れがある場合
  @Dist({@PConf(‘‘Phase1’’, ‘‘HomeOnly’’)})
    int counter;
  @Dist({@PConf(‘‘Phase1’’, ‘‘HomeExec’’)})
    int bar() { return counter; }
}

class DistNode0 extends DistNode<Phase> {
  @Dist({@PConf(‘‘Phase1’’,’’CacheExecWithRetry’’)})
  int foo() {
    if(....) {
        .....
    } else {
       this.localObject.bar();
    }
  }
}
```

図 4 プロキシ上のメソッド実行（サンプル）
Fig. 4 Usage of `CacheExecWithRetry` method (sample case).

び出された場合も，`TryHomeCallException` が `foo()` によって捕捉され，本体での `foo()` の再実行が行われることになる．

前述以外にも以下のメタ修飾子を準備している．

- AssertOfHomeCallOnly：プログラマが，メソッド呼び出しが本体オブジェクトのある PE でしか起きないと保証．標準ルールとする．将来，保証内容を確認実行するモードを導入する予定．
- AssertOfNoExec：当該局面では実行されないとプログラマが保証．将来，保証内容を確認実行するモードを導入する予定．
- ParallelExec：前述の `DistBase` にのみ許されたメソッド．

最後に，局面更新とメソッド実行が並行に行われた場合について．あるメソッドの実行中に，並行して局面更新が行われたとする．この場合，そのメソッドは局面更新前のキャッシュを利用し続ける可能性がある．元来，プロキシを利用するのは，排他制御などを必要しない計算であり，その意味では以前にキャッシュされたデータを利用する可能性があるのも当然である．

確実に，局面遷移とメソッド実行を排他的に実行したいのであれば，fork-join などを自身で行うか，導入予定の PhaseKeep 機能を利用するとよい．各 PE では，`become(Phase)` メソッド以外に，`acquirePhaseKeep()`，`releasePhaseKeep()` を提供し，`aquire` してから `release` されるまでの期間，`become(Phase)` の実行をブロックすることができる．`become(Phase)` と異なり，`acquire`，release は PE 間通信の必要なく実行され，代わりに become 操作の際に，各 PE の PhaseKeep 状況を見て become 操作を行う．

5. 各種集合データ構造

DistCollection は，複数のデータを分散した状態で保持するため，その整合性確保と実行効率向上のため，各種データ構造に応じた最適化を行う必要がある．

集合データ構造の効率化の研究としては，現在 Jakarta Commons Collections Package[9] に属する FastArrayList などの研究が存在する．この研究は，共有メモリ計算機上で排他制御区間を短くするため，複数の実行モードを備えた集合系ライブラリであるが，分散環境の場合，さらなる効率化の工夫が必要である．

本論文では，DistCollection の例として DistArray，DistSet，DistHash，DistTree を提案する．

DistArray：分散配置された配列データ構造で，サイズは生成時に指定したサイズで固定される．DistArray< T >，DistArrayInt などを提供．以下のような各種モードが存在する．

- 本体配置：各要素の本体を PE に分散配置する指定方法．Block，Cyclic，Centralize を提供．
- データキャッシュ：要素のキャッシュを配置するか否か．HomeOnly，HomeExec，Cache，CacheOnDemand，Reduce(*Ope*) などを提供．

HomeOnly モードでは，i 番目要素の担当 PE でのみ get(i)，put(i，elem) の実行を許し，HomeExec モードでは，担当 PE 以外で呼ばれた場合に，遠隔呼び出しを行う．つまり，get() の返り値や put() の引数については，コピーが送受信される．一方，Cache，CacheOnDemand モードでは，get(i) に対しては i 番目要素のキャッシュを返し，加えて put() の呼び出しを禁止する．最後の Reduce であるが，これは，各 PE でローカルな配列としてアクセスしてもらい，局面終了時に各要素について各々Reduce を行うモードである．

図 5 は Moldyn プログラムにおける DistArray の利用例である．まず，モード指定を行うための DArrayManager がメタ修飾により作られている．DistArray は Moldyn クラス内で利用されており，要素型とともにデータ配置のための DArrayManager が指定されている．注意しておきたいのは，フィールド one に修飾されているのは，Moldyn クラスの one というフィールドのキャッシュ制御であり，DistArray の制御は，DArrayManager によって制御されていることである．このクラスの場合，初期化局面において one ならびにその要素が初期化され，その後，PE 間にデータ分散し，Others，Calc の両計算を繰り返すことになる．

