非対称分散共有メモリ上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†

我々は更新型共有メモリアクセスを含む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 は読み出しに関しては従来の共有仮想メモリ方式と同じだが、書き込みに関してはコンシステンシ維持コードを埋め込む。

従って、様々な最適化が可能になる。コンパイラがプロトコルに応じてコンシステンシ維持コードを生成することで、プロトコル切替えを行なうことができる。コンシステンシ維持コードの数を削減することにより、書き込みのオーバーヘッドを削減できる。

我々は ADSM 用のコンパイラとランタイムシステムを AP1000+ 上に作成し、実験によりコンパイラの最適化の有効性を評価する。

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 provides high-speed-update distributed memory access by software. As for the read, the ADSM prepares the page-size cache and exploits the virtual shared memory mechanism. As for the write, the ADSM executes a sequence of instructions for managing consistency after the corresponding store instruction.

On the ADSM, various optimizations can be performed. The compiler can generate the instructions for managing consistency according to the consistency protocol we want to implement. The compiler reduces the overhead of the shared write by reducing the number of instructions for managing consistency.

We have implemented prototypes of the compiler and the runtime system for the ADSM on a multicomputer Fujitsu AP1000+. We evaluate the effectiveness of the ADSM scheme by experiments.

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、分散環境でメッセージパッシングを使用してプログラミングを行なうのは多大な労力を必要とする。

そこで上記の労力を軽減するために、共有メモリモデルを提供する必要が生じる。共有メモリモデル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プログラマが気を配る必要はない。

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モデルを提供するために、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 (**Asymmetric Distributed Shared Memory:ADSM**) を提案してきた [12]。ADSM では共有メモリの読み出しに関しては従来の共有仮想メモリ方式に従い、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。従って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が必須になる。

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。

2. 非対称分散共有メモリ

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

† 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

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。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、 diff 方式の問題を解決するために、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 (ADSM) を提案してきた。 ADSM では読み出しと書き込みの実現モデルが異なっていて、これが名前の由来である。

- 共有領域への読みだし
  従来のソフトウェア DSM と同様にページ単位のキャッシュを用意して、プロセッサのロード命令で行なう。ページトラップでキャッシュミスを検知して対処する。
- 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、コンシステンシ維持のためのコード列 (遠隔メモリアクセスや付加的なメモリ操作) を実行コードとして埋め込む。

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え
  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え [11] をサポートする。
- コンシステンシ維持コード列のコンバイニング
  コンシステンシ維持コードに必要なのはストアアドレスとサイズだけである。従って全ての共有書き込みを一連のコード列に変換する必要はない。例えば、ある同期区間において連続した共有アドレスへの一連の書き込みがあるとする。一連の書き込みは、コンシステンシ維持コードの情報としては一個の大きな書き込みがある場合と等価である。
  従って各書き込みはただのストア命令に変換して、最後の書き込みのみ、ストア命令の後に一連の書き込みの結果を反映するコンシステンシ維持コードを埋め込むといった最適化を行なうことが可能になる。

3. プロトコルの実現

我々は ADSM 上で 3 種類のプロトコルの実装を行なった。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。

(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。 Home-Based LRC(HLRC) [9] という diff を使用した実現モデルが既に存在する。

・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。

4. 共有領域への書き込みのオーバーヘッドの削減

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。共有領域への書き込みのオーバーヘッドを削減するために、コンシステンシ維持コードの数を減らす最適化を行なう。

4.1 possibly-shared な書き込みの数の削減

ポインタを介しての書き込みがあると、それは共有領域に対してなされる可能性がある。従ってポインタを介しての書き込みに対しては、"動的にアドレスを調べてそこが共有領域の時にのみコンシステンス維持コードを実行するコード"を埋め込む必要がある。全てのポインタを介しての書き込みにこのコードを埋め込むことは高オーバーヘッドにつながる。

我々は上記のオーバーヘッドを削減するために、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ことにより、ポインタが指し得る値を計算する [2], [7](ポインタ解析)。コンパイラはポインタ解析の情報を使用して以下のコード生成を行なう。

