# 局所性を考慮した共有メモリ並列計算機上の 並列 BIBOP 型メモリアロケータ

遠藤 敏夫 †,††　田浦 健次朗 †††　米澤 明憲 †††

† 東京大学大学院理学系研究科

†† 日本学術振興会特別研究員

††† 東京大学大学院情報理工学系研究科

本稿は共有メモリ並列計算機における Big bag of pages(BIBOP) 型メモリアロケータの並列化方式である Locality-aware page shared(LPS) 方式を提案する。本方式の目標はスケーラビリティ、良好な局所性、高メモリ利用効率を達成することである。分散共有メモリ (DSM) における局所性とメモリ利用効率はトレードオフの関係にあり、 LPS 方式はそのバランスを調節することができる。ユーザが与える任意の定数 $k$ に対して、 LPS 方式は最もメモリ利用効率がよい場合に比べ $k$ 倍までメモリを消費する。メモリを多く消費することによって、各スレッドがローカルメモリを確保できる機会を増やす。これを実現するために LPS 方式は空きページをスレッド個別に管理する。そしてメモリ消費量と $k$ に依存する閾値を比較してリモートページを獲得するか新たにメモリを消費するか判断する。 DSM 型計算機 Origin 2000 上の実験を通して、 $k$ の値によって局所性とメモリ利用効率のバランスを調節できることを示す。また、 LPS 方式は空き領域をスレッド個別に管理するためにスケーラビリティは良好であり、 48 スレッドにおいて逐次時の 24 倍の速度向上を達成することが分かった。

# Locality-aware Parallel BIBOP Memory Allocator on Shared Memory Multiprocessors

Toshio ENDO †,††　Kenjiro TAURA †††　Akinori YONEZAWA †††

† Graduate School of Science, The University of Tokyo

†† Research Fellow of the Japan Society for the Promotion of Science

††† Graduate School of Information Science and Technology, The University of Tokyo

This paper presents a parallel algorithm of a BIBOP memory allocator on shared memory multiprocessor, named Locality-aware page shared(LPS). The goal is scalability, locality, and high memory utilization. LPS method can control the tradeoff between locality and memory utilization on distributed shared memory (DSM) machines. The memory consumption of LPS allocator is $k$ times larger than that of the most economical method (where $k$ is a given constant). By using more memory, each thread can allocate local memory in a higher probability. To achieve this property, each thread maintains free pages locally. By comparing the current memory consumption and a threshold derived from $k$, each thread determines whether it should obtain remote pages or consume new pages. The experimental results on Origin 2000 DSM shows that users of LPS method can control the balance between locality and memory utilization by adjusting $k$. In LPS methods, each thread maintains free small objects for good scalability. It achieves 24 fold speedup with 48 threads.

## 1 はじめに

並列アプリケーションプログラムの多くは動的にメモリ確保を頻繁に行なうため、その性能はメモリアロケータの性能に大きく影響を受ける。しかし OS 標準ライブラリの malloc 関数など、広く使われているメモリアロケータの多くは並列化されていない。このためそれらをそのまま利用すると並列アプリケーションのスケーラビリティを大きく低下させることがある。メモリ確保処理自体の性能に加え、アプリケーションの局所性も考慮する必要がある。 SGI Origin 2000 [8] などの分散共有メモリ (DSM) 型計算機においてはメモリの配置によってアクセスコストが異なるため、な

るべく要求者スレッドに対してローカルなメモリ領域を渡すのが望ましい。一方アロケータがプログラムの必要量よりも大量にメモリを消費してしまうと他プロセスへ悪影響を及ぼす。

本稿は Big bag of pages(BIBOP) 型アロケータの並列化方式である Locality-aware page shared (LPS) 方式を提案する。目標は、メモリ確保処理のスケーラビリティ、アプリケーションの局所性、メモリ利用効率の全てを達成することである。並列アロケータにおいてこの三点に大きく影響する、空き領域をスレッド間で独立にするか共有するか、という管理方式について議論する。 BIBOP 方式において空き領域はオブジェクトレベルとページレベルの 2 レベルのフリーリストによって管理されるので、素朴な並列化方式として以下の選択肢が考えられる (図 1 参照)。

