非対称分散共有メモリ上における最適化コンパイル技法の評価

丹羽純平[†] 稲垣達氏[†,☆]
松本尚[†] 平木敬[†]

我々は保護された高速なユーザー通信／ユーザー同期を実現する"非対称分散共有メモリ:ADSM"を提案してきた．ADSM は読み出しと書き込みの実現モデルが別々で，読み出しは通常の仮想共有メモリ方式と同様であるが，書き込みに関してはコンシステンシ維持コードが埋め込まれる．書き込みの自由度が高いから，様々な最適化が可能になる．我々はコンシステンシ維持コードの数を静的／動的に削減することで，書き込みのオーバーヘッドを削減する最適化技法を提案する．汎用並列オペレーティングシステム SSS–CORE と AP1000+ 上に作成したコンパイラ並びにランタイムシステムにおいて SPLASH-2 の LU-Contig を使って評価を行なった．実行時間は静的な最適化により 80% 向上し，更に動的な最適化をおこなうことで 30% 向上した．

Performance Evaluation of Compiling Techniques on Asymmetric Distributed Shared Memory

Junpei NIWA,[†] Tatsushi Inagaki,[†,☆] Takashi Matsumoto[†] and Kei Hiraki[†]

We have proposed an "Asymmetric Distributed Shared Memory: ADSM", that realizes user-level protected high-speed communications/synchronizations. In the ADSM, the shared-read is based on a cache-based shared virtual memory system. As for the shared-write, instructions for consistency management are inserted after the corresponding store instruction. Therefore, various optimizations can be performed. We propose an optimizing method of reducing overheads for consistency management. The algorithm coalesces a sequence of consistency management instructions statically/dynamically. We have implemented the prototype of the compiler and the runtime system for the ADSM on a multicomputer Fujitsu AP1000+ and the general-purpose massively-parallel operating system: SSS–CORE. The performance evaluation using LU-Contig of SPLASH-2 shows that the execution time is reduced by 80% using static optimization and it is further reduced by 30% using dynamic optimization.

1. はじめに

近年，分散メモリ型計算機が重要な計算資源になりつつある．しかしながら，分散環境におけるメッセージパッシングを使用したプログラミングには多大な労力が必要である．共有データ位置の静的な予測が困難であり，通信をおこなう際に明示的にメッセージパッシングライブラリを使用する必要がある．また，共有データのコンシステンシの維持にも常に注意を払わなければいけない．

上記の労力を軽減するために，分散メモリ型メモリ計算機上で共有メモリ機構を提供する必要が生じる．共有メモリ機構では共有データは単一のアドレス空間でアクセスされるから，データの位置やコンシステンシの維持に関してユーザが気を配る必要がない．更に性能の観点からもメッセージパッシングより広義共有メモリ（遠隔メモリ操作を可変粒度高機能メモリ操作としてユーザに見せる）の方が定性的に優れている[13)]．

ユーザ／コンパイラにとっては共有メモリ機構がハードウェアで実現されようがオペレーティングシステムで実現されようが，基本性能が同じならば問題はない．共有メモリ機構をハードウェアで実現するにはどうしても実装コストが問題になる．近年，実装コストの低いソフトウェアで共有メモリ機構を実現する方式（ソフトウェ

† 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
☆ 現在，日本アイ・ビー・エム（株）東京基礎研究所
Presently with IBM Research, Tokyo Resesrch Laboratory

ア DSM)[3),4),6)] が注目を集めている.これらは狭義共有メモリ(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)であり,ユーザレベルのアプリケーションが利用可能な最適化の可能性が制限されてきた.

我々は,特殊な通信同期ハードウェアを仮定しない分散メモリ環境で効率の良い共有メモリ機構を提供するために,保護され仮想化された高速なユーザ通信/ユーザー同期を実現する非対称分散共有メモリ(**Asymmetric Distributed Shared Memory:ADSM**)を提案してきた[11)].ADSM 上では共有メモリの読み出しに関しては従来の仮想共有メモリ方式に従い,共有メモリの書き込みと同期はユーザコードに一連の命令シーケンスを静的に挿入する.そこで共有メモリの書き込みや同期を一連の命令シーケンスに変換する最適化コンパイラとそれをサポートするランタイムが必要になる.それらによって狭義共有メモリでは実行できなかった種々の最適化への道が開かれる.