(1) *must-shared*

ポインタが常に共有領域☆を指している時の書き込みを *must-shared* と呼ぶ。必ずストアの後にコンシステンシ維持コードを埋め込む。

(2) *local*

ポインタが常に局所領域☆☆を指している時の書き込みを *local* と呼ぶ。コンパイラは何もしない。

(3) *possibly-shared*

ポインタが局所領域も共有領域も指している時の書き込みを *possibly-shared* と呼ぶ。"動的にアドレスを調べてそこが共有領域の時にのみコンシステンス維持コードを実行するコード"を埋め込む。

4.2 コンシステンシ維持コードのコンバイニング

多くのアプリケーションではループ等において連続した共有アドレスへのストアを行なう場合があり、更なる最適化が可能である。

RC モデルでは共有領域への書き込みは、プログラムが同期点に到達するまで他のプロセッサに伝達する必要がない。従って、必ずしも共有領域への書き込みの都度、コンシステンシ維持コードを埋め込む必要はない。一連の *must-shared* な書き込みが連続したアドレスに対して行なわれていて、一連の書き込みの間に同期コード／リターンコードが入っていない場合には、一連の書き込みはコンシステンシ維持コードの情報として、一個の大きな書き込みがある場合と等価になる。従って、コンシステンシ維持コードをコンバイニングして一個のコンシステンシ維持コードにまとめることができる。

コンシステンシ維持コードのコンバイニング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 *must-shared* な書き込みを含む 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) コンバイニングを行なわずにコードを生成する。

(1) $Depth := n$。

(2) If $Depth = 0$ THEN 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。

(3.0) 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いならば、コンシステンシ維持コードをコンバイニングする。

例を挙げて説明すると、二個のコンシステンシ維持コード $I1 = I(A, S)$,$I2 = I(A', S')(A < A')$ が存在した時、もし $A' < A + S$ なら $I1$ と $I2$ は一個のコンシステンシ維持コード

☆ 共有変数や共有ヒープ領域

☆☆ 局所変数や局所ヒープ領域

$I(A, A' - A + S')$ にコンバイニングする。

(3.1) 同期コード / リターンコードがあれば (3) へ。

(3.2) $I(A, S)$ で表される各コンシステンシ維持コードに対して

(a) 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。

(b) 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、コンバイニング可能であり、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) $Depth := Depth - 1$ (3) へ。

上記のアルゴリズムを例を使って説明する。以下のループネストは SPLASH-2 [8] の LU のコードである。

```
for (j = 0; j < n; j = j + 1) {
  for (i = 0; i < n; i = i + 1) {
    if (n - i <= edge) {
      ibs = edge;
      ibs = n - edge;
      skip = edge;
    } else {
      ibs = block_size;
      skip = block_size;
    }
    if (n - j <= edge) {
      jbs = edge;
      jbs = n - edge;
    } else
      jbs = block_size;
    ii = i / block_size
         + (j / block_size) * nblocks;
    jj = i % ibs + (j % jbs) * skip;
    rhs[i] = rhs[i] + a[ii][jj];
  }
}
```

`double` の配列`rhs` は共有領域にあり、プロトコルは`LRC` で`CMI` はコンシステンシ維持コードとする。上述のアルゴリズムを適用すると

- まず以下のコードに変換される。

```
for (j = 0; j < n; j = j + 1) {
  for (i = 0; i < n; i = i + 1) {
    ......
    rhs[i] = rhs[i] + a[ii][jj];
    CMI(LRC,&rhs[i],sizeof(double));
  }
}
```

