汎用超並列オペレーティングシステム SSS–CORE 上の非対称分散共有メモリにおけるコンパイル技法

丹羽 純平　稲垣 達氏　松本 尚　平木 敬

本稿では、 ADSM 用の最適化コンパイラにおいて書き込みのオーバーヘッドを削減するための手法を提案する。 AP1000+ 上でコンパイラとランタイムのプロトタイプの実装を行ない、実験により本手法の有効性を評価する。

1 はじめに

従来のソフトウェア分散共有メモリ (DSM)[6]では、共有領域への書き込みに対してコンパイラが単なるストア命令を用意して、ページの書き込みトラップルーチンでコンシステンシ維持コードを実行していた。共有領域への書き込みの都度トラップを起こすオーバーヘッドを削減するために、同期区間内のページの差分を計算する diff方式が提案されてきた[4]。ページの書き込みトラップは同期区間内で一回に押えられる。しかし、 diff を作成するためには、ページのコピーを生成したり、ページの修正された部分とは無関係にページ全体を調べなければならず、オーバーヘッドが大きい。データ転送の軽いハードウェアを使用して、共有領域への書き込みの結果を全て home に転送し、 home を常に最新の状態に保つ (Automatic update release consistency(AURC)[3]) ことで、 diff 方式が抱える問題は解決できる。

我々は、特殊な通信同期ハードウェアを仮定しない環境で、メモリベース通信機能を使った新しいソフトウェア DSM である非対称分散共有メモリ (**Asymmetric Distributed Shared Memory: ADSM**) を提案してきた[12]。 ADSM では、共有領域からの読みだしは従来のソフトウェア DSM と同様、仮想記憶機構を利用してプロセッサのロード命令によって実現されるが、共有領域への書き込みはストア命令のトラップではなく、コンパイラが生成する明示的なコンシステンシ管理コードで実現される。従って、コンパイラが共有領域への書き込みを一連のコード列に変換する際、以下のような種々の最適化が可能になる。

- ページ単位のプロトコル切替
  コンパイラがストアの後に挿入するコンシステンシ維持コードを切替えることによって、様々なプロトコルを実現できる。従って ADSM ではページ単位のプロトコル切替[11]をサポートする。
- コンシステンシ維持コード列の coalescing
  ある同期区間において連続した共有アドレスへの書き込み列がある場合、コンシステンシ管理コードの列を、連続領域に対する一つの管理コードに変換することによって、実行時のオーバーヘッドが軽減される。
- 通信パケットのコンバイング
  送り先が同じパケットをコンバイニングしてメッセージ数を減らし、更新型メモリアクセスのオーバーヘッドを削減する。

Compiling Techniques for ADSM on General-Purpose Massively-Parallel Operating System: SSS–CORE
Junpei NIWA, Tatsushi INAGAKI, Takashi MATSUMOTO, Kei HIRAKI, 東京大学大学院理学系研究科情報科学専攻, Department of Information Science, Faculty of Science, University of Tokyo.
コンピュータソフトウェア, Vol., No.(), pp.–.
1997 年月日受付.

2 プロトコルの実現

実行時には、書き込みの検知と書き込みの結果の回収を効率良く行なうために、書き込みの履歴 (*write history*) を管理する。 *write history* は書き込みのアドレスと書き込みのサイズの組で表現される。我々はADSM 上で 3 種類のプロトコルを実装した。

(1) LRC プロトコル

TreadMarks[5]の Lazy Release Consistency (LRC)[4] プロトコルと基本的に同じである。コピーページへの書き込みの反映を実際にデータが必要になる (acquire) 時点まで遅延して、ページフォールト時には必要な差分のみを取り寄せる。TreadMarks とは以下に示すように実現方法が異なる。

・ 書き込みの検知

TreadMarks では実行時に OS が検知する。最初の書き込みをトラップして、トラップルーチンでページ全体のコピー (twin) を作成する。

ADSM ではコンパイラが共有書き込みをストア命令と *write history* を登録するコード列に変換する。

・ 書き込みの結果の回収

TreadMarks では差分が必要となった時点で現在のページと twin を比較して diff に差分の情報を格納する。通信量を最小にするために diff は明示的に同期をとって GC しない限り除去しない。

我々は登録された *write history* の情報を元にプロセッサが必要とする差分を計算する。この計算は作成された *write history* の量に比例する。作成されたログは明示的に同期を取らないと除去できない。従って大量のメモリが消費されるので GC が必要になる。