```
@DArray({@PAConf(‘‘Init’’,’’HomeOnly’’,’’Centralize’’),
         @PAConf(‘‘Others’’,’’HomeOnly’’,’’Cyclic’’),
         @PAConf(‘‘Calc’’,’’Cache’’,’’Cyclic’’)})
class DArrayManager implements Seasonal<Phase> {}

class Moldyn extends DistBase<Phase> {
  @Dist({@PConf(‘‘Init’’,’’HomeOnly’’),
         @PConf(‘‘Others, Calc’’,’’Cache’’)})
  DistArray<Particle, DArrayManager> one;

  init() {
    one = new DistArray<Particle, DArrayMnaager>(size);
  }
}
```

図 5 DistArray の利用例 (Moldyn)
Fig. 5 DistArray (sample: Moldyn).

DistSet，DistHash： これらのデータ構造も DistArray 同様，データ分散やキャッシュ配置を指定できる．ただし，index がないため，データ分散指定やデータ重複の可能性に注意が必要となる．

- 本体配置：Centralize，HashBase（ハッシュ値に応じて配置場所を決定），PutBase（後述）
- データキャッシュ：HomeExec，IncreaseOnly，Cache，CacheOnDemand などが提供される．
- データ整合性：PutBase モードでは，データ整合性に関して Optimistic モードを指定可能．

Centralize や HashBase といった本体配置法の場合，各要素がどの PE に配置されるべきか分かるため，その PE で整合性確認をとることができる．一方，PutBase は，put() で登録されるとまずはその PE に配置するモードである．ただし，逐次プログラムと同じような実行を目指すなら，put() の際も重複するデータや hash key が存在しないか，まず全 PE に確認すべきである．ただし，それでは対象データがそもそも重複しえない場合においても，高速な実行が期待できない．このため，put() による重複の可能性を無視した Optimistic モードを導入する．このモードでは，通常モードへの遷移時に要素重複の確認が行われる．get()，remove() についても，まず自 PE 内の検索を行い，対象が見つからない場合にのみ全 PE での検索を行う．

要素データのキャッシュ効率化に関しても，put()，remove() の有無によって状況が異なる．よって，put()，remove() を禁止して，get() 系操作の高速化を図る Cache モード，remove() 系のみを禁止して，要素の CacheOnDemand を行う IncreaseOnly モード，キャッシュの利用を行わない HomeExec モードなどの導入を予定している．

DistTree： 本環境では，木構造の提供も準備している．ただし，今のところ計画しているライブラリは，

使用法にいくらか制限のあるものである．まず，木のルートから 1 つずつ子供を追加する形でしか木を大きくできず，また，できあがった木を切り放して複数の木として扱うことは許していない．これは，現在の実装が TreeNode の ID として，ルートノードからの相対位置を利用しているからである．また，キャッシュ方針としても，複雑なものを準備しておらず，指定した段数のノードを各 PE に分散配置するといった指定法しか提供していない．**図 7** は，6 章で紹介する BH プログラムの記述例である．MyNode では，システム提供の 8 分木データ構造をもとに，コンストラクタ部に，本体データ配置指定と，キャッシュ配置（Cache(5) は 5 段目までキャッシュする意味）を行っている．また，メソッドに関する指定では，calcAcc() のように深さに応じたモード指定をできるようにしている．

6. プログラミングサンプル

本論文では，提案手法の有効性を図るうえで，3 つのアプリケーションを対象に分散環境への移植操作を行い，記述面の有効性を検討する．対象とするのは，Java Grande Benchmark [1] の逐次版プログラムの Moldyn，RayTracer と，Barnes-Hut 法[7] タイプの N 体問題プログラム（BH）である．Moldyn，RayTracer に関しては，mpiJava を利用した場合のコードも公開されている．

Moldyn： Moldyn の例については，すでに図 2，図 3，図 5 で取り上げてきたので，ここでは概観のみ述べる．各粒子はオブジェクト（Particle）で表現され，配列に格納されている．この配列自体は，オリジナルでは大域変数として扱われているが，共有データとして扱うため DistBase である Moldyn クラスのフィールドとして格納した．

また，計算も複数の計算ステップから構成されており，今回は Init，Others，Calc と切り分けている．mpiJava のプログラムにおいても，同様の箇所でバリア同期や Collective 通信が行われている．

このプログラムは，各 PE が各局面で必要とするデータ領域は明確で，適切なデータ再配置をプログラマが指定すれば，各局面の中では PE 間通信をいっさい行うことなく計算可能である．つまり，MPI 的なアプローチが有効に機能する例といえる．

RayTracer： このプログラムは，どちらかというと共有データはあまり持たない，分割統治型のプログラムともいえる（**図 6**）．RayTracer では，Init，App，Valid の 3 局面が存在する．また，scene などのほとんどのフィールドは，Init 局面で Home によって初期化されるため HomeOnly モードで，App 局面では各 PE から並列に読み込まれ変更も行われないため Cache モードで実行するとよい．

