計算機アーキテクチャ 141－2
（2001．1．26）

OpenMP向けコンパイラ支援ソフトウェアDSMの性能評価

佐藤 茂久、 草野 和寛、 佐藤 三久
新情報処理開発機構 つくば研究センタ
{sh-sato, kusano, msato}@trc.rwcp.or.jp

OpenMP API を用いた共有メモリ並列プログラムを、SMP PC を高速なネットワークで接続した SMP クラスタ上で透過的に実行するコンパイラ支援ソフトウェア DSM を研究・開発している。コンパイラが、共有データのノード間での一貫性を保証するためのコードをプログラム中に挿入すると共に、ソースレベルの解析に基づいて一貫性制御コードの最適化を行なうことを特徴とする。本稿では、OpenMP C 版の NAS 並列ベンチマークの中の、共役勾配法のプログラム (CG) を用いて、性能と最適化の効果を評価する。評価には Pentium II Xeon プロセッサを用いた 4 ウェイ SMP を 8 台、Myrinet で接続した SMP クラスタを用いた。わずかなソースコードの修正と、手続き間解析や並列データフロー解析に基づいた最適化により、4 ノード×2 スレッドで、逐次実行の 3.77 倍の性能が得られた。しかし、それ以上の性能向上のためにはデータ配置の最適化が必要と思われる。

Performance Evaluation of the Compiler-directed Software DSM for OpenMP

Shigehisa Satoh, Kazuhiro Kusano, and Mitsuhisa Sato
Tsukuba Research Center, Real World Computing Partnership

We are developping a compiler-directed software DSM symtem that enables transparent execution of shared-memory parallel programs using OpenMP API on PC-based SMP clusters. The compiler inserts coherence control codes into a program and optimize them based on source-level analysis. In this paper, we present performance evaluation using OpenMP C version of the CG benchmark in the NAS Parallel Benchmarks. We evaluated performance using an SMP cluster consisting of Pentium II Xeon processors and Myrinet interconnection network. We obtained 3.77 times speedup for 4 nodes of 2-way SMPs with simple code modification and aggressive compiler optimizations based on interprocedural analysis and parallel dataflow analysis. However, it seems that optimizations for data placement is needed to obtain more performance.

1. はじめに

我々は、共有メモリ並列プログラミングの標準 API である OpenMP を用いて記述した並列プログラムを、コモディティ・ハードウェアを用いた SMP クラスタ上で透過的に実行することができるコンパイラ支援ソフトウェア DSM(以下、CSDSM と呼ぶ) を研究・開発している [1)~5)]。単一のプログラムを再コンパイルするだけで様々な計算機上で実行することができる、シームレスな並列プログラミング環境を実現することが本研究の目的である。

CSDSM の特徴は、コンパイラがプログラムを解析し、実行時に共有データの一貫性制御を行なうためのコードをプログラム中に挿入することにある。同時に、ソースレベルの情報を用いて不要な一貫性制御の削除などを行なうことができるため、アプリケーションの特徴に応じた最適化が可能である。OpenMP を用いた場合、プログラムの並列性が構造的に記述されていることと、緩いメモリコンシステンシモデルを用いていることから、他の共有メモリプログラムよりもコンパイラによる最適化を効果的に行なえる。その一方、コンパイラによる最適化が効果的に行なえない場合、良好な性能が得られない場合もある。

本稿では、CSDSM が苦手とする、不規則な共有データ参照を行なうプログラムを用いて性能を評価する。不規則問題の例として、並列処理の性能評価に広く用いられている NAS 並列ベンチマーク [7)] の中の CG と呼ばれる共役勾配法のプログラムを用いる。CG の計算の中心は疎行列とベクトルの積であり、その疎行列の参照が不規則な参照パターンを持つ。

SMP PC を Myrinet で接続した SMP クラスタを用いて、実行性能を評価し、最適化の有効性を検証した。その結果、4 ノード×2 スレッドで、逐次実行の 3.77 倍の性能が得られた。しかし、ノードやスレッドをさらに増やしても一貫性制御のオーバーヘッドのために性能は向上せず、それ以上の性能を得るためにはデータ配置の最適化が必要と思われる。

