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

松田 聡[†,☆] 鎌田 十三郎[††]

プログラムの高速化には，CPU とメモリとの速度差を緩和するキャッシュメモリの有効利用が重要である．共有メモリ型並列計算機では，データの一貫性保持のためのキャッシュの無効化が存在するため性能への影響も大きく，しばしばプログラマ自身がキャッシュを意識したオブジェクト配置を行ってきた．本研究では，フィールドアクセス情報を利用してクラスを分類・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ことで，この作業の自動化を目指す．本論文では，Sun HotSpot VM の自動メモリ管理機構に深さ優先コピー方式やアクセス傾向別領域を導入し，各種アプリケーションを通してその有効性を評価する．

Cache-conscious Memory Management for Shared-Memory Multiprocessors

Satoshi Matsuda[†,☆] and Tomio Kamada[††]

The processor/memory speed gap increases the importance of cache utilization for efficient program execution. To achieve scalable speed-up of parallel programs on shared-memory multiprocessors, the programmer is often forced to do cache-conscious allocation of memory objects avoiding for frequent cache invalidation. This paper proposes an automatic object arrangement scheme on memory management systems using field access information of the target program. We customize the memory management system of Sun HotSpot VM to introduce depth-first copying GC and memory chunk for frequently read objects, and evaluate the performance of the system using several applications.

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，キャッシュミスを削減することがプログラムの高速化につながる．特に共有メモリ型並列計算機では，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，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☆☆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ともな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．

そこで，キャッシュを意識した配置を自動メモリ管理機構上で行うことで，この作業の自動化ならびに並列プログラムの高速化を目指す．本研究では，従来のキャッシュを意識したオブジェクト配置方法に加え，並列計算機向けにコヒーレンスミスの削減も行う．そのためのアプローチとして，オブジェクトの読出し（read）・書込み（write）アクセス傾向からクラスを分類・分割し，メモリ割当て・ガーベジコレクション（GC）時にアクセス傾向の異なるオブジェクトを別々の領域に配置することで，その後のプログラム実行におけるキャッシュミスの削減を図る．

本提案を実現するにあたり，既存の Java 処理系である Sun HotSpot VM[8] の自動メモリ管理機構に対して，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を実装した．本論文では，アクセス情報が取得可能であると仮定してクラスを分類・分

† 神戸大学大学院自然科学研究科情報知能工学専攻
The Graduate School of Science and Technology, Kobe University
†† 神戸大学工学部情報知能工学科
Department of Computer and System Engineering, Faculty of Engineering, Kobe University
☆ 現在，富士通株式会社
Presently with FUJITSU LIMITED

☆☆ 本論文では，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．

割し，本プロトタイプの性能評価・考察を行う．

本論文の構成を以下に示す．2 章では，関連研究をまとめ，キャッシュを意識したオブジェクト配置について検討する．3，4 章では，本研究のアプローチならびにプロトタイプ実装について述べる．5 章では，複数の並列プログラムを通して評価を行う．最後に 6 章で，まとめと今後の課題について述べる．

2. 関 連 研 究

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 たとえば，参照関係にあるオブジェクトどうし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どうし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．

本節では，HotSpot VM のメモリ管理機構のベースである copying GC を中心に，既存の研究について上記 3 点に着目してまとめ，共有メモリ型並列計算機に適したオブジェクト配置方法を検討する．

2.1 空間的局所性の向上

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[4]．そのため，たとえば木構造の場合，木の兄弟節点どうし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[14]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[9),12)]．これらは，直接参照関係にあるオブジェクトどうし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．メモリを順に割り当てる Coying GC はもともと局所性は高いが，たとえば文献 13) 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 のオブジェクトどうしを 1 つのグループとしてメモリを割り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．たとえば文献 6) 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どうし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どうしの局所性を向上させることが可能となる．

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 5) 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[6)]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，スレッドごと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．たとえば，今回利用する HotSpot Server VM のように，各スレッドごと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ごと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．

