分散記憶並列計算機における局所ごみ集めのスケジュール方式について

田浦 健次朗† 米澤 明憲†

専有された，分散記憶並列計算機において，局所ごみ集めのスケジュール戦略が性能に与える影響について考察し，実験結果を報告する．分散記憶並列計算機におけるごみ集め（GC）方式は一般に，**局所 GC** と**大域 GC** を組み合わせて用いる．各プロセッサはメモリのある一部の領域を，他のプロセッサの協調なしに回収するために局所 GC を行い，複数のプロセッサにまたがるごみを回収するのに大域 GC（参照カウントや大域マークスイープ）を行う．局所 GC のそもそもの動機は，特に大規模な並列計算機では高価である，プロセッサ間の協調や通信をできるだけ避けることであるため，それらは一般に各プロセッサによって独立にスケジュールされることが多い（**独立スケジュール方式**）．しかし，我々の実験結果はそのようなスケジュール方式は，通信遅延に敏感なアプリケーションの性能を大きく低下させることを示している．この原因は，そのようなスケジュールによって，GC 中のプロセッサが，入ってくるメッセージに対して反応が遅くなることにある．一方で，全プロセッサが同時にごみ集めを行う，**同期スケジュール方式**は，それによって余分な同期やごみ集めのための仕事が生じるにもかかわらず，はるかに頑強な性能特性を示す．さらに，アプリケーションの通信挙動を観測することで，実行時に望ましいスケジューリング戦略を選ぶことが可能であることを示す．

On Local Garbage Collection Scheduling Schemes for Distributed Memory Parallel Computers

Kenjiro Taura† and Akinori Yonezawa†

We investigate performance impact of local garbage collection scheduling strategies on parallel applications running on distributed memory, dedicated parallel computers. Garbage collection schemes on distributed memory computers generally combine a *local* and a *global* collection; each processor performs a local collection to reclaim a part of its memory without coordinating processors and a global collection (*e.g.*, reference count or global mark-and-sweep) to detect garbage that spans multiple processors. Since the motivation of local collections is to avoid coordination and communication between processors, which are expensive especially on large-scale distributed memory computers, they are commonly scheduled *independently*, eliminating communication between processors altogether. Our experiments show, however, that such a scheduling strategy significantly slows down latency-sensitive applications, due to the fact that collecting processors become unresponsive to incoming messages. On the other hand, *synchronized* schedule of local collections, in which all processors perform local collections at the same time, exhibits much more robust performance, despite the additional synchronization and possible extra collections. Finally, it is shown that the application's communication behavior can guide the preferred scheduling strategy at runtime.

1. はじめに

分散記憶並列計算機におけるごみ集め（Garbage Collection；以下 GC）方式として，数多くの方式が提案されている[1),12),19)]．それらは大域マークスイープ方式[11),13)]と，参照カウント方式[3),7),18)]に大別される．どちらの方式をとるにせよ，それらは多くの場合**局所 GC** を併用している．局所 GC で回収可能なごみには制限があるが，一方でプロセッサ間の協調動作や通信をほとんど，またはまったくせずに，ごみを回収できる．これは，通信オーバヘッドの大きいメッセージ交換型計算機，プロセッサが協調して GC を行うためのコストが大きい大規模な並列計算機環境などで有望な方式である．

既存のシステムで局所 GC がどのようにスケジュールされるかについて明示的に述べている文献はあまり存在しないが，それは一般的にプロセッサごとに**独立に**スケジュールされていることが多い（**独立スケジュール**）．つまり，各プロセッサが必要に応じて自律的に局所 GC を起動する．これはそもそも局所 GC が，プロセッサ間の協調を少なくして GC を可能にするため

† 東京大学
University of Tokyo

の方式であることを考えれば，自然なことである．

本論文ではこの，局所 GC のスケジュール方式に焦点を当てる．具体的には，上で述べた独立スケジュールに対して，**同期スケジュール**と呼ばれる，各プロセッサが同時に局所 GC を行う方式とを比べ，スケジュール方式がアプリケーションの性能全体に与える影響を実験によって示す．

我々の GC は，Boehm らによる 1 プロセッサ用の保守的 GC ライブラリ（以下，Boehm GC）[4),5)] を基にしており，並列オブジェクト指向言語 ABCL/f 用の GC として実装されている．実験は 256 プロセッサからなる富士通 AP1000+ [10)] 並列計算機上で行った．実験結果およびその解析から得られる知見は以下のとおりである．

- アプリケーションの性能は局所 GC のスケジュール方式に大きく影響を受ける．特に，同期通信を頻繁に行い，プロセッサ内に並列性を持たないアプリケーションは，独立スケジュールによって大きく性能が損なわれる．これは局所 GC がばらばらに起こることによって，局所 GC 中のプロセッサが他のプロセッサからのメッセージに反応しなくなり，それによってメッセージに余分な遅延が生ずるためである．
- 同期スケジュールに必要な，GC 開始時の同期のコストは，我々が実験した環境（256 プロセッサまで）においては大きくない．同期は単なる 1 対 N の通信によって行われる．したがって，つねに同期スケジュールを行うという単純なスケジュール戦略ですら，許容できる戦略である．
- さらに，アプリケーションの通信挙動を実行時に調べることで，適切な局所 GC スケジュールを自動的に選択する，**適応的**スケジュールも有効に機能する．**適応的**スケジューラは，基本的には同期スケジュールを用い，アプリケーションが遅延に対して強いと判断したときには独立スケジュールを用いる．

本論文の構成は以下のとおりである．2 章で背景について述べる．3 章で局所 GC のスケジュール方式について詳しく述べる．4 章と 5 章で実験の設定と結果を述べる．6 章で関連研究について述べ，7 章でまとめを述べる．

