コンパイラが支援するソフトウェア DSM 機構:ADSM と UDSM の性能評価

丹羽純平[†,††] 松本尚[†] 平木敬[†]

我々はコンパイラが支援する 2 つのソフトウェア共有メモリ機構を提案してきた。1 つは、読み出しミス時のみ TLB/MMU の支援を必要とするページベースのキャッシュ機構で、もう 1 つは読み書きともに完全にユーザレベルのコードで実現されるキャッシュ機構である。いずれの機構も、アプリケーションプログラムの情報を元にコンパイラが最適化を施す。手続き間ポインタ解析により共有アクセスを検知して、キャッシュエミュレーションコードを挿入し、手続き間冗長性削除の枠組みで、冗長なキャッシュエミュレーションコードを取り除き、coalescing 等によりキャッシュエミュレーションコードの粒度を大きくする。我々は上記の最適化コンパイラのプロトタイプを作成して、ワークステーションクラスタ上に 2 つのソフトウェア共有メモリ機構のランタイム実装した。SPLASH-2 ベンチマークを用いた実験によりコンパイラの最適化の効果を確かめ、コンパイラで支援された共有メモリ機構が高い高速化率を得ることを確かめた。

Evaluation of Compiler-Assisted DSM Schemes:ADSM and UDSM

Junpei NIWA,[†] Takashi Matsumoto[†] and Kei Hiraki[†]

We have proposed two compiler-assisted software-cache schemes. One is a page-based system (Asymmetric Distributed Shared Memory: ADSM) which exploits TLB/MMU only in the cases of read-cache-misses. Another is a segment-based system (User-level Distributed Shared Memory: UDSM) which uses only user-level checking codes and consistency management codes for software-cache. Under these schemes, an optimizing compiler directly analyses shared memory source programs, and performs sufficient optimization. It exploits capabilities of the middle-grained or coarse-grained remote-memory-accesses in order to reduce the number and the amount of communications and to alleviate overheads of user-level checking codes. It uses interprocedural points-to analysis and interprocedural redundancy elimination and coalescing optimization. We have implemented the above optimizing compiler for both schemes. We also have implemented runtime systems for user-level cache emulation. Both ADSM runtime system and UDSM runtime system run on the SS20 cluster connected with the Fast Ethernet(100BASE-TX). We have revealed that both schemes achieve high speed-up ratio with the SPLASH-2 benchmark suite.

1. はじめに

近年、安価で高性能なワークステーション (WS) やパーソナルコンピュータ (PC) が市場に出回るようになった。その結果、ネットワークで相互接続して使用する計算機クラスタが、価格性能比の良さから並列計算環境としても注目を浴びつつある。

我々の目的は、計算機クラスタ (i.e., 分散メモリ環境) で、共有メモリモデルに基づいた並列プログラムを効率良く実行することである。共有メモリモデルの効率の良い実現には、遠隔メモリの内容をアクセスしたプロセッサの近くにキャッシュするハードウェア機構ないしはソフトウェア機構が必須である。しかし、既存の方式は様々な問題点を抱えている。

- ハードウェアを付加することで実現する方針はどうしても実装コストが問題になる。また共有されるメモリは I/O Interface 上に取らざるをえない。
- TLB/MMU を利用したオペレーティングシステム (OS) のみによる実装は、逐次のコンパイラの使用を前提としている。したがって、実行コードのリモートメモリアクセスはプロセッサの単純な load/store に変換されてしまい、コンシステンシ管理操作はユーザからは見えない。そのためアプリケーションレベルの最適化の情報を反映させることができない。
- アプリケーションのバイナリを解析して、プライベートでない load/store に対してコンシステンシ管理コードを挿入する方針も存在する。false sharing を避けるために、コンシステンシ管理は細粒度で行なわれ、無駄な転送は起こらない。しかし、短い長さのメッセージが頻繁に発行されるので、高いバンド幅のネットワークが必要になり、汎用のネットワークで効率良く実現するのは困難である。

我々は、OS ベースのソフトウェア DSM と同様に、主記憶の空き領域を遠隔ノードのデータをキャッシュするのに使用する。ユーザプログラムから最適化が施せるように、このキャッシュのコンシステンシ管理はユーザレベルで行なう。我々は、最適化コンパイラがアプリケーションのソースコード

† 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
†† 日本学術振興会特別研究員
JSPS Research Fellow

