OpenMP向けコンパイラ支援ソフトウェアDSM

佐藤茂久† 草野和寛† 佐藤三久†

共有メモリ並列プログラミングの標準APIであるOpenMPを用いた並列プログラムを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分散共有メモリ（DSM）システムを提案する．SMPクラスタの各ノードで実行するプロセスの仮想アドレス空間の一部を共有アドレス空間と見なし，ソフトウェアで細粒度の一貫性制御を行う．コンパイラ支援ソフトウェアDSMでは，実行時に一貫性制御を行うためのコードをコンパイラがプログラム中に埋め込む．同時に，効率の良い一貫性制御を行うための最適化も行う．ソースレベルの解析により共有データの参照パターンを求め，一貫性制御の削除や，明示的なリモートコピーの生成を行う点が本稿の方法の特徴である．本方式をPCとMyrinetで構成したSMPクラスタ上に実装し，性能を評価した．その結果，規則的なデータ参照を行うアプリケーションで，4ウェイSMPを8台接続したSMPクラスタで，逐次実行の6.15倍から9.90倍の性能が得られた．さらに，共有データの参照パターンに基づいた最適化は，共有データの書き込みの多いプログラムで特に効果の高いことも分かった．

A Compiler-directed Software DSM for OpenMP

Shigehisa Satoh,† Kazuhiro Kusano† and Mitsuhisa Sato†

We present an OpenMP implementation which is based on a compiler-directed software distributed shared memory system. The system enables the transparent execution of shared-memory parallel programs on top of a cluster of SMP nodes. A part of virtual address space in each process running on an SMP cluster is virtually shared across nodes and the coherency of the shared region is maintained by software at a fine granurality. The OpenMP compiler inserts coherence control codes into the program and optimize them by using source-level information. The compiler removes coherence control codes and/or inserts explicit remote copies without runtime checks if the sharing pattern of the shared data is known at compile-time. We evaluated the performance of our approach using a prototype system on a PC-based SMP cluster connected via Myrinet. With eight 4-way SMP nodes, we obtained speedups of 6.15 to 9.90 over serial execution for programs that have regular memory access patterns. It is also found that the compiler optimization using sharing-pattern information is quite effective for shared-write intensive programs.

1. はじめに

近年のマイクロプロセッサやネットワーク技術の進歩により，低価格で高性能な並列システムを容易に構築できるようになった．そこで，コモディティ・ハードウェアで構成した，対称マルチプロセッサ（Symmetric Multi-Processors; SMP）やクラスタ（SMPクラスタを含む）を用いた高性能計算が注目されている．

並列プログラムを記述する際の並列プログラミングモデルとしては，POSIXスレッドなどの共有メモリモデル，MPIやPVMなどのメッセージパッシングモデル，そして両者を組み合わせたハイブリッドモデルが主に用いられている．メッセージパッシングモデルやハイブリッドモデルは並列処理の専門家以外が用いるのは難しく，また専門家であっても開発コストは共有メモリモデルより大きくなる．そこで，分散メモリシステム上にハードウェアまたはソフトウェアによって共有アドレス空間を実現した分散共有メモリ（Distributed Shared Memory; DSM）システムが研究・開発されている[1),2)]．DSMでは，スケーラビリティとプログラミングの容易さを両立することができると考えられている．そのため，共有メモリモデルは，様々なプラットフォームでシームレスに利用可能な並列プログラミングモデルとして期待されている．

共有メモリモデルのプログラミングインタフェースとしては，POSIXスレッドのようなマルチスレッドライブラリがよく用いられている．また，Javaでは言語レベルでスレッドをサポートしている．これらのスレッドを用いたプログラミングは様々なアプリケー

† 新情報処理開発機構つくば研究センタ
Tsukuba Research Center, Real World Computing Partnership

ションに適用できる柔軟性がある反面，プログラミングはそれほど容易ではない．高性能計算の要求される科学技術計算ではループの並列化が中心となる場合が多いため，並列化指示文を用いた並列化で十分な記述ができることが多い．ベクトル計算機では，自動ベクトル化コンパイラとベクトル化指示文により容易に高性能が得られたことで，広範囲に利用されるようになった．並列化指示文による共有メモリ向け並列化はベクトル化からの移行もしやすく，科学技術計算分野では十分受け入れられると思われる．ところが，従来は並列化指示文の仕様を各社が独自に定義しており互換性がなかった．そこで，並列化指示文の標準仕様として，OpenMP API[3] が 1997 年に発表された．