- All-shared(AS): 両フリーリストをスレッド間で共有する。スケーラブルでないので本稿では触れない。
- Page-shared(PS): オブジェクトレベルフリーリストを各スレッド個別にし、ページレベルフリーリストをスレッド間共有する。
- All-local(AL): 両フリーリストを各スレッド独立にする。

AL 方式はスケーラビリティと局所性が最も良い一方、メモリ利用効率が悪い。スレッド間で空き領域が一切共有されないためアロケータは最悪の場合に必要メモリ量の $P(=$ スレッド数$)$ 倍メモリを消費してしまう。 PS 方式は小さい空き領域をスレッド個別に管理することによりスケーラビリティを、空きページを共有することにより高メモリ利用効率を達成する。しかし空きページは同確率で任意のスレッドによって再利用されるため DSM における局所性は良くない。

本稿が提案する LPS 方式は PS 方式の局所性を改良したものである。この方式では各スレッドはリモートページよりもローカルページを優先的に確保することができる。これを実現するために、LPS 方式は PS 方式よりも余計にメモリを消費する可能性がある。ユーザによって与えられたある定数 $k(k \geq 1,$ 許容消費倍率と呼ぶ$)$ に対して、本方式の消費メモリ量は PS 方式の $k$ 倍以下であることを保証する。ユーザはこの許容消費倍率を調節することによって局所性と消費メモリ量のトレードオフを自由に調節することができる。

本稿ではガーベージコレクション (GC) の存在を前提としているが、議論のほとんどは手動によるメモリ解放を行なう場合にも適用できる。

以降、 2章ではアルゴリズムの詳細について述べる。 3章で PS, AL, LPS の各方式のメモリ消費量の理論上上限について解析する。 4章で Origin 2000 上での実験結果を述べる。 5章で関連研究について触れ、6章で結論を述べる。

## 2 アルゴリズム

本稿が対象とする並列アプリケーションプログラムは pthread などのカーネルレベルスレッド (以下、スレッド) によって並列化されているとする。各スレッドはそれぞれ別のプロセッサ上で動作することを仮定して議論する。各スレッドが確保するオブジェクトは単一の共有ヒープにおかれ、各スレッドはヒープ中の任意のオブジェクトにアクセスすることができる。

実験環境である DSM 型計算機 Origin 2000 は複数のノードから成り立ち、各ノードはプロセッサと物理メモリノードを持つ。リモートメモリへのアクセスはローカルメモリへのアクセスよりも 2 〜 4 倍程度低速である。このためアプリケーションの性能を向上させるためには、アロケータは確保者スレッドにとってローカルなメモリ領域を割り当てることによって局所性を良くすることが望ましい。

Origin 2000 において、アドレス空間中のメモリは 16KB のページ単位で、物理メモリノードのいずれかに配置される。あるプロセッサ上のスレッドがページを OS から獲得する (そしてスワップインを起す) と、OS はページをそのプロセッサにとってローカルな物理メモリノードに配置する。本稿の議論において、並列プログラム実行中にスワップアウトは起らないと仮定する。つまり、一度 OS から獲得したページを OS に返さず、また決まったメモリ配置は不変とする。

以降では PS, AL, LPS の各並列アロケーション方式について説明する。まず共通事項を 2.1 節で説明する。その次に 2.2節で素朴な方式である PS, AL 方式について、 2.3節で本稿が提案する LPS 方式について述べる。

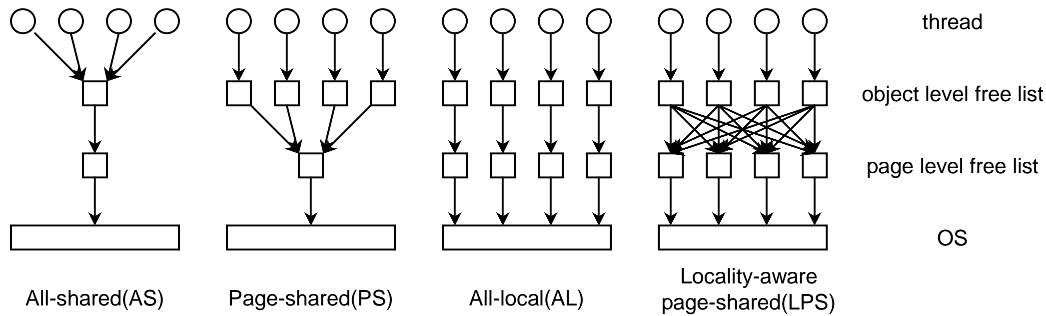


図 1: 並列 BIBOP 型アロケータの設計方式。左から All-shared(AS), Page-shared(PS), All-local(AL), Locality-aware-page-shared(LPS, 提案方式)。矢印はメモリ確保要求を表す。