を直接解析して、アプリケーションの持つ情報をランタイム/OS にフィードバックして、性能向上を図る方針を取る。つまり、最適化コンパイラがソースコードを解析して、共有データアクセスを検出し、キャッシュのコンシステンシ管理コードとそれに伴う通信コードを明示的に大きな粒度で埋め込むコード生成を行う[9]。

我々は上記の最適化コンパイラが支援する 2 種類の共有メモリ機構 (ADSM[12] と UDSM[13]) を提案してきた。本稿では、両機構を定性的に比較し、コンパイル技法のフレームワークを述べて、 SPLASH-2 ベンチマークによる両機構の定量的な性能評価を行なう。

2. ADSM と UDSM

Asymmetric Distributed Shared Memory (ADSM) は、 OS 支援の共有メモリ機構の流れを汲んでいて、読み出しミス時のみ TLB/MMU の支援を必要とするページベースのキャッシュ機構である。

- 共有読み出しは TLB/MMU を利用した方式によって単純な load 命令としてコードが生成される。キャッシュミスはページフォールトとして検知され、ページフォルトハンドラからユーザレベルのキャッシュミスハンドラを呼び出すことで処理される。
- 共有書き込みに関しては、コンパイラが共有メモリアクセスが必要な書き込みを検知して、キャッシュコンシステンシ管理コードを明示的に実行コード内に挿入する。

書き込みのコンシステンシ管理操作はユーザレベルのコードとして挿入される。したがって、静的に最適化をかけることが可能になり、アプリケーションの情報を反映させられないという従来の OS 支援の DSM が抱える問題を解決する。

User-level Distributed Shared Memory (UDSM) はバイナリ解析支援の共有メモリ機構の流れを汲んでいて、共有領域への読み出し / 書き込みともにユーザレベルのコードで実現されるキャッシュ機構である。コンパイラが共有アクセスが必要なメモリアクセスを検出して、キャシュのチェックコードとコンシステンシ管理コードをユーザレベルのコードとして実行コードに埋め込む。

- チェックコードはブロックの状態を検査して、 Invalid の時は通信を行なって最新のコピーを入手する。共有アクセスを必要とする全ての load/store の前に挿入される。
- コンシステンシ管理コードは共有アクセスを必要とする全ての store の後に挿入される。該当ブロックに書き込みが起こったことを記録しておき、必要ならば他プロセッサに伝える。

バイナリ解析に比べ意欲的な最適化を施せる。 1 つには、ソースを直接解析するので、より正確な共有アクセスの検知ができる。もう 1 つには、ループレベル情報を元にユーザレベルのコンシステンシ管理コードとそれに伴う通信の粒度を大きくする最適化をかけられる。

以下に両機構の相違点を挙げる。

1) ブロックサイズ (コンシステンシ単位)

ADSM ではキャッシュミスをページフォールトとして検出する必要があるために、共有領域のキャッシュブロックサイズはページサイズと等しくなければならない。 UDSM ではユーザレベルのコードでチェックするから、キャッシュブロックサイズは任意である。ただし、あまり小さすぎてもユーザレベルのコンシステンシ管理情報の操作が重くなり、メッセージ長の短い通信が頻発してネットワークがすぐに飽和する可能性がある。逆にあまり大きすぎても、 false sharing が発生して、トラフィック量が増大する可能性がある。最適なブロックサイズの選択が重要である。

2) 共有読み出し

ADSM ではデータが Valid の時はソフトウェアによるオーバーヘッドは零であり、 Invalid の時はトラップが引き起こされる。一方、 UDSM ではデータが Valid/Invalid に関わらず、ブロックの状態を検査するための一定のオーバーヘッドがかかる。しかし、規則的な読み出しを行なうような場合には、キャッシュチェックコードに静的に最適化をかけることが可能になりオーバーヘッドは緩和される。したがって、非常に多くの不規則な読み出しがあるようなプログラムを実行する場合には、 ADSM の方が有利である。

緩和されたメモリモデルに基づく UDSM ではチェックコードは必ずしも load 命令の直前に置く必要はない。チェックコードをできる限り前に置くことで、キャッシュミス時の通信の発行を速めることができ、プリフェッチの効果が得られる。 ADSM ではキャッシュミス時の通信は、実アクセスの時に行なわれる。つまり、 UDSM は ADSM に比べてキャッシュミス時のレイテンシを短くする可能性を持つ。