2. 背　　景

前章の冒頭で述べたとおり，並列および分散環境における GC は参照カウント方式と分散マークスイープ方式に大別されるが，どちらの方式を用いるにせよ，実際に稼動している GC のの多くは局所 GC の併用を前提としている．それは以下で述べるように，GC 全体の効率を上げるのに多くの場合役に立つ．以下に参照カウント法，分散マークスイープのそれぞれについて，局所 GC の果たす役割についてまとめておく．

2.1 遠隔参照カウント

参照カウント法と局所 GC を組み合わせて用いる場合，通常参照カウントは他のプロセッサからの参照に対してのみ管理される（**遠隔参照カウント法**）．つまり，プロセッサ P のメモリ内に割り当てられたオブジェクトに対し，P 以外のいくつのプロセッサがそれを参照しているか（遠隔参照カウント）を記録する．遠隔参照カウントが 1 以上のオブジェクトは局所 GC の際，ルートの一部と見なされ，回収されない．そうでないオブジェクトは，局所 GC によって（他のルートから到達可能でない限り）自然に回収される．

あるオブジェクト O の遠隔参照カウントが減少するのは，あるプロセッサが O をもはや参照していないことを検出したときである．これを検出するためにも局所 GC を用いる．つまり，あるプロセッサ P が局所 GC を行った結果，P のルートから到達可能なオブジェクトが発見される．P が使用している遠隔参照は，単にそれらのオブジェクトが使用している遠隔参照の集合である．そうでない遠隔参照はもはや P は用いていないことになるので，そのような遠隔参照が指すオブジェクトの参照カウントを減らすことができる．

ここで注意すべきは，局所 GC によって局所参照に対しては参照カウントを保持する必要がなくなることである．もし局所 GC を行わないとすると，遠隔参照の消滅を検出するためにはその遠隔参照を保持しているオブジェクトの消滅を検出しなくてはならず，それを GC なしで行うにはそのオブジェクトに対する局所参照も含めたすべての参照の数を数えなくてはならなくなる．このオーバヘッドは非常に大きいのでほとんど用いられない．局所 GC の併用はそうするかわりに，プロセッサ内のグラフ全体を時折ルートから走査して，ルートから到達可能な遠隔参照を並べあげる操作であるととらえることができる．この意味においてローカル GC の併用は参照カウント方式にとっては，ほぼ必須のものといえる．

2.2 大域マークスイープ

大域マークスイープは局所 GC を分散環境に単純に拡張したものである．局所 GC が局所参照をたどっていくのと同じように，大域マークスイープは各プロセッサのルートから局所参照をたどっていき，遠隔参照に遭遇したら参照先のオブジェクトを持つプロセッ

サに「マークメッセージ」を送る．マークメッセージを受け取ったプロセッサはマークメッセージが示すオブジェクトからさらに局所参照をたどりつづける．

局所 GC は，参照カウント方式の場合と同様に，それに対する遠隔参照が残っているオブジェクトをルートの一部と見なして GC を行う．参照カウント方式の場合と違い，遠隔参照が取り除かれるのは，大域マークスイープ時だけである．

一般にマークスイープ方式の効率（= 一定量のメモリを回収，再利用するのに必要な仕事量）は，

$$\frac{\alpha L}{H-L}$$

で近似できる．ここで L は，その GC で走査されるオブジェクトの合計の大きさ（たとえばバイト数），H がヒープサイズである．α は 1 バイトをマークするコストを表し，したがって分子がマークスイープ 1 回あたりの仕事量を表している☆．一方，分母はマークスイープ 1 回で回収されるメモリ量を表しているので，全体として 1 バイトを回収，再利用するのに必要な仕事量を表している．

α は定数だが，局所 GC におけるそれと，大域マークスイープにおけるそれとでは大きく違う．局所 GC では α は大体，局所メモリを 1 バイト分マークするコストを表しているのに対し，大域マークスイープでは，遠隔参照に対して送られるマークメッセージのオーバヘッドを含んでいる．

大域マークスイープ法に局所 GC を併用することは，局所 GC が小さな α で領域を回収することで，GC 全体の効率を上昇させる効果を持つ．

2.3 局所 GC のスケジュール方式

局所 GC のスケジュール方式としてすぐに考えられるものとして，局所 GC を全プロセッサで同時に行う方式（**同期スケジュール**）と，各プロセッサが必要になったときに自律的に行う方式（**独立スケジュール**）がある．局所 GC のそもそもの意義——プロセッサ間の協調を最小限にする——を考えると，直感的には独立スケジュールが勝ると考えるかもしれない．しかし実際には独立スケジュールでは，GC 中のプロセッサが他のプロセッサからのメッセージに応答しなくなり，これによって他のプロセッサのアイドル時間が増大する危険性がある．そのような状態が頻発するならば，それらのプロセッサもその時間を GC に使う——つまり，同期スケジュールを行う——方が良いことになる．

以降ではこの仮説を実験によって示し，さらに正しいスケジュールの方式を実行時に獲得する方式を示していく．

3. ごみ集めの方式

本章では，4 章の実験で用いる，局所ごみ集めのスケジュール方式について詳述する．実験に用いられた GC は大域マークスイープと組み合わされており，参照カウントは管理していない．

3.1 輸 出 表

ほとんどの分散記憶並列計算機上の GC システム同様，他のプロセッサから指されているかもしれないオブジェクトを各プロセッサが保守的に見積もるために，各プロセッサは**輸出表**と呼ばれる表を管理する．

オブジェクトが最初にあるプロセッサに生成されたときはそれは輸出表に登録されていない．オブジェクト O のプロセッサ P が別のプロセッサに O の参照を送るとき，O が輸出表に，まだ登録されていなければ，登録される．

