2D–03

汎用超並列オペレーティングシステム SSS-CORE
— コンパイラによる通信最適化技法 —

丹羽 純平　稲垣 達氏　松本 尚　平木 敬
東京大学大学院理学系研究科情報科学専攻

1　はじめに

SSS-CORE ではメモリベース通信機能 [5],[6],[7] 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[5]) 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。本稿では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、 SSS-CORE 上で実装して評価を行なう。

2　非対称分散共有メモリ

非対称分散共有メモリでは読み出しと書き込みの実現モデルが異なっている[1]。

- 共有領域への読みだし
  従来のソフトウェア DSM と同様にプロセッサのページ管理機構を利用したページ単位のキャッシッングをおこなって、プロセッサの load 命令で実行する。従ってキャッシュヒット時はオーバーヘッドがない。共有されているページのキャッシュミスはページトラップで検知して対処する。
- 共有領域への書き込み
  書き込まれるデータの従うべきプロトコルに従って、遠隔メモリアクセスや付加的なメモリ操作といったコンシステンシ維持のためのコード列 (コンシステンシ維持コード) を実行コードとして埋め込む。埋め込まれたコンシステンシ維持コードにより明示的なパケット通信を行なう。

共有領域への書き込みをストア命令から一連のコード列に変換してしまうために従来の狭義共有メモリ (遠隔メモリ操作をプロセッサレベルのメモリ操作としてユーザに見せる) で行なわれてきた最適化に加えて以下の最適化が可能になる。

- 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、コンシステンシ維持コードの列を連続領域に対する一つのコンシステンシ維持コードに変換することによって、通信のオーバーヘッドと実行時の書き込みを管理するオーバーヘッドが軽減される。
- 通信パケットのコンバイニング
  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すことで、通信のオーバーヘッドが削減される。

3　共有領域への書き込みの検知

我々のコンパイラは、 SPLASH や SPLASH-2 で用いられている C 言語に PARMACS というマクロを加えたプログラミングモデルをサポートしている。このモデルではユーザは同期プリミティブによる明示的な同期、 **G_MALLOC** オペレーションによる共有領域の動的な確保を行なうことができる。共有領域への書き込みの検知は以下のような手順からなる。

1. 手続き間ポインタ解析 [1],[4] によって G_MALLOC オペレーション (共有領域の動的な確保) の値を推移的に指す可能性のあるポインタを検出する。
2. 上記で検出されたポインタを用いた書き込みの後に、コンシステンシ維持コードを挿入する。

4　共有領域への書き込みのオーバーヘッド削減

コンシステンシ維持プロトコルとして LRC(lazy release consistency) モデル [2] を仮定した場合、共有領域への書き込みのコンシステンシ維持コードを次の同期ポイントに到達するまでに、できるだけ連続したものにまとめ、冗長なものを省きたい。このときの冗長性の検出は共通部分式の削除と同様に [3]、プログラムの制御フローグラフの大域的な availability, anticipatability を計算することによって行なう。

4.1　手続き間解析

共有領域への書き込みを一つ固定する (各データフロー変数は真偽値を取る)。プログラム中の各文 i のどれが該当する共有書き込みを行なうかどうかが、先行するポインタ解析によって分かっているものとすると、その情報から以下の変数の値が求められる。

- $\mathrm{COMP}(i)$: 文 i が指定された共有領域への書き込みを行なう。
- $\mathrm{TRANS}(i)$: 文 i が上下の文に情報を伝える。文 i が同期である場合か、共有書き込みのパラメータを変更する場合には偽になる。

これらから文 i における共有書き込みの availability (前の同期ポイントから文 i の間で共有書き込みが行なわれる) および anticipatability(文 i から次の同期ポイントの間に共有書き込みが行なわれる) は次のように計算される。

$$
\begin{aligned}
\mathrm{AVIN}(i) &= \prod_{p\in \mathrm{pred}(i)} \mathrm{AVOUT}(p) \\
\mathrm{AVOUT}(i) &= \mathrm{COMP}(i) + \mathrm{TRANS}(i)\cdot \mathrm{AVIN}(i) \\
\mathrm{ANTOUT}(i) &= \prod_{s\in \mathrm{succ}(i)} \mathrm{ANTIN}(s) \\
\mathrm{ANTIN}(i) &= \mathrm{COMP}(i) + \mathrm{TRANS}(i)\cdot \mathrm{ANTOUT}(i)
\end{aligned}
$$