本論では,ADSM 上で書き込みのオーバーヘッドを静的/動的に削減するためのコンパイル技法/ランタイムサポートを提案する.また,提案した技法の有効性を AP1000+ とワークステーションクラスタ上の汎用並列オペレーティングシステム SSS–CORE[11)] における実験によって評価する.

2. 非対称分散共有メモリ

ADSM では読み出しと書き込みの実現モデルが異なっている[☆]

- 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ングを行なって,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おこなう.

共有領域への書き込みをストアー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維持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を削減する.
- 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し,更新型メモリアクセスのオーバーヘッドを削減する.

3. プロトコル(SAURC)の実現

我々は ADSM 上に三種類のプロトコルの実装を行ない評価した[15)].ソフトウェアで Automatic update release consistency(AURC)[3)] をエミュレートするプロトコル “SAURC” が AP1000+ 上の実験においては最も良い結果を示した.また,SAURC は三者の中で実装が最も容易であり,実験の結果メモリ使用量も最も少ないプロトコルであることが判明した.この評価法に基づき,本論では SAURC プロトコルについて詳しく扱う.

SAURC では各ページに home ノードが存在する.home ノード以外にコピーページを有するプロセッサが一台の時は,互いに更新しあう(CopySet-2 プロトコル)が,それ以外の時はコピーから home ノードへの更新のみが伝えられる(CopySet-N プロトコル).これにより home ノードは常に最新の値に保たれる.home ノード以外のコピーページは WriteNotice[5)] を使用した Lazy Release Consistency(LRC)[4)] と同様な無効型プロトコルで管理される.

3.1 書き込みのコンシステンシ維持コード

バッファをノード台数個用意する.

- アドレスからページ番号(P)を求める.P への初めての書き込みの場合には WriteNotice を作成する.
- 自分が P の home ノードでない場合,「アドレスとサイズと中身」を P の home ノードに対応するバッファにコピーする(ただし,オーバーヘッド削減のため,過去に同じアドレスへの書き込みがあったかどうかは調べない).
- 自分が P の home ノードであり,かつ CopySet-2 である場合,アドレスとサイズと中身を,コピーを

☆ これが名前の由来である.

☆☆ loop transformation の一つである,loop coalescing(loop tiling の逆操作)が名前の由来である.

持つノードに対応するバッファにコピーする（ただし，オーバーヘッド削減のため，過去に同じアドレスへの書き込みがあったかどうかは調べない）．

書き込みの都度バッファの中身を転送しないで，バッファが一杯になった時点/同期点に到達した時点でバッファの中身をそのまま対応するノードに転送する．つまりランタイムレベルでメッセージのコンバイニングをおこなう．同期点に到達した場合はコンシステンシ維持完了の確認のためにメモリバリアを張る．対応するノードは転送されたデータを元に自分のページを更新し，TimeStampを更新する．更新後に転送されたデータは棄却する．

3.2 アクセスミス

ページフォールトが発生した場合，該当ページのhomeノードにページ要求のリクエストを発行する．該当ページのhomeノードはその要求を受け取ると，ページの中身とTimeStampをページフォールトを起こしたノードに転送する．

3.3 同 期

各ロックにはround-robinで割り振られたロック管理者が存在する．自分がロックを持っていない限り，全てのロックacquireのメッセージはロック管理者に転送し，管理者の元で逐次化されて管理される．ロック管理者はロックacquireのメッセージを受け取ると，そのメッセージを“最後にロックacquireのメッセージを発行したノード”にフォワードする．ロックをaquireした時には，ロックをreleaseしたノードが生成したWriteNoticeの情報が伝えられる．それを元に対応するページを無効化する．ただし，TimeStampの比較を行なってTimeStampが古いページのみ無効化する[3]．

各ノードはバリアに入ったら，バリア管理者に欠けているWriteNoticeの情報をバリア管理者に転送する．バリア管理者は全てのノードから情報を集めて対応するページの無効化をおこなう．然る後に，各ノードに欠けているWriteNoticeの情報を各ノードに転送する．各ノードは転送されたWriteNoticeの情報を元に該当するページの無効化をおこなう．ただし，TimeStampの比較を行ないTimeStampが古いページのみ無効化する[3]．

