アーキテクチャパラメータを利用した並列 GC の性能予測

遠藤 敏夫

1999 年 10 月 6 日

共有メモリモデルをプログラマに提供するモデルは様々な種類があり、対象型共有メモリマシン (SMP), 分散共有メモリマシン (DSM), 分散メモリマシン + ソフトウェア分散共有メモリなどの上で、同一のプログラムを動かすことができる。しかし「性能に関する移植性」は保証されない。同じプログラムを同数プロセッサで動かしても、マシンによって得られる台数効果が全く異なるのはざらである。それは一般のユーザプログラムであっても、言語ランタイムシステムでも起こる問題であり、筆者は並列 GC のマークフェーズを対象に研究を行っている。どんなアーキテクチャでも最適な性能を示す方式を提案するのが理想だが、その前段階として、それぞれのアーキテクチャでの性能の理由づけ (SMP では速度向上するのに DSM では頭打ちになるのはなぜか etc.) を定量的に行う。アーキテクチャパラメータから性能の予測ができるようになれば、未知のアーキテクチャ上での性能を知ることも可能になる。

性能に最も影響を及ぼす違いは、メモリアーキテクチャの違いである (図 1)。 SMP である Enterprise 10000 と、 DSM である Origin 2000 のアーキテクチャは以下の通りである (数値の出所は、メーカーの資料、本 [3]、自分の実験などごちゃまぜで、あくまで目安)。

Enterprise 10000:

- メモリアクセスのレイテンシ: 500–800 ns (どのアドレスも均等だがばらつきが見られた)
- うち、メモリノード占有時間: 48 ns
- メモリ配置: アドレスによってインタリーブされる
- 二次キャッシュラインサイズ: 64 バイト

Origin 2000:

- メモリアクセスのレイテンシ: local 400 ns, remote 600 ns 以上
- うち、メモリノード占有時間: 200 ns
- メモリ配置: page 単位で、最初に触ったプロセッサのローカルメモリとなる
- 二次キャッシュラインサイズ: 128 バイト

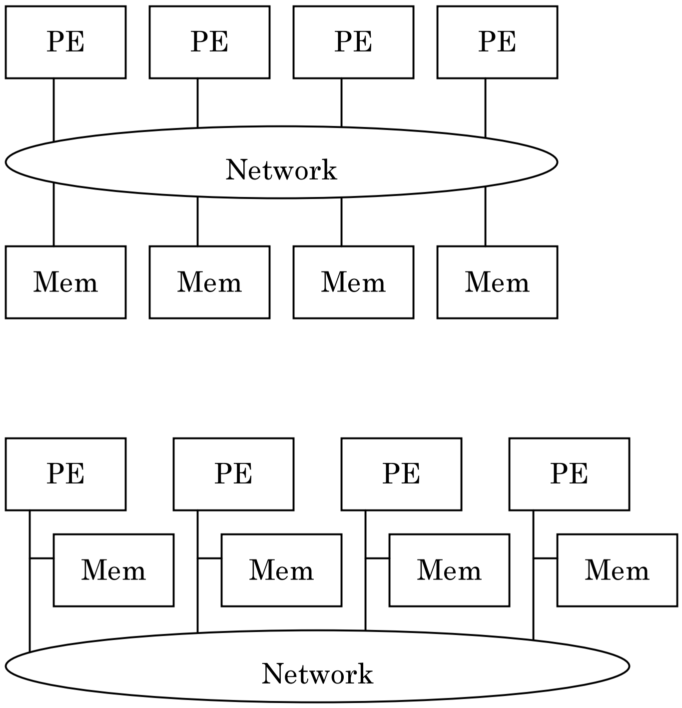

図 1: SMP のメモリアーキテクチャ (上) と DSM のメモリアーキテクチャ (下)

現在の並列 GC の実装では、 Enterprise で良い台数効果を得られる場合でも、 Origin で頭打ちになってしまう場合がある。この原因は、 Origin の local/remote のレイテンシの差よりも、使用中メモリオブジェクトをたくさん持っているメモリノードへのアクセス集中の影響が大きい。一方 Enterprise では、自動的にメモリノードへのアクセスは平均化され、アクセス集中は起りづらいようだ... ということまでは以前に分かっていた。