局所 GC は輸出表をルートの一部と見なして，登録されているオブジェクトおよびそこから局所ポインタによって到達可能なオブジェクトは回収しない．いったん輸出表に登録されたオブジェクトが除去されるのは大域マークスイープによってのみである．

3.2 基本的なスケジュール戦略

Boehm GC を含む，多くの 1 プロセッサ用の GC は，GC をどの時点で起こすべきかどうかを以下のように判断する．

現在のヒープがあふれたときに，そのプロセッサの現在のヒープサイズを H，最後の GC が行われてからそのときまでに割り当てられたメモリの量を A，f は調節可能な定数として，

$$A > \frac{H}{f}$$

であれば GC を行い，そうでなければ GC を行わずにヒープを拡張する．つまり，システムは隣接した 2 回の GC の間に少なくとも H/f だけのメモリが確保されるように，必要ならばヒープを拡張する．

たとえば $f = 3$ として，アプリケーションが M だけの長寿命（GC を生き残る確率がほぼ 1）なオブジェクトを保持し，その他に短命（GC を生き残る確率がほぼゼロ）なオブジェクトを割り当てつづけるとする．この状態では H は最終的に $1.5M$ 程度に落ち着き，アプリケーションは $0.5M$ $(= H/3)$ の短命なオブジェクトを生成しては，GC を行って $0.5M$ の領

☆ 厳密にはここでスイープのコストを無視している．それは回収されたメモリを再利用するための手続きを，実際のメモリ割当て要求まで遅らせること（遅延スイープ）によって，近似的には α の増大に還元できる．

域を開放する，という定常状態になる．

これは多くの1プロセッサ用のGCが行っている典型的な経験則で，アプリケーション全体に占めるGCのコストの上限を保証しつつ，かつメモリの使用量をアプリケーションが実際に使うデータに比例したものに保つ．我々はこの方針を，分散記憶の並列計算機用に，以下のように拡張した．

基本的な方針は，GCを行うのはあるプロセッサが最後のGCから H/f のメモリ割当てを行い，かつそのプロセッサのヒープサイズが全プロセッサ中の最大のヒープサイズに十分近い場合に，GCを行うというものである．具体的には以下のようである．

まず，全プロセッサ中で最大のヒープサイズを $H_{\max}$ として，それを全プロセッサに通知しておく（通知は定期的に，たとえば大域マークスイープを行った際に行う）．あるプロセッサでヒープがあふれた際には，そのヒープサイズが，c を1より少し小さい定数として，$cH_{\max}$ 以下であれば無条件に $H_{\max}$ まではヒープの拡張を許すものとする（実験では $c = 0.8$ としている）．たとえば，あるプロセッサが10 MBのヒープを持っている場合は，8 MB以下のヒープを持つプロセッサは，GCを行うことなく10 MB程度までヒープを拡張する．この方針は同期スケジュール戦略でも独立スケジュール戦略でも用いられる．

この方針は，各プロセッサが必要とするメモリ量が大体同じくらいであると仮定すれば妥当な方針であると考えられる．つまりあるプロセッサがある量のメモリを必要としたとき，他のプロセッサもいずれ同様の量のメモリを必要とする，という仮定である．この仮定の下では，この方針が必要とするメモリの量は妥当なものである．一方この方針を用いないとすると，同期スケジュール戦略は不必要に頻繁なGCを強いられてしまうだろう．なぜならば，1つでもヒープの小さな，したがってGCを頻繁に起こすプロセッサがいると，GCの頻度はそのプロセッサが起こすGCの頻度まで増大してしまうからである．

3.3 局所GCの独立/同期スケジュール戦略

この方針の下であるプロセッサがGCを行うと判断したときに，**独立スケジュール**においては，GCを行うと判断したプロセッサが，他のプロセッサへ通知することなしに局所GCを行う．一方，**同期スケジュール**においては，GCを行うと判断したプロセッサが，他の全プロセッサに局所GCを行う命令を送信し，すべてのプロセッサが局所GCを行う．

適応スケジュールは，各々の局所GCを独立に行うか同期させて行うかを実行時の情報をもとに判断する．局所GCが独立に行われることで生じる効果は，各プロセッサが局所GCの間，外からのメッセージに応答しなくなり，その結果そのメッセージの返答を待つプロセッサが遅延を被るということである．このことによる性能の低下は同期メッセージを頻繁にやりとりするようなアプリケーションでは大きく，通信をほとんどしないか，あるいは通信をしても遅延に耐えるだけの十分なノード内並列性があるようなアプリケーションでは少ない．適応スケジュールの基本的な考え方は，独立してGCをスケジュールすることによる性能の低下を，アプリケーションの通信挙動を見ることで見積もることである．

基本的な判断基準は以下のとおりである．

(1) 他に判断基準がないとき（プログラム開始時や不明確なとき）は同期スケジュールを行う．
(2) アプリケーションが通信遅延に強いと判断したときは，徐々に独立スケジュールに移行する．
(3) 独立スケジュールが誤りであると判断したときは，ただちに同期スケジュールに戻る．

主な問題は2番目，つまり現在同期スケジュールを行っているアプリケーションが実際には遅延に強いことを，どのように判断するかである．これを知るために，ある時間帯においてプロセッサが何回アイドル状態を経験したかを判断の材料にする．詳細は以下で述べるが，プロセッサが活性状態とアイドル状態を多数回繰り返すような状態では，アプリケーションは遅延に弱い，というのが基本的な考えである．これはその多数回のアイドル状態はメッセージの受信待ちによって生じており，そのメッセージの処理が，受信先のプロセッサが局所GCによって遅れると，その分アイドル時間が増える，との仮定に基づいている．この仮定の下で，各プロセッサは，ある区間で生じた各アイドル状態が，後に述べるある確率で，局所GCに対応する時間だけ長くなったときに，どのくらい性能が低下するかを予想する．各プロセッサはこの数値を1つのマスタープロセッサに通知し，ほとんどのプロセッサが小さな性能の低下を予想したときにアプリケーションが通信遅延の増大に強いと判断する（**図1**）．