以下では、まずコンパイラ支援ソフトウェア DSM の概要を述べる。次にソフトウェア一貫性制御の概要と、コンパイラによる一貫性制御の最適化について述べる。そして、CG を用いて最適化の効果を評価する。最後にまとめと今後の課題を述べる。

2. コンパイラ支援ソフトウェア DSM

2.1 システム構成

CSDSM の構成要素を図 1 に示す。

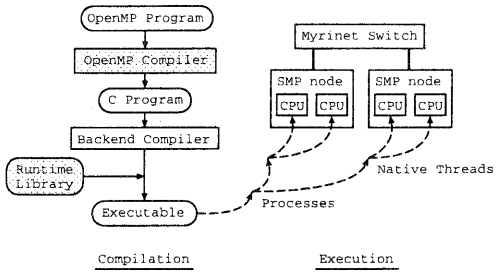

図 1　コンパイラ支援ソフトウェア DSM

OpenMP コンパイラは OpenMP プログラムを、CSDSM の実行時ライブラリを用いたマルチスレッドプログラム (C プログラム) に変換する。その際、共有データの一貫性の制御を行なうコードを挿入すると共に、ソースレベルの解析情報を用いて一貫性制御コードの最適化を行なう。生成された C プログラムは既存の C コンパイラでコンパイル・リンクする。そして得られた実行可能プログラムを、SMP クラスタの各ノードでそれぞれ一つのプロセスとして実行する。各プロセスは、ノード内でネイティブスレッドを一つ以上起動する。各ネイティブスレッドは OpenMP のスレッドを割り当てられ、実行する。

各ノードで実行されるプロセスはそれぞれ独自の仮想アドレス空間を持っている。CSDSM では、その仮想アドレス空間の一部を共有アドレス空間とみなし、その空間内に配置された共有データの内容が各ノードで同一に見えるよう、ソフトウェアで制御する。

各プロセスは逐次プログラムと同様に、テキスト (実行する命令)、静的データ、スタック、プロセス内のヒープ (局所ヒープ) を持つ。その他にノード間で共有される共有ヒープを持つ。仮想的な共有アドレス空間は静的データと共有ヒープからなり、この領域のデータが一貫性制御の対象となる。

次節で共有アドレス空間の一貫性制御の方法を述べる。CSDSM の詳細と関連研究については、文献[1)~5)]を参照して頂きたい。

2.2　一貫性制御の概要

静的データと共有ヒープからなる共有アドレス空間は、さらにページとラインと言う二種類の単位で管理される。

CSDSM のページとは、ホームノードの割当の単位である。ただし、本稿の実験ではホームノードの最適化は行なっていないため、全てのページのホームはノード 0 である。ノード 0 はホームノードである他に、共有ヒープの管理、逐次領域の実行、`master` および `signle` 構文の実行も担当する。

CSDSM のラインとは、一貫性制御の単位である (現在のライン長は 64 バイト)。各ノードはライン毎に状態フラグを持ち、対応するラインの最新のコピーを持っているか否かを管理している。実行時の一貫性制御はこの状態フラグを元にライン単位で行なう。ただし、ラインの一部 (ワード単位) のみを更新することもできる。アドレスの連続したラインの処理は一括して行なえるため、ラインの倍数の可変長ブロックを一貫性制御の単位としていると考えることもできる。

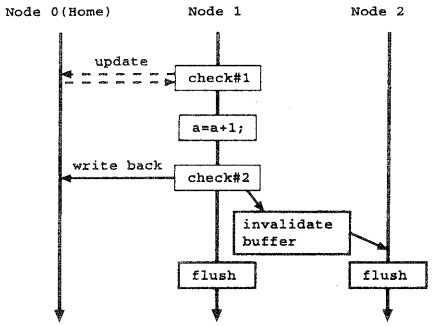

図 2　チェックコードの基本操作

コンパイラがプログラム中に挿入する一貫性制御コード (チェックコードとも呼ぶ) には、次の三種類の基本操作がある。

読み込み前チェック　参照する領域の状態フラグを調べ、最新のコピーを持っていなければ、ホームノードから最新の値をコピーする。

書き込み前チェック　読み込み前チェックと同様だが、ライン全体を書き換える場合には古い値は必要ないため、コピーは行なわずに状態フラグの更新のみ行なう。

書き込み後チェック　新しい値をホームノードへ書き戻し、自分とホーム以外のノードの持つコピーを無効化する。

これらはいずれもアドレスと参照する領域のサイズを引数として呼び出され、最初に参照する領域が共有アドレス空間か否かを判定する。これは、ポインタ間接参照などでは、静的な解析で共有データか否かを正確に決定することができないためである。