```
public enum RayTracerPhase{
     Init, App, Valid
}

class RayTracer extends DistributedBase<RayTracerPhase>{
  @Dist({@PConf(''Init, Valid'', ''HomeOnly''),
        @PConf(''App'',''Cache'')})
    Scene scene;

  @Dist({@PConf(''Init, Valid'', ''HomeOnly''),
        @PConf(''App'',''Reduce(Sum)'')})
    long checksum;

  @Dist({@PConf(''Init, Valid'', ''HomeOnly''),
        @PConf(''App'',''Reduce(Sum)'')})
    int[ ] row;

  @Dist({@PConf(''Init, Valid'', ''NoExce''),
        @PConf(''App'',''ParalellExec'')})
    void JGFapplication( ){ /* 主要計算 */ }

  void JGFrun(int size){
    JGFsetsize(size);
    JGFinitialise();
    become(RayTracerPhase.App);
    JGFapplication();
    become(RayTracerPhase.Valid);
    JGFvalidate();
  }
}
```

図 6 RayTracer サンプルコード
Fig. 6 Sample code of RayTracer.

一方，checksum フィールドは，App 局面で各 PE によって並列に加算が行われており，最終的に必要なのは，全 PE での総和である．つまり，Reduce(Sum) モードを使う典型例である．また，row フィールドは RayTracer の最終結果となる画像データが格納される配列である．App 局面では，各 PE が自分の担当領域に対してのみ書き込みを行う．このような場合にも，Reduce（Sum）モードが有効である．

BH： BH の計算局面は，Const，CofM，Calc と分かれており，Const でまず空間（Node）を必要に応じて再帰的に 8 分割させながらリーフの空間に粒子（Particle）が 1 つしか所属しないようにする．次に CofM で Node の重心を求め，その後，主要計算局面である Calc において，各 Particle にかかる力を Node の 8 分木データ構造をもとに計算する．その際，十分遠い空間に対しては，空間全体の重心と Particle の座標との間で力の近似計算を行う．つまり，不規則なデータ構造とアクセス領域を有する，MPI などの苦手とするプログラムといえる．

このプログラムを分散化するうえで（図 7），8 分木の上から 2 段目（64 ノード）の段階で分散配置し，3 段目以降は同じ PE に配置されるように指定している．また，Const の際は，まず座標データに基づ

```
// システム提供の 8 分木データ構造
class TreeNode8<Phase> implements DistNode<Phase> {
  public long distNodeID();
  public Tree(); // Root Node Only
  public Tree(Tree parent, index i);
  int depth();
  TreeNode getChild(int i);
  TreeNode getParent();
}
// 以下, ユーザプログラム
enum MyPhase implements Phase { Const, CofM, Calc }

class MyNode extends TreeNode<MyPhase> {

 @DTree(@HomeDist(``DistMethod1(2,Block)''),
        {@PConf(``Const, CofM'', ``Home''),
          @PConf(``Calc'', ``Cache(5)'')})
  MyNode(MyNode parent, int i, ..) { super(parent, i); .. }

 @Dist({@PConf(``Const'', ``HomeOnly''),
       @PConf(``CofM, Calc'', ``AssertOfNoExec'')
  MyNode makeChild(int i, ..) {
    return new MyNode(this, i, ..);
  }
 @Dist({@PConf(``Const, CofM, Calc'', ``CacheExec'')})
  MyNode getChild(int i) { return (MyNode)super.getChild(i); }