### 2.1 並列 BIBOP 方式の概要

BIBOP 型アロケータにおいてヒープは複数のページから成り立ち、各ページは同一サイズのオブジェクトのみを格納する[1]。アロケータはオブジェクトレベルとページレベルの 2 レベルのフリーリストを管理する。ページサイズよりも小さい空き領域は前者で、大きい空き領域は後者で管理される。また、各ページ毎にホームスレッドとオーナースレッドを記録する。前者はページの最初の確保者、後者はページの最近の確保者である。各ページのホームは不変であり、オーナーはプログラム実行中に変化しうる。

PS, AL, LPS の全方式において、各スレッドがそれぞれオブジェクトレベルフリーリスト群を管理する。この理由の一つはメモリ確保処理のスケーラビリティを得るためである。またこれにより、同一ページ内のオブジェクトは同一スレッドによって確保されたオブジェクトを含むことができ、不要なキャッシュ無効化を防ぐことができる。ページレベルフリーリストの管理については次節以降で説明する。

スレッド $i$ がメモリ確保要求を行なったとき、アロケータの処理の流れは以下のようになる。ここではページサイズより小さい領域の確保について述べる。

1. スレッド $i$ のオブジェクトレベルフリーリストに空き領域があればそれを返す。リストが空で失敗したら次へ。
2. ページレベルフリーリストに空きページがあれば 1 ページ取り出し、そのページのオーナーをスレッド $i$ とする (そのページがプログラム開始後初めて確保されたのであればホームも設定する)。そのページから要求されたサイズの領域を取り出して返す。リストが空で失敗したら次へ。
3. OS から新ページを獲得する[2]か、 GC を行なって空き領域を増やす。

本稿の実験では、 GC 時に全スレッドを停止して協調的に GC を行なう。 GC によって空き領域ができると、ページ単位の空きはページレベルフリーリストに格納される (後述)。ページサイズよりも小さい空き領域は、その領域を含むページのオーナースレッドのオブジェクトレベルフリーリストに格納され、後のメモリ確保要求のために再利用される。

### 2.2 素朴な並列化方式とその問題

まず各方式のアロケータのメモリ利用効率を議論するために利用ページ量と消費ページ量を定義する。ある時点での利用ページ量 $u$ は、並列アプリケーションプログラムによって確保されたオブジェクトが存在しているページ数とする。 $u$ のうちスレッド $i$ がオーナーであるページ数を $u_i$ とする ($u = \sum_i u_i$)。消費ページ量 $a$ は、プログラム開始からその時点までにアロケータが OS から獲得したページ数とする。常に $a \geq u$ であり、 $a$ が $u$ に近いほどアロケータのメモリ利用効率は良いと見なす。

PS, AL, LPS の各方式の間の差異は空きページの管理方式にある。 PS 方式では全ての空きページを唯一

[1] 実際にはページを使いすぎないようにきりのいいサイズに切り上げて管理する

[2] 実際の実装ではシステムコール回数を削減するために複数ページを同時に要求し、別のリストに保存しておく

のページレベルフリーリストが保持する。GC などによって空きになったページは任意のスレッドによって再利用される。このためこの方式ではアプリケーションプログラムの局所性は劣る。

AL 方式では各スレッドがページレベルフリーリストを管理する。あるページが空になると、そのページのオーナースレッド ($i$ とする) のページレベルフリーリストに格納される。そのページを再利用することができるのはスレッド $i$ のみとする。各ページのオーナーは不変なので局所性は良いが、その一方で以下のような場合に消費ページ量が増大してしまう。