4. コンパイラアルゴリズム

我々のコンパイラが扱うのは，緩和されたコンシステンモデルに基づく共有メモリにおける明示的な並列プログラムである．共有メモリ上の明示的な並列プログラムでは，プログラマがアプリケーション依存の情報を使って動的な負荷分散をおこなうといった利点がある．具体的な最適化の情報はプログラム中の同期命令，コンシステンシ維持のプロトコル，及び共有アドレスの使用方法それぞれに，分散して含まれていることになる．どのくらいコンパイラに分かり易い形でプログラムが書かれているかによって，可能な最適化の範囲やコンパイルにかかる時間が影響を受ける．

我々のコンパイラはSPLASH-2[9]で使用されている，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．このプログラミングモデルでは共有領域の動的な確保やスレッドの生成や明示的な同期を実行できる．

4.1 共有領域への書き込みの検出

コンパイラが共有領域への書き込みをコンシステンシ維持コードを含んだコード列に変換する．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]の技法を用いる．手続き間ポインタ解析は解析時間が長くなるものの次のようなメリットがある．

- 手続き間のalias情報を元に手続き間でコード移動をおこなうことができる．
- 共有書き込みの候補者の正確な情報が得られるから，後のデータフロー解析のコストが減少する．

手続き間ポインタ解析はlocation setと呼ばれるデータ構造を用いた前進型データフロー問題を解く．ポインタの代入があればpoints-to情報（ポインタの指す，指される関係☆）が生成される．ポインタがdereferenceされた時点で，コントロールフローグラフを遡ってpoints-to情報が初めて計算される．手続き呼び出しの都度，同じ計算をするのを防ぐために，パラメータ間の別名関係が異なる時のみ，再計算をおこなう．パラメータ間の別名関係が同じならば☆☆，既存の結果を適用するだけでいい．

手続き間ポインタ解析を行なって，G_MALLOCオペレーション（共有領域の動的な確保）が生成するポインタの値を保持しうる変数検出し，該当する書き込みの後にコンシステンシ維持コードを挿入する．

5. 共有領域への書き込みのオーバーヘッドの削減

連続した共有領域に対する書き込みをできるだけまとめて処理して，冗長なコンシステンシ維持コードを除去することができれば，共有領域への書き込みのオーバーヘッドが削減できる．先行するポインタ解析により，全ての共有書き込みは検知されているものとする．

☆ location setで表現される

☆☆ 値が同じである必要はない

5.1 コンシステンシ維持コードの静的なコアレシング

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同期コードを間に含まない場合，コンシステンシ維持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維持コードと等価である．

以下に誘導変数の情報を利用してアフィンなメモリアクセスの検知を行ない，ループレベルのコンシステンシ維持コードのコアレシングをおこなうアルゴリズムを示す．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 $I(A,S)$ で表す．まず，必ず実行される共有書き込みを含む 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．

(0) 各共有書き込み毎にコンシステンシ維持コードを生成
(1) $Depth := n$
(2) $Depth = 0$ ならば終了
(3) $depth = Depth$ を満たす各ループ (L) に対して以下を実行する
(3.1) 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コアレシングする
(3.2) 同期コード／リターンコードがあれば (3) へ
(3.3) $I(A,S)$ で表される各コンシステンシ維持コードに対して
- 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
- 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，コアレシング可能であり，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）

(4) $Depth := Depth - 1$ として (2) へ

以下にコアレシングのアルゴリズムを適用した例を挙げる．

```
for (i = 0; i < n; i = i + 1) {
    a[i] = a[i] + alpha * b[i];
    I (&a[i], sizeof (double));
}
```

$\Downarrow$ Static Coalescing Optimization

```
for (i = 0; i < n; i = i + 1) {
    a[i] = a[i] + alpha * b[i];
}
I (&a[0], n * sizeof (double));
```

5.2 コンシステンシ維持コードの動的なコアレシング

一連の共有領域への書き込みが以前に書きこんだ領域を上書きしてしまうが，静的には検知できない場合が存在する．また，静的には連続領域に書き込みを行なっていると解析できないが，連続領域に書き込みを行なっている場合も存在する．この両者の場合，動的なコンシステンシ維持コードのコアレシングによりオーバーヘッド削減が可能である．