$\mathrm{pred}(i), \mathrm{succ}(i)$ はそれぞれ文 i に先行、後継する文の集合を与える。これらによって、文 i の後ろに共有書き込みへのコンシステンシ維持コードを挿入するかどうかを表す変数 $\mathrm{INSERT}(i)$ は次のような式で表される。

$$
\mathrm{INSERT}(i) = \mathrm{AVOUT}(i)\cdot\neg\left(\prod_{s\in \mathrm{succ}(i)} \mathrm{AVOUT}(s)\right)\cdot\neg\mathrm{ANTOUT}(i)
$$

availability は前進型データフロー方程式、 anticipatability は後退型データフロー方程式に従うので、全体の計算は二つのパスに分かれる。手続き間で計算を行なう際に、最も簡単な方法は手続き呼び出しにおいて呼び出される手続きの出口における availability を手続き呼び出しの COMP 変数とみなす方法である。サブルーチンの出口での INSERT は、 AVOUT が呼び出し元に伝搬されたかどうかを考慮して計算される。この単純な方法では anticipatability の計算は各手続き内で実行される。全体の計算は手続き呼び出し木を bottom-up に走査する順で行なわれる。さらに詳細に計算する場合は、 availability および anticipatability を手続き間で計算する。すなわち、手続き呼び出しの context における availability および anticipatability をパラメータマッピングによって呼び出される手続きに反映する。

General-Purpose Massively-Parallel Operating System: SSS–CORE — Compiler optimization for communication overhead reduction —
Junpei NIWA, Tatsushi INAGAKI, Takashi MATSUMOTO, Kei HIRAKI
niwa@is.s.u-tokyo.ac.jp

[1]これが名前の由来である

4.2 連続領域に対する書き込みの表現

各共有書き込みを、共有書き込み集合 (shared write set) によって表す。共有書き込み集合は三つ組

$$W = (f, s, C)$$

で表される。f は共有領域の先頭アドレス、s は一回の書き込みによるサイズ、C は f, s を生成する不等式制約の集合であり、具体的にはプログラム中のループに対応する。この表現を用いると、連続アドレスへの書き込みのコンシステンシ維持コードをまとめるコアレシング変換は

$$(\&a[i], 1, \{0 \le i < n\}) \rightarrow (a, n, \emptyset)$$

と表現される。コアレシングが可能である必要条件は、不等式制約には明示されないが、誘導変数の刻み幅 c に関する条件式

$$f(i + c) - f(i) \le s$$

が成立することである。同様にしてテキスト上の複数の連続した書き込みをまとめる fusion 変換は

$$(\&x[2 * j], 1, \emptyset) \circ (\&x[2 * j + 1], 1, \emptyset) \rightarrow (\&x[2 * j], 2, \emptyset)$$

と表される。また、同じアドレスへの書き込みが繰り返される場合を Fourier-Motzkin 消去によって検出可能な場合がある。例えば、

$$(\&a[k + j * stride], n - k, \{k + 1 \le j < n,\ 0 \le k < n\}) \rightarrow$$

$$(\&a[j * stride], n, \{1 \le j < n\})$$

では、始めの共有書き込み集合が $k = 0$ の場合に全て覆われていることがわかる。

前述の各データフロー変数を、共有書き込み集合を値に取るように計算することで、複数の共有書き込み間の冗長性の検出およびコンシステンシ維持コードの最適化を行なう。大域データフロー解析において区間解析の考え方を用い、ループ (区間) のサマリを計算する際にコアレシング変換及び冗長なインデクスの除去を行ない、書き込み集合の和を計算する際に fusion や包含関係による冗長性の検出を行なう。書き込み集合間の包含関係も Fourier-Motzkin 消去を用いて計算する。

5 実験と評価

我々は非対称分散共有メモリのコンパイラとランタイムシステムのプロトタイプを NOW 版の SSS-CORE に実装してきた。SSS-CORE の各クラスタノードは Axil 320 model8.1.1(Sun SS20 互換機, 85MHz SuperSPARC × 1) からなり、Fast Ethernet SBus Adapter2.0 を追加して 100BASE-TX のスイッチで Fast Ethernet 接続されている。ユーザーレベルの保護された高速なメモリベース通信 [5],[6],[7] を実行し、ピークバンド幅は 11.2MBytes/sec である。