例えば各スレッドによる利用ページ量 $u_i$ の遷移が図 2(B) のようなプログラムを考える。PS 方式ではスレッド 1 が $m$ ページを解放した後スレッド 2 がそれらを再利用することができるため、アロケータによる消費ページ量は $a = m$ となる。一方 AL 方式ではアロケータは各スレッドのためにそれぞれ $m$ づつページを消費してしまうため、$a = 2m$ となる。一般に AL は最悪の場合 PS の $P$(= スレッド数) 倍ページを消費しうる。

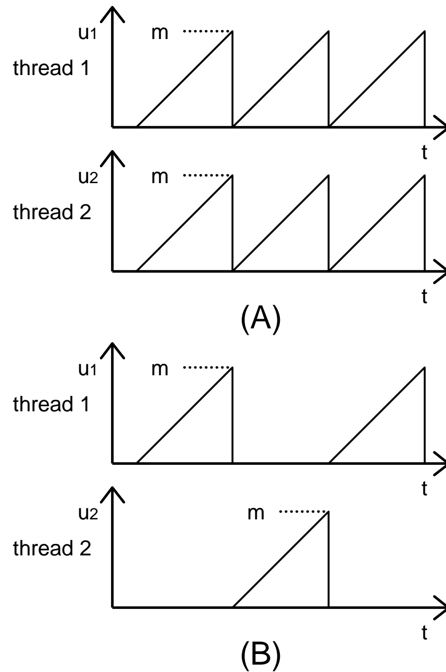


図 2: グラフは並列プログラムによるページ利用量の遷移の例を示す。(A) は PS, AL ともアロケータによる消費ページ量が同等であるようなプログラムの様子。(B) のようなプログラムでは、AL は PS のスレッド数倍メモリを消費してしまう

### 2.3 提案方式

本節ではメモリ利用効率と局所性のバランスを調節可能な LPS 方式を提案する。LPS 方式の特徴は、PS 方式よりメモリを積極的に消費することにより局所性を向上させることである。与えられた許容消費倍率 $k(k \geq 1)$ に対して、LPS 方式の消費ページ量が PS 方式の $k$ 倍以下かつ AL 方式の消費量以下であることを保証する。

LPS 方式は AL 方式と同様に各スレッド毎にページレベルフリーリストを管理する。プログラムの実行中の任意の時点で、各スレッドのページレベルフリーリストはそのスレッドにとってローカルな空きページのみを含む。これは、GC によってできた空きページをそのホームスレッドのページレベルフリーリストに格納することによって実現される。

スレッド $i$ によるページ確保処理 (2.1節の 2.) の詳細は以下の通りである。

- 2-1. 自リストに空きページがあればそれを返し、終了。失敗したら次へ。
- 2-2. 後述の関数 *heap-expansion-allowed()* が真を返したらすぐ 3. へ。偽であれば 2-3. へ。
- 2-3. 他スレッドのリストを次々に検査し空きページを探す。成功した場合はそのページのオーナーを $i$ とする。失敗したら 3. へ。

*heap-expansion-allowed()* は、$a < k \max l$ かつ $a < \sum_i \max u_i$ のとき真を返す関数である。ここで $a$ はアロケータによる消費ページ量、$l$ は最近の GC で生き残った全ページ数、$\max l$ はプログラムの実行開始からその時点までの $l$ の最大値である。

直観的には、$\max l$ はプログラムが必要とするメモリ量の近似値である。LPS 方式は、$a$ がその $k$ 倍未満であれば、積極的に OS から新ページを獲得する。$k$ が大きいほどリモートページの代りに新ページを獲得する可能性が増加するため、局所性は向上する。このように LPS 方式では、ユーザはプログラムの性質に応じて $k$ を調節することによりメモリ利用効率と局所性のバランスを変更することができる。

## 3　メモリ消費量解析

本章では、各方式のアロケータによる消費ページ量の解析を行なう。PS，AL，LPS の各方式の消費ページ量 $a_{ps}, a_{al}, a_{lps}$ について、プログラムによる利用ページ量 $u, u_i$ との関連を述べる。

本来、利用ページ量はプログラムの挙動だけでなく GC の起るタイミングに影響されるため、アロケータ方式の違いに影響を受けるが、議論を簡潔にするためにどの方式でも $u, u_i$ の遷移の様子は同じと仮定する。本章ではどのような $u, u_i$ の遷移のもとでも、各時点で $a_{lps} \leq ka_{ps}$ かつ $a_{lps} \leq a_{al}$ を満たすことを示す。