現在、以下のようにマークフェーズ時間予測ができている。これらは毎回の GC 直前にヒープを検査するようなコードを実装することにより、行われている。

- GC 中に起こるであろうキャッシュミス数を見積もる。
- DSM の場合、 GC 中に起こるであろうキャッシュミス数の、対象メモリノードごとの統計を取る。
- GC 中の総仕事量 (ほぼ live object size) とクリティカルパス (ほぼオブジェクトグラフの深さ) を見積もる。
- 以上の情報とアーキテクチャパラメータから、マーク処理時間を予測する。
- 実測値との違いに愕然とする (笑)。

今回は、 Origin 上での性能予測について話をする。

1　並列マークアルゴリズムの説明

共有ヒープとその中のオブジェクト、それぞれのオブジェクトに対するマークビットは、どのプロセッサからもアクセスすることができる。各プロセッサは 1 つづつマークスタックを持ち、全プロセッサが協調的に負荷分散をしながらマーク処理を行う。

各プロセッサは自分のルートからたどれるオブジェクトを、自分のマークスタックを利用してたどる。各マークビットの検査 / 更新を atomic に行う必要がある。マークスタックが空になってしまったら、他のプロセッサのマークスタックの底から一つオブジェクトを盗んで仕事を続ける。全プロセッサのマークスタックが空になったら、マークフェーズは終了である。

Cilk を知っている人は、「あぁ同じような実行方法だな」と思ってもらえれば充分。

2　性能予測の概要

分散メモリ計算機において性能に大きく影響を及ぼすのが send/receive であるように、共有メモリ計算機ではメモリアクセスを考慮しなければまっとうな性能予測はできない。

以下の章では、 GC 直前のヒープ状況から、そのヒープについて並列マーク処理を行う場合の実行時間を予測するモデルについて述べる。

具体的な方法は、

- ヒープ状況からキャッシュミス数を予測する。
- DSM では、それと同時にキャッシュミス数の対象ノードごとの統計をとる。
- ヒープ状況から総仕事量と、クリティカルパス長を得る。
- キャッシュミスレイテンシやキャッシュライン長などのアーキテクチャパラメータ定数を参考文献から得る。
- これらの情報を、簡易 LoPC モデルに代入して実行時間を予測する。

前もって予測が困難であるとか、影響が少なさそうという理由で考えていない要素は

- DSM での、 local/remote のレイテンシの差
- invalidation によるキャッシュミス数の増加
- 仕事探しの時間
- 終了判定の時間
- SMP でのメモリコンテンション
- プロセッサ間仕事移動によるキャッシュミス数の増加 (下の章で説明)

などである。

3 性能予測方法 (1): キャッシュミス数の予測

GC が行う全てのメモリアクセスのうち、キャッシュヒットした場合は他プロセッサへの影響はないし、アクセス時間も一定で短い。これ以降問題にする「キャッシュミス」とは 2 次キャッシュミスを指し、「キャッシュヒット」は 1 次キャッシュヒット / 2 次キャッシュヒットを指す。

GC が行うメモリアクセスの主な対象は、

- ヒープ中のオブジェクト (read only)
- マークビット (read/write)
- マークスタック (read/write)

である。このうちマークスタックはほとんどの場合、持ち主プロセッサのみにアクセスされ、領域もオブジェクトグラフの最大深さのオーダーと小さいため、ほぼキャッシュヒットすると考えて以降無視する。

残りのオブジェクト、マークビットに対するのキャッシュミス数を知りたい。これが、規則的なアクセスをするプログラムであったら話は楽で、 (アクセスしたサイズ / キャッシュラインサイズ) がキャッシュミス数になる。しかし GC では仮に live object 合計サイズが分かったとしても、キャッシュミス数を知ることはできない。オブジェクトを depth first にたどる順番が、キャッシュミス数に効いてくるからである。