図 2 は、ノード 1(ホームではない) で共有データ `a` の更新を行なう場合のチェックコードの動作を図示している。変数 `a` への代入文の前に実行されるチェックコード (`check#1`) は、読み込み前チェックと書き込み前チェックの両方の操作を行なう。すなわち、状態フラグを調べ、無効なラインがあればその全てに対してホームノードから最新の値をコピーする。一方、代入後のチェックコード (`check#2`) は、書き換えた値をホームノードに書き戻し、他のノードへ無効化要求を送る。無効化要求はバッファに蓄積され、ノード 1 が `flush` 操作に到達するまでに送信される。無効化要求を受信したノードは、そのノードが `flush` 操作を実行するまでに無効化処理を完了する。

以上は基本的なチェックコードであり、これらを特化したものや、無効化プロトコルではなく更新プロト

コルで処理するものなども用意している。それらは、コンパイラによる最適化や、実行時ライブラリの内部で使用する共有データの一貫性制御で用いられる。

3. コンパイル時最適化

OpenMP コンパイラでは主に以下の処理を行なう。

プログラム構造の正規化： 後続の処理で解析や最適化を行ないやすくするため、制御フローやデータ参照の正規化を行なう。

OpenMP 指示文の変換： SMP 向けのコンパイルと同様に指示文の変換を行なう。OpenMP の機能を実現するためのライブラリは、SMP 向けと同じインターフェイスでも、内部では DSM の実行時システムを用いたコードになっている。

共有データのメモリ割り付け： 複数のスレッドに参照され得る自動変数を共有領域に割り付け直したり、動的なメモリ割付・解放の関数 (`malloc()` 等) を共有ヒープに対する操作に置き換える。

一貫性制御コードの生成と最適化： 共有データである可能性のあるデータの参照に対する一貫性制御コードを生成と最適化を行なう。

CSDSM のコンパイラによる最適化の中心は一貫性制御コードの最適化であるが、その他にも OpenMP 指示文の最適化 (不要指示文削除等) や、逐次プログラムと同様の定数伝搬等の最適化も行なう。

次節以降では、一貫性制御コードの最適化を以下の三段階に分類し、それぞれの最適化の概要を述べる。

局所最適化 同期区間内の解析で得られる情報のみを利用して最適化を行なう。

手続き間解析に基づく最適化 プログラム全体に対して指示文の意味を考慮した手続き間解析を行ない、それを利用した最適化を行なう。

並列データフロー解析に基づく最適化 スレッド間のデータ依存関係を解析し、それを利用した最適化を行なう。

3.1 局所最適化

OpenMP のように緩いメモリコンシステンシモデルに従う共有メモリ並列プログラムでは、メモリ同期点 (OpenMP では `flush` 操作) を越えない範囲 (同期区間と呼ぶ) では、スレッド間の相互作用を考慮することなく逐次プログラムと同様の最適化が行なえる。そのようなメモリコンシステンシモデルを生かした最適化は、従来の CSDSM でも行なわれており、以下のような最適化手法が効果的である。

チェックコードのマージ 連続した領域の参照に対する複数のチェックコードを一つのチェックコードにまとめる。

チェックコードの冗長性削除 同期区間内で同一領域へのチェックコードが複数回連続して実行される時、その一つを除いて削除する。

```
void sparse(...,  double rowstr[], ...
            int firstrow, int lastrow, ...) {
    int j, n, nrows;
    ...
C0: _check_before_write(&rowstr[1], n*4);
S0: nrows = lastrow-firstrow+1;
C1: _check_before_read(&rowstr[1], (nrows+1)*4);
C2: _check_before_write(&rowstr[2], nrows*4);

L1: for (j = 1; j < n+1; j++) {
S1:   rowstr[j] = 0;
    }
L2: for (j = 2; j < nrows+2; j++) {
S2:   rowstr[j] = rowstr[j]+rowstr[j-1];
    }
L3: for (j = nrows; j >= 1; j--) {
S3:   rowstr[j+1] = rowstr[j];
    }

C3: _check_after_write(&rowstr[1], n*4);
C4: _check_after_write(&rowstr[2], nrows*4);
    ...
}
```