 @Dist({@PConf(``Const'', ``HomeOnly''),
              @PConf(``CofM, Calc'', ``AssertOfNoExec'')}),
  void insert(int val) { ... }

 @Dist({@PConf(``Const, CofM'', ``AssertOfNoExec''),
        @PConf(``CofM, Calc'',
               ``0: CacheExecWithRetry, 3: CacheExec'')
              //深さ 0-1 では Retry, 3- では caller 任せ
       })
  public AccInfo calAcc(Particle target) {
    ...
    if (....) {
      return new AccInfo(....);
    } else {
      AccInfo sumup = new AccInfo();
      for(int i= 0; i< 8; i++) {
        MyNode child = getChild(i);
        if(child != null ) {
          AccInfo result = child.calcAcc(target);
          sumup.add(result);
        }
      }
      return sumup;
    }
  }
}
```

図 7 BH の記述例
Fig. 7 Sample code of BH.

いて Particle の再配置を行ってから木構造の構築を行っている．このため Const，CofM に関しては，完全に PE 内で計算を行うことができる．一方，Calc では，Node の上から 5 段目までが全 PE にキャッシュされ，その下位についてはキャッシュは行わず，遠隔メソッド呼び出しを利用している．これは，木の上層部については全般的なアクセスが行われるが，木の深い部分まで必要となるのは近傍に限られるからである．ただし，たとえば，境界付近の Particle については，他の PE 上のデータを必要とすることもしばしばある．今回，木の上層のみをキャッシュし，Node#calcAcc() メソッドは，Cache により実行可能