具体的な計算法は以下のとおりである．現在システムは同期スケジュールを行っており，これから同期GCを行うところであると仮定する．各プロセッサに対して最後の局所GCから現在までの区間に各プロセッサが経験したアイドル状態の回数を n とする．そして，この区間に対する**性能低下指数** D を以下で定義する．

$$D = \frac{npG}{2(L+G)} \quad (\text{ただし，} p = G/(L+G)).$$

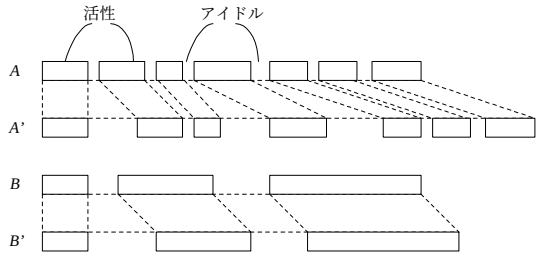


図 1 アプリケーションの，遅延の増大に対する耐性を判定する方法．各線分がプロセッサの活性状態，その間がアイドル状態を表す．A は頻繁に活性 ↔ アイドルを推移し B はそうではない．各アイドル状態はメッセージの受信待ちであり，そのメッセージは送信プロセッサで局所 GC が起こると，その分遅れて到着する．各アイドル状態がある確率（図では 1/2）で長くなると，A の性能は B に比べて大きく損なわれる（A' および B'）．すなわち A は B に比べて遅延の増大に弱い．

Fig. 1 Estimating latency-tolerance of applications. Each line segment represents an active period whereas a gap an idle period. A frequently transits between active and idle states, whereas B does not. In each idle period, the processor is waiting for a message, which will be delayed if the sender is performing local GC. Suppose each idle period is thus prolonged by a local GC with a probability (1/2 in the figure); A would suffer more severely than B (see A' and B'). That is, A is more sensitive to an increased latency than B is.

ここで，G は 1 回あたりの局所 GC にかかる時間，L は隣接する 2 つの GC 間の時間の見積もりである（これはプログラム実行中の時間を計測して得る）．したがって，$p = G/(L+G)$ は，1 つのメッセージが，宛先プロセッサが局所 GC 中に到着する確率を表す．そのようなメッセージは平均して $G/2$ だけの余分な遅延を被る．したがって，np と $G/2$ をかければその区間で独立スケジュールを行っていたときに生じたであろう遅延の総量が見積もられる．それを区間の長さ $L+G$ で割ったものが D である．まとめると，D はその区間がどれだけの割合で遅くなるかを表している．

D のほかに，各プロセッサの利用率（非アイドル時間の割合）もマスタープロセッサに集められる．マスターは以下の条件のときに独立 GC が有効であると考える．

(1) ある閾値（実験では 0.3）よりも大きい D を報告したプロセッサが全プロセッサ数のある割合（実験では 1/8）よりも少なく，

(2) ある閾値（実験では 0.5）よりも少ない利用率を報告したプロセッサが全プロセッサ数のある割合（実験では 1/2）よりも少ない．

条件 (1) は遅延が増えたときに被害を受けるプロセッサが少なく，したがってアプリケーションが遅延に強いことを表している．条件 (2) は多くのプロセッサがアイドルであり，したがって GC のたびに大域的な同期を行うことはどのみちほとんど問題にならないような場合を除外している．

一方，システムが独立スケジュールを行っているときは各プロセッサは各局所 GC の間のプロセッサの利用率を監視している．もし利用率がある閾値（$= 0.5$）よりも少なければ，そのことをマスタープロセッサにつげる．マスターはそのような通知がある閾値（$=$ 全プロセッサ数の 1/16）よりも増えた時点で，同期スケジュールに戻る．

実際の状態遷移は 3 状態（-1，0，および 1 の 3 状態）の飽和カウンタで制御する．システムが独立 GC が望ましいと判断するとカウンタを 1 減らし，-1 に達した時点で，独立スケジュールに移行する．これによってアプリケーションがたまに遅延に強い部分を持つだけで，スケジュール方針が意味なく振動することを抑止している．一方，独立スケジュールを行っている状態から，プロセッサ利用率の低下を判断すると，ただちに同期スケジュールに移行する．

本方式の制限について：明らかにこの方式は，同期スケジュールと独立スケジュールの**どちらが良いか**を正しく判断しているわけではない．基本的な仮定は，（後の実験結果が示すように）同期スケジュールはほぼつねに安全であるので，よく分からないとき，あるいは独立スケジュールを行ってもあまり利益がないとき（プロセッサの利用率が小さいとき）は，つねに同期スケジュールを行えばよいというものである．そして，プロセッサの利用率が高く（したがって GC の仕事量を削減することが全体の性能向上につながる可能性が高い），かつ独立スケジュールを行うことの性能低下がほとんどないと判断される場合のみ，独立スケジュールを行う．

性能低下を判断するために用いた方式にも制限がある．基本的な考えは頻繁にアイドル状態に陥る（つまりアイドル状態とそうでない状態を頻繁に繰り返す）アプリケーションは，メッセージに対する返答の遅れによって被る性能低下が大きく，一方稀にしかアイドルにならないアプリケーションは，たとえそのアイドル時間自体は長かったとしても，それらにいくらかの遅延が加わったところで大して影響を受けない，というものである．

アイドル状態に陥る回数 n で，その影響を判断するという上で述べた方式は，通信の遅延が増大しても，アイドル状態に陥る**回数**自体はかわらない，ということを仮定している．これは，(1) ほとんど通信をしないアプリケーション，(2) 通信の遅延隠蔽を行わず，ノー