3.1 で述べた方法によるコンシステンシ維持コード内の操作では，オーバーヘッドを考慮して以前に同じアドレスに書いたかどうかを調べずに，アドレスとサイズと中身をバッファに書き込んでいた．この方式は静的に全ての連続領域への書き込みを解析できるような場合にオーバーヘッドが一番少ない．しかし，静的な解析では検出できないような連続領域への書き込みが多い場合はこの方式はオーバーヘッドが大きくなる．静的な解析では検出できない連続領域への書き込みが多い場合でも SoftwareDirtyBit 方式[10] を利用することでオーバーヘッドを削減できる．

SoftwareDirtyBit 方式では，各共有アドレスがそのアドレスへの書き込みが行なわれたかどうかを反映する bit（SoftwareDirtyBit）を持っている．コンシステンシ維持コード内で該当アドレスとサイズと中身をバッファに書き込むかわりに，該当アドレスに対応する SoftwareDirtyBit の領域を算出してサイズ分の SoftwareDirtyBit をセットする（図 **1**）．

3.1 で述べた方法では対応するバッファに該当アドレスとサイズと中身を書き込んでしまうため，バッファが一杯になると対応するノードへの通信が発生する，つまり update が生じる．共有領域への書き込みが以前に書きこんだ領域を上書きしてしまうような場合が何度も繰り返されると，無駄なメッセージが飛び，無駄な update が発生する危険がある．SoftwareDirtyBit 方式では，SoftwareDirtyBit をセットするローカルな計算のオーバーヘッドが課せられるだけで済む．

同期点に到達した時に，SoftwareDirtyBit の領域を

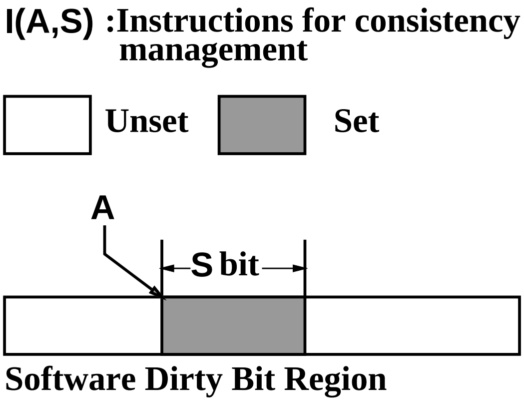


図 1 ソフトウェア・ダーティー・ビット
Fig. 1 Software Dirty Bit

探索することで自分が修正した部分を検出する．ただし，この部分は 3.1 で述べた実装には存在しないオーバーヘッドである．つまり 3.1 の実装ではバッファの中身がそのままメッセージのパケットになっていたが，SoftwareDirtyBit 方式ではメッセージのパケットを作成しなければいけない．しかしながら，この時点では自分が書き込んだ共有領域が最大限にコアレシングされて検出される．パケットを生成したら対応するノードに転送をおこなう．すなわち，SoftwareDirtyBit 方式では update は可能な限り lazy に行なわれる．

6. 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+ と NOW 版の SSS–CORE[16] に実装してきた．本実験ではプロトコルは SAURC を使用し，アプリケーションとして SPLASH-2[9] の中の Kernel の LU-Contig を採用し，データの home ノードの配置の最適化を行なった．バックエンドコンパイラとして gcc-2.7.2 を使用し最適化のオプションは “-O2” とした．

6.1 コンパイル時間

我々のコンパイラは，PARMACS のマクロを加えた C 言語によって書かれた，明示的な並列プログラムを入力とする．まず型チェック（型検査）を行ない，コントロールフローグラフを生成して（CFG），手続き間ポインタ解析を行ない (別名解析)，手続き内コアレシング（コアレシング）をおこなう．コンパイル時間の結果を表表 **1** に示す．入力プログラムはいずれも SPLASH-2 の kernel ベンチマークである．コンパイルは Sun SparcStation20 でおこなった．実際のプログラムに対しても，手続き間ポインタ解析の時間が実用に耐え得る時間内におさまっていることがわかる．

表 1 コンパイル時間（単位 秒）
Table 1 Results of compilation time (in seconds)

プログラム	行数	型検査	CFG	別名解析	コアレシング
LU	983	0.78	0.31	4.24	0.19
Radix	788	0.53	0.19	2.62	0.13
FFT	1,007	0.48	0.19	0.94	0.13

