2004－ARC－156　(2)
2004／2／2

コンパイラが支援するソフトウェアDSMにおける プリフェッチ機構

丹　羽　純　平†

ソフトウェア DSM は、汎用の分散並列環境において、実行時に共有アドレス空間を提供できるため、幅広いアプリケーションを扱うことが可能である。ソフトウェア DSM では、遠隔メモリアクセスのレイテンシの削減のために、遠隔ノードのデータを自ノードの局所メモリにキャッシュする。

本稿は、ソフトウェア DSM において、アプリケーションプログラムのソースを直接解析する最適化コンパイラの支援により、遠隔メモリアクセスのレイテンシを更に削減する手法を提案する。すなわち、遠隔メモリアクセスのレイテンシを削減するための、プリフェッチを行うコンパイル技法を提案し、それを可能にするインタフェイスを導入する。

上記のコンパイル技法を最適化コンパイラ（RCOP）に実装し、ギガビットイーサ接続された汎用の PC クラスタ上にランタイムを構築した。PC クラスタにおいて、SPLASH-2 ベンチマークを用いた実験により、本最適化の効果を検証した。

Prefetch Mechanism in Compiler-Assisted Software DSM System

JUMPEI NIWA†

Software Distributed Shared Memory (S-DSM) provides shared address space at run-time and accepts a wide range of applications on parallel computer systems with commodity hardware. Software DSM caches remote data in the local memory in order to reduce remote-memory-access latency.

This paper proposes the methods for reducing remote-memory-access in S-DSM by utilizing an optimizing compiler that directly analyzes explicitly parallel shared-memory source programs. That is to say, this paper suggests the compiling techniques of issuing prefetch for remote-memory access and introduces the interface that enables prefetch mechanism.

I have implemented this compiling technique in optimizing compiler, Remote Communication Optimizer :RCOP. I also have implemented the lightweight runtime systems on PC cluster connected with the Gigabit Ethernet (1000BASE-T). The experimental results using the SPLASH-2 benchmark suite show that the prefetch technique is effective.

1. はじめに

本研究の目的は、計算機クラスタのような、汎用のネットワークで接続され、固有の共有メモリ機構を持たない分散並列環境において、共有メモリモデルに基づいた書かれた明示的に並列なプログラム (明示的に並列な共有メモリプログラム) を効率良く実行することにある。そのためには、ソフトウェア分散共有メモリ（Software Distributed Shared Memory: S-DSM）[4),5)](図 1 参照) と呼ばれる、遠隔ノードのデータを自ノードのメモリにキャッシュする (ソフトウェアキャッシュ) 機構が求められる。

従来の S-DSM では、逐次コンパイラによるアプリケーションのコード生成が大前提になっている。すなわち、コヒーレンス管理操作（複数のノード間で共有データの内容に矛盾が生じないような制御）を必要とする共有メモリアクセスは局所メモリアクセスと同様にプロセッサの load ／ store で実現される。コヒーレンス管理操作を必要とする共有メモリアクセスはトラップによって検知され、コヒーレンス管理操作はトラップハンドラ内で OS により実行される。したがって、コヒーレンス管理操作はアプリケーションからは見えなくなり、アプリケーションからコヒーレンス管理操作に最適化をかけることができない。

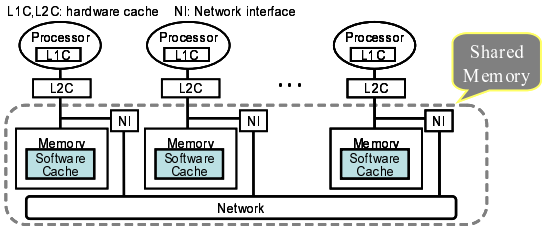


図 1　ソフトウェア DSM

初めて、遠隔ノードにあるデータにアクセスする場合には、キャッシュミスが発生し、データを持ってい

† 科学技術振興機構さきがけ研究 21「機能と構成」領域
PRESTO, Japan Science and Technology Agency

るノードにキャッシュブロックリクエストが転送される。従来の S-DSM では、実際にアクセスが行なわれる時になって初めて、トラップハンドラによって上記の操作が行われる。しかも、データが到着するまで、待つだけである。ソースを直接解析すれば、共有メモリアクセスを検知することが可能な訳だから、実メモリアクセスよりも、もっと前に非同期式リクエスト (プリフェッチ) を発行することが可能になり、通信と計算をオーバラップさせることができるのではないか？と考察した。