(2) SAURC プロトコル

ソフトウェアで Automatic update release consistency(AURC)[3]をエミュレートする。

・書き込みの検知

HLRC では実行時に OS が検知する。最初の書き込みをトラップして、トラップルーチンでページ全体のコピー (twin) を作成する。

ADSM ではコンパイラが共有書き込みをストア命令と *write history* を登録するコード列に変換する。

・書き込みの結果の回収

HLRC では release の時点で diff を作成し home に転送する。 home は差分をページに適用した後 diff を捨てる。

ADSM では release の時点で登録された *write history* の情報を元に、プロセッサが必要とする差分を計算し home に転送する。 home は差分をページに適用した後に *write history* を捨てる。

3 HYBRID プロトコル

ページフォールトの際、 LRC では差分を取り寄せるために二個以上のプロセッサと通信しなければならない場合が出てくる。 SAURC では home と通信するだけでいいが、ページ全体を取り寄せねばならないため、差分のみを転送するより高いバンド幅が必要とされる。

我々は LRC と SAURC の抱える問題を克服する新しい HYBRID プロトコルを提案する。 HYBRID プロトコルはページフォールト時に home と通信するだけで良くて、なおかつ差分のみを取り寄せる方式である。

・書き込みの検知

コンパイラが共有書き込みをストア命令と *write history* を登録するコード列に変換する。

・書き込みの結果の回収

release の時点で登録された *write history* の情報を元に、プロセッサが必要とする差分を計算し home に転送する。 home は差分をページに適用した後に *write history* の情報を保持しておく。

ページフォールトの際、 home はページフォールトを起こしたプロセッサの時計を受け取り、保持しておいた *write history* の情報を使用して、ページフォールトを起こしたプロセッサに欠けている部分を計算する。

3 共有領域への書き込みのオーベーヘッドの削減

コンパイラが共有領域への書き込みを検出するために、手続き間ポインタ解析[2],[8]の手法を用いる。これは共有変数のreferenceや動的な共有領域の確保を源とした前進型手続き間データーフロー解析である。これによって共有アドレスへのポインタを保持し得る変数を検出し、該当する書き込みの後にコンシステンシ管理コードを挿入する。

一連の共有領域への書き込みが連続したアドレスに対して行なわれていて間に同期コードを含まない場合、コンシステンシ管理コードの情報としては、連続した大きな領域全体に対する一つのコンシステンシ管理コードと等価である。以下にコンシステンシ維持コードのcoalescingを行なうアルゴリズムを述べる。

ループには$depth$があり、n重ループネストの最外ループの$depth$は1で最内ループの$depth$はnとする。コンシステンシ維持コードはアドレスとサイズの組(A,S)で表す。まず、必ず実行される共有書き込み[†1]を含むn重ループネストを列挙して、各ループネストに対して以下を実行する

1. 各共有書き込み毎に管理コードを生成。
2. $Depth := n$。
3. $Depth = 0$ならば終了。
4. $depth = Depth$を満たす各ループ(L)に対して以下を実行する。
   (a) ループLの中にコンシステンシ維持コードが複数存在して、そのアドレス部分が連続であり、コード間に同期コード/リターンコードがなければ、コンシステンシ維持コードをcoalescingする。
   (b) 同期コード/リターンコードがあれば4.へ。
   (c) $I(A,S)$で表される各コンシステンシ維持コードに対して
      i. A(アドレス部分)がループ不変である場合: ループLからコンシステンシ維持コードを削除して、ループLの外側にコンシステンシ維持コードを挿入する。
      ii. AがループLの誘導変数である場合: もしAのストライド(A_{stride})がS以下であれば、coalescing可能であり、ループLからコンシステンシ維持コードを削除して、ループLの外側にコンシステンシ維持コードを挿入する。挿入されるコンシステンシ維持コードは$I(A_{low},(c-1)*A_{stride}+S)$と表現される(ただし$c$はループ$L$の回数で$A_{low}$は$A$の最小値)。
5. $Depth := Depth - 1$として4.へ。

coalescing変換を行なうと、領域$\{w \mid A \leq w \leq A+S\}$がページを跨る場合がある。そこでコンシステンシ維持コード内で、動的に領域wが属する各ページに対して必要な処理を行なっている[†2]。

4 関連研究

オブジェクトベースのソフトウェアDSMであるMidway[1]では各共有データは同期変数に束縛されている。lockのacquireの時点で、そのlockに束縛されたデータの変更のみが伝えられる(Entry Consistency)。Midwayではコンパイラが共有領域への書き込みをストア命令と定まったコード列[†3]に変換するだけである[10]。従ってプロトコルの切替えやコード列のコンバイニングといった最適化は行うことはできない。