図 3 局所最適化の例

チェックコードのループ外移動 ループ中で繰り返し参照されるデータのチェックコードをループ外に移動し、一回だけ実行されるようにする。

これらの最適化に必要な情報は同期区間内の解析のみでもある程度は取得できる。しかし、ポインタが共有データを指す可能性があるか否かや、途中に手続き呼び出しを含むような場合には、局所的な解析だけでは精度の高い情報が得られない。そのため、後述の手続き間解析や並列データフロー解析を行なうことによって、これらの最適化を更に効果的に行なえる。

図 3 は、チェックコードのマージと冗長性削除が組合わせた最適化の例を示す。なお、本稿で用いる例はいずれも CG ベンチマークで実際に行なわれた最適化を元にしている。

この例では、複数のループで参照される配列 `rowstr[]` に対する読み込みと書き込みのチェックコードが挿入されている。配列 `a` の要素 `n` から `m` までの集合を $a[n : m]$ と表すことにする。読み込み前のチェックコード `C1` は、ループ `L2` で読み込まれる $rowstr[2 : nrows + 1]$ と $rowstr[1 : nrows]$ に加えて、ループ `L3` で読み込まれる $rowstr[1 : nrows]$ を合わせた、$rowstr[1]$ から始まる $nrows + 2$ 個の要素に対するチェックを表している。読み込み前・書き込み前のチェックコードはできるだけ速く実行されるように配置されるが、`C1` では参照範囲を表すのに変数 `nrows` が使用されるため、`nrows` への代入文である文 `S0` の直後に挿入される。

実際のプログラムでは、このように配列の参照範囲をコンパイル時定数だけでなく変数を含んだ式のまま解析すること (シンボリック解析) が必要である。

3.2 手続き間解析に基づく最適化

近年の最適化コンパイラでは、手続き間解析・最適化は必須の技術となっている。CSDSM でも手続き間情報を利用した最適化は重要であり、以下のような解

析・最適化を行なっている。

ネストレベル解析とクローニング　逐次領域と並列領域とで一貫性制御のプロトコルを変えたり、逐次領域にある不要な指示文を削除するため、ネストレベルの解析を行なう。逐次領域と並列領域の両方で実行される関数はクローニングを行ない、ネストレベルに応じた最適化を可能にする。

手続き間ポインタ解析　逐次プログラムと同様に別名関係を求める他に、ポインタ間接参照が共有領域を指すか否かを解析する。動的に割り付けられるデータに対しては、メモリ割り付け関数の呼び出し文脈毎に区別した名前付けを行ない、別々に割り付けられたデータ同士を区別する。OpenMP ではプライベートデータを動的に割り付けることができないため、プライベートなポインタにしか参照されない動的データを検出し、共有ヒープではなく局所ヒープに割り付けるように最適化を行なう。

手続きの参照変数解析　逐次プログラムの手続き間解析と同様に、手続きが直接・間接に参照 (定義または使用) する変数を検出し、手続き呼び出しを越えた最適化を可能にする。同一変数に対して共有版とプライベート版の参照が存在する場合には、それらを別の変数として区別した解析を行なう。

並列領域の参照変数解析　手続きと同様に並列領域に対しても、その中で参照される変数を解析する。ただし、並列データフロー解析を行なわない場合、スレッド間のデータフローについては安全な近似を行なう。

これらの情報を元に共有データを以下のように分類し、参照パターン毎に異なる方法で一貫性制御を行なう。

逐次領域でのみ参照される変数　このような変数はデフォルトでは共有データでも、一貫性制御は不要である。

並列領域では使用のみされる変数　定義が逐次領域にしかないような変数は、並列領域内での動的な一貫性制御は不要であり、並列領域の入口で各ノードが最新の値を持っていることが保証できれば良い。