本稿では、ユーザレベル最適化を可能にするフレームワーク[6),12)] の元で、インタフェイスを改良し、最適化コンパイラが明示的に並列な共有メモリプログラムのソースを直接解析して、プリフェッチを行うコヒーレンス管理コードを自動的に挿入し、ランタイムが効率良くそれを実行する方式を提案並びに実装する。

まず、プリフェッチを可能にするインタフェイスについて述べ、次に、プリフェッチを行なうコンパイル技法について述べる。実験により、プリフェッチの効果を測定し、最後に関連研究とまとめを述べる。

2. プリフェッチを可能にするインタフェイス

まず、二つのコンパイラベースの S-DSM 機構を述べ、次にプリフェッチを可能にする改良点を述べる。そして、コヒーレンス管理コードについて述べる。

Asymmetric DSM[12)]

ユーザレベルの最適化を可能にするインターフェイスの一つが Asymmetric DSM（ADSM）である。

- ADSM では、既存の OS ベースのシステムと同様に（ソフトウェア）キャッシュミス/ヒット判定にページ管理機構で実現
- 書き込み時のコヒーレンス管理操作は、store 命令とは分離して、ユーザレベルのコードとして最適化コンパイラが自動的に挿入し、緩和されたメモリモデルを活用して最適化を適用
- コヒーレンス単位（ソフトウェアキャッシュのブロック）はページであるために、false sharing と不必要なデータ通信が問題

ADSM では、実アクセスまで、キャッシュミスヒット判定が起こなわれず、また、キャッシュブロックが到着するまで、トラップハンドラの中で、待つことになる。もちろん、その間は、他ノードのリクエストに対応したりはするが、自ノードの計算自体は進まない。

User-level DSM[6)]

もう一つのインターフェイスである User-level DSM（UDSM）では、ノード間の通信オーバヘッドを最小にするために、コヒーレンス単位（ソフトウェアキャッシュのブロック）をユーザ定義のセグメントにしている。

- キャッシュミス/ヒット判定並びに書き込み時のコヒーレンス管理操作は全てユーザレベルコード (コヒーレンス管理コード) で実現
- コヒーレンス管理コードは対応する load/store 命令とは分離され、ADSM と同様に、最適化コンパイラが自動的に挿入し、最適化を適用
- コヒーレンス管理コードのオーバヘッドをいかに押さえるかが鍵

UDSM では、実アクセスの前に、キャッシュミスヒット判定を行うことが可能である。従来の実装では、ADSM と同様に、データが到着するまで、自ノードの計算は進まなかった。つまり、ミスしたら、計算は、データが来るまでその場で待つという方針を取っていた。

というのは、もし、非同期式なリクエストを発行すると、データが到着したかどうかを判定するコードが必要になるため、コヒーレンス管理コードのオーバヘッドが増大する危険を有していたからである。

2.1 Hybrid DSM

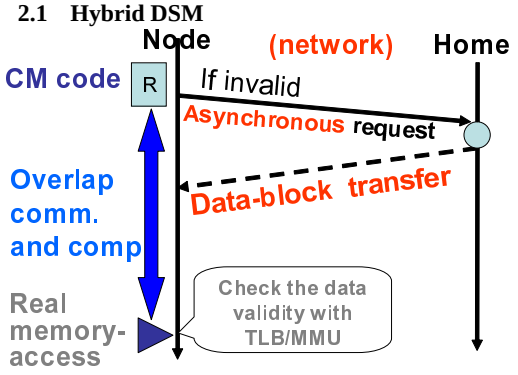


図 2 HDSM の共有読み出し時のランタイムの動作

ADSM と UDSM のハイブリッド方式:Hybrid DSM（HDSM）を用いると、プリフェッチが可能になる。すなわち、

- UDSM と同様に、キャッシュミス／ヒット判定並びに書き込み時のコヒーレンス管理操作を全てユーザレベルのコードで実現
  ADSM と同様に無効なキャッシュはアンマップ
- キャッシュミス時のリクエストは非同期式に転送
- キャッシュブロックが到着したかどうかは、ページ管理機構を利用してチェック