6.2 AP1000+

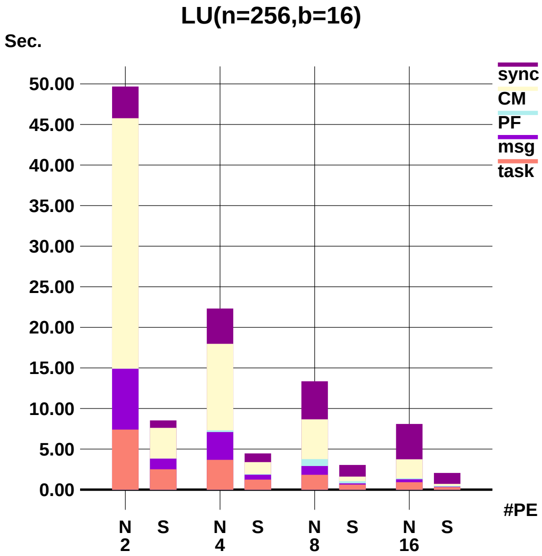
各ノードは 50MHz SuperSPARC（二次キャッシュなし, 16MBytes メモリサイズ）でネットワークは 2 次元トーラスでバンド幅は各リンク毎に 25MBytes/sec である．AP1000+ の OS はユーザレベルの割り込みハンドラをサポートしていないため，仮想記憶機構を利用できない．今回の実装では共有ページへのアクセスの前にページの有効性をチェックするコードを挿入した．もし有効でないページにアクセスしたら，ユーザーレベルのページフォールトハンドラを呼ぶ．外部からのメッセージに割り込みで反応することができないから，リモートノードからのメッセージは polling で処理する．コンパイラがループのバックエッジと関数呼びだしの所に polling のコードを挿入した．

図 **2** は静的なコアレシングの効果を調べたものである．**S** はコアレシングしたもの，**N** はしないものを示す．問題のサイズは 256×256 行列でブロックサイズが 16 である．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．静的なコアレシングにより，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．コアレシング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維持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．

6.3 SSS–CORE

ワークステーションクラスタ上の汎用超並列オペレーティングシステム SSS–CORE で実験を行なった．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（**MBCF**:Memory Based Communication Facilities）[11),12),14)] を実行し，ピークバンド幅は 11.2MBytes/sec である．メモリベースシグナルを使用してリモートからのメッセージに対処する．現時点ではページフォールトは AP1000+ の実装と同様にソフトウェアで検出している．

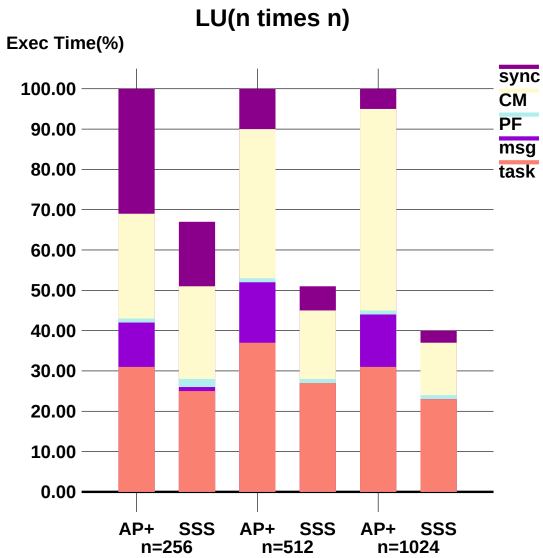
図 **6.3** は AP1000+ との比較を行なった結果であ



sync	バリアなど同期処理にかかる時間
CM	コンシステンシ維持コードの時間
PF	ページフォールト処理にかかる時間
msg	task 実行中にメッセージを処理した時間
task	アプリケーション本来の計算時間

N	コアレシングしない
S	手続き内で静的なコアレシング

図 2 静的なコアレシングの効果 (AP1000+)
Fig. 2 Effects of static coalescing (AP1000+)



sync	ロックなど同期処理にかかる時間
CM	コンシステンシ維持コードの時間
PF	ページフォールト処理にかかる時間
msg	task 実行中にメッセージを処理した時間
task	アプリケーション本来の計算時間