PS 方式における共有ページレベルフリーリスト中のページ数を $d$ とする。空きページは任意のスレッドの確保要求に使われるため、アロケータが OS からページを獲得する ($a_{ps}$ が増加する) 必要があるのは $d = 0$ のときのみである。このとき $u$ はその時点までの最大値であり、 $a_{ps} = \max u$ が成り立つ。ここで max はプログラム開始から各時点までの最大値を表すとする。

AL 方式において、スレッド $i$ のページレベルフリーリスト中のページ数を $d_i$ とする。 AL 方式では空きページのスレッド間移動は起らないので、各スレッドによる消費ページ量を $a_{al,i} = u_i + d_i$ とすると、この値はスレッド独立に決まる。各スレッドは $d_i = 0$ になればすぐに OS からページを確保する。このため $a_{al,i} = \max u_i$ であり、 $a_{al} = \sum_i \max u_i$ となる。

以下、 LPS 方式のメモリ消費量を考察する。

以下で $a_{lps} \leq ka_{ps}$ であることを示す。スレッド $i$ がページを確保しようとして $d_i = 0$ であることを発見したときの動作は以下のうちのどれかである。

1. $a_{lps} < k \max l$ のときは OS からページを獲得することが許されるので、 $a_{lps}$ は増加する。
2. $a_{lps} \geq k \max l$ かつ $d > 0$ のとき、他スレッドは空きページを持っている。このとき空きページを他スレッドから獲得するので $a_{lps}$ は増加しない。
3. $a_{lps} \geq k \max l$ かつ $d = 0$ のとき、新ページを OS から獲得するしかない。この場合は $a_{lps} = \max u$ が成り立つ。

以上の考察により、 $a_{lps}$ の値が増加するのは (1), (3) の場合のみなので、常に $a_{lps} < k \max l$ または $a_{lps} = \max u$ が成り立つ。$l$ の定義から $l < u$ なので $k \max l < k \max u$、 $k \geq 1$ から $\max u \leq k \max u$ が成り立つ。よって $a_{lps} \leq k \max u = ka_{ps}$ が成り立つ。

また、 *heap-expansion-allowed()* 関数の条件 $a_{lps} < \sum_i \max u_i$ より、 $a_{lps} \leq a_{al}$ が言える。以上から、 LPS 方式の消費ページ量は 2.3節で述べた通り $a_{lps} \leq ka_{ps}$ かつ $a_{lps} \leq a_{al}$ となることが分かった。ここで $k = 1$ のときに PS 方式の消費量と同等に、 $k$ が充分大きいときには AL 方式と同等になる。 $k$ がその中間の場合にはメモリ消費量も中間となる。

## 4　性能評価

本稿で説明した PS，AL，LPS の各方式を、筆者らがこれまでに実装した並列ガーベージコレクション (GC) ライブラリ [4, 10] に組み込むことにより実装した。このライブラリは C や C++ のための保守的 GC ライブラリである Boehm-Demers-Weiser GC [3] を基に並列化拡張したものである。アプリケーションプログラムは malloc の代替関数 GC_malloc を呼ぶことにより本並列アロケータを利用することができる。

本並列アロケータの性能を DSM 型計算機 Origin 2000 上の実験により評価する。 Origin 2000 は複数のノードから成り立ち、1 ノードは 195MHz の R10000 プロセッサ 2 つとメモリノードから成り立つ。本章ではまず各方式のメモリ確保処理のスケーラビリティについて述べ、その次に各方式のメモリ消費量、局所性を比較する。

### 4.1　実験に用いたプログラム

2 つのベンチマークプログラムと 2 つのアプリケーションを作成して、実験を行なった。これらは C++ 言語で記述されている。アプリケーションの記述には StackThreads/MP スレッドライブラリ [9] を用いた。このライブラリは多数のユーザスレッドを固定数のカーネルレベルスレッド上でスケジュールする。実験では各カーネルレベルスレッドをそれぞれ別のプロセッサに束縛する。