キャッシュミスしたページはアンマップしてあるので、ブロックの転送が実アクセスの前に間に合わない場合には、実アクセス時にトラップが発生してブロックの到着を待つ。もし、間に合った場合には、ハンドラが当該ブロックをマップして更新していることになるので、何事もなく実アクセスが行われ、計算は続いていく。

コヒーレンス管理コード

HDSM において、コヒーレンスを維持するため最適化コンパイラは 2 種類のユーザレベルのコヒーレンス管理コードを挿入し、それらはランタイムによって実

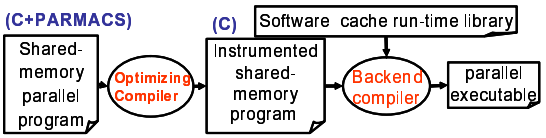


図 3 コード生成プロセス

行される。

- 読み出しの発行（read commitment）
  - `R(address, size)`
    読み出しのコヒーレンス管理操作を実行するコードであり、読み出しの前に挿入される。（ソフトウェア）キャッシュのミス／ヒット判定を実行し、ミス時にはハンドラを起動し、非同期式のリクエストをホームに転送する
- 書き込みの発行（write commitment）
  - `W(address, size)`
    書き込みのコヒーレンス管理操作を実行するコードであり、書き込みの後に挿入される。他ノードへの無効化情報にこのブロックを加えたり、ホームにデータを転送する等の処理を行なう

引数は、共有アクセスが行われる可能性のある開始アドレス（`address`）とデータサイズ（`size`）からなり、コンパイラが「連続した領域 $\{x \mid \texttt{address} \leq x < \texttt{address} + \texttt{size}\}$ に対してアクセスがある」ことをランタイムに通知する。ランタイムは実行時にこの領域が共有領域かどうかを確かめて、共有領域であった場合には上述のコヒーレンス管理操作を行う。

3. プリフェッチを可能にするコンパイル技法

まず、ベースとなる最適化コンパイラの概要について述べ、次に、プリフェッチを可能にする、read commitment を用いた最適化について述べる。

3.1 最適化コンパイラの概要

本稿で使用する、ソフトウェア分散共有メモリを支援する最適化コンパイラ Remote Comunication OPtimizer(RCOP)[8),13)] の概要を述べる。

RCOP の目的は、ソフトウェアキャッシュ管理に伴う通信と命令のオーバヘッドを削減するコードを生成することである。そのために、ソースコードを直接解析して、

- 共有アクセスを漏れなく、かつ、できるだけ正確に検知する。
- 緩和されたメモリモデルをの元で、ループ、手続き呼び出しといったプログラムの階層構造を活用し、複数のコヒーレンス管理コードの一括発行を行う。

ことにある。

RCOP のフレームワークについて以下に述べる。入力は、Lazy Release Consistency :LRC モデル[4)] に基づいて書かれた、明示的に並列な共有メモリプログラムで、API は PARMACS[1)] という並列化マクロで C を拡張したものである。最適化コンパイラは手続き間別名解析を行い、区間解析、冗長性削除の手法を活用し、読み出し (read commitment)、書き込み (write commitment) に関して、共有アクセス集合を使用して、fusion、coalescing、redundant index elimination によるループレベルの最適化を行い、サマリを手続き間で計算する。

3.2 Read commitment を用いた最適化

HDSM において、LRC モデルの元で、共有アクセス集合を使用して、プログラムの階層構造を活用し、複数のコヒーレンス管理コードをできる限り早く一括して発行する最適化について述べる。

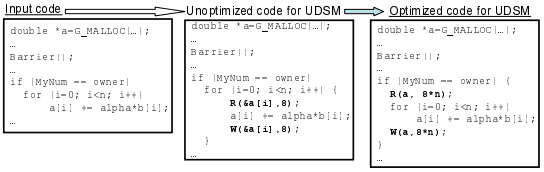


図 4 UDSM におけるコード生成例

まず、図 4 に示してある例を用いて UDSM 用のコードを生成する例を述べる。RCOP が UDSM 用のコードを生成する場合には共有アクセスを検知した時点で `R(&a[i], 8)` というチェックコードが挿入される。それは、共有アクセス集合を用いて、$S = (\&a[i], 8, \emptyset)$ と表記される。それを元に、一イテレーション分のサマリを求めて (それは S に等しい)、それから、ループのインデクス情報 (不等式制約) を付加して、ループ全体のサマリ S' を求めると、