Eager RCをソフトウェアで実装したShasta[7]では、アセンブラレベルのinstrumentationによって局所領域以外のアクセスにコンシステンシ維持コードが挿入される。batching Miss CheckやInvalid Flag Techniqueといった種々の最適化を行なっているが、ループレベルのコンシステンシ維持コードのcoalescingは行なっていない。

5 実験と評価

我々はADSMのコンパイラとランタイムシステムのプロトタイプをAP1000+に実装してきた。

†1 共有書き込みが条件節の中にあるような場合は必ず実行される訳ではない

†2 連続だからページの跨ぎ方のパターンは少ない

†3 該当アドレスのSoftware dirty bitをセットする

AP1000+ のセル OS はユーザレベルの割り込みハンドラをサポートしていないため、仮想記憶機構を利用できない。しかし、コンシステンシ維持コード数を削減するコンパイル技法は仮想記憶機構とは無関係なので、AP1000+ 上の実験でもコンパイル技法の効果を確認することができる。

今回の実装では、コンパイラが、共有ページへのアクセスの前にページの有効性をチェックし、もし有効でないページにアクセスしたらユーザーレベルのページフォールトハンドラを呼ぶコード列を挿入した。また、リモートノードからのメッセージはポーリングで処理した。ポーリングのコードはコンパイラがループのバックエッジ及び各関数呼びだしに挿入する。

アプリケーションとして SPLASH-2 [9]のカーネルのうち LU-Contig, Radix, FFT の三つを使用した。LU は 256 × 256 行列でブロックサイズが 16、Radix はデータが 262,144 個で基数が 1,024、FFT は 16,384 点である。各図において GC は LRC プロトコルにおける GC の時間、sync はロックやバリアなど同期処理にかかる時間、CM はコンシステンシ管理コードの時間、PF はページフォールト処理にかかる時間、msg は task 実行中のポーリングでメッセージを処理した時間、task はアプリケーションプログラム本来の計算時間を表す (現在の実装ではポーリングとページテーブルチェックのオーバーヘッドを含む)。グラフの縦軸は実行時間 (秒)、横軸はプロセッサ台数である。

図 1は SAURC プロトコルで管理コードの coalescing の効果を調べたものである。O は coalescing したもの、N はしないものを示す。LU, FFT ではボトルネック部分のオーバーヘッド軽減に成功している。coalescing によってデータ参照の局所性が変化すること、またコンシステンシン管理コードへの手続き呼び出しは task にカウントされていることから task も減少している。

また、図 2は coalescing を行なった状態でプロトコルを変えたときの様子である (全ての共有領域についてプロトコルは同じ) †4。L は LRC、A は SAURC、H は HYBRID を示す。

†4 radix の 16PE の HYBRID はデータ領域が不足のため測定不能

全体としてソフトウェア DSM の特徴でもあるが、台数とともに同期、ページフォールトの時間が増大してくる。この部分を以下に押えるかが今後の課題である。

LRC では GC のオーバーヘッドが大きく、しかも、台数とともにオーバーヘッドが増大している。原因の一つとして LRC では局所的に書き込みの履歴を除去できないという点が挙げらる。ページを全部書き潰すような時は履歴などいらないから、ランタイムを改良して必要ない時には履歴を作らないようにするといった改善を行なう予定である。

コンシステンシ管理コードのオーバーヘッドが主要な割合を占める場合には、より遅延的なプロトコル (LRC,HYBRID) が有利であり、結果的に LU, FFT の 2 台の実行では HYBRID が速い。しかし、全体的にはプロセッサに対してネットワークが相対的にかなり高速で、二次キャッシュを持たず書き込みのコストが大きいという AP+ の環境では SAURC が最も効率的である。

6 まとめ

本稿では ADSM 上で共有書き込みの際のオーバーヘッドを削減するコンパイル技法を提案し、AP1000+ 上に実装したランタイムシステムを用いて最適化の手法が有効であることを確認した。今後は当研究室で開発中の汎用超並列オペレーティングシステム SSS-CORE 上に実装し、評価を行なう方針である。

参考文献

[1] Bershad, B. N., Zekauskas, M. J., and Sawdon, W. A.: The Midway Distributed Shared Memory System, *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, February 1993, pp. 528–537.

[2] Emami, M., Ghiya, R., and Hendern, L. J.: Context-Sensitive Interprocedural Points-To Analysis in the Presence of Function Pointers, *Proc. of '94 Conf. on PLDI*, June 1994, pp. 242–256.

