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

A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines

遠藤 敏夫 Toshio ENDO　田浦 健次朗 Kenjiro TAURA　米澤 明憲 Akinori YONEZAWA

東京大学大学院 理学系研究科
Graduate School of Science, the University of Tokyo

概 要

共有メモリ並列計算機上でのマークスイープ法ガーベージコレクタ (GC) の実装と性能評価について報告する。実装した GC は並列 GC であり、共有ヒープ中のオブジェクトに対して全プロセッサが協調的に GC 処理を行なう。本 GC の性能評価を対称型共有メモリ計算機 Ultra Enterprise 10000 と分散共有メモリ計算機 Origin 2000 上で行った。本 GC はスケーラビリティを実現するために GC 処理の動的負荷分散を行う。また、処理のボトルネックを避ける工夫を提案し実装した。これにより GC の台数効果は、Enterprise 10000 上で 64 プロセッサを用いた時に 14 〜 30 倍となった。一方 Origin 2000 では、Enterprise 10000 上と全く同じ実装では十分な性能を得られないことが分かった。これは特定のメモリモジュールにメモリアクセスが集中するからと推測される。この推測に基づいてマークビットをメモリモジュール間で均等に配置した結果、性能の向上が見られた。

1 はじめに

汎用並列プログラミング言語の実装の研究において、ガーベージコレクタ (GC) の性能に関する研究はまだ少ない。特に、計算機が大規模になるにつれ重要になる GC のスケーラビリティに関してはまだ報告されていない。本稿では共有メモリ並列計算機のためのマークスイープ GC の実装と性能評価について報告する。共有メモリ計算機のための従来の GC 方式の多くは、特定の 1 プロセッサに GC の仕事をさせユーザプログラムと同時に動作させる。これに対し本稿で実装する方式は GC 処理を並列化してそのスケーラビリティに注目する。GC が要求されるとユーザプログラムを全て停止させ、全プロセッサが協調的にマーク処理、スイープ処理を行なう。また GC 時間を短縮するために動的負荷分散を行なう。本 GC の実装を既存の GC ライブラリである Boehm GC ライブラリ [1]のソースを基にして行なった。そして性能測定を対称型共有メモリ計算機 Ultra Enterprise 10000 と、分散共有メモリ計算機 Origin 2000 上で行なった。

第 2節で Boehm GC ライブラリについて述べ、実装した並列 GC 方式とスケーラビリティを得るための技法を第 3節で説明する。第 4節で実験環境について述べ、第 5節で性能評価を行なう。第 6節で関連研究を挙げ、第 7節でまとめる。

2 Boehm GC ライブラリ

Boehm GC ライブラリ [1]は C や C++ プログラムから利用できるマークスイープ GC である。プログラムはヒープからメモリオブジェクトを確保する際に GC ライブラリが提供する関数を用いる。そのオブジェクトは不要になると自動的に解放される。

ヒープ中の各オブジェクトにマークビットと呼ば

れるフラグが用意されている。Boehm GC ではマークビットはオブジェクトとは別の箇所に、ビットマップの形式で保持されている。GC 処理が要求されると、ルート (スタックやレジスタ) からポインタによって到達可能なオブジェクトを次々にマーク (マークビットが '0' なら '1' に) する。再帰的にポインタをたどるためにマークスタックというデータ構造を用いる。マークスタックには常に、「それ自身はマークされたが、未マークなオブジェクトを指すかもしれないオブジェクト」のポインタを含む。

マークスタックが空になりマークフェイズが終了するとスイープフェイズを行なう。本来のスイープ処理は未マークのオブジェクトを解放することだが、Boehm GC ではこの段階ではページ単位でマークビットを検査し、完全に空なページを検出してページごとのフリーリストを作成することのみを行なう。オブジェクト単位の解放は、後のメモリ要求の際に必要に応じて行なわれる。

3 並列 GC アルゴリズム

GC 処理が要求されると、まず全てのユーザプログラムをシグナルを送ることによって停止する。各プロセッサはそれぞれマークスタックを持ち、それを用いて各自のルートを起点にマーク処理を行なう。マークビットが '0' か検査し '1' にする処理は不可分に行なう。複数プロセッサから到達可能なオブジェクトは、初めにそれを見つけたプロセッサによってマークされる。GC 時間を短縮するために本 GC では、マークスタックがタスクプールの役割を果たしていることに注目して動的負荷分散を導入する。各プロセッサはマーク処理中に定期的にマークスタックに含まれる仕事の一部をバッファに移動する。自分のマークスタックが空になったらバッファから仕事を取得し、マーク処理を続ける。全プロセッサのマークスタックと仕事をやりとりするためのバッファの全てが空になったらマークフェイズを終了する。