3. Remote Communication Optimizer

我々は共有メモリベースのプログラムに対する最適化コンパイラ Remote Communication OPtimizer(RCOP) を開発してきた。入力となる共有メモリプログラムは、 PARMACS[2] という並列化マクロで拡張した C 言語によって記述される。 PARMACS はタスク生成、共有メモリの確保、同期 (バリア、ロック、ポーズ) のプリミティブを提供する。

RCOP は、与えられた共有メモリプログラムを解析し、プログラム中の共有メモリアクセスを検出するのであって、並列プロセスの問題を解くのではない。与えられたプログラムの各スレッドの副作用として、共有メモリアクセスを検出する。この時点で、正しい実行に必要な全てのキャッシュチェックコード / コンシステンシ管理コードを生成することができる。

共有メモリのコンシステンシモデルは、 Lazy Release Consistency(LRC) モデル[10] に従う。 LRC モデルでは、共有メモリへの書き込みの結果は、書き込みの後の同期ポイントに到達するまで、他のプロセッサに反映されない。従って、キャッシュチェックコード / コンシステンシ管理コードを置く場所に自由度が発生するので、最適化の可能性が開かれる。

この自由度を利用して、手続き間の冗長性の検出を行ない、区間解析を行なって、キャッシュチェックコード / コンシステンシ管理コードの発行をまとめる。この最適化は実行時の通信のベクトル化に相当している。 RCOP は、ユーザレベルのキャッシュチェックコード / コンシステンシ管理コードが挿入された共有メモリプログラムを、 C 言語のソースの形で生成する。最後に、プラットフォーム上の逐次コンパイラが、生成されたコードをコンパイルし、 DSM のランタイムライブラリとリンクして実行形式を生成する。

コンパイラは両機構を効率良く実現するために

(1) 全ての共有アクセスの検知
(2) 冗長なキャッシュチェックコード / コンシステンシ管理コード[☆]の除去
(3) 粒度の大きなチェックコード / 管理コードの発行 (通信の一括化につながる)

をする必要がある。

3.1 共有アクセスの検知

入力言語は C 言語であるから、共有アドレスはポインタ変数に代入されたり、手続き間に渡って使用されることもある。全ての共有アクセスを検知するために RCOP は手続き間 points-to 解析[16),17)] を適用している。

共有メモリアクセスを検出するには、共有メモリを確保するプリミティブ (G_MALLOC) の返り値を追跡すればよい。 G_MALLOC の値を使う可能性のメモリアクセスが共有アクセス

☆ 以下それぞれチェックコード、管理コードと記述する

である。該当するメモリアクセスの前にチェックコードを挿入し、書き込みの場合には管理コードをその後に挿入する。

3.2 冗長性の除去

チェックコード / 管理コードは共有領域のアドレスと書き込まれるデータのサイズをパラメータとするユーザレベルの手続きである。冗長なチェックコード / 管理コードを取り除く最適化は冗長性の除去として定式化できる[1)]。冗長な管理コードの除去は[9),13)] を参照されたい。

冗長なチェックコードの除去について以下に簡潔に述べる。共有読み出しを一個固定する。手続き間ポインタ解析の結果から、各文 i について以下の真偽値を定数として求める。

$\mathrm{COMP}(i)$　文 i が共有読み出しを行なう

$\mathrm{TRANS}(i)$　文 i が共有読み出しを上下の文に伝える

文 i が同期プリミティブである場合は $\mathrm{TRANS}(i)$ は偽である。文 i がコンシステンシのチェックコードのパラメータを更新する場合も $\mathrm{TRANS}(i)$ は偽である。 上述の定数の値から以下の 5 種類のデータフロー変数を図 1の式を用いて計算する[☆]。

AVIN　先行する全ての文から共有読み出しが文 i に伝えられた

AVOUT　文 i から後続の文に共有読み出しが伝わる

ANTIN　文 i から先行する文に共有読み出しが伝わる

ANTOUT　後継する全ての文から共有読み出しが文 i に伝えられた

INSERT　文 i の前にコンシステンシチェックコードを挿入する

$$\mathrm{ANTOUT}(i) = \prod_{s \in \mathrm{succ}(i)} \mathrm{ANTIN}(s)$$

$$\mathrm{ANTIN}(i) = \mathrm{COMP}(i) + \mathrm{TRANS}(i) \cdot \mathrm{ANTOUT}(i)$$

$$\mathrm{AVIN}(i) = \prod_{p \in \mathrm{pred}(i)} \mathrm{AVOUT}(p)$$

$$\mathrm{AVOUT}(i) = (\mathrm{COMP}(i) + \mathrm{AVIN}(i)) \cdot \mathrm{TRANS}(i)$$

$$\mathrm{INSERT}(i) = \mathrm{ANTIN}(i) \cdot \neg\Big(\prod_{p \in \mathrm{pred}(i)} \mathrm{TRANS}(p) \cdot \mathrm{ANTIN}(p) \Big) \cdot \neg\mathrm{AVIN}(i)$$