並列領域で定義・使用ともされる変数　このような変数の最適化には、並列データフロー解析が必要である。

このように変数毎に並列領域での定義・使用の有無を求めることにより、並列領域で書き換えられることのない変数に対しては、並列領域内での動的な一貫性制御が不要となる。

3.3 並列データフロー解析に基づく最適化

明示的な共有メモリ並列プログラムに対する、スレッド間の相互作用を考慮したデータフロー解析を**並列データフロー解析**と呼ぶ [3),4)]。並列データフロー解析を行なうことにより、次のような最適化が可能になる。

```
void conj_grad(..., double r[], ...) {
  int j;
  double alpha;

  ...
#pragma omp for
  for (j = 1; j <= lastcol-firstcol+1; j++) {
    r[j] = x[j];
  }
#pragma omp for
  for (j = 1; j <= lastcol-firstcol+1; j++) {
    r[j] = r[j] - alpha*q[j];
  }
#pragma omp for reduction(+:rho)
  for (j = 1; j <= lastcol-firstcol+1; j++) {
    rho = rho + r[j] * r[j];
  }
  ...
}
```

図 4　並列データフロー解析の例

非共有データの検出　配列を参照するループの並列処理では、配列全体としては共有されていても、配列の要素単位で見ると特定のスレッドにしか参照されない場合が良くある。配列の要素単位で複数のスレッドに参照され得るかを解析し、共有されない要素に対する一貫性制御を省略する。

明示的データ転送　共有データが複数のスレッドに参照される場合でも、いつどのスレッドが定義した値をいつどのスレッドが使用するかが検出できる場合がある。そのような場合には、定義したスレッドから使用するスレッドへ明示的にデータを転送することで、状態フラグのチェックを省略すると共に、通信の遅延を隠蔽することが期待できる。

メモリ同期点を越えた最適化　メモリ同期点で同期する必要のある必要最小限の変数を求め、その他の変数に対しては同期点を越えて最適化できるようにする。OpenMP ではバリア同期などで暗黙に `flush` 操作が行なわれることが多いため、このような最適化の機会は多い。

図 4 は、並列データフロー解析により、スレッド間のデータ依存が生じないことが検出できる例である。ここで配列 `r[]` は複数の並列ループで参照されるが、それぞれのループの範囲が同一であり、スケジューリングも同一 (デフォルトの `static` スケジューリング) であるため、各要素は同じスレッドにしか参照されない。そのため、配列 `r[]` の参照に対する一貫性制御は不要である。これは単純な例であるが、より複雑な場合でもスレッド間のデータ依存関係を検出できる [3),4)]。

4. 性能評価

4.1 ベンチマークプログラム (CG)

我々は CSDSM を含めた OpenMP の処理系の評価のために、並列計算機の性能評価に良く利用される

```
void conj_grad(int colidx[], int rowstr[], ...
               double a[], double p[], ...) {
  int j, k;
  extern int lastrow, firstrow;
  ...
#pragma omp for
  for (j = 1; j <= lastrow-firstrow+1; j++) {
    double tmp = 0.0;
    for (k = rowstr[j]; k < rowstr[j+1]; k++) {
      tmp = tmp + a[k]*p[colidx[k]];
    }
    q[j] = tmp;
  }
```

図 5 CG プログラムのカーネル

NAS 並列ベンチマーク (NPB) の OpenMP C 版を作成した☆。本稿の性能評価では、そのうちの CG(クラス A) を使用する。

CG は共役勾配法のプログラムであり、計算の中心は構造的でない疎行列とベクトルの積である。この疎行列の参照が添字の配列を用いた間接参照であるため、参照パターンをコンパイル時に検出できず、静的な最適化の妨げとなる。

図 5 は、CG のプログラムの中心となるループを示す。ここで問題となるのは配列 `p[]` に対する参照 `p[colidx[k]]` である。添字が静的に求められないため、`p[]` に対するチェックコードの最適化が行なえず、一要素毎にチェックコードが実行される。そのため、このままコンパイル・実行するとチェックコードのオーバーヘッドが極めて大きくなる。

CG ではこのループの他に、これと同じ形式のループと、前節の例のように配列を規則的に参照する並列ループがいくつかある。図 5 のように並列化した SMP 向けのコードを元にし、CSDSM に適した並列化の方法と、コンパイラによる最適化の効果を評価していく。