$$S' = S \multimap \{0 \leq i < n\} = (\&a[i], 8, \{0 \leq i < n\})$$

となる。その際に、S' は Coalescing という最適化で

$$S' \rightarrow S'' = (\&a[i], 8 * n, \emptyset)$$

となる。そして、ループ全体のサマリ S'' は、更にコントロールフローグラフ (CFG) をさかのぼろうとする。しかし、条件式が偽になる時は読み出しが実行されないので、冗長性削除のデータフロー方程式[8),13)] にしたがって、条件分岐で分かれた直後に対応する read commitment : `R(&a[i], 8*n)` が挿入される。

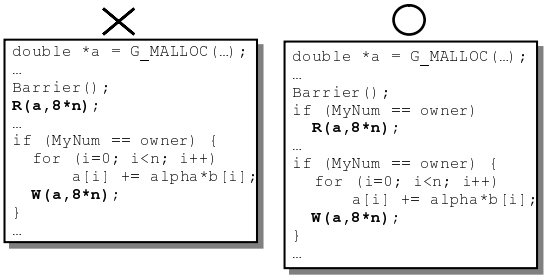


図 5 HDSM におけるコード生成例 (左：悪い例、右：良い例)

レイテンシ削減のために、図 5 の左側にあるように、制御を無視して更に上に移動する最適化も考えられる。しかし、そのような最適化を選択した場合には、制御式を無視しているため、制御が if 文の中を通らない場合には、無駄なチェック、特に、通信を誘発する危険がある。

本最適化では、上記のような危険を防いだ上で、レ

$$\mathrm{ANTOUT}(i) = \bigsqcup_{s \in \mathrm{succ}(i)} \mathrm{ANTIN}(s) \quad (1)$$

$$\mathrm{ANTIN}(i) = \mathrm{COMP}(i) \vee (\mathrm{TRANS}(i) \wedge \mathrm{ANTOUT}(i)) \quad (2)$$

$$\mathrm{AVIN}(i) = \bigsqcup_{p \in \mathrm{pred}(i)} \mathrm{AVOUT}(p) \quad (3)$$

$$\mathrm{AVOUT}(i) = (\mathrm{COMP}(i) \vee \mathrm{AVIN}(i)) \wedge \mathrm{TRANS}(i) \quad (4)$$

$$\mathrm{INSERT}(i) = \mathrm{ANTIN}(i) \wedge \neg\Big(\bigsqcup_{p \in \mathrm{pred}(i)} (\mathrm{TRANS}(p) \wedge \mathrm{ANTIN}(p)) \Big) \wedge \neg \mathrm{AVIN}(i) \quad (5)$$

($\mathrm{pred}(i)$ は文 i に先行 (precede) する文の集合であり、$\mathrm{succ}(i)$ は文 i に後継 (succeed) する文の集合である)

図 6 冗長な read commitment を削除するデータフロー方程式

イテンシ削減のために、read commitment を早く発行することを目標にする。すなわち、もし、if 文の条件式 (例の場合には、`(MyNum == owner)`) が副作用を持たない場合には、条件式を複製し、チェックコードに加えたコード、すなわち、条件付きチェックコード“`if (MyNum == owner) R(a, 8*n)`”を挿入して、更に上へと持ち上げる。その結果、図 5 の右側のようなコードが生成される。

本最適化を行うための、read commitment の冗長性削除を行うデータフロー方程式を求める (図 6)。各変数の意味を以下に簡潔に述べる。簡単のために、read commitment を一つ固定して（`R(a,s)`）、対応する共有アクセス集合 $S = (a, s, \emptyset)$ を考える。

UDSM におけるコード生成と同様に、別名解析の結果から、各文 i に対して、着目した領域 $\{x \mid \mathtt{a} \leq x < \mathtt{a} + \mathtt{s}\}$ に対して以下の論理定数を求める。

$\mathrm{COMP}(i)$　文 i が着目した連続共有領域に対して読み出し (当該共有読み出し) を行なう

$\mathrm{TRANS}(i)$　文 i を越えて read commitment `R(a,s)` を移動することはプログラムの意味を変更してしまう。

文 i が同期プリミティブである場合、または、read commitment のパラメータ（`a` もしくは `s`）を更新する場合に、$\mathrm{TRANS}(i)$ は偽となる。