図 1　Dataflow equation to remove redundant check codes

まず、 $\mathrm{Anticipatability}(\mathrm{ANTIN}, \mathrm{ANTOUT})$ を後退型データフロー方程式で求める。次に $\mathrm{Availability}(\mathrm{AVIN}, \mathrm{AVOUT})$ と INSERT を前進型データフロー方程式で求める。

- INSERT を、冗長なものをまとめるためになるべく読み出しより前で真にしたい。
- INSERT が真になるのは、同期やパラメータの書き換え後か、制御が分かれた直後 (条件分岐で分かれた直後とかループボディの最初) である。

更に手続き間でこれを計算する。呼び出しグラフを主手続きから深さ優先で走査する。 callee を解析するときには、手続き呼びだしにおける caller の情報をマッピングする。 callee が閉じた (closed) 手続きである時、その手続きの入り口における ANTIN の値が、手続き呼びだしの COMP になる。

3.3 ループ構造を用いた共有アクセスの一括発行

同期区間 (acquire してから release するまでの間[10)]) において、一連の共有アクセスが連続領域に対してなされる場合には、対応する管理コード / チェックコードをまとめて発行する。

一連の管理コード / チェックコードを (f, s, C) の三つ組として表す (これを共有アクセス集合と呼ぶ)。 f と s は管理コード / チェックコードの先頭アドレスとサイズを表し、 C はキャッシュ管理コード / チェックコードが含まれるループの情報 (誘導変数の不等式集合) を表す。 Points-to 解析によって共有アクセスが検出された直後は、各集合はただ一つの共有アクセスを含む。つまり $s = 1, C = \emptyset$ である。

区間解析[3)] の手法を用いて、ループ構造で階層化された Control Flow Graph に対して、論理演算を集合演算に拡張して上述の冗長性の削除を行なう。更に共有アクセス集合を用いて以下の最適化を施す[☆☆]。ループのサマリがループ外に伝えられるとき、 C に共有アクセスを囲むループを表す不等式制約が加えられる。詳細は[9),13)] を参照されたい。

Coalescing　イテレーション間で連続したアドレスに対する管理コード (チェックコード) をまとめる。誘導変数だけではなく連続変数[9)] も利用できる。

Fusion　異なる共有アクセスから発行された管理コード (チェックコード) をまとめる。

Redundant Index Elimination　共有アクセス集合内の管理コード (チェックコード) が重なるような場合に冗長なインデクス変数を削除してコード数を減らす。

4. ランタイムシステム

我々は、 WS クラスタの上の汎用オペレーティングシステム SSS–CORE 上に ADSM/UDSM のランタイムライブラリを実装した。ランタイムに関しては ADSM と UDSM に関して差異はない。 SSS–CORE/WS クラスタにおけるランタイムの基本方針は CellOS/AP1000+ におけるそれ[9)] と同一である。

1) キャッシュコヒーレンスプロトコル

Automatic Update Release Consistency(AURC)[8)] を明示的な通信コードによって、ソフトウェアエミュレーションする SAURC[12)] プロトコルを採用する。ブロック毎にホームノードが存在して、ホームノード以外のノードはブロックのコピーを持つことができる。コピーの更新はコンシステンシ管理コードによって常にホームのブロックに伝えられ、ホームのブロックは最新の状態に保たれる。コピーブロックは無効化型プロトコルで管理される。無効なブロックにアクセスしたらホームから最新の内容を取り寄せる。 LRC 系の他のプロトコルも使用可能であるが、実験により SAURC が無効化型プロトコルの中では優れていることが確かめられている[14)]。

さらに、キャッシュチェックコード / コンシステンシ管理コードの中身を変更することによって全く別なプロトコルをサポートできる。ホームオンリー (書き込み時にコピーを作らない) プロトコルは書き込みのときの fetch-on-write によるトラフィックを回避するのに有効である[12)]。現時点では、プロトコルの切り替えはコンパイラが生成したソースをユーザーが手で書き換えることで実現している。