- $Depth := 2$
- $depth$=$Depth$(=2) となるループには、I(`&rhs[i]`,`sizeof(double)`) で表されるコンシステンシ維持コードしか存在しないので次のステップへ進む。
- A=`&rhs[i]`, S=`sizeof(double)` である。`&rhs[i]` は誘導変数であり、そのストライドは S に等しい。よってコンバイニング可能であり、コンシステンシ維持コードをこのループから除去してこのループの外側に挿入する。ループの回数は n で A の最小値は`&rhs[0]` だから、挿入されるコンシステンシ維持コードは I(`&rhs[0]`,`n*sizeof(double)`) で表される。従って以下のコードが生成される。

```
for (j = 0; j < n; j = j + 1) {
  for (i = 0; i < n; i = i + 1) {
    ......
    rhs[i] = rhs[i] + a[ii][jj];
  }
  CMI(LRC,&rhs[0],n*sizeof(double));
}
```

- $Depth := 1$
- $depth$=$Depth$(=1) を満たすループには I(`&rhs[0]`,`n*sizeof(double)`) で表されるコンシステンシ維持コードしか存在しないので次のステップへ進む。
- A=`&rhs[0]` で S=`n*sizeof(double)` である。`&rhs[0]` はループ不変である。よってコンバイニング可能になり、コンシステンシ維持コードをこのループから除去してこのループの外側に挿入する。従って以下のコードが生成される。

```
for (j = 0; j < n; j = j + 1) {
  for (i = 0; i < n; i = i + 1) {
    ......
    rhs[i] = rhs[i] + a[ii][jj];
  }
}
CMI(LRC,&rhs[0],n*sizeof(double));
```

- $Depth := 0$ → 終了。

この最適化によりコンシステンシ維持コードの実行回数は $\frac{1}{n^2}$ に削減される。

5. 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☆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。また、Midway ではプログラマがコードに`shared` か`private` か annotation を付けることで *possibly-shared* な書き込みの数を減らしている。

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+ に実装してきた。AP1000+

☆ 該当アドレスの Software dirty bit をセットする

の OS はユーザレベルのページフォールト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できる。

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する。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。

OS がユーザレベルのページフォールトハンドラをサポートしてないため、外部からのメッセージに割り込みで反応することもできない。従って、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。

本実験では 8 ノードを使用して、 SPLASH-2 [8] の 3 個の kernel(LU-Contig,Radix, FFT) を走らせた。

- LU-Contig(表 1) →疎行列のブロック化された LU 分解 (256×256 行列で、ブロックサイズは 16×16)
- Radix(表 2) →整数のラディックスソート (key の数が 262144 で radix が 1024)
- FFT(表 3) → 複素数の一次元 FFT(データ点数が 16384)

全ての場合においてポインタを介してのアクセスは *local* か *must-shared* に分類され、 *possibly-shared* は存在しなかった。

各表の見方であるが、 LRC(*) は LRC プロトコルを選択した時の結果、 SAURC(*) は SAURC プロトコルを選択した時の結果、 HYBRID(*) は HYBRID プロトコルを選択した結果である。 *(NO) は最適化 (コンシステンシ維持コードのコンバイニング) を行なわない時の結果であり、 *(O) は最適化を行なった時の結果である。

- “task” →計算時間
- ”message” → polling によって捕まえたリモート処理するのに費やした時間と polling で捕まえたメッセージ数と総メッセージ数の比
- “PF” →ページフォールト時間とその回数
- ”CM” → コンシステンシ維持コードにかかった時間とコンシステンシ維持コードの数
- “sync” → lock/unlock や barrier といった同期処理にかかった時間
- ”GC” → GC に費やした時間とその回数

AP1000+ は二次キャッシュがなく、一次キャッシュが write through なので、プロセッサがキャッシュされたデータに対して行なった変更は全て外部に反映される。従ってストアのコストが比較的高い。”CM” ではストア命令が頻繁に行なわれるため、”CM” が比較的大きくなる。

各プロトコルに対して最適化を施した方が良い結果をもたらしていることが分かる。コンシステンシ維持コードの数が減ると、”CM” が減って、 *write notice* の量は減少する。”PF” や”sync” や”message” における多くの操作は *write notice* の量に依存しているので、コンシステンシ維持コードの数が減ると、”CM”、”PF”、”sync”、”message” は全て減少する。