```
class Target implements Seasonal<Phase> {
  // 自動追加メソッド
  void sendObject(Phase p, ObjectOutputStream out) {
    // 局面ごとの送信ルーチン
  }
}
class TargetProxy extends Target { // 自動追加クラス
  int foo() { // 自動生成 (override)
    phase = DistBase.currentPhase();
    switch(phase) {
      phase1: return super();
      phase2: throw Exception();
    }
  }
  // 追加メソッド
  void receiveObject(Phase p, ObjectInputStream in) {
    // 局面ごとの受信ルーチン
  }
}
```

図 8 プロキシ実装
Fig. 8 Implementation image of proxy.

な範囲は実行し，対象データのキャッシュがなくなった時点で遠隔メソッド呼び出しを実行すべく，0，1 および 2 段目では CacheExecWithRetry に，それ以下では CacheExec モードに設定している．下位にいたるまで CacheExecWithRetry にしていないのは，5 段目のノード呼び出しに対して遠隔呼び出しを行うと PE 間通信が増加するため，木の根本付近に遡ってやり直すことで PE 間通信を減らしたほうが効率的なためである．この効果については，7.2 節で解説する．

7. 試験的評価

7.1 プロトタイプ実装

現時点ではシステムは完成していないため，本論文ではプロトタイプ実装に基づいた性能評価を行う．性能評価に先立ち，現在の実装状況について概説する．

DistNode，DistCollection の実現： DistNode のプロキシは，メタ修飾情報をもとにして自動生成されなくてはならない．ただし，現在は一部自動化されただけの状態にある．今回の実装では，**図 8** のように，プロキシは対象オブジェクトとフィールド構造を共有し，キャッシュ転送用メソッド（send/receiveObject()）の追加と，既存メソッドの override のみを行っている．フィールドアクセスに関しても，今回の実装ではいっさいの動的チェックを行っていない．一方で，メソッド起動時は，コード例のように現在の局面を取得し，その値に応じた挙動を行っている．この局面の取得（currentPhase() 呼び出しとして記述）に関しては，文献 17) で紹介したコード変換に基づく高速スレッドローカルと同様の実装を行う予定である．このため，今回の実装においても，DistBase の参照を引数に追加したプログラムを記述することで，対応している．

次に，DistCollection について．この場合も各 PE 上に要素管理を行うプロキシが配置され，各種操作に対し，実行モードに応じた振舞いが実装されている．たとえば，get() に対しては，本体もしくはキャッシュが存在すればその参照を返し，さもなくば Exception を発生する．

このように，DistNode，DistCollection のメソッド呼び出しに関してのみ，局面分岐などのコストを必要としている．

PE 間通信の実現： キャッシュ送受信機能と，各 DistNode の識別機構を提供する必要がある．

局面遷移の際は，DistBase の本体およびプロキシ間でデータ転送が行われる．その際，Seasonal オブジェクトに対しては各局面ごとの転送ルーチンが実行される．また，データ転送に関しては，今回の評価では moldyn の例に限り内部実装に mpiJava を用いている．それ以外の場合では PE 間で協調したスケジューリングは実現されていない．

次に，DistNode の ID について．本実装では DistBase 内に輸出入表に相当する ID と本体もしくはプロキシの連想表を準備して管理を行っている．ただし，この表情報の Garbage Collection については，現時点では未対応である．また，木構造の ID については，基本的に全ノードが DistNode であるため普通に処理したのでは ID 管理が面倒である．このため，要素の削除や切り放し機能を制限した API を設定し，簡略化した ID 管理を行っている．今回の実装では，NodeID はルートノードの ID とルートからの相対位置を整数にエンコードしたものを用いている．BH で用いた 8 分木構造では，深さ 0–9 段に制限された ID であり，深さ情報 4 bit と各層の情報 $3 \times 9 = 27$ bit で表現している．ID 連想表にはルートのみを登録し，そこから木構造をたどることで対応するプロキシへの到達を可能にしている．

7.2 実行性能評価

本節では，6 章で紹介したプログラムに対し，7.1 節で説明したコード変換を半ば手作業で行い，実行性能測定を行った結果を紹介する．また，mpiJava のコードが提供されている Moldyn，RayTracer については mpiJava で実行した場合との比較を行った．一方，BH については，mpi 環境に移植すること自体面倒であるため，代わりに共有メモリ版プログラムを共有メモリ計算機上で実行した場合との比較を行った．