2) 論理タイムスタンプの不使用

LRC や AURC プロトコルのようにタイムスタンプベクトルを用いた方式では、その複雑さのために同期情報量が増大する。したがって頻繁に同期を取るようなプログラムでは各同期プリミティブの操作が重くなってしまうという欠点がある。

各ブロックに関して、どこの区間でどのプロセッサに書き込まれたかどうかの情報は管理せず、単純に書き込みされたかどうかの情報のみを管理する。

AP1000+ における実装とは異なる点を挙げると、

1) パケットコンバイニング

☆ $\mathrm{pred}(i)$ は、文 i に先行する文の集合であり、 $\mathrm{succ}(i)$ は、文 i に後継する文の集合である

☆☆ Coalescing, Fusion の名前は類似のループ変換から来ている

AP1000+ の場合と同様にコンシステンシコード毎にホームに通信を発行する方針では、コンパイラの最適化が効かない場合に細粒度のパケットが飛び交ってしまう。汎用のネットワークで接続された WS クラスタではネットワークがすぐ飽和してしまう危険がある。そこで、コンシステンシ管理コード内で一旦アドレスとサイズとデータをバッファ上にセーブして、同一宛先のものはマージしてしまう。つまり、動的に通信粒度を大きくするパケットコンバイニングを実行する。

2) リモートリクエストの処理

AP1000+ では、Polling によって検出を行なっていた。つまり、ループのバックエッジと関数呼び出しの所に、メッセージキューをチェックするコードを挿入していた。

SSS–CORE/WS クラスタでは、リモートリクエストは割り込みによって検知され、ユーザレベルハンドラが呼ばれて処理をする。SSS–CORE により、割り込みのハンドラの atomicity は保証されている。

5. アプリケーション

我々は SPLASH-2 ベンチマーク[18] からいくつかのアプリケーションを用いて性能測定を行った。各アプリケーションに対して、ソースプログラムのアノテーションに記されている「共有メモリのホームの指定」を行った。以下のプログラムはソースを変更した。

LU-Contig では部分行列 (i, j) のオーナーと部分行列 (j, i) のオーナーを交換した。**Radix** ではホームオンリープロトコルを使用した。**FFT** は、もとのソースでは、行列の転置操作が、受取側が自分の計算する部分を読み出すように書かれている。これはキャッシュブロックサイズが大きいと激しい false sharing を招く。我々は、これを送り側が相手に書き込むように修正した。また、ホームオンリープロトコルを使用した。**Raytrace** では、ray ID を更新する部分のロックの操作がボトルネックになる。しかし、ray ID は実際の計算には使われていない。我々はこの部分のロック操作を取り除いた。**Ocean** ではグリッドの分割を変更した。正方形状のサブグリッドに分割しないで、列方向で分割して、横長のサブグリッドを生成する (Ocean-Rowwise と呼ぶ[4])。**Barnes** では木を作成する部分を逐次に行なう[6] よう変更した。

6. 実験と性能評価

6.1 実行環境

RCOP は、ADSM 用のコードも UDSM 用のコードも生成できる。バックエンドコンパイラとして gcc-2.7.2 を使用し、最適化オプションは "-O2" で実行した☆。

WS クラスタの各ノードは 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。OS は汎用超並列オペレーティングシステム SSS–CORE を使用した。SSS–CORE では、保護と仮想化の機能を保存したまま他ノードのメモリをユーザレベルで直接操作するメモリベース通信 (**MBCF**:Memory Based Communication Facilities)[11),12)] を実行する。メモリベース通信はパケットの到着保証と順序保証のプロトコルをサポートしていて、通信性能は、ピークバンド幅が 11.93(MBytes/sec) で、ラウンドトリップレイテンシは 49(μsec) である。

リモートからのリクエストメッセージの処理には、高機能メモリベース通信の一つであるメモリベースシグナル (仮想化された遠隔呼び出し) を使用している。メモリベース通信ではコンバインされたパケットの転送もサポートしている。したがって、パケットコンバイニングの最適化も高速に実行できる。

☆ Ocean-Rowwise は除く

6.2 問題サイズと逐次実行時間

各プログラムの逐次実行時間と、1 台で並列プログラムを実行した時の時間と、ADSM/UDSM におけるオーバーヘッドを示したのが表 1である。逐次実行時間は並列プログラムに NULL マクロを適用したものを計測して得られる。UDSM のブロックサイズは Raytrace, Water-NS, Water-SP が 1KB、それ以外は 4KB とした。ADSM のブロックサイズは 4KB 固定 (ページサイズ) である。