「オブジェクトをたどる順番」は、マークアルゴリズムが固定されていて、かつ 1 プロセッサであれば一意に決まる。この場合のキャッシュミス数は、実際にマークアルゴリズムを 1 プロセッサで動かして見れば当然分かる。これを求めるため、キャッシュラインが 1 つしかないようなキャッシュシミュレータを実装して、 GC 直前にヒープをなめる (というか、実際にマークしてみる) ことにした[1]。

ちなみに、このキャッシュシミュレータが外れる原因はいくつかあるが、特に大きなものは、 line A, line B, line A の順でアクセスしたときに 2 回目の A もミスだと思ってしまう点である。ここではこのような「外れ」は保留にしておく。

さて、ここで「1 プロセッサの場合が分かっても、並列に GC を行った場合のミス数が分からなければしかたない」という問題が出てくる。ある一つのヒープ状態があったとして、それに対し 1 プロセッサでマークシミュレートを行った時のミス数予測値を M_1 とする。 P プロセッサでマークを行い、それぞれのプロセッサに先ほどと同じキャッシュシミュレータが実装されていると仮定するときの、ミス数予測値を M_P とする (全プロセッサでの合計)。この並列マークシミュレータは、本物のマークアルゴリズムと同じ順番でオブジェクトをアクセスし、負荷分散も行うとする。非決定性のせいで、 M_P は一定とは限らない。

ここで、 M_P と M_1 の関係について、以下の未証明の定理 (のようなもの) を述べる。

- 並列マーク中に Cache invalidation が一切起こらず、
- 全てのオブジェクトの reference count が 1 以下である

[1] ここまでしておいて性能「予測」と名乗るのは心が痛むが

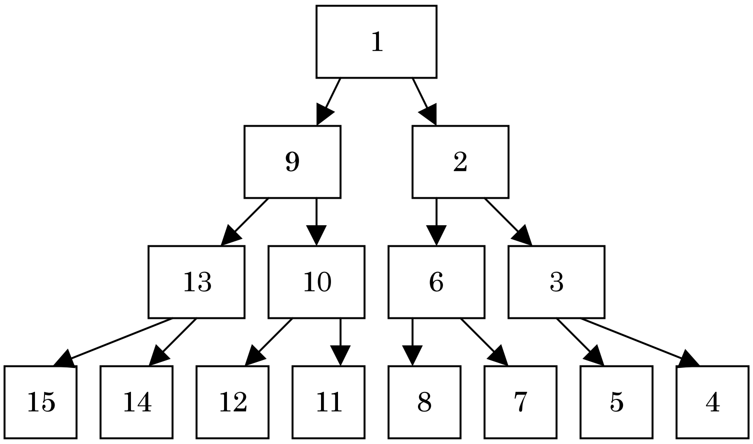

図 2: ヒープ状況の例。オブジェクトについている数字はスキャンされる順番を示す

と仮定する。マークアルゴリズム、負荷分散方式は前章で述べたものとする。この時、並列マーク中に全プロセッサで起こった仕事移動回数を S とすると、 $M_P \leq M_1 + S$ である。

この定理 (のようなもの) がなぜ成り立つのかについて直観的な説明をする。

図 2のようなヒープに対してマーク処理を行うことを考える。一番上のオブジェクトがルートとする。オブジェクトについている数字は、 1 プロセッサで GC した場合にそのオブジェクトがスキャンされる順番を示す[2]。この順番から先ほどの M_1 が一意に求められる。オブジェクト k とオブジェクト $k+1$ が同じキャッシュラインにのっていれば $k+1$ アクセス時はキャッシュヒット、のっていなければミス、という具合である。

さて、並列の場合を考える。仕事は全部プロセッサ A が持っていて、プロセッサ B はプロセッサ A から仕事を一回盗むものとする。仕事を盗むときの規則では、相手のスタックの底からオブジェクトを一つ盗むことになっているので、 B が盗みうるのはオブジェクト 9, 13, 15 のいずれかである。(B からさらに盗まれない限り)B がオブジェクト 9 を盗んだ場合は、 B はオブジェクト 9 からオブジェクト 15 までを番号の通りにスキャンすることになる。他を盗んだ場合も同様。つまり、アクセスの連続という点に注目すると、 1 プロセッサの場合に比べてオブジェクト 8 とオブジェクト 9 の一カ所のみの連続がとぎれることになる。このとき、 M_1 と M_P の関係は、