分散計算環境としては，1000 Base-T のネットワークで接続された 16 台の PentiumIV 3.2 GHz/Linux（kernel2.4.18）/1 GB memory の環境で，JDK 1.5.0 beta2 にオプション -server -Xmx512M を付加して実行させた．一方，共有メモリ計算機としては，Sun Fire 12 K を利用した．

表 1 MolDyn 実行時間 [sec]
Table 1 Elapsed time (Moldyn) [sec].

種別		become Calc	Calc	become Others	Others	Loop Total
逐次版		-	727	0	0.1	728
MPJ 版	1	-	765	0.1	0.1	766
	2	-	370	0.5	0.1	371
	4	-	183	0.8	0.1	184
	8	-	101	1.3	0.1	103
	16	-	51	2.4	0.1	54
本方式	1	0.0	635	0.1	0.1	635
	2	1	297	0.4	0.1	298
	4	2	155	0.6	0.0	158
	8	3	79	1.6	0.0	84
	16	4	40	2.3	0.0	46

Moldyn：表 1 が 108000 粒子を対象としたときの実行時間である．各局面ごとの計算時間のうち，MPJ 版における becomeCalc，becomeOthers は，対応する MPI Collective 通信に要した時間を示している．

MPJ も本方式もデータ配置などは基本的に同じであるため，主要計算局面である Calc では同じような速度向上が見られており，16 台で 14–15 台分の台数効果を得ている．絶対時間として，本方式の方が高速な傾向を示しているのは，逐次版および MPJ 版では static フィールドであった粒子配列を，本方式では DistBase のインスタンスフィールドに変更し，加えて，各メソッドの引数に DistBase の参照を付加したためと思われる．

become 時間に関しては，becomeOthers は MPI 同様の速度向上を果たしている．これは内部実装に mpiJava を利用したためである．一方で，MPI では becomeCalc に相当する時間がない．これは Others 局面の計算を PE 間で分担せずに，全作業を各 PE がそれぞれ行うことで通信を削減できたためである．このように，通信を行う代わりに，全 PE で重複して計算を行った方が効率的な場合も存在する．

RayTracer：表 2 が全実行時間である．このプログラムにおいても，主要計算局面がほぼ計算のすべてを占めており，また PE 間通信もほとんどともなわないため，MPJ 版では高い台数効果を得ている．一方で，本方式では PE 台数が多くなったときの実行時間が遅くなっている．これについては，1PE では 0.2 秒程度であった JIT の compilation time（java.lang.management.CompilationMXBean により測定）が，16PE では 1.1 秒も要するようになっ

表 2 RayTracer 実行時間 [sec]
Table 2 Elapsed time (RayTracer) [sec].

PE	本方式	MPJ
1	32.175	30.678
2	16.167	15.329
4	8.511	8.387
8	5.146	4.242
16	3.306	2.331

表 3 BH 実行時間 [msec]
Table 3 Elapsed time (BH) [msec].

PE	Const	CofM	Cache	Calc	Total
1	2073	533	0	18433	20735
2	991	156	55	10538	11760
4	422	97	100	5441	6137
8	205	66	146	2953	3470
16	135	27	275	1981	2497

ていたことが直接の原因と考えている．本方式ではデータ送受信にマルチスレッドを多用しており，何らかの理由で compilation time の増加につながったのかと想像しているが，詳しいことは分かっていない．

BH：表 3 が 50 万粒子を対象とした BH の実行時間であり，また，**図 9** の各ラインは，それぞれ分散環境における計算局面と，分散環境における全実行と，共有メモリ計算機上で共有メモリ版プログラムを実行した際の全実行に関して，その速度向上比を示したものである．

グラフからも分かるように，この BH では共有メモリ計算機上でも完全な台数効果を示していない．他方，分散環境におけるプログラムは，共有メモリ環境とほぼ同様の台数効果を示しており，適切な分散環境移植が実現できたといえる．また，BH の台数効果が限定的であったのは，対象プログラムでは負荷分散が偏っていた（中央付近の粒子に，より多くの計算を要する）ためだと思われるため，今後，プログラム自体のデータ配分などを調整し，より高い台数効果を目指したいと考える．