Ptree ベンチマークでは各スレッドはバリア同期をとりながらそれぞれ木構造を繰り返し作成する。木のノードオブジェクトの大きさは 16 バイトである。

Vserver ベンチマークでは各スレッドは仮想的なタスクスケジューラからタスクを 1 つずつ受け取り次々

に実行する。タスクはメモリを大量に使うもの (メモリ型タスク) と、メモリをほとんど使わないもの (プロセッサ型タスク) の 2 種類がある。与えられた定数 $v$ に対して、タスクスケジューラは、同時に最大 $v$ 個のスレッドにメモリ型タスクを、それ以外のスレッドにプロセッサ型タスクを与える。実験では各メモリ型タスクは (20MB/$v$) のメモリを利用し、タスク終了後にはそのメモリを破棄することができる。このためこのベンチマークの利用メモリ量は $v$ に関わらず約 20MB となる。$v$ が大きければ各スレッドはいつもほぼ同程度の量のメモリを利用し、図 2(A) に近くなる。$v$ が小さければ、利用メモリ量はスレッド毎にばらつき、図 2(B) に近くなる。実験ではスレッド数 $P = 12$ とした。

CKY アプリケーションは自然言語の文法規則ファイルと文章を入力とし、可能な構文解析木を全て求める [6]。実験では 36〜97 語の文を 85 文解析する。

Cube アプリケーションは Rubik's cube パズルの近似解を枝刈りつき幅優先探索によって求める。

### 4.2 メモリ確保処理のスケーラビリティ

Ptree ベンチマークを用いてメモリ確保処理のスケーラビリティを示す。図 3は ptree の速度性能を、PS 方式、AL 方式、LPS 方式 (k=1) について示す。また libc の malloc を用いたときの性能も示す。横軸はスレッド数を表し、縦軸は 1 秒あたりのメモリ確保回数を表す。計測時間からは GC 時間 (libc では free 時間) を省いているので、縦軸はメモリ確保のピークスループットと考えてよい。実験により、Libc は逐次速度は本アロケータよりも 1.3 倍程度速いが、並列時には全くスケーラビリティを得られないことが分かった。それに対し、フリーリストをスレッド毎に管理する PS, LPS, AL ではメモリ確保処理並列化の効果が見られる。PS のスループットが他より低いのは共有のページレベルフリーリストに対するアクセスコストのためと考えられる。現在の実装では AL では 48 スレッドで逐次時の 28 倍、LPS では 24 倍のスループットを達成している。

### 4.3 消費メモリ量と局所性

Vserver ベンチマークを用いてメモリ消費量と局所性のトレードオフを議論する。図 4は vserver における消費メモリ量 (消費ページ量とページサイズの積) を示す。横軸は前に述べた定数 $v$ を表し、縦軸はアロケータが OS から確保したメモリ量 (単位は megabytes) を表す。グラフの折れ線はそれぞれ PS 方式、 AL 方式、そして許容消費倍率 $k$ を様々に変えた LPS 方式を表す。

PS 方式における消費メモリ量は $v$ に関わらずほぼ一定で済んでいる[3]。一方 AL 方式では $v$ が小さくなりスレッド間の利用メモリ量が不公平になるにつれ、アロケータはメモリを大量に消費してしまう。$v = 1$ の場合はほぼ $20MB \times P = 240MB$ のメモリを消費してしまう。 LPS 方式では、 $k = 1$ のときには PS 方式とほぼ同等の結果であり、 $k$ が増加するにつれ消費メモリ量は増加し AL 方式に近づく。しかし AL 方式と違い、 $v$ が小さくなってもほとんどの場合に PS 方式の消費量のほぼ $k$ 倍でおさえられている。以上のように、 LPS 方式は許容消費倍率 $k$ の値により消費メモリ量を調節できることが分かった。

図 5は vserver ベンチマークにおける局所性を示す。横軸は図 4と同じく定数 $v$ を表す。縦軸は、各スレッドがページを確保したときに、そのページのホームが別のスレッドだった確率 (以降、リモートページ確保率) を表す。

AL 方式では常にローカルページを確保することができるのでリモートページ確保率は 0 である。逆に PS 方式ではリモートページ確保率は $v$ に関わらず 0.9 程度になってしまい、局所性は悪い。また LPS 方式におけるリモートページ確保率は AL 方式と PS 方式の中間であることが分かる。$v$ または $k$ が大きいほど、リモートページ確保率は低くて済む。