OpenMP を用いた並列プログラムは，SMP や，CC-NUMA（Cache-Coherent Non-Uniform Memory Access）のようなハードウェア DSM では，コンパイラがネイティブスレッドを用いた並列プログラムに変換することで，実行可能プログラムを生成する．しかし，ハードウェアでは共有アドレス空間をサポートしていないクラスタなどで共有メモリプログラムを実行するためには，何らかの方法で共有メモリから分散メモリへの変換を行う必要がある．HPF[4] では，言語レベルでは共有アドレス空間を提供しているが，コンパイラがそれをメッセージパッシングコードに変換し，分散メモリ用のプログラムとして実行される．これは共有メモリから分散メモリへの変換を静的（コンパイル時）に行っているといえる．一方，近年注目されているソフトウェア分散共有メモリ（ソフトウェア DSM）では，実行時にソフトウェアで仮想的な共有アドレス空間が実現される．各ノードの局所メモリ内に持つ共有データのコピー（キャッシュ）の一貫性を実行時に保証することによって，共有メモリから分散メモリへの変換が動的（実行時）に行われる．

本稿では，コンパイラ支援ソフトウェア DSM と呼ぶソフトウェア DSM の一種に基づいた，SMP クラスタ向けの OpenMP の実装を提案する．

コンパイラ支援ソフトウェア DSM は，コンパイラがプログラム中の共有データの参照を検出し，一貫性制御のためのコードを挿入するとともに，一貫性制御のオーバヘッドを削減するための最適化を行うことを特徴とする．さらに，我々の方法では，プログラムの並列性を考慮したデータフロー解析を行い共有データの参照パターンを求め，それを利用して効率の良い一貫性制御を行えるように最適化を行う点が従来のコンパイラ支援ソフトウェア DSM と異なっている．従来のソフトウェア DSM では，共有データの一貫性の制御をすべて実行時に行っていたが，我々の方法では，一貫性制御を可能な限り静的に行い，それが困難な場合にのみ動的に行う．すなわち，コンパイル時に共有データのスレッド間のデータフローが正確に解析できる場合には，そのデータを書き込むスレッドと読み込むスレッドの間で明示的にデータ転送を行い，そうでない場合には従来のように実行時に一貫性制御を行うプログラムを生成する．

本方式により，OpenMP プログラムをソースコードの書き換えなしに透過的に SMP クラスタ上で実行することができる．OpenMP のスレッドは，SMP クラスタの各ノードで実行される各プロセス内のスレッドによって実行される．各プロセスの仮想アドレス空間の一部は共有アドレス空間として扱われ，一貫性制御コードによってノード間の一貫性が保証される．また，ページやオブジェクトではなく，ラインと呼ぶ固定長の小さな領域を単位に，細粒度の一貫性制御を行う．連続した複数のラインの処理を一括して行うことで，可変長ブロックのように扱える．参照パターンに応じた様々な一貫性制御方法（プロトコル）を用意し，コンパイラが効率の良い一貫性制御コードを生成しやすくする．

本方式のプロトタイプを SMP PC を Myrinet ネットワークインタフェースに接続した SMP クラスタ向けに実装し，規則的なデータ参照を行うプログラムを用いて性能評価を行った．その結果，共有データの読み込みが中心のプログラムでは従来のコンパイラ支援ソフトウェア DSM と同様の最適化のみでも良好な性能が得られた．しかし，共有データの書き込みの多いプログラムでは，従来の最適化手法だけでは一貫性制御のオーバヘッドが大きく，共有データの参照パターンをもとに効率の良い一貫性制御を行うことで良好な性能が得られることが分かった．

以下では，まず 2 章で OpenMP の特徴を述べる．次に本システムの概要を 3 章で示す．主要な構成要素である実行時ライブラリとコンパイラの詳細を，それぞれ 4 章と 5 章で述べる．プロトタイプシステムを用いた性能について 6 章で論じる．関連研究を 7 章で述べ，最後にまとめと今後の課題を述べる．

2. OpenMP の特徴

OpenMP API[3] は並列化指示文・実行時ライブラリ・環境変数からなる共有メモリ並列プログラミング API である．現在までに，Fortran と C/C++をベース言語とした仕様が発表されている[5),6)]．