以上に述べた並列マーク処理により、プロセッサ台数が少ないうちは良好な台数効果が得られるが、安直な実装では処理のボトルネックのためプロセッサが多いときに性能が頭打ちになってしまう。これに対し以下の改良を行なった。

- 負荷の分散単位が 1 つのオブジェクトであるため、大きいオブジェクトがあると負荷の均衡が崩れる。大きいオブジェクトを小さく分割して複数のポインタとしてマークスタックに格納することにした。これによって細粒度な負荷均衡が可能になる。
- マーク処理の終了判定を 1 つの共有カウンタによって実装すると、プロセッサが多くなるとボトルネックになってしまう。局所的なフラグを用いた実装を行なった。
- マークビットの更新の際のロック待ち時間が非常に長い場合が見られた。ロックをかける代わりに、プロセッサの不可分更新命令 (Ultra SPARC の compare & swap、R10000 の load-linked, store-conditional) で直接マークビットを更新するようにした。

スイープフェイズについては、ヒープを細かく分割しプロセッサが部分ヒープを取り合って処理することによって並列スイープを行なう。各プロセッサは部分ヒープに対してマークビットを検査し、ページのフリーリストを作成する。この処理は局所的に行なうことができる。最後に各部分ヒープのフリーリストを連結する。スイープフェイズが終ったらユーザプログラムを再開する。

4 実験環境

Sun Ultra Enterprise 10000 は 64 台の Ultra SPARC プロセッサ (250MHz) を持つ対称型共有メモリ計算機である。プロセッサとメモリモジュールは 10.7GB/s のバンド幅のクロスバーで接続され、プロセッサは任意のアドレスに対して均等な時間でアクセスできる。

SGI Origin 2000 は 195MHz の R10000 プロセッサを持つ分散共有メモリ計算機である。実験では 64 プロセッサ構成の計算機を使用した。2 つのプロセッサが 1 ノードを構成し、ノードごとにメモリモジュールがある。ノードはハイパーキューブ状に接続され、近いノードのメモリほどレイテンシが小さい。各メモリモジュールのバンド幅は 0.78GB/s である。各メモリページは、そのページを初めてアクセスするプロセッサのノードのメモリモジュールに配置される。

5 性能評価

実験では以下の並列アプリケーションを C++ で記述して用いた。詳細については [3]を参照のこと。Enterprise 10000 では Solaris thread を、Origin 2000 では pthread を用いて並列化を行なった。

BH Barnes-Hut アルゴリズムを用いて N 体問題を解く。ステップ毎に粒子の位置を基に 8 分木を作り、その木を用いて各粒子にかかる力を計算する。実験では粒子数 20000、ステップ数 50 とした。