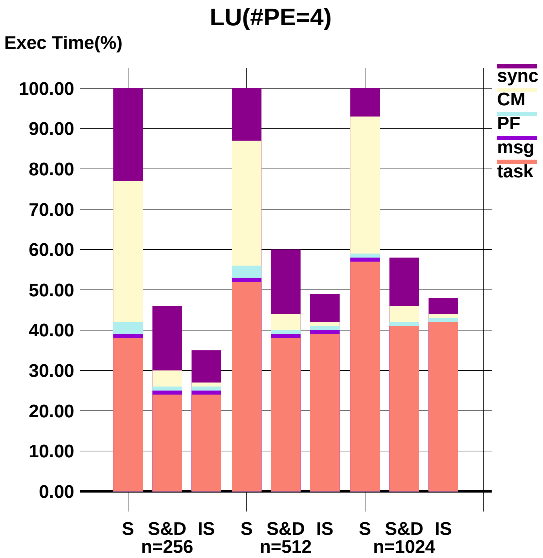
AP+	AP1000+ での実験データ
SSS	SSS-CORE での実験データ

図 3 SSS–CORE と AP1000+ の比較 (#PE=4)
Fig. 3 Comparison between results of SSS–CORE and those of AP1000+(#PE=4)

る．プロセッサ台数は 4 台で行列の一辺のサイズを 256，512，1024 と変化させた．縦軸は実行時間で AP1000+ の時間で正規化している．SAURC は通信量の多いプロトコルであるからネットワークが高速な AP1000+ に有利であるが，SSS–CORE のメモリベース通信機能が十分性能を発揮していることからプロセッサのクロック比に近い結果が得られる．問題サイズを大きくすると二次キャッシュのついている SSS–CORE の方がデータの局所性から更に高速になる．

図 4は動的なコアレシングの効果を調べたものである．最適化の内容は同一ページの表を参照されたい☆．プロセッサ台数は 4 台で，行列の一辺のサイズを 256，512，1024 と変化させた．縦軸は実行時間で静的にコアレシングしたもので正規化している．動的にコアレシングを実行することで実行時間の削減に成功している．動的にコアレシングを実行することでコンシステンシ維持コードの時間 (CM) は減少する．ただし，バリアに到着した時に自分が修正した部分を検出するオーバーヘッドが加算されて同期処理の時間 (sync) は増大している．SSS–CORE ではメモリベース通信で kernel が update をしてくれる．その時間は task にも含まれているため，update の量が減少したことにより task の時間も減少する．現実装での静的なコアレシングは手続き内のループに限られている．LU は手続きの呼び出しがネストしていて，手続き間で更にコアレシング可能な場合が非常に多いため，動的なコアレシングが効果的に作用する．表 2は各ノードが転送するメッセージの数量とコンシステンシ維持コードの実行回数の平均である．静的なコアレシングをおこなうことでコンシステンシ維持コードの実行回数は大幅に減少する．更に動的なコアレシングをおこなうことにより，手続き間で静的にコアレシングをおこなうのと同程度にデータ転送量を削減できる．

☆ IS のみ人手でコード生成を行なった



sync	バリアなど同期処理にかかる時間
CM	コンシステンシ維持コードの時間
PF	ページフォールト処理にかかる時間
msg	task 実行中にメッセージを処理した時間
task	アプリケーション本来の計算時間

S	手続き内で静的なコアレシング
D	動的なコアレシング
IS	手続き間で静的なコアレシング

図 4　動的なコアレシングの効果（SSS–CORE）
Fig. 4　Effects of dynamic coalescing (SSS–CORE)

表 2　LU 分解（n=512,b=16）における最適化の効果
Table 2　Effects of optimization methods on LU-Contig (n=512,b=16)

最適化	メッセージ数	転送量（Mbytes）	#CM
N	189035	212.07	11184770
S	88694	99.0	744928
S&D	2350	6.45	744928
IS	3550	6.38	2830

#CM: コンシステンシ維持コード実行回数

7. 関 連 研 究

7.1　OS ベースのソフトウェア DSM

従来のソフトウェア 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た．ここが ADSM とは大きく異なる点である．LRC では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部を調べなければいけない．このオーバーヘッドは無視できない．AURC[3] では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 ノードに転送し，home ノードを常に最新の状態に保つことで，diff 方式が抱える問題を解決している．