### 4.4 アプリケーションの性能

表 1に 48 スレッド用いたときの CKY, Cube アプリケーションの性能を示す。消費メモリ量とリモートページ確保率については、期待通り LPS 方式は PS 方式と AL 方式の中間の結果となった。$k$ が大きいほど消費メモリ量は大きくなるが、局所性は良くなる。CKY の実行時間 (ただし GC 時間を除く) は、 AL が最も速く、 PS が遅い。この差には、メモリ確保処理のスケーラビリティと、局所性の差によるアプリケーションのメモリアクセスコスト差の両方が含まれると考えられる。 AL は PS より約 8%、LPS($k = 4$) は PS より約 7% 高速である。一方 Cube ではリモート

[3]消費メモリ量が 20MB を越えてしまう場合がある理由は、手動メモリ管理ではなく GC を用いているために、タスク終了から時間が経ってから実際の解放処理が行なわれているためと考えられる

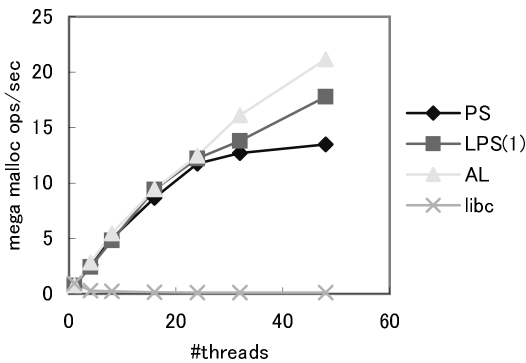


図 3: Ptree ベンチマークにおけるメモリ確保速度。横軸はスレッド数を表し、縦軸は秒あたりの malloc 回数 ($\times 1,000,000$) を表す

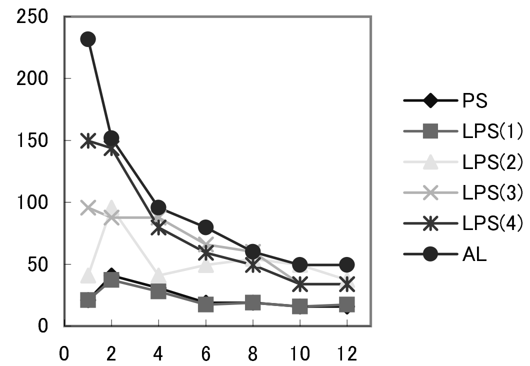


図 4: Vserver ベンチマーク (12 スレッド) における消費メモリ量。横軸は同時にメモリ確保可能なスレッド数 $v$、縦軸はメモリ消費量 (MB)。 LPS の括弧内の数値は許容消費倍率 $k$ を表す

確保率の減少にも関わらず、実行時間にほとんど差は見られなかった。これは、各スレッドが、他スレッドによって確保されたオブジェクトをアクセスする機会が多く、アロケータの局所性向上による利益を得ることができなかったためと考えられる。

## 5 関連研究

OS の標準ライブラリのアロケータの多くは並列化されていないため、メモリ確保を大量に繰り返す並列プログラムの多くはそれぞれメモリ管理ルーチンを自作している。 Apache HTTP サーバ [5] もその一つである。このアプローチはプログラマへの負荷が高い。

共有メモリ並列計算機上の汎用並列アロケータはこれまでいくつか提案されている。 Larson らの並列アロケータ [7] はスケーラビリティを達成するために複数ヒープを用いて共有資源へのアクセスを削減する。ページ単位の空き領域は任意のスレッドに再利用されうる。 Boehm の並列 BIBOP 型アロケータ [2] はオブジェクトレベルフリーリストの一部を各スレッドが持ち、他の資源はスレッド間で共有する。 Berger らの並列 BIBOP 型アロケータ [1] はスケーラビリティとメモリ利用効率に注目し、スレッド固有ヒープと共有ヒープの両方を用いる。空ページだけでなく部分的に空いたページもスレッド間で共有するのでメモリ利用効率は本稿のアロケータよりも良い。その一方、アプリケーションがキャッシュ無効化の影響を受ける可能性が高くなる。彼らはアロケータの消費メモリ量の理論上の上限と実験結果を示した。