- $M_P = M_1$。 1 プロセッサの場合でも 8 と 9 は別キャッシュラインだった。
- $M_P = M_1 + 1$。 1 プロセッサの場合 8 と 9 が同一キャッシュラインだったのに、並列化のせいでミスするようになってしまった。

のいずれかとなる[3]。

つまり、 1 回の仕事移動のせいで、 M_P と M_1 の間に差が 1 つできうる。さらにこの後仕事移動がおこった時でも、 1 回の移動で起こる「連続アクセスの途切れ」は 1 個所以下であるため、上の式が成り立つ。

実装レベルの話としては、今使っているアプリケーションでは (仕事移動回数) $\ll$ (live object 数)

[2]マークされる順番は別で、違う番号がふられるのだが、同様な議論が成り立つので省略する

[3]並列化のせいでミス数が減ることも、可能性としてはありうると思う

であることを利用して、 $M_P = M_1$ として予測に用いている。また事実上、仕事移動回数を本番マーク前に知ることはできない。

このキャッシュミス数予測の精度は、以下の通り。

- nbody ではほぼ一定して、実測値よりも 2 倍ほど多く予測されてしまう。これは、 line A, line B, line A... というアクセスパターンが予想外に多いためらしい。
- cky ではプロセッサが増えるほど離れていってしまう。実測値の方がどんどん多くなっていく。invalidation が原因か？

なお、オブジェクト確保時、マークビットマップ確保時に確保者プロセッサを記録しておき、キャッシュシミュレータはそれを見てミス対象ノードごとの統計をとる。

4 性能予測方法 (2): 実行時間の予測

LoPC モデル [4] の用語を使って説明する。 LoPC モデルは分散メモリマシンのためのモデルなのだが、ここでは DSM に対応させたものを説明する[4]。

このモデルでは、各プロセッサ i は以下のような動作を繰り返すとする (図 3)。

- 平均時間 R_{wi} の間、通信のない局所的な仕事をする。
- その後、 V_{ij} の確率でプロセッサ j(この場合 DSM なので今後メモリノード j という) にメッセージを送信する。片道の latency は S_l である。
- Ack が返ってくるまでプロセッサ i は何もしない。
- メモリノード j は受け取ったメッセージを順番に処理する。一つのメッセージ処理に平均 S_o 時間かかる。
- メッセージ処理が終わったらメモリノード j は Ack を送り、プロセッサ i が受け取る。

この一セットを round trip と呼ぶ。

一回の round trip にかかる平均時間は $R_i = R_{wi} + S_l + \sum_j V_{ij} R_{qj} + S_l$ となる。 R_{qj} は j にメッセージが届いてから処理が終わるまでの平均時間で、 $R_{qj} \geq S_o$ である。

さて、今回の GC 性能予測の話をこのモデルに持ち込む。上で述べたメッセージ送信が、キャッシュミスに相当する。ただし、 DSM でのキャッシュミスは本来はメッセージの一往復では済まないのだが (home と owner の両方を経由したり、 invalidate メッセージのため)、ここでは一往復と計算する。さらに、 S_l はどのメモリノードに対しても一定とするので、 local/remote のレイテンシの差は残念ながらモデルに入っていない[5]。また、 V_{ij} は i 方向には一定であると仮定する。 V_j と書ける。 j 方向には一定ではない。これは、メモリアクセスの集中を表現するためである。

[4]CPU が受信処理をしなくて良い分、オリジナルより簡単になる

[5]分散メモリマシンでは、そもそも local に対するメッセージ送信という概念がなかったので、 local/remote の差は歴然とあった。 DSM では、ローカルメモリアクセスであろうとも contention が起りうるので、メッセージ送信として扱わなければならない