ド内並列性のないアプリケーション，(3) 各プロセッサ内にたくさんの並列性を持つアプリケーション，などにはよくあてはまる．(1) と (3) については，それらのアプリケーションはどのみち，n の値は非常に小さいため，通信の遅延が増えても，n が大きく変わることはない．(2) については，遅延隠蔽がなければ，n の値は大体通信の回数で近似される（つまりほぼすべての同期通信が，1 回のアイドル状態を生ずる）ため，それは遅延の長さにはよらない．

我々の定式化がうまく機能しないアプリケーションの 1 つの例は，「ある程度まで」の遅延に耐えるように設計されたアプリケーションである．たとえばデータの先行要求（prefetch）や，少数のスレッドを各プロセッサに持たせることである程度の遅延に耐えるようにしたアプリケーションがそれにあたる．そのようなアプリケーションが，同期スケジュールされた状態では完全に遅延を隠蔽してしまったとすると，結果として同期スケジュールされた状態ではほとんどアイドル区間を生ずることなく動き，システムは独立スケジュールに移行してしまうかもしれない．この点についてはさらなる考察および改良が必要である．我々のベンチマークにはそのようなアプリケーションは含まれていない．

うまく機能しないアプリケーションのもう 1 つの例として，アイドル時間を投機的な計算（ほかに仕事があるうちは行わないが，アイドル時間になると起動されるような計算）によって利用するようなアプリケーションがあげられる．このような状態ではプロセッサはほとんどアイドルにならないため，システムは独立スケジュールに移行する．そのような状態では実際には各プロセッサは GC による遅延の増大を被っているにもかかわらず，その遅延を余分な計算によって隠してしまうため，システムは余分な計算に多くの時間を費やすことになる．このようなアプリケーションに対しては，投機的な計算を実行中それをアイドル時間として計上するような仕組みによって対処できるだろう．

4. 実　　験

4.1 設　　定

実験は分散記憶並列計算機 AP1000+ [10] 上で，並列オブジェクト指向言語 ABCL/f [20] を用いて書かれた 4 つのアプリケーションを用いて行った．以下に AP1000+と ABCL/f について簡潔に述べる．

4.1.1 AP1000+

AP1000+ [10] は分散記憶並列計算機で，各要素プロセッサはクロック周波数 50 MHz の SuperSparc と 16 MB のメモリを搭載している．我々が実験に用いたシステムは 256 個のプロセッサを搭載している．それらは 25 MB/秒（point-to-point）のバンド幅を持つ 2 次元トーラスネットワークで結合されている．通信ライブラリは AP1000+に備わっているものを用いている．メッセージの 1 往復に要する遅延 + オーバヘッドは 40 μs（または 2,000 プロセッササイクル）程度である．

用いた OS は仮想記憶のないシングルタスクの OS で，1 つの並列ジョブはすべての CPU とメモリを専有することができる．したがって，メッセージの受信が他のプロセスによって遅れることはない．

4.1.2 ABCL/f

ABCL/f はオブジェクト指向言語を，future 呼び出しによる動的なスレッド生成（非同期呼び出し）と並列オブジェクトによって拡張したものである．並列オブジェクトは動的に生成することができ，遠隔参照によってプロセッサ間で共有できる．したがって大域 GC の主な対象である．ABCL/f はメソッドおよび通常の手続きを両方提供しており，それらはともに同期，非同期に呼び出すことができる．非同期呼び出しが ABCL/f において並列度を生成する方法である．ABCL/f の実装は非常に多数のスレッドを許容する [21]．各プロセッサに何千，あるいはそれ以上のスレッドを生成しても，それによってシステムが深刻に遅くなることはない．

4.1.3 アプリケーション

表 1 に用いたアプリケーションおよびその主なデータ構造，並列性，支配的な通信の挙動を示す．

BH [2),8),22),23)] は，階層的な木構造による近似を用いた並列の多体シミュレーションである．これはいわゆる SPMD 的な同期アプリケーションであり，各プロセッサは頻繁に同期通信を行い，ノード内に並列性は存在しない．

CKY [27] は文脈自由文法の並列構文解析アルゴリズムである．元々の逐次の CKY アルゴリズムは，Cocke, Kasami, および Younger によって提案された [15),24)]．並列アルゴリズムの概説は [17] にある．CKY はメモリ割当てを多用するアプリケーションである．通信は頻繁でかつ同期的である．各プロセッサ内の並列度はプロセッサ数や入力に依存する．

RNA [16] は RNA の 2 次構造を塩基配列から予測する，木探索アルゴリズムである．RNA はいわゆる非同期的アプリケーションで，各プロセッサが非常に多数のプロセッサ内並列性を持つ．通信は

表 1 並列アプリケーションの簡単な記述
Table 1 A brief description of parallel applications.

	記述	主なデータ	並列性	通信
BH	Barnes-Hut 法	BH 木のノード，粒子，スタックフレーム	SPMD	頻繁，同期的
CKY	CFG 構文解析	CKY 表，構文解析木，スタックフレーム	通信する多数のスレッド	頻繁，同期的
RNA	RNA2 次構造予測	途中解，スタックフレーム	並列再帰呼び出し	頻繁ではなく，非同期的
GA	遺伝的アルゴリズム	ワーカーと遺伝子	SPMD	稀，同期的