ADSM を使用した場合のオーバーヘッドは 0∼9% にとどまる。ADSM のオーバーヘッドはユーザレベルで共有書き込みを実現するために必要なコード列のオーバーヘッドである。UDSM を使用した場合のオーバーヘッドは 3 ~ 20% である。最適化により、読み出し書き込み全てユーザレベルのコードで実現される UDSM のオーバーヘッドでさえ、十分耐えられるレベルまで緩和している。更にこのオーバーヘッドは台数が増えるに連れて、減少するものである。

ADSM と UDSM のオーバーヘッドの差が、ユーザレベルで共有読み出しを実現するために必要なコード列のオーバーヘッドである。FFT において ADSM のオーバーヘッドが UDSM のそれより大きくなっている。ホームオンリープロトコルを実現するにあたって、UDSM はキャッシュチェックコードを省略するだけでいいが、ADSM はページの状態を変更するシステムコールを呼ばなければいけない分、オーバーヘッドが大きくなる。

Barnes は主要な構造体のメンバーへのアクセスの中で、coalescing できないものが存在するため、UDSM のオーバーヘッドは 17% に達する。Ocean-RW は隣接ブロックの一要素へのアクセスといった最適化のかけられないアクセスの数が多いため、UDSM のオーバーヘッドが 12% に達する。

キャッシュチェックコードは、ソース内でインライン展開されるが、その数が多い場合には、バックエンドコンパイラに負担がかかる。Ocean-Rowwise はキャッシュチェックコードがインライン展開された結果、ファイルサイズが大きくなり過ぎて "-O2" オプションをつけてコンパイルできなかった。

6.3 ADSM と UDSM の定量的比較

図 2は RCOP による最適化とユーザによるプロトコル指定とランタイムの最適化とを行った時の ADSM と UDSM における 8 台と 16 台の台数効果を表している。全体としては、いずれか一方が優れていると判断することはできない。個々のアプリケーションを見ていく。

LU-Contig は、各プロセッサが担当する部分行列内の要素が、連続した共有アドレス上に配置される。したがって、読み出し / 書き込みともに手続き間に渡って coalsecing が適用可能であり、いずれも高い高速化率を示している。ADSM の方が若干良い結果が得られている。結果の差はキャッシュチェックコードのオーバーヘッドである。

FFT と Radix は、通信量 / 計算量の比が大きく、高い通信バンド幅を必要とするプログラム[4),7)] であり、計算機クラスタにおいては高性能を得ることは困難とされている。にもかかわらず、ユーザがプロトコル指定 (ホームオンリー) を行うことで台数効果を得ている。FFT の方は UDSM の方が高性能を示している。ホームオンリーの実現に関して、6.2 で述べたように、ADSM の方が UDSM より高オーバーヘッドだからである。

Water-NS では、水分子の分子間力を計算する。水の分子配列をブロック分割しており、各プロセッサの共有データの参照は規則的であり、coalescing が可能である。また参照の局所性も高く、高速化率につながっている。Water-SP では、Water-NS と同じ問題を空間分割で解く。各プロセッサが担当する分子や空間の構造体は 1KB 以下である。いずれの場合も ADSM の時はブロックサイズがページサイズ (4KB) である

表 1　The problem size and sequential execution time (sec)

program	problem size	sequential	ADSM parallel 1PE	overheads (%)	UDSM parallel 1PE	overheads (%)
LU-Contig	2048^2 doubles	435.62	436.34	1	464.38	10
Radix	4M integer keys	6.49	6.53	1	6.85	7
FFT	1M complex doubles	19.14	20.86	9	19.79	3
Raytrace	balls4, 128^2 pixels	171.41	171.44	0	175.80	3
Barnes	2^{15} bodies	55.71	57.20	3	66.51	17
Water-NS	4096 molecules	479.63	487.49	2	498.49	4
Water-SP	4096 molecules	53.23	54.92	3	58.79	10
Ocean-RW	258^2 ocean	20.76	21.67	4	24.47	12