CKY 自然言語の文章とその言語の文法を入力としてうけとり、各文について可能な全構文木を出力する。実験では 10 〜 40 語からなる

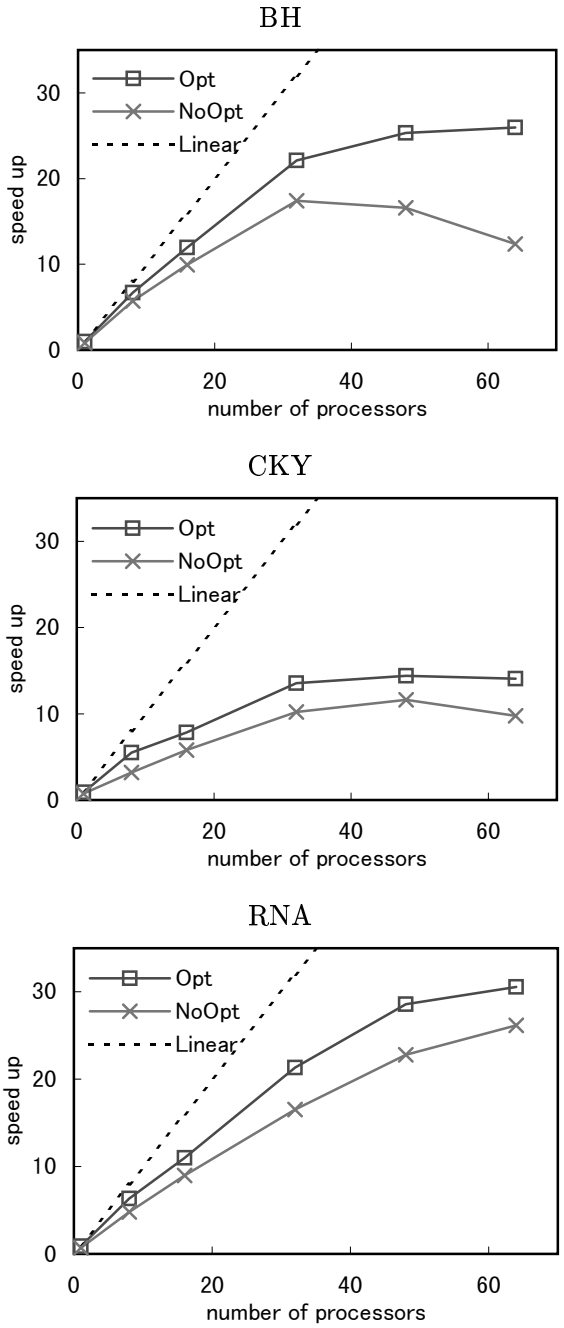

図 1　Enterprise 10000 上の GC の台数効果

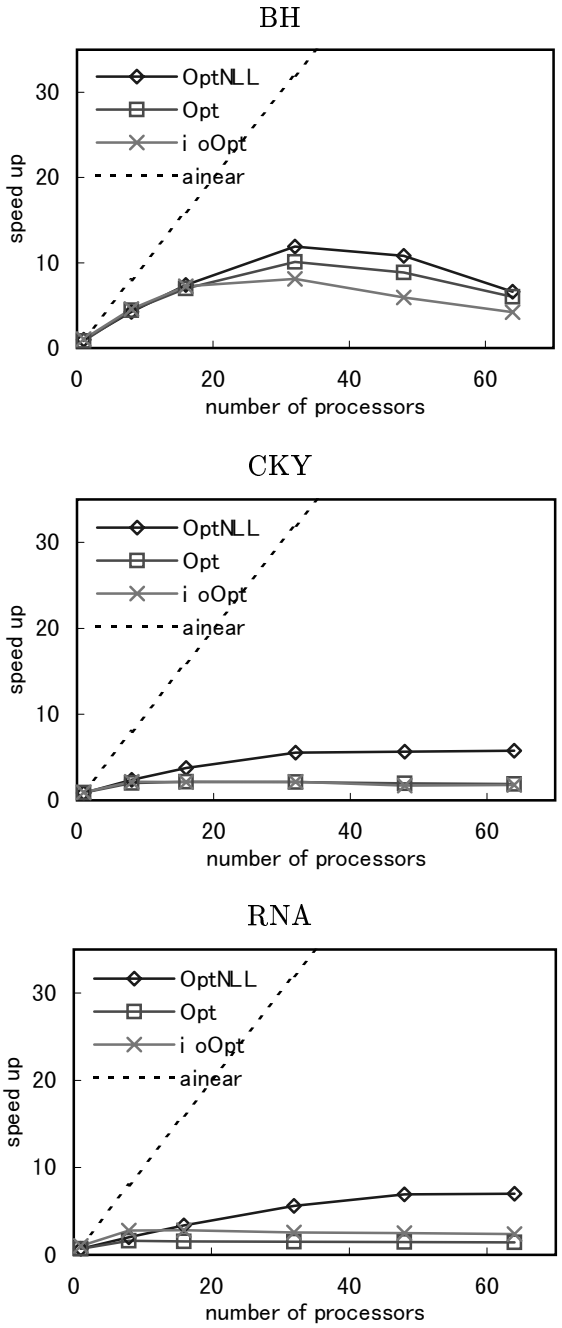

図 2　Origin 2000 上の GC の台数効果

文を 256 文用いた。

RNA　　RNA の二次構造を予測する。スタック領域の位置とエネルギーを入力として受けとり安定解を求める。実験ではスタック領域数 119 とした。

本稿では GC 単体の台数効果について議論する。マーク処理の時間は一般にはマークされたオブジェクトの総数と合計サイズ、スイープ処理の時間はヒープサイズに比例する。よってそれらの仕事量と並列 GC にかかった時間から GC 処理の速度を求めることができる。詳細は [3] [4]を参照のこと。