以下ではコンパイラ支援ソフトウェア DSM を実現

するうえで重要な点を中心に，OpenMP の仕様の概要を述べる．

2.1 実行モデル

OpenMP は実行モデルとして**フォーク・ジョインモデル**を採用している．**マスタースレッド**と呼ばれる単一のスレッドによってプログラムの実行が開始される．プログラム中の `parallel` 構文によって指定された並列領域の入口でスレッドの集合（**チーム**）が生成され，複数のスレッドの実行が開始される．並列領域の最後でそれらのスレッドが合流し，そのうちの 1 つのスレッドがマスタースレッドとして実行を継続する．並列領域を実行するスレッドの数は実行時に決定することもできるが，単一の並列領域の実行中にスレッド数が変化することはない．

2.2 メモリモデル

OpenMP のメモリモデルで重要な点は，緩いメモリコンシステンシモデル[7] を採用していることである．`flush` 指示文，あるいは他の指示文で暗黙に行われる `flush` 操作によって共有データが同期される．

あるスレッドが書き換えた共有データを他のスレッドが読み込む場合，書き込み側と読み込み側の両方で `flush` 操作が必要である．したがって，基本的には Lazy Release Consistency と考えることができるが，`flush` 指示文で同期する変数を指定した場合にはその変数のみ同期されるので，Entry Consistency ということもできる．ただし，配列の要素単位で `flush` の指定はできず，また，バリア同期などで必ず暗黙の `flush` 操作が行われるため，プログラマが共有データの同期を詳細に制御することは困難である．

このような緩いメモリコンシステンシモデルでは，メモリの同期点を含まない区間ではスレッド間の相互作用を考慮することなくメモリアクセスの最適化が行える．たとえば，メモリ参照の順序を変えたり，共有データの値をレジスタに割り付けるなどの最適化が可能になる．さらに，スレッド間の相互作用を考慮したデータフロー解析を行う際にも，相互作用をメモリ同期点でのみ考慮すればよいために効率良く解析できるという利点がある．

2.3 同 期 構 文

同期構文としては，バリア同期のための `barrier` 指示文，排他制御のための `critical` 指示文と `atomic` 指示文，そして DOACROSS 型ループのための `ordered` 指示文が用意されている．このうち，スレッドが特定の順序で同期されるのは `ordered` のみであり，Post/Wait のような汎用的なイベント同期のための構文はない．ただし，ライブラリ関数のロック機構を用いて複雑な同期を記述することもできる．

2.4 ループ並列化

OpenMP はループの並列処理が中心となる科学技術計算の並列化に適している．ループの並列処理は，`for` 構文で指定されたループの各反復を各スレッドが分担することによって行われる．並列実行されるループでは，`ordered` 構文や `critical` 構文で明示されなければ，ループの反復間のデータ依存関係はない，すなわち DOALL 型のループであると見なされる．各反復のスレッドへの割り当て方は，`schedule` 節によって指定でき，指定のない場合は処理系のデフォルトのスケジュール方式が採用される．本稿では，スケジュール方式は `static` のみ考える．

3. システム概要

我々の提案する，コンパイラ支援ソフトウェア DSM に基づいた SMP クラスタ向けの OpenMP の実装のために，OpenMP コンパイラと実行時ライブラリを設計・開発した．まずこの章でシステムの概要を述べ，その後の 2 つの章で OpenMP コンパイラと実行時ライブラリの機能について述べる．

図 1 は，OpenMP プログラムのコンパイルと実行の手順を示す．本稿ではベース言語を C 言語として説明する．入力となる OpenMP プログラムは OpenMP コンパイラによって，マルチスレッドプログラムに変換される．このマルチスレッドプログラムは実行時ライブラリの呼び出しを含む C プログラムである．それをさらに既存の C コンパイラでコンパイルし，実行時ライブラリとリンクすることで実行可能プログラムを得る．生成された実行可能プログラムは，SPMD 方式の並列プログラムであり，SMP クラスタの各ノードで 1 つずつ別のプロセスとして実行される．各プロセスは既存のスレッドライブラリを用いて複数のスレッ

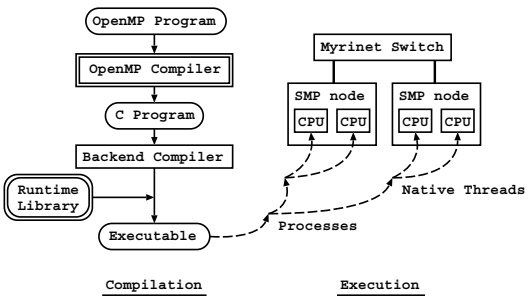

図 1 OpenMP プログラムのコンパイルと SMP クラスタでの実行

Fig. 1 Compilation and execution of OpenMP programs on an SMP cluster.