4.2 SMP クラスタ: COMPaS II

本稿の評価には PC ベースの SMP クラスタ COMPaS II を用いた。Intel Pentium II Xeon プロセッサ (500MHz) の 4 ウェイ SMP をノードとし、Myricom 社の Myrinet を介して 8 ノードが接続されている。Myrinet 用の通信ライブラリとして NICAM[6] を用いる。各ノードは 1GB 以上の主記憶を持ち、それぞれ 4 ウェイにインターリーブされている。OS は Red Hat Linux 6.2 であり、カーネルのバージョンは 2.2.16 である。ノード内の並列処理には LinuxThread(POSIX スレッドライブラリと互換) を用いる。OpenMP コンパイラが生成した C プログラムをコンパイルするバックエンドコンパイラは、GNU C コンパイラ (egcs1.1.2) を用い、最適化オプションは `-O3 -funroll-loops -malign-double` とした。

4.3 評価方法

CG の性能を、次の五段階に分類し、それぞれのコードを作成して測定した。

BASE: SMP 向けのコードをそのままコンパイルし、局所最適化のみ行なう。

LOCAL: 不規則参照される配列のプライベートコピーを作成し、局所最適化のみ行なう。

IPA: 上記 LOCAL に加えて手続き間解析に基づく最適化を行なう。

PDA: 上記 IPA に加えて並列データフロー解析に基づく最適化を行なう。

IDEAL: 配列に対する一貫性制御を全て省略したもの。

元となった OpenMP C 版 CG のコードは SMP 上で開発されたものであり、逐次プログラムへの変更を最小限に押えたものである。しかし、そのままでは前述のように共有データに対する不規則参照が含まれており、最適化の障害となる。

CG の場合、不規則な参照をされる配列は、読み込みは不規則だが書き込みは規則的である。そこで、不規則参照するときにはプライベート配列にコピーしたものを参照するように修正したのが LOCAL である。具体的には、図 5 の配列 `p[]` をプライベートコピーの参照に置き換えた。ただし、この `p[]` のコピーは他の配列と同様に引数として渡すようにした。そのため、局所解析のみではプライベートデータであることがわからず、チェックコードが生成されてしまう。

LOCAL と同じソースに対して、最適化のレベルを上げていったのが IPA と PDA である。IPA では、手続き間解析によってプライベートデータと並列領域で書き換えられない共有データが検出され、それらに対する一貫性制御が削減される。さらに PDA ではスレッド間のデータ依存関係を解析し、並列領域で書き換えられる共有データに対する不要な一貫性制御も削除できる。

最後の IDEAL は、制御フローを確定するのに最小限の一貫性制御コードのみ挿入したコードであり、命令の実行数は変わらないが、計算結果は正しくない (1 ノードの時は正しい)。これは一貫性制御のオーバーヘッドが最小化された場合の性能の目安として示している。

4.4 実験結果と考察

図 6 はそれぞれのコードの実行性能を逐次版からの相対性能で示している。ただし、LOCAL 全体と IPA の一部のデータは示していない。グラフで $n \times m$ と表示されているのは n 個のノードでそれぞれ m 個のスレッドを用いて、合計 nm 個のプロセッサを用いたと言う意味である。また、逐次版の実行時間は 39.32 秒であった。

☆ OpenMP C 版 NPB は **http://pdplab.trc.rwcp.or.jp/pdperf/Omni/benchmarks/NPB/** で公開している。なお、NAS の Jin らが NPB をベースに開発した PBN3.0 と呼ばれる新しいベンチマークには、OpenMP Fortran 版が含まれている [8]。

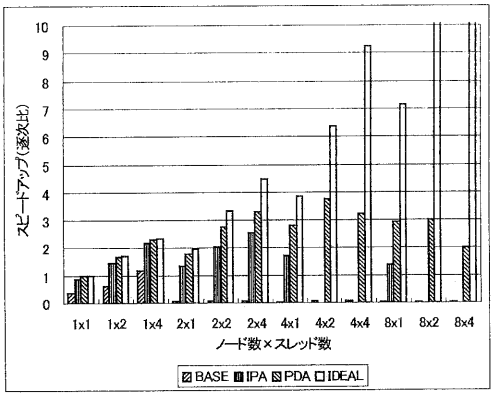

図 6 CG の実行性能

まず、BASE では、1x1(1 ノードで 1 スレッド) のときに逐次版の 2.86 倍の時間がかかっている。これは最内側ループにチェックコードが入っているため、チェックコードの命令実行のオーバーヘッドが生じているためと考えられる。BASE の 2 ノード以上はグラフでは殆んど見えないが、速度向上率にすると 0.04 倍から 0.07 倍、実行時間にすると 604 秒から 997 秒と言う膨大な時間がかかっているためである。これは、配列の要素単位で無効化・更新を行なっているためと考えられる。