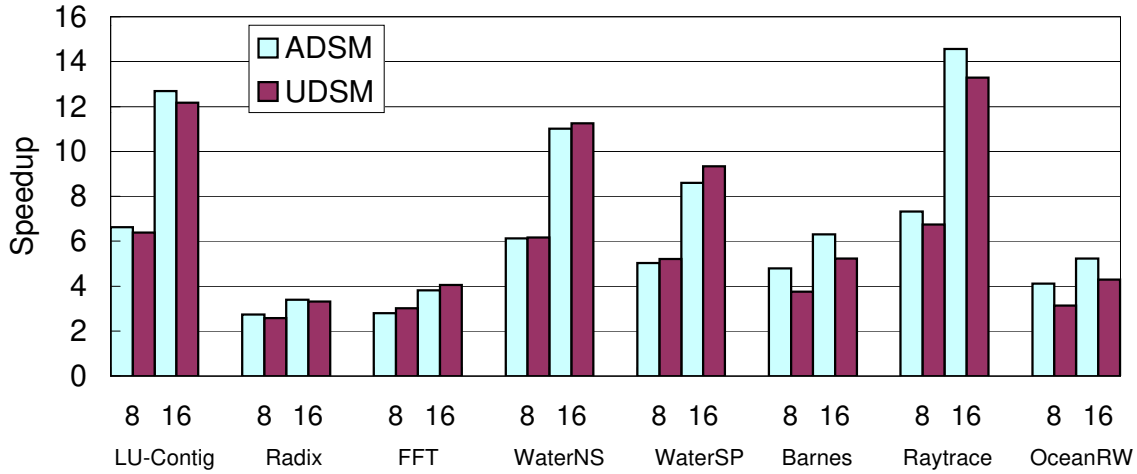

図 2　Speedups on 8 and 16 procs

ため、キャッシュミス時に余分なデータを取ってきてしまい、ネットワークに負担がかかる。UDSM ではブロックサイズを 1KB にすることで、データ転送量を削減できる。この効果は台数が増えるにつれて顕著になる。

Barnes は、八進木を用いて N 体問題を計算する。木の作成は逐次に行なう。一括発行できない共有読み出しが多いことと、ブロックのホームをうまく指定することによりキャッシュミスの回数が低く押さえられることにより、ADSM の方が良い結果を与えている。

Raytrace は並列にレイトレーシングを行なう。共有データの参照は不規則だが、物体のデータは無効化されないので、キャッシュミスのオーバーヘッドはほとんどない。同期のホームの指定により、同期のオーバーヘッドもほとんどなく、並列度が非常に高い。両者の差が、ユーザレベルのキャッシュチェックコードのオーバーヘッドである。

Ocean-RW は、二次元配列を横長ブロックに分割するので、coalescing 可能な共有アクセスが含まれる。しかし、他プロセッサの所有する隣接ブロックの要素にアクセスする機会が多く、それは連続とはならない。さらに、共有メモリアクセス全体の数も多いので、ADSM の方が UDSM より高速になっている。UDSM では、キャッシュチェックコードをできる限り前に移動する最適化によって、キャッシュミスの応答時間を改善している。その効果により台数が増えるにつれて両者の差は縮まってくる。

ハードウェア DSM 用に書かれた SPLASH-2 の 8 個のアプリケーションにおいて、両機構ともコンパイラの支援によって台数効果を挙げていることが分かる。LU-Contig, Radix, Barnes, Raytrace, OceanRW は ADSM の方が良い結果を与えている。これらはキャッシュミスが少ないため、Valid 時にソフトウェアオーバーヘッドのない ADSM の方に有利である。FFT, WaterNS, WaterSP は UDSM の方が良い結果を与えている。キャッシュミスの時に無駄な転送がないということが主な原因である。

7. 関連研究

コンパイラが、共有メモリプログラムを直接解析してソフトウェア DSM を支援する方式はいくつか存在する。

Shasta では、逐次コンパイラが生成したアセンブラレベルのソースを入力とする[15]。彼らは仮想記憶機構を用いず、全てユーザレベルで実現している。オーバーヘッド緩和のために様々な工夫をしている。しかし、コンシステンシ管理の粒度は細かく、ループレベルの最適化は行なっていない。そのため、メッセージ長は短く、高いバンド幅のネットワークが必要になる。

Dwarkads 達は我々のように明示的に並列に書かれたプログラムを入力としている☆[5]。Regular section analysis を用いて変数のアクセスパタンを解析している。その情報を TreadMarks の DSM システムにフィードバックして通信の一括化や、コンシステンシ管理のオーバーヘッド除去を行なっている。ただし、手続き間解析を行なっておらず、誘導変数を用いた解析しかしておらず、連続変数を用いた最適化は行なっていない。