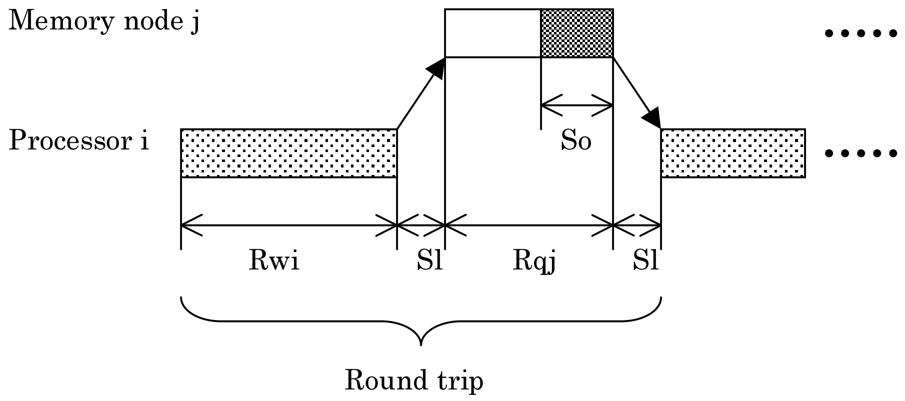

図 3: 簡略化 LoPC でのプロセッサの挙動。 Round trip は局所的仕事とメッセージの往復から成り立つ

S_o, S_l はアーキテクチャパラメータである。 V_j は、キャッシュミス対象ノードごとの統計から得られる。

R_{wi} は以下のように考える。仮にキャッシュミスが起らないで 1 プロセッサでマークしたときの時間を T_1 とする。クリティカルパス長を T_∞ とする。これらは、 live object サイズなどを数えることにより求めることができる。 Cilk の研究 [1] により、キャッシュミスを考えない P プロセッサでの実行時間は $T = T_1/P + T_\infty$ で押さえられることが分かっている。この時間中にキャッシュミスが均等なタイミングで起ると仮定すれば、 $R_{wi} = T/(M_P/P)$ となる (この値は i に対し一定。また、以前に述べた通り、 $M_P = M_1$ としてしまう)。

ここまで分かれば、本来の LoPC モデルであれば、待ち行列理論を使って平均待ち時間 R_{qj} を求める方程式をたてて[6] ... とすすむのだが、以下のような単純な式でも、あまりひどい誤差はでないであろうことを予備実験で確認した。

コンテンションがないと仮定した時の予測実行時間は、 $T_m = T + (M_P/P) \times (S_l + S_o + S_l)$ である。コンテンションがあればもっと時間は長くなる。一方、最も込み合うメモリノードに注目すると、このノードが常に何かしらのメッセージを処理しているとしても、最低 $T_n = max(V_j)M_P S_o$ の時間はかかる。コンテンションを考慮にいれた予測実行時間を、 $T_c = max(T_m, T_n)$ とする。以上。

さて、 T_c と実測値時間を基に GC 速度を測った結果が図 4である。特に cky でひどいことになっている。前半 (キャッシュミス数予測) と後半 (ミス数を基に時間予測) の両方にまだ問題がある... という程度まで分かっている状況である。

5 関連研究

不規則的にタスクを生成するプログラムの性能予測の研究では、 Cilk の話を外せない。 [1] では総仕事量とクリティカルパスのみから実行時間の upper bound を求めていたが、メモリの影響を考慮に入れていなかった。最近、 [5] でメモリ階層を考慮に入れた性能予測を行っている。流し読みした限りでは、 memory contention は考慮に入れていないように見える。

