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

Compiling Techniques for ADSM on General-Purpose Massively-Parallel Operating System: SSS–CORE

丹羽 純平　稲垣 達氏　松本 尚　平木 敬
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

概 要

SSS-CORE ではメモリベース通信機能を使って新しいソフトウェア分散共有メモリ (非対称分散共有メモリ) をユーザに提供する。非対称分散共有メモリでは共有領域への読み出しは、従来のソフトウェア DSM と同様にロード命令で行なわれるが、共有領域への書き込みに関してはコンパイラが検知して一連のコード列に変換する。この書き込みのコード列の結合や削除を行なってオーバーヘッドを削減する方法を提案し、 AP1000+ 上に実装して予備評価を行なった。

1 はじめに

従来のソフトウェア分散共有メモリ (DSM) [4]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 [2]。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) [1]) 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 [8]。 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 [7]をサポートする。
- コンシステンシ維持コード列のコアレシング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。

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 [3]の Lazy Release Consistency (LRC) [2]プロトコルと基本的に同じ。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 の実装と異なり、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。 TreadMarks と同様、ログの除去には明示的な同期による GC が必要である。
2. SAURC プロトコル
   ソフトウェアで Automatic update release consistency(AURC) [1]をエミュレートする。 登録された *write history* の情報を元に、 home が必要とする更新箇所を計算し、書き込みの都度ではなく release の時点に、一つのメッセージにコンバイニングして 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複数のプロセッサと通信しなければならない場合がある。 SAURC では home と通信するだけでよいが、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 プロトコルを提案した。 HYBRID プロトコルはページフォールト時に home と通信するだけでよく、なおかつ差分のみを取り寄せる方式である。

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 [5]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。

†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

(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 $I\left(A_{low}, (c-1) * A_{stride} + S\right)$ 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。

コアレシング変換を行なうと、領域 $\{w \mid A \geq w \geq A + S\}$ 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は領域 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。

4 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。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 [6]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)。 L は LRC、 A は SAURC、 H は HYBRID を示す。 LRC では GC のオーバーヘッドが大きい。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。

5 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] Iftode, L., Dubnicki, C., Felten, E. W., and Li, K.: Improving Release-Consistent Shared Virtual Memory

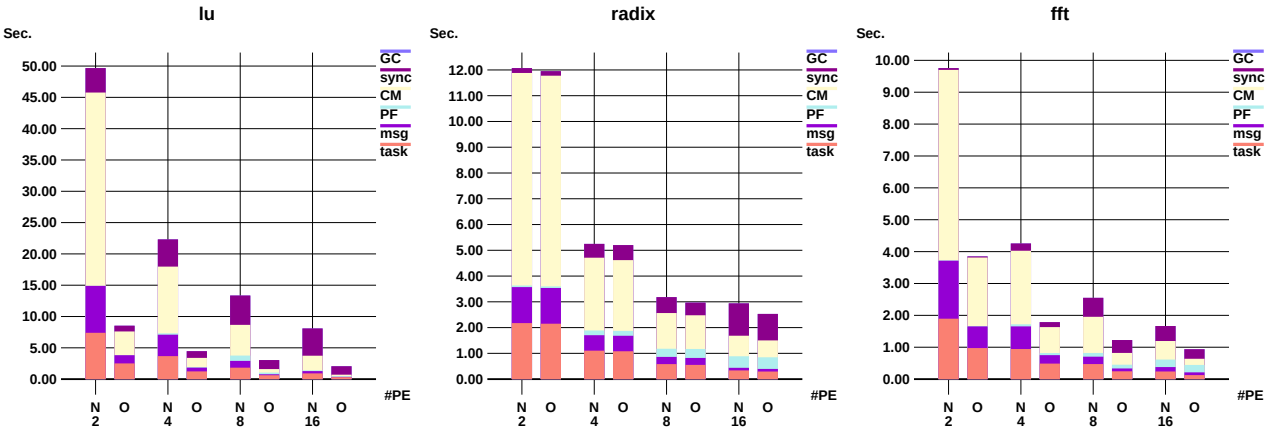

図 1　最適化の効果 (プロトコルは SAURC)

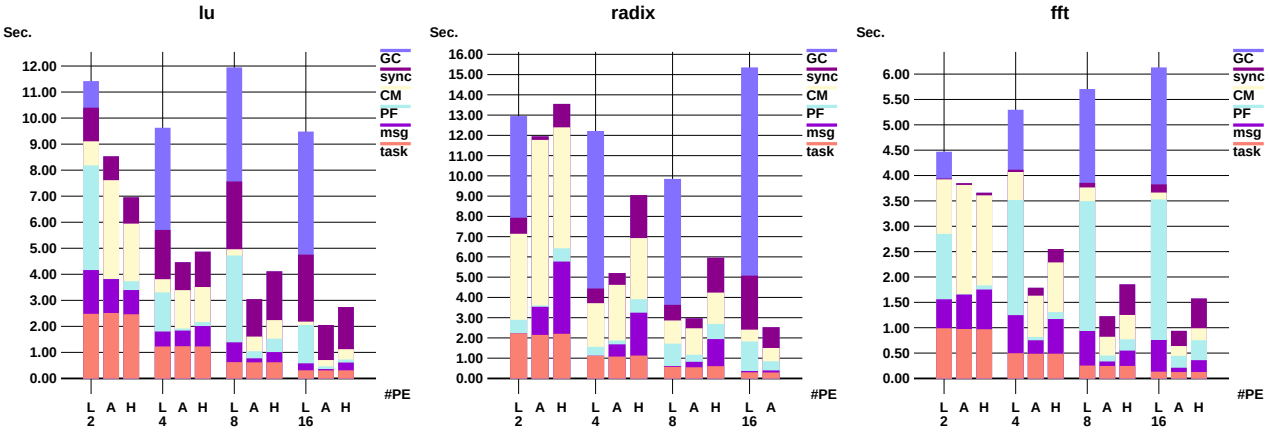

図 2　プロトコル間の較差 (最適化あり)

using Automatic Update, *Proc. of the 2nd Inter. Symp. on HPCA*, February 1996.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.