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どうし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ともな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．これに対し文献 11) は，レイアウト変更の自動化を目指した研究である．その中で提案されている固定レイアウト法は，文献 5) を並列向けにしたもので，クラスを頻繁に読み出されるフィールドを格納するクラス（Light クラス）と頻繁に書き込まれるフィールド・あまりアクセスされないフィールドを格納するクラス（Heavy クラス）とに 2 分割し，Light オブジェクトどうしを詰めて，Heavy オブジェクトからは離れた位置に配置することで，キャパシティミスに加えコヒーレンスミスの削減を図るものである．しかし，自動メモリ管理機構との統合は行われていない．

3. アプローチ

3.1 オブジェクト配置の検討

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どうし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，2 章で紹介したキャッシュ効率に関する 3 つの項目に対し，ひととおりの対応を講じることができる．(1) については文献 11) で提案した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．

(1) で利用するクラス分割などの手法はすでに提案済みの研究であるが，自動メモリ管理機構との統合に向けた検討はされておらず，今回，広く実用されている HotSpot Server VM に対して実装技術の検討を行い，加えて深さ優先コピー方式と併用した場合の効果について評価を行った．一方で，2 章で紹介した他の関連研究との比較/統合については今後の課題とする．

3.2 自動メモリ管理機構との統合に向けて

3.1 節での検討をふまえ，本研究ではオブジェクトの読出し・書込みアクセス傾向でクラスを分類し，アクセス傾向別にオブジェクトを配置する．その実現にあたり，以下の 2 つの機構からなるシステムを構築する．

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，4 章で述べる．

3.3 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出し・書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出し・書込みの行われるフィールドがそれぞれ存在し，フィールドごと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 2 章で紹介した固定レイアウト法[11]によるクラス分割を行った後，分類することにする．以下，固定レイアウト法について簡単に説明する．

まず，読出し・書込みアクセス数に応じて各フィールドを W/R/N 変数の 3 つに分類する．そして，もとのクラスを R 変数のみを含む Light クラスと W，N 変数を含む Heavy クラスとに 2 分割する．また，分割した Heavy クラスにアクセスできるようにするため，Light クラス内に Heavy クラスへの参照を加え，Heavy クラスに配置されたフィールドへのアクセスはこの参照を介して行う．