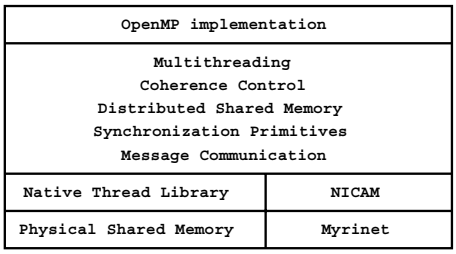

図 2 実行時システムの構成要素
Fig. 2 Components of the runtime system.

ドを生成する．

図 2 は，実行時システムの構成要素を示す．各ノード内では物理共有メモリと既存のマルチスレッドライブラリを用いて並列処理を行う．ノード間のデータ共有や同期のための通信は，Myrinet を介したリモートデータ転送を用いて行う．そのために，高速なリモートメモリ通信を行う通信ライブラリ NICAM[8] を用いた．これらのハードウェアとシステムライブラリを用いて，SMP クラスタ上でのマルチスレッド実行や分散共有メモリを実現，さらにその上に OpenMP の実行時システムを実装した．

我々のシステムは共有メモリ向けの OpenMP プログラムをプログラマが書き換えることなく，透過的に分散メモリシステム上で実行することを目的としている．OpenMP コンパイラを通してコンパイルされたプログラム中のデータに関しては，コンパイラで適切な変換を行うことにより，ノード間で共有することができる．しかし，OpenMP コンパイラでコンパイルされていないライブラリ内の共有データ（`errno`，`stdout` など）や，OS の管理下にあるリソースのノード間での共有はサポートしていない．

4. 実行時ライブラリ

SMP 向けの OpenMP の実装では，OpenMP の指示文の機能を実現するための関数と，OpenMP のライブラリ関数からなる実行時ライブラリが必要となる．我々のライブラリも，OpenMP の指示文の実装は SMP 向けの実装と同等の API を用いて実現する．しかし，その API を SMP クラスタ上で実現するための機構と，一貫性制御のための機構が新たに必要になる．

SMP クラスタ向けの実装に必要な機能は，以下のように分類できる．

(1) メッセージ通信
(2) スレッド管理
(3) 同期機構
(4) 共有仮想メモリ
(5) 一貫性制御

これらの各機能について，この章の残りで説明する．

4.1 メッセージ通信

ノード間の通信は，基本的には受信側ノードのスレッドが介在することのないリモートコピーにより行う．しかし，受信側のスレッド（ノード）で何らかの処理が必要な場合には，リモートコピーを用いて実装したメッセージ通信を用いる．これは汎用的なメッセージ通信機構を提供するものではなく，本システムで必要な処理に特化した機能である．

メッセージには以下の種類がある：

- 共有ヒープの伸長
- 共有ページの割当てと解放
- 共有ヒープのメモリブロックの取得と解放
- 論理スレッドの起動
- ロック操作
- 無効化要求
- プログラムの終了

共有空間内にノードの組ごとのメッセージキューを持ち，キューに直接メッセージをリモート書き込みすることによりメッセージを送信する．たとえば，ノード 0 からノード 1 へメッセージを送信する際には，ノード 1 の持つノード 0 用のキューに書き込む．

メッセージの受信は，スレッドがポーリングを行うことにより行う．メッセージの受信専用のスレッドはなく，ノード内のどのスレッドでもメッセージの受信を行える．メッセージの送り手は特定のスレッドであるが，受け手はノード内のどのスレッドでもよい．メッセージがリモート書き込みされた後，最初にポーリングを行ったスレッドが受信する．スレッド 0（マスタースレッド）は特別な役割を持つが，メッセージの受信に関しては同一ノード内の他のスレッドと対等である．ただし，論理スレッドの起動だけは受け手のスレッドを指定して行う．

Shasta[9] や ADSM[10] ではポーリングを関数呼び出しやループの反復ごとに行っているが，我々の実装では実行時ライブラリの関数が呼び出された際にのみ行った．これは，並列領域で受信するメッセージの多くが無効化要求であり，直後のメモリ同期点まで処理を遅らせることができることと，簡単な実験を行ったところポーリングを頻繁に行うと性能が低下したためである．

4.2 スレッド管理

SMP クラスタ上でのマルチスレッドプログラムの実行を，ノード内では POSIX スレッドや Solaris ス

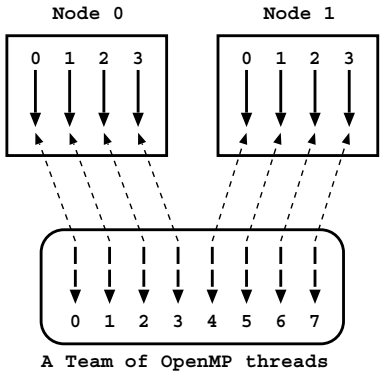

図 3 スレッドの割当て
Fig. 3 Mapping of threads.