いずれも並列アプリケーションに対するスケーラビリティに注目している一方、 DSM 型計算機における局所性については考慮していない。これに対して本稿のアロケータはスケーラビリティ、メモリ利用効率、局所性に注目する。

## 6 おわりに

本稿は共有メモリ並列計算機のための BIBOP 型アロケータの並列化方式である Locality-aware-page-shared(LPS) 方式を提案し、性能を示した。目的はスケーラビリティ、 DSM における良好な局所性、高メモリ利用効率を達成することである。メモリ利用効率と局所性はトレードオフの関係にある。 LPS 方式は、空きページを共有する PS 方式に比べて消費メモリ量が $k$ 倍 ($k$ はユーザが指定できる許容消費定数) 以下であることを保証する。そして PS 方式よりメモリを積極的に消費することにより局所性を向上させる。

DSM 型計算機 Origin 2000 上での実験を通して、消費メモリ量と局所性の調節がうまく働くことを示した。また実験によって、メモリ確保処理のスループットは 48 スレッドで逐次時の 24 倍であり本方式がスケーラブルであることを示した。

今後は GC のタイミングと消費メモリ量の関係を詳細に研究する必要がある。さらに局所性の理論上の上限や許容消費定数 $k$ の自動選択に関する研究を進める予定である。

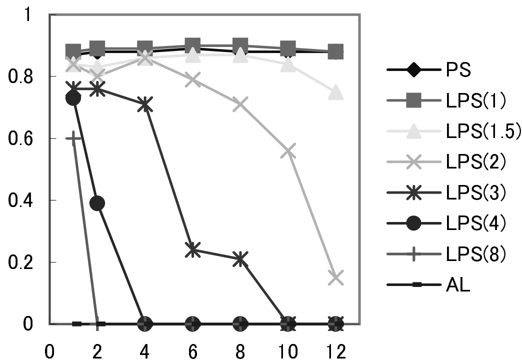


図 5: Vserver ベンチマーク (12 スレッド) におけるリモートページ確保率。横軸は同時にメモリ確保可能なスレッド数 $v$ を表す

**謝辞** 実験のために東京大学医科学研究所ヒトゲノム解析センターの計算機を利用させて頂きました。

| | 実行時間 | 消費<br>メモリ量 | リモート<br>確保率 |
|---|---|---|---|
| CKY/PS | 22.1 | 37.3 | 0.90 |
| CKY/LPS(1) | 21.7 | 33.2 | 0.25 |
| CKY/LPS(2) | 21.0 | 59.8 | 0.12 |
| CKY/LPS(4) | 20.7 | 67.3 | 0.00 |
| CKY/AL | 20.4 | 67.3 | 0.00 |
| Cube/PS | 24.5 | 11.4 | 0.68 |
| Cube/LPS(1) | 25.0 | 10.2 | 0.54 |
| Cube/LPS(2) | 24.7 | 14.5 | 0.00 |
| Cube/LPS(4) | 24.2 | 16.3 | 0.00 |
| Cube/AL | 24.2 | 18.4 | 0.00 |

表 1: アプリケーションの性能 (48 スレッド)。各アロケータ方式について、実行時間 (秒、 GC 時間除く)、消費メモリ量 (MB)、リモートページ確保率を示す

## 参考文献

[1] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[2] Hans-Juergen Boehm. Garbage collector scalability. http://www.hpl.hp.com/personal/-Hans_Boehm/gc/scale.html.

[3] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[4] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of ACM/IEEE Conference on High Performance Networking and Computing (SC97)*, November 1997.

[5] The Apache Software Foundation. Apache http server project. http://httpd.apache.org.

[6] T. Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, 1965.

[7] Per-Ake Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 176–185, October 1998.

[8] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, 1997.

[9] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating futures into calling standards. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 60–71, May 1999.

[10] 遠藤 敏夫, 田浦 健次朗, 米澤 明憲. 大規模共有メモリ並列マシンにおけるスケーラブルなマークスイープ法ガーベージコレクタ. In 日本ソフトウェア科学会第 *15* 回大会論文集, September 1998.