図 1～2は 2 種類の計算機でそれぞれ 3 つのアプリケーションを動作させた時の性能を示す。アプリケーション実行中に GC は複数回おこるので、全 GC 処理の仕事量の合計と GC 時間の合計から求めた速度をグラフに示す。台数効果の基準は並列化のオーバヘッドを含まない逐次 GC である。

グラフの「Opt」は負荷分散と第 3節で示したボトルネックを解消する改良を行なった場合の性能である。「NoOPT」は負荷分散は行なうが改良

を行なわない場合の性能である。図 1は Enterprise 10000 上での性能を示す。改良前の「NoOpt」の場合は BH と CKY で 32 〜 48 プロセッサを超えると GC 速度が落ちてしまう。「Opt」は特に BH で効果が大きく、 64 プロセッサで 26 倍程度の台数効果を得られた。

図 2は Origin 2000 上での性能を示す。「Opt」を見るとどのアプリケーションも良い性能を得られず、特に CKY と RNA では最大 2 〜 3 倍程度の台数効果しか得られていない。この一因は、 Origin が分散共有メモリ計算機であるにもかかわらず本 GC がオブジェクトのローカル / リモートを考慮せずにマーク処理を行なっていることと考えられる。しかしプロセッサが増えると GC 速度が下がるのはレイテンシの違いだけでは説明できない。 Origin が Enterprise と違う点として、プログラムのメモリ確保の仕方によってはメモリページの配置がモジュール間でばらつきがでることが挙げられる。この場合、 GC によるメモリアクセスが特定のモジュールに集中しボトルネックになるのではないかと推測される。そうであれば、メモリページの配置をモジュール間で均等に近付ければ性能が上がる可能性がある。 GC が頻繁にアクセスする主なメモリ領域は、ヒープ中のメモリオブジェクトとマークビットマップである。このうちオブジェクトの配置は GC が勝手に決めるべきではない。もう一方のマークビットマップを、対応するヒープアドレスによってラウンドロビン方式でモジュールを決め均等に配置した。その場合の結果をグラフの「Opt+RR」に示す。「Opt」に比べて特に CKY と RNA で性能が向上しており、推測したメモリページのばらつきが速度の低い一因であることが分かった。しかしこれだけでは BH での性能の頭打ちは解決できていないし、 CKY や RNA での性能も十分とはいえないため、今後の改善が必要である。

6 関連研究

提案されてきた共有メモリマシンのための GC の多く ([2] [6]など) は、特定の 1 プロセッサに GC の仕事をさせ、ユーザプログラムと同時に動作させる。これによりユーザプログラムの停止期間を短くすることができる。しかしこの方式ではマシンが大規模になるにつれ GC 時間が長くなり、 GC によるメモリ解放がメモリ確保に間に合わなくなる可能性がある。

これに対し、 GC そのものを並列化する方式としては [7] [5]などが提案されている。 [7] [5]は両者とも動的負荷分散を行なう。しかし性能評価を小規模なマシンやシミュレータ上でのみ行なっており、大規模なマシンでおこる性能上の問題については言及していない。また分散共有メモリ上での性能についても言及していない。

7 おわりに

共有メモリ計算機上で並列マークスイープ GC を実装した。 GC のスケーラビリティを得るために、動的負荷分散を行ないながらマーク処理とスイープ処理を並列に行なう。処理のボトルネックを避ける改良によってスケーラブルな GC を構築できた。 Enterprise 10000 上で 64 プロセッサで 14 〜 30 倍の台数効果を得た。しかし Origin 2000 ではそのままでは良い性能を得られない。その一因はメモリ配置のばらつきであり、マークビットの配置を均等にすることによって性能を改善できる。しかしそれでも十分とはいえないため、メモリアクセスの挙動をより深く調査し性能の改善を図るのが今後の課題である。

参考文献

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

[2] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multithreaded implementation of ML. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–123, January 1993.

[3] Toshio Endo. A scalable mark-sweep garbage collector on large-scale shared-memory machines. master thesis, Department of Information Science, The University of Tokyo, February 1998.

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

[5] Akira Imai and Evan Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1030–1040, September 1993.

[6] James O'Toole and Scott Nettles. Concurrent replicating garbage collection. In *Proceedings of 1994 ACM Conference on LISP and Functional Programming*, pages 34–42, 1994.

[7] 渦原 茂. 共有メモリ型マルチプロセッサにおける並列ガーベージコレクション. In 情報処理学会研究報告 *90-ARC-83*, pages 205–210. 並列／分散／協調処理に関するサマー・ワークショップ (SWoPP '90), July 1990. (in Japanese).