レッドなどの既存のマルチスレッドライブラリ，ノード間では SPMD 方式を組み合わせることによって実現した．実行可能プログラムは，各ノードで同時に，しかし独立したプロセスとして起動される．各プロセスは，ノードごとの指定されたスレッド数（通常はプロセッサ数）のスレッドを，既存のマルチスレッドライブラリを用いて起動する（これらを物理スレッドと呼ぶ）．物理スレッドはプログラムの起動から終了まで実行され続ける．

OpenMP の並列領域におけるスレッドは仮想スレッドと呼ぶデータ構造で表し，それぞれの物理スレッドに特定の仮想スレッドの実行を指示するメッセージを送ることで実行される．たとえば，プログラムの逐次実行部ではただ 1 つの物理スレッドが逐次実行部に相当する仮想スレッドを実行し，他の物理スレッドは仮想スレッドの割当てを待っている．並列領域の入口でチーム内のスレッド数分の仮想スレッドが作成され，それぞれの物理スレッドに割り当てられ，並列に実行される．コンパイル時に最適化を行いやすくするため，スレッドの割当ては静的に行っている．

図 3 は仮想スレッド（点線の矢印）の物理スレッド（実線の矢印）への割当てを示す．8 本の OpenMP スレッドからなるチームを，4 つずつのスレッド（CPU）を持つ 2 つのノードで実行する場合を表している．仮想スレッド 0 はノード 0 の物理スレッド 0 に割り当てられ，以下ノード 0 内のスレッドから順に仮想スレッドが割り当てられる．これにより，並列ループのスケジューリングが `static` であれば，ループの各反復がどのノードおよびスレッドで実行されるかをコンパイル時に特定できるようになる．

4.3 同期機構

同期機構としては，排他制御のためのロック機構と，同一並列領域内のスレッド間のバリア同期機構を，リモートコピーとノード内のロック機構を組み合わせて実装した．ノード間でのロックの要求や解除のための通信は，前述のメッセージ通信を用いて実現した．バリアとロックがあれば，OpenMP のすべての同期構文が実現できる．

バリアは，極性バリア[11] のアルゴリズムを用いて，ストアの代わりにリモートコピーを用いて実装した．これは本来 SMP 用のアルゴリズムで，大規模なクラスタでの利用には適していない．

ロックは，それを作成したノードがホームノードとなりロックの管理を行う．ロックの操作はホームノードに対してメッセージを送ることで行う．それらのメッセージの処理は，ホームノードのいずれかのスレッドがポーリングを行ったときに行われる．

ロックの獲得要求時にそのロックが使用中だった場合，ロックの持つキューに獲得要求が追加される．ただし，スレッドがロックを取得しようとするときに，そのロックをすでに同じノードのスレッドが取得している場合，ロックのホームノードへはメッセージを送らず，そのノード内のキューにロックの取得要求を登録する．ロックを保持するスレッドは，それ解放する際にまず同じノード内のキューを調べ，そのロックに対する取得要求があれば，ロックをホームへ返さずに同一ノード内のスレッドへ直接渡す．これにより，ロック操作のためのノード間通信を削減できる．

4.4 共有仮想メモリ

SMP クラスタでの実行の際には，各ノードでそれぞれ異なるプロセスが実行されるため，ノードごとに仮想アドレス空間が存在する．それらの仮想アドレス空間の一部を共有アドレス空間と見なし，ノード間で共有データの一貫性を維持する．同一ノード内の複数のスレッド間では，物理共有メモリを用いてメモリを共有する．共有データの一貫性制御の方法については次節で述べ，この節ではその他の共有仮想メモリに関する機能について述べる．

各プロセスの仮想アドレス空間は，**図 4** に示すように複数のセグメントに分割される．セグメントには，スタック，テキスト，ダイナミックセグメント，静的データ，共有ヒープ，プライベートヒープがある．このうち，スタックとテキスト，ダイナミックセグメント（共有ライブラリやスレッドのスタックに利用される）は従来のプロセスと同様に配置され，ノード間で共有はされない．プライベートヒープは従来のプロセス単位のヒープであり，これも共有されない．