7.2　コンパイラベースの DSM

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☆に変換するだけである[10]．プロトコルの切替えやコンシステンシコード列のコアレシングといった最適化は実行していない．

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コアレシングは行なっていない．

8. ま と め

我々は 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た: 1) 誘導変数の情報を利用してアフィンなメモリアクセスの検知を行なってコンシステンシ維持コードを静的にコアレシングする，2) 静的にコアレシングできない場合でも SoftwareDirtyBit 方式を使用してランタイムで動的にコアレシングする．

我々は AP1000+ と SSS–CORE 上にコンパイラとランタイムシステムのプロトタイプを作成した．プロトコルとして SAURC を選択し，プロトタイプ上で SPLASH-2 の LU 分解を走らせて，この技法が有効であることを確認した．

ただし AP1000+ のような，ネットワークがプロセッサに対して比較的高速で書き込みのコストが高い環境では動的なコアレシングが必ずしも効果を発揮するという

☆ 該当アドレスの Software Dirty Bit をセットする

保証はない．どのような環境においても一番効果的な最適化は静的なコアレシングを手続き間でおこなうことである．手続き間ポインタ解析の雛型を使用すれば低コストで実装することが可能であり，現在実装中である．

更に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を作成して，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おこなう予定である．

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) (Feb. 1993).
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).
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 (Feb. 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).
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 (Jan. 1994).
6) Li, K. IVY: A Shared Virtual Memory System for Parallel Computing, Proc. of the 1988 ICPP (Aug. 1988).
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 (Oct. 1996).
8) Wilson, R. P. and Lam, M. S. Efficient Context-Sensitive Pointer Analysis for C Programs, Proc. of '95 Conf. on PLDI (June 1995).
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).
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 (Nov. 1994).
11) 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 (Aug. 1996).
12) 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 (Nov. 1996).
13) 松本 尚, 平木 敬 共有メモリ vs. メッセージパッシング, 情報処理学会研究報告, 第 97-ARC-126 巻 (Oct. 1997).
14) 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 (June 1997).
15) 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 (Aug. 1997).
16) 平木 敬, 他 汎用超並列オペレーティングシステムカーネル SSS-CORE の研究: 成果報告書, 平成 6 年度情報処理信仰事業協会、独創的情報技術育成事業報告書 (1995).

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．

(平成 9 年11月7日受付)

(平成10年 4 月7日採録)

丹羽　純平

1972 年生. 1997 年東京大学大学院理学系研究科情報科学専攻修士課程終了. 同年 4 月から同大学学院理学系研究科情報科学専攻博士課程に在籍. 並列化／最適化コンパイラに関する研究に従事．他に並列計算機アーキテクチャ，並列分散オペレーティングシステムに興味を持つ．

稲垣　達氏

1970 年生. 1995 年東京大学大学院理学系研究科情報科学専攻修士課程終了. 1998 年同博士課程終了. 同年 4 月から日本アイ・ビー・エム（株）東京基礎研究所に入社．並列化／最適化コンパイラに関する研究に従事. 他に並列計算機アーキテクチャ，並列分散オペレーティングシステムに興味を持つ．

松本　　尚（正会員）

1962 年生．1985 年東京大学工学部計数工学科卒業．1987 年大阪市立大学大学院理学研究科物理学専攻修士課程修了．日本アイ・ビー・エム（株）東京基礎研究所研究員を経て，1991 年 11 月より東京大学大学院理学系研究科情報科学専攻助手．並列計算機アーキテクチャ，並列分散オペレーティングシステム，最適化コンパイラに関する研究に従事．他に数値計算による制約解消系，グラフィックス，ニューラルネットワーク等に興味を持つ．電子情報通信学会，日本ソフトウェア科学会，ACM 各会員．

平木　　敬（正会員）

1976 年東京大学理学部物理学科卒業．1982 年同大学大学院理学系研究科物理学専攻博士課程修了．理学博士．1982 年通商産業省工業技術院電子技術総合研究所入所．1988 年より 2 年間 IBM 社 T．J．Watson 研究センタ客員研究員．1990 年より東京大学理学部情報科学科（現在大学院理学系研究科情報科学専攻）に勤務．現在，超並列アーキテクチャ，超並列超分散計算，並列オペレーティングシステム，ネットワークアーキテクチャなどの高速計算システムの研究に従事．日本ソフトウェア科学会会員