表 3 の局面ごとの実行時間からは，`Calc` 局面に遷移する際の `Cache` のコストが，PE 台数とともに増加していることが分かる．本プログラムでは，木構造の送信手続きが複雑なため，内部実装に mpiJava を用いていないことも影響している．

次に，考察として MPI で実装した場合の性能を推察し，本実装との比較を行う．今回のプログラムでは，最初に 8 分木を全 PE で協力して作成し，その後，木の上層部だけをキャッシュして計算を行った．このため，`Calc` 局面では，16PE の場合に合計 2000–3000 回の遠隔メソッド呼び出しが行われていた．ただし，対象粒子が 50 万であることを考えると，稀に遠隔メソッド呼び出しが行われていたといえる．

一方で，MPI では動的なメッセージへの対応は面倒なため，これを避けようとすると，木構造全体を各 PE で保持する必要がある．今回同様，まず協力して木を構築し，その後全 PE にキャッシュを配布したケースを想定する．今回の木構造は，合計 40–50 MB 程度のデータ構造であるため，各 PE が純粋にデータ受信だけを行ったとしても，実効 500 Mbps 程度のバンド幅で 1 秒弱の時間が必要となる．本プロトタイプ実装上では測定の結果 2.7 秒を要していた．現在，16PE 時の Calc の計算時間は 2 秒程度であり，そのオーバヘッドは大きい．今回の実行結果と比較すると 2 割程度の速度低下は必至である．別の方式として，そもそも協力して木を作成せずに，同じ木構造を全 PE 各々に作成した場合を検討する．ただし，木の構築には 1PE で 2 秒近く要するため，さらに粒子座標データの送信が加わると，速度低下は必至といえる．

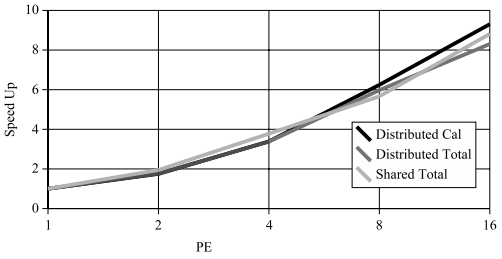


図 9 BH 性能比較
Fig. 9 BH scalability.

つまり，この例については，主要なアクセス領域のみを最初にキャッシュし，それ以外については必要に応じた通信を行うことが，高速化につながっているといえる．

8. 議　　論

関連研究との比較： まず，従来の MPI などによるプログラミングとの差について考える．一番の違いは，本方式では，プログラマが各フィールドやメソッドについて理解している（と思っている）性質を，メタ修飾を利用してそのまま記述させるという方法をとっている点である．一方，MPI などを利用した場合，プログラマの理解に基づいて，データ再配置という操作を記述させている．このため，我々の方法では，ユーザの意図は本来正しかったのか，実行時に確認するという手段がとりうるが，MPI ではそのようなアプローチをとりえない．

次に，本研究同様リンクデータ構造の共有イメージを提供しようとした研究として Replicated Method Invocation [11] を紹介する．我々が一部データをキャッシュしたのと違い，全 PE でつねに共通のイメージを保持しようとする点が特徴である．このため，メソッドが書き込みをともなうと分かった時点で，そのメソッドを全 PE 上で実行させる．ただし，更新をともなう作業を PE 間で分担させたい場合に，しばしば問題が起きると考えられる．

局面記述について： 現在のメタ修飾記述は必ずしもコンパクトとはいえず，引き続き記述法の改善が必要だと考える．また，文献 15)，16) で行ったように，局面記述から各コード片の実行可能局面を求め，そこから各局面におけるフィールドの read/write の有無などを解析することは可能である．各フィールドのキャッシュ方針が決まれば，あるメソッドが `CacheExec` 可能か自動判定できる場合も多い．ただし，解析精度の問題もあり，解析結果だけに頼った自動最適化は有効でないケースも多い．特に分散環境では無駄な通信が実行速度低下に直結するため，本研究では，局面ごとの各フィールドの性質をプログラマに直接指示させることで，プログラマの意思を明瞭に反映できる方式を採用した．今後，局面解析機構の併用によるメタ修飾の簡略化などについて考察を進めたいと考えている．

逐次プログラムからの移植性： まず，本提案方式では逐次プログラムからの移植性を重視し，局面の導入やメタ修飾を行うことによる，分散環境への移植実現を目指した．メタ修飾されたプログラムも，修飾を無視すれば，逐次もしくは並列プログラムとして機能するようにデザインされている．

一方で，並列化・分散化の際の注意点も存在する．まず，本方式は純粋な共有メモリ空間を提供してはいない．`DistNode` と呼ぶデータ構造については，共有空間においてユニークな ID を割り当て，遠隔メソッド呼び出し機能による仮想的な共有空間を提供している．ただし，`DistNode` 以外の一般データ構造については，あくまでコピーによる別オブジェクトが送られるのであり，共有関係は存在しない．これは，JavaSpaces のようなオブジェクトベースの共有メモリ機構と同様である．このため，**図 10** の `method0()` は `method1()` のようなメソッドに改変して実行する必要がある．これは，RMI などの遠隔呼び出しでも起きる問題である．

次に，排他制御について．共有データに対して読み書きが混在して行われている状況では，本環境上ではプログラマは本体上での実行を指示することになる．当然，ボトルネックが発生するような状況については，