上述の定数の値から、プログラム内の任意の点 p(CFG の任意のエッジ) に対して、以下の二種類のデータフロー変数を計算する。

Availability　p にいたる、“ある” パスにおいて、当該共有読み出しが発行される。ただし、あるパスを選択する条件式が副作用を持たない場合に限る。もし副作用を有する場合には、“ある” を “全て” に置きかえて考える。つまり、UDSM 上で冗長な読み出しを削除するデータフロー方程式の Availability と同様の意味を持つ。

Anticipatability　p からはじまる、“ある” パスにおいて、当該共有読み出しが発行される。ただし、あるパスを選択する条件式が副作用を持たない場合に限る。もし副作用を有する場合には、“ある” を “全て” に置きかえて考える。つまり、UDSM 上で冗長な読み出しを削除するデータフロー方程式の Availability と同様の意味を持つ。

$\mathrm{ANTIN}(i)$　文 i を実行する前の Anticipatability

$\mathrm{ANTOUT}(i)$　文 i を実行した後の Anticipatability

$\mathrm{AVIN}(i)$　文 i を実行する前の Availability

$\mathrm{AVOUT}(i)$　文 i を実行した後の Availability

$\bigsqcup$ は、パスの合流点における “条件付き和” を表す。すなわち、合流点に至る全てのパスにおいて、パスを選択する条件式が副作用を持たない場合には、あるパスにおいて当該共有読み出しが発行されている時に、条件式の情報を付加して真を返す。

逆に、あるパスにおいて、それを選択する条件式が副作用を持つ場合には、$\bigsqcup$ は “論理積” を表す。すなわち、条件を考慮せず、全てのパスにおいて当該共有読み出しが発行されている時に真を返す。

文 i を実行する前に read commitment を実際に挿入するかどうかを示す真偽値を取るデータフロー変数を $\mathrm{INSERT}(i)$ と表す。Read commitment は、$\mathrm{INSERT}(i)$ が真になる。すなわち、

- 当該共有読み出しが anticipatable であり、かつ
- 先行する文のどれかにおいて、
  - 当該共有読み出しが anticipatable ではない
  - 同期プリミティブが実行される☆

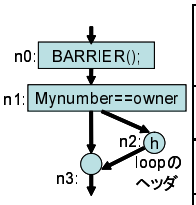
  のいずれかが成立し、かつ
- 当該共有読み出しが available でない

ような点 (中間語における文の境界) に挿入する。

手続き間でデータフロー方程式 (**図 6**) を計算する場合には、従来の RCOP と同様に、呼び出しグラフを主手続きから深さ優先で探索する。callee を解析するときには、call_site（手続き呼び出し）における caller の情報がマッピングされる。

実際には、各データフロー変数を真偽値ではなく、共有アクセス集合の集合 $\{S_1, S_2, \ldots, S_n\}$ として表す。概念的には、個々の要素（共有アクセス集合）S_i について図 6 の演算を行って得た真理値の集合を求めることになる。データフロー方程式（図 6）の論理演算は全て集合演算になる。

☆ 共有変数を引数に取る同期プリミティブが存在するから



Node No. → Property ↓	n0	n1	n2	n3
ANTIN	$\emptyset$	$\{S_2\}$	$\{S_1\}$	$\emptyset$
ANTOUT	$\{S_2\}$	$\{S_2\}$	$\emptyset$	$\emptyset$
AVIN	$\emptyset$	$\emptyset$	$\emptyset$	$\{S_2\}$
AVOUT	$\emptyset$	$\emptyset$	$\{S_1\}$	$\{S_2\}$
INSERT	$\emptyset$	$\{S_2\}$	$\emptyset$	$\emptyset$

図 7 図 4 のデータフロー方程式の解

図 4 の例を実際にこの方程式で解く過程を示したのが、**図 7** である。図の左が対応する CFG であり、右が共有アクセス集合の集合である。$S_1 = (a, 8*n, \emptyset)$、$S_2 = (a, 8*n, MyNum == owner)$ とする。条件節内の区間 (ループ) に関しては解析済みであるとする。CFG の下流から上流に向かって、ANTOUT, ANTIN を求め、CFG の上流から下流に向かって、AVIN, AVOUT を求め、最後に INSERT を求める。