しかし，文献 11) においては W/R/N 変数の分類基準を示していない．そこで，本論文の評価環境においては，

- 書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込みアクセス数のうち，20%以上書込みアクセスされる変数．
- 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%以上読出しアクセスさ

```
// 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 1 節点のデータ構造（カウンタ付き 2 分木）
Fig. 1 Program code (Binary tree: node).

表 1 節点のアクセス情報（カウンタ付き 2 分木）
Table 1 Access information (Binary tree with counters).

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

れる変数.

- N 変数：W，R 以外の変数.

と定める．そのうえで，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，5 節の評価で使用しているカウンタ付き 2 分木の節点（**図 1** 左）のクラス分割を考える．**表 1** 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出し・書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（読出し + 書込み）に対する各フィールドの読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込みアクセス数に対する各フィールドの書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 1 右）.

3.4 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，6) 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．

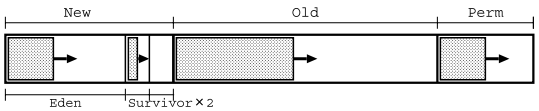

図 2 ヒープ構造（SUN HotSpot Server VM）
Fig. 2 Heap layout (SUN HotSpot Server VM).

したがって，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 であることを示し，`FreqReadAry` はそのクラスのオブジェクト配列が FreqRead であることを示す．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.

4. メモリ管理・配置機構の実装

4.1 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，4.2 節以降で今回の改良について述べる.

図 2 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[1)]．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 空間とからなる．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（どちらかはつねに空）．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

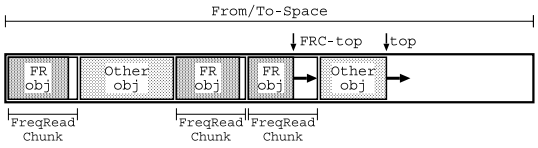

図 3 FreqRead Chunk
Fig. 3 FreqRead Chunk.

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）．

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十分な領域が Old 空間にない場合に起動される．なお，GC の並列化はされていない．

4.2 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 からメモリを割り当てる．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．したがって，**図 3** 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 導入にともなうメモリ割当て・GC の変更について説明する．メモリ割当てにおいては，まず割り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 から割り当てる．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 とまったく同様であり，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 空間から割り当てる．

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 3 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．

4.3 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．

5. 評　　価

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 12 K 上とで評価を行う（**表 2**）．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 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

表 2 実行環境
Table 2 Experiment environments.

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

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 などはどの 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．

5.1 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 1）．したがって，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．今回のように，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 サイズは初期値（8 KB）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 3** 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，そのうちの 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 1 CPU では depth が速いが，CPU 数が増加すると，FRdepth のほうが速くなっている．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，

- 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，1 CPU 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 数が多いときには，探索において depth より FRdepth のほうが高速であり，無効化の影響を回避した効果といえる．

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．したがって，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 4**）．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

表 3 カウンタ付き 2 分木の実行結果（insert(gc)/search [sec]）
Table 3 Performance evaluation for binary tree with counters (insert(gc)/search [sec]).

#cpus		1	2	4	8	16
Enter-prise	breadth(no-split)	6.11(0.90)/5.36	4.13(0.90)/3.08	2.94(0.89)/1.77	2.26(0.88)/1.05	1.94(0.85)/0.66
	depth(no-split)	5.50(0.89)/4.02	3.78(0.90)/2.28	2.78(0.89)/1.29	2.16(0.88)/0.78	1.85(0.85)/0.50
	FRdepth(split)	5.79(0.97)/4.40	3.88(0.95)/2.28	2.83(0.94)/1.24	2.24(0.90)/0.74	2.01(0.92)/0.45
Fire	breadth(no-split)	3.27(0.52)/3.36	2.30(0.52)/1.94	1.60(0.51)/1.12	1.32(0.49)/0.67	1.11(0.48)/0.47
	depth(no-split)	2.74(0.46)/2.38	1.97(0.46)/1.36	1.44(0.46)/0.80	1.14(0.45)/0.45	1.09(0.44)/0.35
	FRdepth(split)	3.30(0.50)/2.71	2.17(0.47)/1.45	1.52(0.48)/0.78	1.19(0.45)/0.45	1.07(0.44)/0.32

表 4 カウンタ付き 2 分木の実行結果：オーバヘッド評価（insert(gc)/search [sec]）
Table 4 Overhead evaluation for binary tree with counters (insert(gc)/search [sec]).

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

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．したがって，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 32 MB 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．

5.2 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[3]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出しである．なお，並列化は各スレッドごとに担当する質点を決めて処理することでなされている．

```
// 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 4 class Node（Nbody Program）
Fig. 4 class Node (Nbody Program).

表 5 アクセス情報（Nbody: Node）
Table 5 Access information (Nbody Program: Node).

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

主要なデータ構造である 8 分木の節点（**図 4** 左）のアクセス数を**表 5** 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．たとえば，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 4

表 6 Nbody の実行結果（insert(gc)/calc [sec]）
Table 6 Evaluation for Nbody Program (insert(gc)/calc [sec]).

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

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．したがって，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** 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．Enterprise では，breadth と depth との構築時の差はあまりないが，若干 depth のほうが速い場合がある．また，計算時においても全体的にあまり変わらないが．breadth のほう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ほう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みられないのは，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，文献 11) において効果がみられたのは，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．

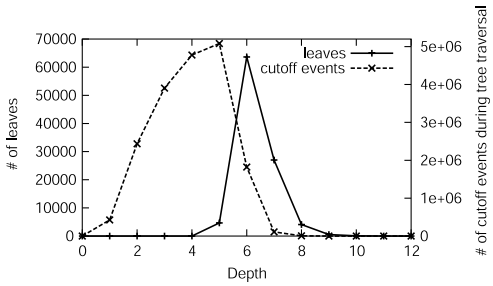

図 5 葉節点の深さと加速度計算時にたどる木の深さ
Fig. 5 Access information of node for each tree depth.