AP1000+ 上の実装では、 SAURC プロトコルが一番いい結果を残している。 GC が必要なプロトコルは LRC のみで、”GC” が非常に大きいということが分かる。 LRC プロトコルは”GC” だけでなく”PF” も他のプロトコルより大きい。

7. 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た。ポインタ解析を利用して *possibly-shared* な書き込みの数を削減し、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る場合に、一連のコンシステンシ維持コードをコンバイニングする方法を提案した。

ADSM 用のコンパイラとランタイムシステムを AP1000+ 上に実装し、 SPLASH2 の 3 個の kernel を走らせて、この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 上に実装していく方針である。

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) (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).

unit : sec.	total	task	message	PF	CM	sync	GC
LRC(NO)	81.892455	1.868542	0.608391	2.267781	3.422607	3.169061	72.577036
			88/2720	104	699040		11
LRC(O)	11.945188	0.627203	0.760216	3.335205	0.239031	2.604353	4.379179
			178/790	232	49640		1
SAURC(NO)	13.556636	1.816375	1.136838	0.830893	4.916014	4.856516	0.000000
			262/437	210	699040		0
SAURC(O)	3.124313	0.618595	0.165564	0.287180	0.562971	1.490004	0.000000
			217/392	210	49640		0
HYBRID(NO)	27.473138	1.828389	4.027283	4.283247	5.994998	11.339222	0.000000
			258/440	204	699040		0
HYBRID(O)	4.114184	0.621423	0.391392	0.515071	0.712049	1.874249	0.000000
			212/356	210	49640		0

表1 LU-Contig(256)

unit : sec.	total	task	message	PF	CM	sync	GC
LRC(NO)	11.428428	0.604499	0.057003	1.069547	1.219771	1.895149	6.582458
			224/4253	283	212093		2
LRC(O)	9.849303	0.574050	0.038313	1.107061	1.143005	0.770975	6.215899
			216/4245	281	196635		2
SAURC(NO)	3.129691	0.581339	0.263697	0.313482	1.364119	0.607055	0.000000
			309/582	470	212093		0
SAURC(O)	2.980493	0.553462	0.298612	0.355711	1.348237	0.424471	0.000000
			294/588	472	196635		0
HYBRID(NO)	6.752322	0.646346	1.657397	0.838957	1.710627	1.898994	0.000000
			307/570	470	212093		0
HYBRID(O)	5.959178	0.605621	1.338247	0.747380	1.547363	1.720568	0.000000
			295/562	468	196635		0

表2 Radix(262,144)

unit : sec.	total	task	message	PF	CM	sync	GC
LRC(NO)	16.785423	0.488462	0.949015	4.071002	0.849016	0.198326	10.229601
			185/740	244	174080		2
LRC(O)	5.705468	0.254218	0.681280	2.559653	0.270242	0.090499	1.849577
			229/740	307	55248		1
SAURC(NO)	2.577256	0.469006	0.217816	0.121456	1.152603	0.616376	0.000000
			296/446	330	174080		0
SAURC(O)	1.210733	0.238359	0.091413	0.116116	0.370441	0.394403	0.000000
			291/443	331	55248		0
HYBRID(NO)	4.263769	0.478066	1.392421	0.167770	1.504440	0.721071	0.000000
			307/418	330	174080		0
HYBRID(O)	1.855387	0.247024	0.301032	0.223589	0.480333	0.603409	0.000000
			262/390	330	55248		0

表3 FFT(16,384)

7) WILSON, R. P. and LAM, M. S. Efficient Context-Sensitive Pointer Analysis for C Prgrams, Proc. of '95 Conf. on PLDI (June 1995).

8) 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).

9) YUANYUAN ZHOU, L. I. and LI, K. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems, Proc. of the 2nd Symp. on OSDI (1996).

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 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, 第 89-ARC-77 巻 (July 1989).

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, 第 96-OS-73 巻 (Aug. 1996).