```
void method0(ResultPlace place) {
  place.x = ....; place.y = ....;
}
ResultObj method1() {
  x = ...; y = ....;
  return new ResultObj(x, y);
}
```

図 10 コードの書き換えを要する例
Fig. 10 Sample code for code rewriting.

```
class Particle {
  static void force(Particle p, Particle q) {
    synchronized(p) {
      synchroniized(q) { /* 計算本体 */; }
    }
  }
  void forceP(Particle q) {
    synchronzied(this) { forceQ(this); }
  }
  void forceQ(Particle p) {
    synchronzied(this) { /* 計算本体 */; }
  }
}
```

図 11 複数オブジェクトのロックの取得
Fig. 11 Sample code for multiple lock acquirement.

排他制御の緩和に向けたアルゴリズム的な改変などが必要となる．別の問題として，現在の実行モデルでは，複数の PE 上のオブジェクトに対してロックをとる際，すこし工夫が必要となる．本プログラムでは，遠隔メソッド呼び出し機能しか提供されていないため，**図 11** の `force()` のようなプログラムは，`p`，`q` の本体オブジェクトが別々に存在する場合，ロックの確保ができない．このため，`forceP,Q()` のようなメソッドに切り分け，遠隔呼び出しを行う必要がある．今回取り上げたアプリケーション例では，そもそも排他制御が必要な状況があまりなかったが，今後，本体機能の移動機能の導入などによる解決策を検討したいと考えている．

9. ま と め

本論文では，分散計算環境においてデータ共有を行うための，新たな計算環境について提案を行った．本方式の特徴は，MPI の Collective 通信に相当するデータ再配置機構を，各オブジェクトに注釈付けすることによって実現することにあり，一般のリンク構造を持つデータ構造をそのままに，キャッシュや reduce 操作といったデータ再配置機能の実現を目指す．

加えて，不規則なデータ構造などが対象であり，各計算において必要と思われるデータが確定できない場合に備えて，遠隔メソッド呼び出し機構も提供することとした．

現在，システム自身は完成していないため，試験的な性能評価を行った結果，Moldyn や RayTracer と

いった MPI に適したプログラムにおいても MPI に匹敵する実行性能が期待できることが分かった．また，BH のように各計算局面においてアクセス対象が確定できない場合については，MPI の Collective 通信を利用した標準的な実装法に比して，効率的な処理を行える可能性が高いことが分かった．

謝辞 本研究の一部は，科学研究費補助金（若手研究 B-14780217）の支援による．

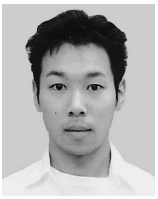
参 考 文 献

1) The Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html
2) Jini Network Technology. http://java.sun.com/developer/products/jini/
3) The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/
4) *mpiJava Home Page*. http://www.hpjava.org/mpiJava.html
5) Top 500 Supercomputer Site. http://www.top500.org/
6) Baker, M., Carpenter, B., Ko, S.H. and Li, X.: mpiJava: A Java interface to MPI, *Proc. 1st UK Workshop on Java for High Performance Network Computing* (1998).
7) Barnes, J. and Hut, P.: A Hierarchical $O(N \log N)$ Force-Calculation Algorithm, *Nature*, Vol.324, pp.446–449 (1986).
8) Harada, H., Ishikawa, Y., Hori, A., Tezuka, H., Sugimoto, S. and Takahashi, T.: Dynamic Home Node Reallocation on Software Distributed Shared Memory, *Proc. HPC Asia 2000*, pp.158–163 (2000). http://www.pccluster.org/
9) The Jakarta Project: commons Collections. http://jakarta.apache.org/commons/collections/
10) Koelbel, C. and Mehrotra, P.: Compiling Global Name-Space Parallel Loops for Distributed Execution, *IEEE Trans. Parallel Distrib. Syst.*, Vol.2, No.4, pp.440–451 (1991).
11) Maassen, J., Kielmann, T. and Bal, H.E.: Efficient replicated method invocation in Java, *Proc. ACM 2000 conference on Java Grande*, pp.88–96 (2000).
12) Sharma, S.D., Ponnusamy, R., Moon, B., Hwang, Y.S., Das, R. and Saltz, J.: Run-time and compile-time support for adaptive irregular problems, *Proc. 1994 ACM/IEEE conference on Supercomputing*, pp.97–106, ACM Press (1994).
13) Sohda, Y., Nakada, H., Matsuoka, S. and Ogawa, H.: Implementation of a Portable Software DSM in Java, *Proc. ACM Java-Grande/ISCOPE 2001 Conference* (2001).
14) Yu, W. and Cox, A.: Java/DSM: a Platform for Heterogeneous Computing, *Proc. Workshop on Java for Science and Engineering Computation*, Vol.43.2, pp.65–78 (1997).
15) 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. JSPP 2002*, pp.245–252 (2002).
16) 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 (2003).
17) 泉　勝，神前宏樹，鎌田十三郎：差分ベースのバイトコード変換と命令再定義機構，*Proc. SACSIS2004*, pp.61–68 (2004).

(平成 16 年 7 月 5 日受付)
(平成 16 年 9 月 19 日採録)

鎌田十三郎（正会員）

1970 年生．1993 年東京大学理学部情報科学科卒業．1995 年同大学大学院理学系研究科情報科学専攻修士課程修了．1998 年同博士課程単位修得退学．1996 年〜1998 年日本学術振興会特別研究員（東京大学）．1998 年より神戸大学工学部助手．博士（工学）．並列・分散処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．



森本　昌治

1981 年生．2004 年神戸大学工学部情報知能工学科卒業．現在，同大学大学院自然科学研究科情報知能工学専攻在学中．並列・分散計算等に興味を持つ．

ニッ森大介

1982 年生．2004 年神戸大学工学部情報知能工学科卒業．現在，同大学大学院自然科学研究科情報知能工学専攻在学中．並列･分散計算や P2P 環境等に興味を持つ．