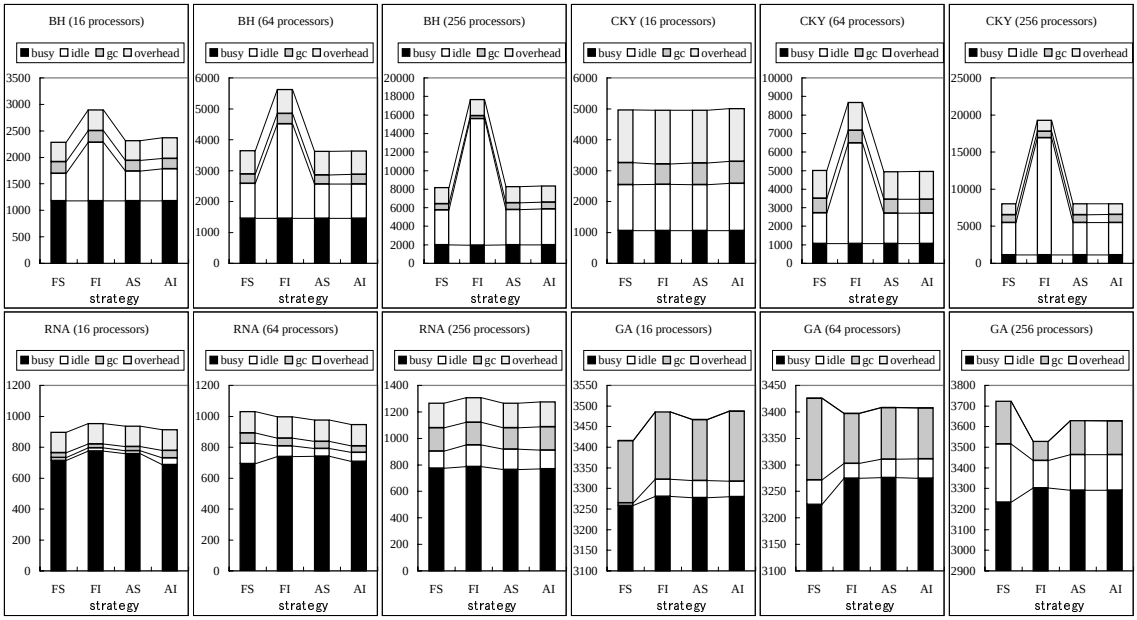


図 2 局所 GC のスケジュール戦略が性能に与える影響．グラフはアプリケーションの時間を活動中，アイドル，GC，および通信コンテクスト切替えその他のオーバヘッドに分けている．時間はすべてのプロセッサでの合計である．*FS* はつねに局所 GC を同期して行う固定同期戦略，*FI* は決して局所 GC を同期しない固定独立戦略を表している．*AS* と *AI* は適応戦略で，初期状態として同期，独立スケジュールをそれぞれ用いるものである．

Fig. 2 Effect of the scheduling strategy of local GCs. Graphs break down application runtime into active, idle, GC, and overhead (communication, context switch, etc.). Times are cumulative over all processors. *FS* refers to fixed synchronous strategy, which always synchronizes local GCs, *FI* fixed independent strategy, which never synchronizes local GCs. *AS* and *AI* refer to adaptive strategies with their initial strategy being synchronous and independent, respectively.

稀で，遅延の長さにはほとんど影響を受けない．

GA[28] は遺伝的アルゴリズムによって多目的最適化問題を解くの並列プログラムである．各プロセッサに 1 つのワーカーがおり，それらはしばらく通信をせずに仕事をし，その後で大域的なデータの交換を行う．GA はほとんど通信をしないアプリケーションである．プロセッサ内に並列性はほとんどなく，通信は同期的である．それでもなお，通信が非常に稀にしか起こらないために，GA は遅延の増大に強い．

4.2 結　　果

図 2 に，以下の 4 つのスケジュール戦略の性能を示す．

固定同期戦略（FS）： つねに局所 GC を同期させる．

固定独立戦略（FI）： 決して局所 GC を同期させない．

適応同期戦略（AS）： 適応的戦略で，初期状態は同期スケジュールを用いる．

適応独立戦略（AI）： 適応的戦略で，初期状態は独立スケジュールを用いる．

適応的戦略は 3.3 節で述べた方針に従って適切なスケジュール方式を選択する．グラフはアプリケーションの実行時間を，活性，アイドル，GC，そして通信，コンテクスト切替えなどのオーバヘッドに分類している．時間は全プロセッサの合計である．

2 つの固定戦略を比べると，どちらかがつねに勝っているわけではない．重要な点は以下のとおりである．

- CKY の 64 もしくは 256 プロセッサ，また BH に

おいてはプロセッサ数によらず，独立スケジュールの結果は非常に悪い．より興味深いことには，それらにおいても GC 時間そのものはまったく増えておらず，増えているのはアイドル時間である．つまり，局所 GC を行っているプロセッサは他のプロセッサからのメッセージに反応しなくなり，結果としてそのメッセージからの返答を待っているプロセッサ上でアイドル時間が生じているのである．

- 固定独立戦略が固定同期戦略よりも良いときはあるが，その差は小さい．我々が実行したアプリケーションの数は 4 つで，この実験結果を持ってこれがつねに成り立つかどうかを結論することはできないが，これはある程度一般的な傾向であるということは以下のようにして分かる．もし各プロセッサで GC を必要とする頻度が大体同じ位であるならば，それらの GC を同期させることは，GC をする時点を少しずらすだけであり，GC 時間の増大にはつながらない．この場合，固定同期戦略がもたらすオーバヘッドは局所 GC 1 回につき 1 回のバリア同期だけである．一方，プロセッサごとのメモリ使用量やメモリ割当てのスピードがまちまちで GC を必要とする頻度がプロセッサごとに大きく違う場合を考える．この場合，同期 GC によって本来 GC を必要としないプロセッサまでが余分な GC を行うことになるため，それによる性能の低下の可能性がある．しかしながら，そのようなアプリケーションは多くの場合，GC を頻繁に必要とするプロセッサが実行時間を支配するであろう．それを認めるならば同期 GC による余分な GC は，他のプロセッサのアイドル時間を GC 時間に置き換えているにすぎず，全体の実行時間にはやはり影響しない．
- 片方の固定戦略がもう片方よりも明らかに優れている場合は，適応戦略は固定戦略のうちの良い方に近い性能を得ている．これは初期のスケジュールがどちらであるかによらない．我々は実際に，それらの場合について適応戦略が正しい方のスケジュール方式を用いていることを確認した．