本実験ではコンシステンシ維持プロトコルとして SAURC[8](Automatic Update Release Consistency をソフトウェアのみで実装したプロトコル) を使用し、アプリケーションとして SPLASH-2 の Application の Water-Spatial を採用した。Water-Spatial は空間分割をした水分子の N 体問題である。水分子の数は 512 で空間は $4 \times 4 \times 4 = 64$ 個のセルに分割される。仮想記憶機構を利用して割り込みでページフォールトを検出する部分は現在実装中である。現時点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するコードを挿入している。今回はチェックコードの最適化を人手で行なった。

グラフ (図 1) の縦軸が実行時間 (秒) で横軸がプロセッサ台数である。**Sync** はロックやバリアなど同期処理にかかる時間で、**CM** はコンシステンシ維持コードの時間で、**PF** はページフォールト処理にかかる時間で、**Msg** は Task 実行中にメッセージを処理した時間で、**Task** はアプリケーションプログラム本来の計算時間である (1 台以外の時は共有ページのチェックコードのオーバーヘッドが加わっている)。どの台数の時もコンパイラの最適化によりコンシステンシ維持コードのオーバーヘッド (CM) は低く押えられていることがわかる。2 台の時には SAURC は update 型プロトコルになるために無用なパケットが飛ぶ可能性があるから、コンシステンシ維持コードのオーバーヘッドが 4 台、8 台の時と比べて相対的に大きい。その代わり、台数が増えるにつれてページフォールトと同期の時間が増大している。

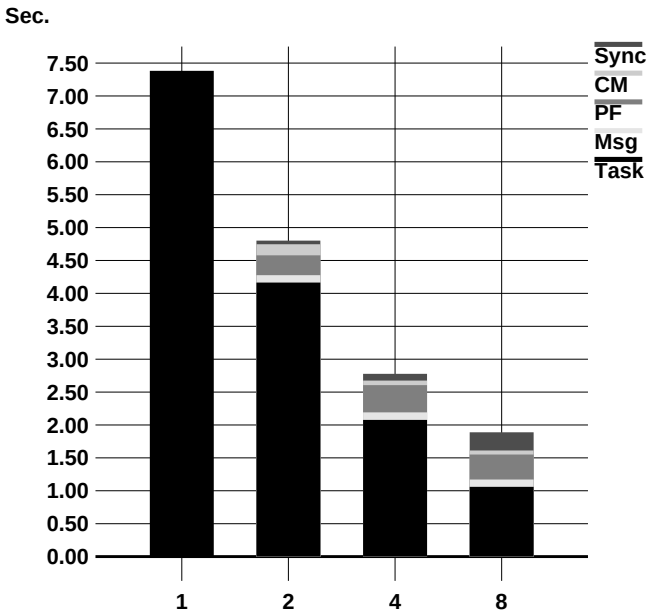

図 1: Water-Spatial の実行時間の内訳

6 おわりに

我々は非対称分散共有メモリにおいて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た。SPLASH-2 の Water-Spatial を使用した SSS-CORE 上の実験で、コンシステンシ維持コードのオーバーヘッドは十分に削減されることを確認した。今後は割り込みでページフォールトを検出する部分を完成し、より多くのアプリケーションを走らせることで更なる性能評価を行なう予定である。

参考文献

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

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

[3] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, Vol. 22, No. 2, pp. 96–103, February 1979.

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

[5] 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, 第 96-OS-73 巻, pp. 115–120, August 1996.

[6] 松本 尚, 駒嵐 丈人, 渦原 茂, 平木 敬. メモリベース通信による非対称分散共有メモリ. コンピュータシステムシンポジウム論文集, pp. 37–44, November 1996.

[7] 松本 尚, 平木 敬. 汎用並列オペレーティングシステムにおける資源保護と 仮想化. 情報処理学会研究報告, 第 97-OS-75 巻, pp. 37–42, June 1997.

[8] 丹羽 純平, 稲垣 達氏, 松本 尚, 平木 敬. 非対称分散共有メモリ上におけるコンパイル技法. 情報処理学会研究報告, 第 97-HPC-67 巻, pp. 121–126, August 1997.