☆ ただし Fortran のプログラムである

8. ま と め

ADSM は OS の支援を必要とするページベース方式であり、UDSM は完全にユーザレベルのコードで実現されるセグメントベースの方式である。本稿ではその差異を定性的に述べ、それらを支援するコンパイル技法を述べ、SPLASH-2 を用いた WS クラスタ上の実験により、両機構の比較を定量的に行なった。

UDSM の性能を左右するのはユーザレベルのチェッキングコードのオーバーヘッドである。逆に ADSM の性能を左右するのは、ブロックサイズがページサイズ固定であるために、通信の際、不必要なデータまで取り寄せてしまうことである。この結果を考慮にいれてプログラムを書いて、プログラムの特質にあった機構を選択するのが最善である。

コンパイラが共有アクセスを検出して、チェックコード / コンシステンシ管理コードをユーザレベルで埋め込み、最適化を施すという方針によって、共有メモリ型並列プログラムが汎用のネットワークで接続された計算機クラスタにおいて十分な高速化率を達成できることが確かめられた。

謝 辞

本研究は情報処理振興事業会 (IPA) が実施している独創的情報技術育成事業の一環として行なった .

参 考 文 献

1) Agrawal, G., Saltz, J. and Das, R. Interprocedural Partial Redundancy Elimination and its Application to Distributed Memory Compilation, Proc. of '95 Conf. on PLDI (June 1995).
2) Boyle, J., et al. *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc. (1987).
3) Burke, M. An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis, *ACM Transactions on Programming Languages and Systems*, **12**, 3 (July 1990), 341–395.
4) D. Jiang, H. S. and Singh, J. P. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors, Proc. of 1997 Principles and Proctice of Parallel Programming (jun 1997).
5) Dwarkadas, S., Cox, A.L. and Zwaenepoel, W. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System, Proc. of the ASPLOS-VII (Oct. 1996).
6) Dwarkadas, S., Gharachorloo, K., Kontothanassis, L., Scales, D. J., Scott, M. L. and Stets, R. Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory, Proc. of the tth IEEE Symp. on High-Performance Computer Architecture (HPCA-5) (Jan. 1999).
7) Erlichson, A., Nuckolls, N., Chesson, G. and Hennessy, J. SoftFLASH: Analyzing the Performance of Clusterd Distributed Virtutal Shared Memory, Proc. of ASPLOS-VII (Oct. 1996).
8) Iftode, L., Dubnicki, C., Felten, E. W. and Li, K. Improving Release-Consistent Shared Virtual Memory using Automatic Update, Proc. of the 2nd HPCA (Feb. 1996).
9) Inagaki, T., Niwa, J., Matsumoto, T. and Hiraki, K. Supporting Software Distributed Shared Memory with a Optimizing Compiler, Proc. of the 1998 ICPP (Aug. 1998).
10) Keleher, P., Cox, A.L. and Zwaenepoel, W. Lazy Release Consistency for Software Distributed Shared Memory, Proc. of the 19th ISCA (May 1992).
11) Matsumoto, T. and Hiraki, K. Performance of Memory-Based Communication Facilities using Fast Ethernet (100BASE-TX), Proc. of the 1997th IPSJ Computer System Symposium (Nov. 1997), (in Japanese).
12) Matsumoto, T., Komaarashi, T., Uzuhara, S. and Hiraki, K. The Asymmetric Distributed Shared Memory using Memory-Based Communication Facilities, Proc. of the 1996th IPSJ Computer System Symposium (Nov. 1996), (in Japanese).
13) Matsumoto, T., Niwa, J. and Hiraki, K. Compiler-Assisted Distributed Shared Memory Schemes Using Memory-Based Communication Facilities, Proc. of the 1998 PDPTA, Vol. 2 (July 1998).
14) Niwa, J., Inagaki, T., Matsumoto, T. and Hiraki, K. Compiling Techniques on Asymmetric Distributed Shared Memory, IPSJ High Performance Computing SIG Notes, Vol. 97-HPC-67 (Aug. 1997), (in Japanese).
15) Scales, D. J., Gharachorloo, K. and Thekkath, C. A. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, Proc. of ASPLOS-VII (Oct. 1996).
16) Wilson, R. P. *Efficient Context-Sensitive Pointer Analysis for C Programs*, PhD thesis, Stanfor University, Computer Systems Laboratory (Dec. 1997).
17) Wilson, R. P. and Lam, M. S. Efficient Context-Sensitive Pointer Analysis for C Programs, Proc. of '95 Conf. on PLDI (June 1995).
18) 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).