5. 議論と制限

我々の実験から得られる基本的な，そして最も重要な結論は頻繁に同期通信を行うアプリケーションの性能は局所 GC の独立スケジュールがもたらすスケジュールのずれによって大きく性能が悪化するということである．我々はこれを局所 GC を同期させることで改善したわけだが，もちろん解決方法がこれ以外にないわけではない．以下にいくつかの選択肢と，それらの潜在的な問題点を述べる．

5.1 インクリメンタルまたは割込み可能な局所 GC

最も素直な方法は，各局所 GC を割込み可能にする，ということである．局所 GC は定期的にネットワークをポーリングし，GC 中も，やってくるメッセージを実行する．これは実際にはインクリメンタル GC[12] の応用にすぎない．これによって生ずるコストは，余分なメモリ，ポーリングのオーバヘッド，そして実装の複雑さである．この方式によって問題が解決されるかどうかはアプリケーションのメモリ使用量と利用可能なメモリ容量に依存する．一般にインクリメンタル GC はそうでない GC に比べて，GC が走っている間にアプリケーションが割当てを行うための，余分のメモリを必要とする（さもなければより頻繁に GC を行う必要があり，それは GC のオーバヘッドを増大させる）．メモリ容量が十分ある場合は問題は顕在化しないが，ひとたびメモリが不足し，あるプロセッサでアプリケーションの進行をしばらく抑制したい状態になったら，やはりそのプロセッサはメッセージに応答しなくなる．そのような状態ではやはり同期した局所 GC が必要である．まとめると，結局のところシステムは，メモリが十分にない場合の「最後の手段」として，同期した局所 GC を持たなくてはならなくなるだろう．

5.2 遅延に強いアルゴリズム

4 章で見たように通信が頻繁でないか，または遅延に強いアプリケーションはスケジュールのずれに影響を受けない．プログラミング言語によっては，コンパイラが通信遅延に強いコードを生成するかまたは，プログラマにそのようなコードを書くよう推奨できるかもしれない．しかし一般に通信遅延に対する耐性はメモリ使用量やスケジューリングオーバヘッドを増加させることによって得られるものである．したがって，通信相手のプロセッサが局所 GC を行っている場合のことだけを考えて，わざわざ他の目的には役立たない並列性をつねに作っておくことはあまり現実的な方法ではない．ここでの遅延遅延は局所 GC 1 回分に相当するもので，それは長いうえに予測が難しい．そのような状況で遅延が必ず隠れるようにするためには，非常に多数の，しかも通常は無駄な並列性を取り出さなくてはならない．

6. 関 連 研 究

並列計算機における GC の性能を，実験結果とともに示した研究は少ないが，文献 14)，29) などがあげられる．

本論文の第1の要点である，独立に局所GCをスケジュールすることによって並列アプリケーションの通信性能が低下し，それを防ぐには局所GCを同期的にスケジュールするのがよい，という知見は，これまでも異なった文脈では，いわゆるcoschedulingの利益として指摘されてきたものである[6),9)]．Coschedulingとは，複数のプロセスを同じタイミングでスケジュールする技法で，これによりそれらのプロセスの間の通信/同期遅延を短く保つことができる．本研究の指摘の1つは，局所GCの同期によって，非GC部分の通信遅延を短く保つことが重要である，ということである．

局所GCを同期させることの潜在的な利益は，これまでのシステムの実装においても認識，指摘されていた[25),26)]が，実験結果とともにこれを明確にした論文は筆者の知る限り存在しない．また，アプリケーションが陥ったアイドル期間の数を数えることで，アプリケーションの通信遅延に対する耐性を予測するという手法は，筆者の知る限りこれまで提案されていない．

7. まとめ

大規模な並列計算機における実際的なGC方式はあまりよく理解されていない．特に，大域同期のコストや，独立した局所GCの利益は誇張されてきた傾向がある．GCなどの複雑なシステムの性能は実験によって，アプリケーションとの相互作用を含めて，十分調査されるべきである．

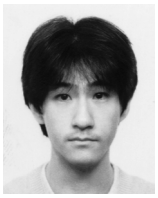
我々の実験は，局所GCの独立スケジュールは実際には非常に危険な戦略であることを示している（256プロセッサで，CKYアプリケーションが150%程度の性能低下）．同期した局所GCは，大域同期や，余分なGCを起動する可能性にもかかわらず，ずっと頑強な性能を示す．そして，簡単な通信挙動の監視によって，局所GCの正しいスケジュール方式を実行時に判断することもできる．