[3] Iftode, L., Dubnicki, C., Felten, E. W., and Li, K.: Improving Release-Consistent Shared Virtual Memory using Automatic Update, *Proc. of the 2nd Inter. Symp. on HPCA*, February 1996.

[4] Keleher, P., Cox, A. L., and Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory, *Proc. of the 19th ISCA*, May 1992, pp. 13–21.

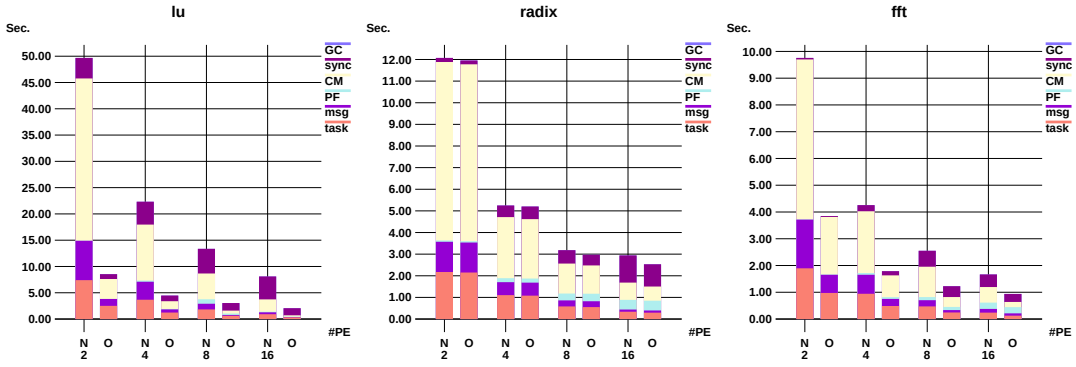

図 1 最適化の効果 (プロトコルは SAURC)

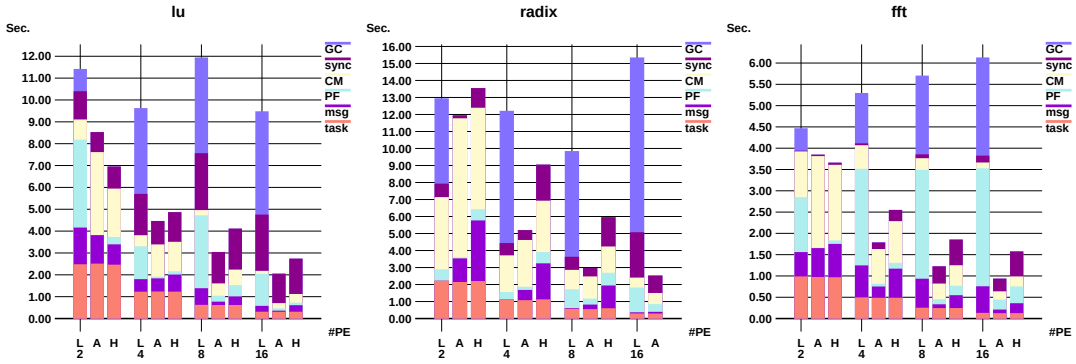

図 2 プロトコル間の較差 (最適化あり)

[5] Keleher, P., Dwarkadas, S., Cox, A. L., and Zwaenepoel, W.: TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, *Proc. of the Winter 1994 USENIX Conference*, January 1994, pp. 115–131.

[6] Li, K.: IVY: A Shared Virtual Memory System for Parallel Computing, *Proc. of the 1988 ICPP*, August 1988, pp. 94–101.

[7] Scales, D. J., Gharachorloo, K., and Thekkath, C. A.: Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, *Proc. of 7th Int. Conf. on ASPLOS*, October 1996, pp. 174–185.

[8] Wilson, R. P. and Lam, M. S.: Efficient Context-Sensitive Pointer Analysis for C Prgrams, *Proc. of '95 Conf. on PLDI*, June 1995, pp. 1–12.

[9] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. of the 22nd ISCA*, June 1995, pp. 24–36.

[10] Zekauskas, M. J., Sawdon, W. A., and Bershad, B. N.: Software Write Detection for a Distributed Shared Memory, *Proc. of the 1st Symp. on OSDI*, November 1994, pp. 87–100.

[11] 松本尚: 細粒度並列実行支援機構, 計算機アーキテクチャ研究会報告, Vol. 89-ARC-77, July 1989, pp. 91–98.

[12] 松本 尚, 駒嵐 丈人, 渦原 茂, 竹岡 尚三, 平木 敬: 汎用超並列オペレーティングシステム SSS-CORE– ワークステーションクラスタにおける実現 –, 情報処理学会研究報告, Vol. 96-OS-73, August 1996, pp. 115–120.