4. 実　　験

本稿で提案した、プリフェッチを可能にする最適化方式を RCOP に実装し、ランタイムを FreeBSD/Linux に移植して、プリフェッチを行えるように改良した。実験を行い、その有効性について検証する。

実験環境を以下に挙げる。

- コンパイラ: RCOP + gcc3.3
  (最適化オプションは “-O3”)
- ノード: Dell PowerEdge 1650 × 8
  (1.26 GHz PentiumIII, 512KB キャッシュ、2GB メモリ)
- OS: FreeBSD 5.1-RELEASE
- ネットワーク：
  NIC は Intel 1000XT で、1000BASE-T のスイッチングハブ (BUFFALO の LSW-GT-8W) で接続
- ランタイム
  SSS-CORE 上のもの[8),13)] を {Linux, FreeBSD}/IA32 に移植。ポータビリティ性を考慮して、通信は TCP/IP プロトコルを使用
- ベンチマーク
  SPLASH-2[11)] から以下の 3 個のアプリケーション (LU-Contig (n=4096), Water-NSquared (n=32768), Raytrace(balls4)) を選択し、文献[9)] を元に、実アプリケーションであるツリーコード[10)] を並列化して、それを使用 (BarnesSpatial(n=32K))

紙面の都合上、8 台における典型的な実行時間のブレイクダウンを 2 つだけ挙げる。

図 8 が ADSM/UDSM/HDSM における LU-Contig の実行時間 (8 台) のブレイクダウンである。“Task” は本来の計算時間を表す、ただし、UDSM と HDSM では read commitment の時間も含まれる。“Msg” はメッセージハンドリングの時間を表す。“C-Miss” はキャッシュミス時の待ち時間を表す。“W” は write commitment の時間を表す。“Sync” は同期時間を表す。三者間で Task の時間にほどんど差がないことから、コンパイラの最適化によって、read commitment の発行のオーバヘッドが低く押さえられていることが分かる。write commitment に関しても同様である。

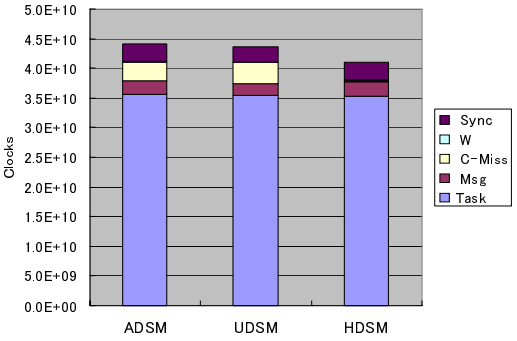


図 8 LU-Contig の実行時間 (8 台) のブレイクダウン

HDSM ではプリフェッチの発行により、Msg の時間が多少増大するものの、C-Miss 時間が大幅に削減していることが分かる。プリフェッチリクエストの平均回数は 10374 回であり、その内、データ転送が実アクセス前に間に合わなかったものは 225 回である。その結果、全体の性能が向上していて、最も良い性能を出している。

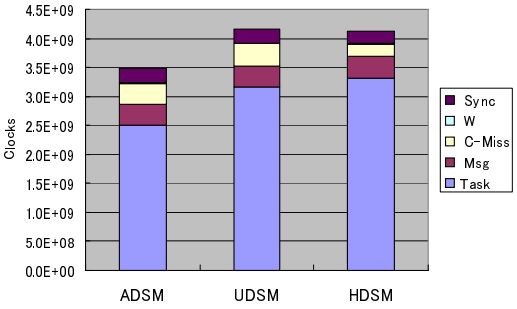


図 9 Raytrace の実行時間 (8 台) のブレイクダウン

図 9 が ADSM/UDSM/HDSM における Raytrace の実行時間 (8 台) のブレイクダウンである。“Task” 等の意味は上と同様である。ADSM より UDSM、UDSM より HDSM と “Task” の時間が増大しているのは、read commitment の発行の命令オーバヘッドが原因である。Raytrace は不規則参照が多いので、ベクトル化によってチェックコードのコストを下げることが困難である。HDSM が UDSM より遅くなるのは、更に制御を複製しているからである。

HDSM ではプリフェッチの発行により、Msg と C-Miss の和が最も少なく押さえられている。その結果、Task のバランスが取れて、“Sync” の時間も減少している。ただし、その分、命令オーバヘッドが増大しているため、プリフェッチ効果は相殺されていることが