次に LOCAL では、不規則に参照される配列をプライベートデータに置き換えているが手続き内解析のみではプライベートで他であることがわからないため、一貫性制御コード自体は挿入され、BASE と性能は大差ない。

手続き間データフロー解析 (IPA) を行なった IPA では、性能が大幅に改善されている。これは主に、並列領域内で書き換えられない共有データを検出できることと、引数の配列がプライベートデータであることを検出できるためである。CG では疎行列の構造と非零要素を保持した配列と、LOCAL で導入したプライベート配列の一貫性制御が削減された効果が大きい。

並列データフロー解析 (PDA) で、スレッド間のデータ依存の生じないデータを検出することで、さらに性能が改善された。しかし、性能向上率は 4 ノード×2 スレッドの時の 3.77 倍が最高であり、それ以上ノード／スレッドを増やすと性能が低下する。その原因を調べるために、一貫性制御を行なわずに SMP クラスタ上で実行できる IDEAL を作成し、性能を比較した。IDEAL では、殆んどの共有データの一貫性制御を行なわないが、それ以外の命令は PDA のコードと同じように実行する。そのため、IDEAL の性能は、命令実行と制御の同期のみ考慮した場合の性能と考えられる。グラフには収まり切らなかったが、8x4 ノードでは逐次実行の 20.4 倍の性能を示した。IDEAL と PDA の性能の差は一貫性制御のオーバーヘッドを表していると考えられる。ノードを増やすほど差が広がるのは、ホームノードの最適化を行なっていないために、ホームでないノードが共有データを参照する割合が高まることと、無効化処理を全ノードに対して行なっているためと考えられる。

5. ま と め

OpenMP C 版の NAS 並列ベンチマークから、不規則なデータ参照パターンを持つ CG を取り上げ、OpenMP 向けコンパイラ支援ソフトウェア DSM の性能評価を行なった。

CSDSM ではコンパイラによる一貫性制御の最適化の成否に性能が大きく依存する。そのため、最適化の困難な不規則データ参照を行なうプログラムでは高性能は得にくい。しかし、プライベートコピーを作成することで共有データに対する不規則な参照をなくし、手続き間解析と並列データフロー解析に基づいた最適化を行なうことによって、ある程度の性能が得られた。これ以上の性能を得るためには、静的・動的なデータ配置の最適化が必要と思われる。

今後は、NPB の他のプログラムでの評価や、さらなる静的・動的な最適化手法の検討を行なっていく予定である。

謝辞 本研究について有益な御討論・アドバイスをして頂いている、Omni コンパイラグループの皆様、筑波大・電総研・東工大などの研究者からなる TEA グループの皆様に感謝いたします。

参 考 文 献

1) 佐藤茂久他, SMP クラスタ向け OpenMP コンパイラ, *SWoPP99*, Aug. 1999.
2) M.Sato, et.al., Design of OpenMP Compiler for an SMP Cluster, *EWOMP99*, Sep. 1999.
3) 佐藤茂久他, OpenMP コンパイラにおけるメモリ一貫性制御の最適化, *JSPP 2000*, June 2000.
4) 佐藤茂久他, OpenMP 並列プログラムのデータフロー解析手法, *SWoPP2000*, Aug. 2000.
5) S.Satoh, et.al., Compiler Optimization Techniques for OpenMP Programs, *EWOMP2000*, Sep. 2000.
6) 松田元彦他, SMP クラスタ向けネットワーク・インターフェース AM 通信, 情報処理学会論文誌, Vol.40, No.5, 1999.
7) The NAS Parallel Benchmarks. **http://www.nas.nasa.gov/Software/NPB/**.
8) H.Jin, et.al., The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance, NAS Technical Report NAS-99-011, Oct. 1999.