参考文献

1) Abdullahi, S.E., Miranda, E.E. and Ringwood, G.A.: Collection schemes for distributed garbage, *Proc. International Workshop on Memory Management*, Vol.637, Lecture Notes in Computer Science, pp.43–81, Springer-Verlag (1992).
2) Barnes, J. and Hut, P.: A hierarchical $O(N \log N)$ force-calculation algorithm, *Nature*, No.324, pp.446–449 (1986).
3) Birrel, A., Evers, D., Nelson, G., Owicki, S. and Wobber, E.: Distributed garbage collection for network objects, Technical Report 116, Digital Systems Research Center (1993).
4) Boehm, H.-J.: Space efficient conservative garbage collection, *Proc. ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp.197–206 (1993). `http://reality.sgi.com/boehm_mti/gc.html`.
5) Boehm, H.-J. and Weiser, M.: Garbage collection in an uncooperative environment, *Software Practice and Experience*, Vol.18, No.9, pp.807–820 (1988).
6) Feitelson, D.G. and Rudolph, L.: Gang scheduling performance benefits for fine-grain synchronization, *Journal of Parallel and Distributed Computing*, Vol.16, No.4, pp.306–318 (1992).
7) Goldberg, B.: Generational reference counting: A reduced-communication distributed strage reclamation scheme, *Proc. ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pp.313–321 (1989).
8) Grama, A.Y., Kumar, V. and Sameh, A.: Scalable parallel formulation of the Barnes-Hut method for n-body simulations, *Proc. Supercomputing '94*, pp.439–448 (1994).
9) Gupta, A., Tucker, A. and Urushibara, S.: The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications, *Proc. international conference on measurement and modeling of computer systems* (*SIGMETRICS*), pp.120–132 (1991).
10) Hayashi, K., Doi, T., Horie, T., Koyanagi, Y., Shiraki, O., Imamura, N., Shimizu, T., Ishihata, H. and Shindo, T.: AP1000+: Architectural support of PUT/GET interface for parallelizing compiler, *Proc. Architectural Support for Programming Languages and Operating Systems*, pp.196–207 (1994).
11) Hughes, J.: A distributed garbage collection algorithm, *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, Vol.201, Lecture Notes in Computer Science, pp.256–272, Springer-Verlag (1985).
12) Jones, R. and Lins, R.: *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons (1996).
13) Juul, N.C. and Jul, E.: Comprehensive and robust garbage collection in a distributed system, *Proc. International Workshop on Memory Management*, Vol.637, Lecture Notes in Com-

puter Science, pp.103–115, Springer-Verlag (1992).
14) Kamada, T., Matsuoka, S. and Yonezawa, A.: Efficient parallel global garbage collection on massively parallel computers, *Proc. Supercomputing '94*, pp.79–88 (1994).
15) Kasami, T.: An efficient recognition and syntax algorithm for context-free languages, Technical report, Air Force Cambridge Research Lab (1965).
16) Nakaya, A., Yamamoto, K., and Yonezawa, A.: RNA secondary structure prediction using highly parallel computers, *Comput. Applic. Biosci.* (*CABIOS*), Vol.11 (1995).
17) Nijholt, A.: Parallel approaches to context-free language parsing, *Parallel Natural Language Processing*, pp.135–167, Ablex Publishing Corporation (1994).
18) Piquer, J.M.: Indirect reference counting: A distributed garbage collection algorithm, *Proc. Parallel Architectures and Languages Europe*, Vol.505, 506, Lecture Notes in Computer Science, pp.150–165, Springer-Verlag (1991).
19) Plainfossé and Shapiro: A survey of distributed garbage collection techniques. *Proc. International Workshop on Memory Management*, Vol.986, Lecture Notes in Computer Science, Springer-Verlag (1995).
20) Taura, K., Matsuoka, S. and Yonezawa, A.: ABCL/f: A future-based polymorphic typed concurrent object-oriented language – Its design and implementation, *Proc. DIMACS workshop on Specification of Parallel Algorithms*, Vol.18, Dimacs Series in Discrete Mathematics and Theoretical Computer Science, pp.275–292 (1994).
21) Taura, K. and Yonezawa, A.: Fine-grain multithreading with minimal compiler support – A cost effective approach to implementing efficient multithreading languages, *Proc. 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '97*), pp.320–333 (1997).
22) Warren, M. and Salmon, J.: Astrophysical N-body simulations using hierarchical tree data structures, *Proc. Supercomputing '92*, pp.570–576 (1992).
23) Warren, M. and Salmon, J.: A parallel hashed oct-tree N-body algorithm, *Proc. Supercomputing '93*, pp.12–21 (1993).
24) Younger, D.H.: Recognition and parsing of context-free languages, *Information and Control*, Vol.2, No.10, pp.189–208 (1967).
25) 近山 隆：Private Communication (1997).
26) 小池汎平，田中英彦：分散メモリ並列計算機上でのジェネレーションスキャベンジング GC，並列処理シンポジウム予稿集，pp.273–280 (1990).
27) 二宮 崇，田浦健次朗，鳥澤健太郎，辻井潤一：並列オブジェクト指向言語 ABCL/f におけるスケーラブルな並列 CKY アルゴリズム，オブジェクト指向コンピューティングワークショップ（WOOC）予稿集 (1997).
28) 比屋根一雄：並列遺伝的アルゴリズムによる多目的最適化問題のパレート最適解集合の生成法と定量的評価法，第 9 回 自律分散システムシンポジウム，pp.295–300，計測自動制御学会 (1997).
29) 六沢一昭，市吉伸行：並列論理型言語 KL1 分散処理系の外部参照管理方式の評価，情報処理学会研究報告 96-PRO-8（SWoPP 予稿集），pp.13–18 (1996).

(平成 11 年 8 月 31 日受付)
(平成 12 年 2 月 4 日採録)



田浦健次朗（正会員）

1969 年生．1996 年より東京大学大学院理学系研究科助手．1997 年東京大学大学院より理学博士取得．並列プログラミング言語の設計，実装に関する研究に従事．

米澤 明憲（正会員）

1947 年生．1977 年 Ph.D. in Computer Science（MIT）．1989 年より東京大学理学部情報科学科教授．超並列・分散ソフトウエアアーキテクチャ等に興味を持つ．共著書「算法表現論」,「モデルと表現」（岩波書店），編著書「ABCL: An Object-Oriented Concurrent System」（MIT Press）等がある．1992〜1996 年ドイツ国立情報処理研究所（GMD）科学顧問，ACM Transaction on Programming Languages and Systems 副編集長，IEEE Parallel & Distributed Technology および Computer 編集委員等を歴任，元日本ソフトウェア科学会理事長．