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 5** 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

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

図 6 class Particle（MolDyn Program）
Fig. 6 class Particle (MolDyn Program).

る．したがって，今回は実装していないが，文献 9), 12) 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．

5.3 MolDyn

プログラム概要：逐次 Java ベンチマークである文献 10) 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，8,788 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** 左）のアクセス情報を**表 7** 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，それぞれを FreqRead/Other クラスに分割する（図 6 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．したがって，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 8,788 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 8** 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 2), 7) 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に対する書込みを行う．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，結果として無効化が生じるためである．

表 7 アクセス情報（MolDyn: Particle）
Table 7 Access information (MolDyn: Particle).

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

表 8 MolDyn の実行結果 [sec]
Table 8 Evaluation for MolDyn [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 12 K					
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

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 7 より，読出しアクセス数が 99%以上もあり，書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いえる．

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 1 CPU 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．したがって，排他制御もそのオブジェクトに対する書込みとして扱うなどの対応が必要である．

5.4 議　　論

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（16 CPU 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うる．

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出し・書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．ただし，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，たとえば文献 5) 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. まとめと今後の課題

本研究では，アクセス傾向を利用して共有メモリ型並列計算機向けのキャッシュを意識したオブジェクト配置を行う自動メモリ管理機構を提案した．実際に，既存の Java の処理系である Sun HotSpot VM に対して深さ優先方式とアクセス傾向別領域とを導入し，複数の並列プログラムを対象に性能評価を行った．今後は，多くのオブジェクトが動的に生成され，混在するようなアプリケーションに対しても有効性の検討を行いたいと考えている．また，バイトコード変換によるクラスの分類・分割の自動化を加え，本システムの完成を目指す予定である．

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) *Tuning Garbage Collection with the 1.3.1 Java^{TM} Virtual Machine*. http://java.sun.com/docs/hotspot/gc/index.html
2) Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y.S. and White, D.: An Efficient Meta-Lock for Implementing Ubiquitous Synchronization, *OOPSLA*, pp.207–222 (1999).
3) Barnes, J. and Hut, P.: A hierarchila O(NlogN) force-calculation algorithm, *Nature*, Vol.324, pp.446–449 (1986).
4) Cheney, C.J.: A Nonrecursive List Compacting Algorithm, *Comm. ACM*, Vol.13, No.11, pp.677–678 (1970).
5) Chilimbi, T.M., Davidson, B. and Larus, J.R.: Cache-Conscious Structure Definition, *PLDI*, pp.13–24 (1999).
6) Chilimbi, T.M. and Larus, J.R.: Using Generational Garbage Collection to Implement Cache-Conscious Data Placement, *ISMM*, pp.37–48 (1998).
7) Dice, D.: Implementing Fast Java^{TM} Monitors with Relaxed-Locks, *the Java^{TM} Virtual Machine Research and Technology Symposium (JVM'01)* (2001).
8) *JAVA HOTSPOT VIRTUAL MACHINE[tm] Sun Community Source Licensing*. http://wwws.sun.com/software/communitysource/hotspot/download.html
9) 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).
10) *The Java Grande Forum Sequential Benchmarks*. http://www.epcc.ed.ac.uk/javagrande/sequential.html
11) 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，*JSPP*, pp.149–156 (2001).
12) Shuf, Y., Gupta, M., Franke, H., Appel, A. and Singh, J.P.: Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times, *OOPSLA*, pp.13–25 (2002).
13) Wilson, P.R., Lam, M.S. and Moher, T.G.: Effective Static-Graph Reorganization to Improve Locality in Garbage-Collected Systems, PLDI, pp.177–191 (1991).
14) 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，*SPA* (2000).

(平成 15 年 2 月 3 日受付)
(平成 15 年 5 月 9 日採録)

松田　聡

1979 年生．2001 年神戸大学工学部情報知能工学科卒業．2003 年同大学大学院自然科学研究科情報知能工学専攻修了．2003 年 4 月より富士通株式会社．並列計算，メモリ管理等に興味を持つ．

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．1996〜1998 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．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．