分かる。

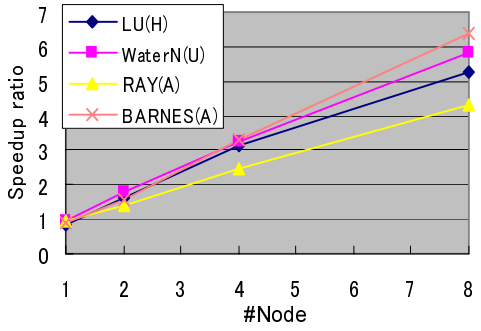


図 10 LAN 上の S-DSM の台数効果

図 10 が S-DSM の 8 台までの台数効果である。ただし、並列 1 台の実行時間ではなく、逐次の実行時間を元に算出している。また、ADSM/UDSM/HDSM の中で最も良いものを採用している。コンパイラの支援があり、適切なインタフェイスを選択すると、汎用の分散環境上でも高性能を出すことは十分可能である。

5. 関 連 研 究

プリフェッチそのものに関する研究としては、Mowry[7] 達の研究が挙げられる。S-DSM におけるプリフェッチの研究としては、Dwarkadas 達[2] の研究が挙げられる。彼らは、コンパイラが明示的に並列に書かれた Fortran のソースを解析することで、トラップベースのソフトウェア DSM：TreadMarks[3] の性能を改善している。Regular section analysis を用いて通信の一括化、無駄なコヒーレンス管理の除去を行っており、プリフェッチも行っている。しかし、手続き呼び出し、条件節が解析の障害になっている。

それに対して、本研究は、手続き間で解析を行っており、条件式が副作用を持たない場合には、条件節内の共有アクセスであっても、コヒーレンス管理コードは条件節の外に移動することが可能である。

6. ま と め

本稿は、S-DSM において、アプリケーションプログラムのソースを直接解析する最適化コンパイラの支援により、遠隔メモリアクセスのレイテンシを更に削減する手法を提案並びに実装し、その効果を測定した。

本手法は、確かに遠隔メモリアクセスのレイテンシを削減するが、その分、命令オーバヘッドを増大させる危険があり、全体の実行時間を遅くするケースがあること確認した。高性能を出すためには、使用する環境やアプリケーションに応じて適切なインタフェイスを選択することが必要である。特に本手法は、WAN のような高レイテンシの環境で有効であると考察される。

参 考 文 献

1) BOYLE,J., BUTLER,R., DISZ,T., GLICKFELD,B., LUSK, E., OVERBEEK, R., PATTERSON, J. and STEVENS, R. *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc. (1987).
2) DWARKADAS,S., COX,A.L.and ZWAENEPOEL,W. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System, Proc. of ASPLOS-VII (Oct. 1996).
3) KELEHER, P., COX, A. L., DWARKADAS, S. and ZWAENEPOEL, W. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems, Proc. of the Winter 1994 USENIX Conf. (Jan. 1994).
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) LI, K. IVY: A Shared Virtual Memory System for Parallel Computing, Proc. of the 1988 ICPP (Aug. 1988).
6) MATSUMOTO, T. and HIRAKI, K. Memory-Based Communication Facilities and Asymmetric Distributed Shared Memory, Proc. of the 1997 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, Los Alamitos, CA (1998), IEEE Computer Society.
7) MOWRY, T., LAM, M. S. and GUPTA, A. Design and evaluation of a compiler algorithm for prefetching, Proc. of ASPLOS-V (Oct. 1992).
8) NIWA,J. *Study on Optimizing Compilers to Support Software Distributed Shared Memory Systems*, PhD thesis, Department of Information Science, Tokyo University (2000).
9) SHAN,H.and SINGH,J.P. Parallel Tree Building on a Range of Shared Address Space Multiprocessors: Algorithms and Application Performance, Proc. of the First Merged Symp. IPPS/SPDP 1998) (1998).
10) TREECODE GUIDE, J. Barnes, http://www.ifa.hawaii.edu/barnes/treecode/treeguide.html.
11) 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).
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) 丹羽 純平, 松本 尚, 平木 敬ソフトウェア分散共有メモリ機構を支援する最適化コンパイラ, 情報処理学会論文誌, **42**, 4 (Apr. 2001), 879–897.