[6] どんな方程式かは [4] 参照

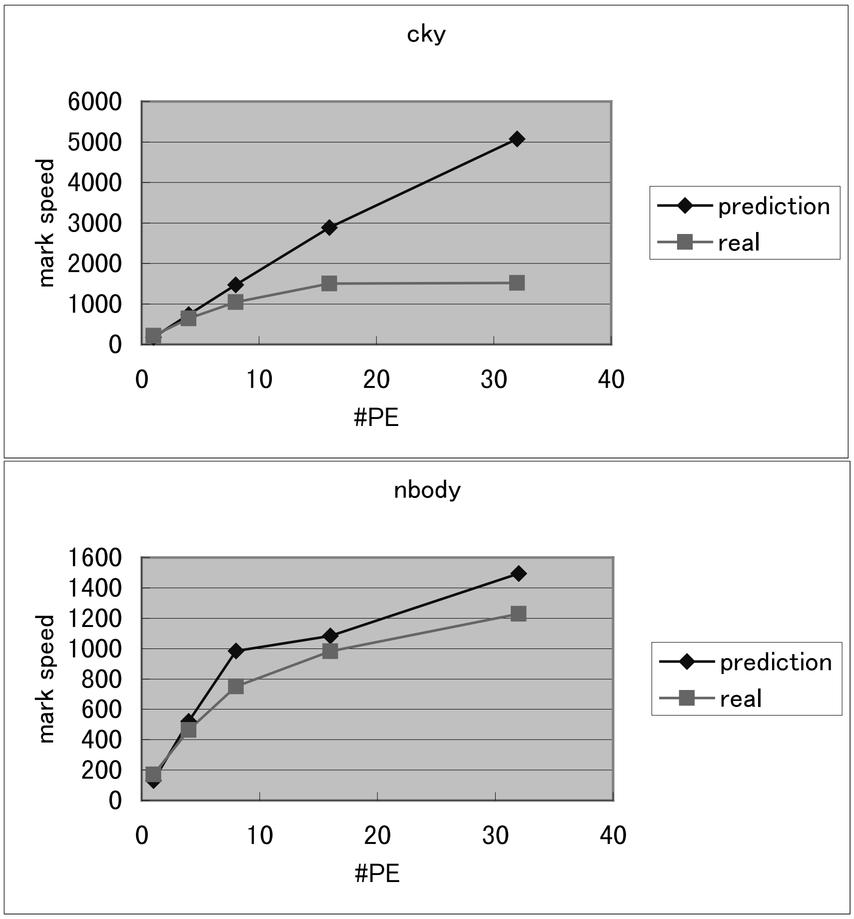

図 4: GC 速度の予測値と実測値の比較。 cky では速度の頭打ちをとらえられておらず、プロセッサが増えるほど予測が外れてしまっている

[2, 4] は分散メモリ並列計算機のアーキテクチャを、プログラムの実行時間に大きく影響を与える要素に限定して、モデル化している。 LogP モデル [2] ではレイテンシ (l), 通信時のオーバヘッドの時間 (o), バンド幅 (g = gap), プロセッサ数 (P) の 4 つのパラメータで計算機の特徴を表す。 LoPC モデル [4] は LogP モデルからバンド幅を省き, 逆にメッセージ受信時の contention による待ち時間を考慮する。 C というパラメータがあるわけではなく、待ち行列理論を使って実行時間を見積もるための方程式を導く手法を提案している。

今回の話は LoPC モデルを分散共有メモリ用に変更 (単純化) したものを参考にしている。本来は方程式を解いて時間を求めるのだが、代わりに簡単な近似式を用いている。

6 おわりに

今の性能予測の精度は全く満足のいくものではない。改良するには、 invalidate を計算に入れる、キャッシュシミュレータの精度を上げるなど、キャッシュミス数の見積もりを変更する必要がある。ただし、キャッシュミス数が正確になったとしても、それだけでは今の予測精度のずれ方を説明できないため、まだ他に考えるべき要素があるはずである。

GC 性能予測以後の話としては、以下のようなふくらませ方が考えられる。

- 多様なアーキテクチャ上に適応して、それぞれ最適に近い性能で動作するような並列 GC 方式の提案を目指す。 GC 性能予測はその前段階である。

- 並列 GC 以外の、一般の並列プログラムの性能予測を行う。 GC は一般の並列プログラムと比べると以下の特徴を持つ。 (1)task graph = object graph であり、 (2) 主にアクセスするメモリ領域が mark bit と object の 2 種類であり、 (3) 同期パターンが単純である。

参考文献

[1] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–358, November 1994.

[2] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[3] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture, A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1999. ISBN 1-55860-343-3.

[4] Matthew I. Frank, Anant Agarwal, and Mary K. Vernon. LoPC: Modeling contention in parallel algorithms. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 276–287, June 1997.

[5] Matteo Frigo. *Portable High-Performance Programs.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.