静的データと共有ヒープは共有データであり，ソフ

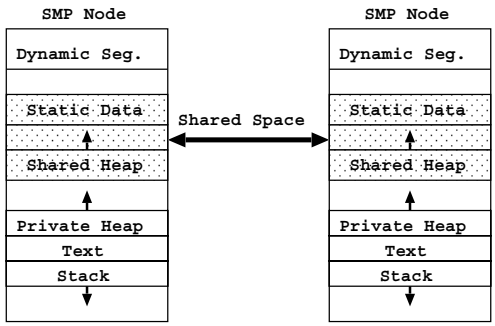

図 4 分散共有メモリのメモリレイアウト
Fig. 4 Memory layout of the distributed shared memory.

トウェアによりノード間の一貫性を維持する．これらの領域の先頭は特定のアドレスに配置され，共有データのアドレスがすべてのノードで同一になるようにする．これは，リンク時にセグメントの先頭アドレスを指定することにより行う．共有ヒープはノード間で共有可能なデータを動的に確保する際に用いられる．従来の動的メモリ管理の関数群（`malloc()` や `free()`）に対応して，共有ヒープを用いた動的メモリ管理のための関数群を用意した．さらに，ページのホームの割当ても行える．共有ヒープの割付けと解放は，単一のノード（ノード 0）で集中管理する．なお，ヒープ管理のアルゴリズムは，Doug Lea の方法[12] をもとにした．

OpenMP ではユーザが動的に確保したデータはつねに共有データと見なされる．そのため，コンパイル時に OpenMP プログラム中の動的メモリの割付けや解放の関数呼び出しを，対応する共有ヒープの割付けや解放の関数の呼び出しに書き換える．しかし，実行時ライブラリが内部で使用するデータおよびスレッド間で共有されないことがコンパイル時に検出できるデータについては，プライベートヒープに割り付ける．

4.5 一貫性制御

この節では，共有データの一貫性制御をソフトウェアで行うための機構について述べる．静的データと共有ヒープからなる共有アドレス空間に配置されたデータは，基本的には，ノード間の一貫性を維持する．OpenMP では緩いメモリコンシステンシモデルを採用しており，`flush` 操作が行われたときにそのスレッドに関して一貫性が保証される必要がある．

共有アドレス空間は，ページとラインと呼ぶ単位に分割して管理する．ページは固定長（4 KB）で，ページを単位にホームノードを定める．ここでいうページは，仮想記憶システムでいうページとは独立である．ホームノードとは，その共有データの最新の値を持っていると見なされるノードである．ただし，共有データの書き込みは直後の `flush` 操作までに完了すればよいため，真に最新の値を持っているとは限らない．各プロセスの仮想アドレス空間内の共有アドレス空間にあるデータは，そのノードがホームであるページに関してはオリジナルのデータであり，ホームでないページのデータはコピーであると見なされる．本稿の実験に用いた実装では，すべてのページのホームノードはノード 0 とした．また，現在の実装ではディレクトリベースのプロトコルではないため，ホーム以外のすべてのノードがコピーを持っている．

ページはさらに，ラインと呼ぶ固定長（64 バイト）の領域に分割される．各ノードはラインごとにそのラインが最新のデータを持っているかを示す状態フラグを持っている．この状態フラグの値をもとに，必要に応じて共有データのコピーの内容がライン単位で更新される．後述するように，連続した領域の参照に対しては，そこに含まれるラインの一貫性制御を一括して行うことができる．そのため，ラインの定数倍の可変長ブロックを単位に一貫性制御を行っていると考えることもできる．

共有データを書き換えた場合には，ライン単位ではなく書き換えた部分だけをホームノードに書き戻す．OpenMP ではデータレースが生じるようなプログラムの動作は不定となるため，書き換えた部分を他のスレッドで同時に参照することはないと仮定してよく，マルチプルライタプロトコルが容易に実現できる．

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は，チェックコードとも呼び，次の 3 種類の基本操作がある．

読み込み前チェック 読み込もうとする共有データの状態をチェックし，最新の値を持っていない場合にはそのデータのホームから最新の値をリモートコピーにより取り寄せる．このときの更新は読み込むデータだけでなくそれを含むライン全体に対して行う．

書き込み前チェック 書き込もうとする共有データの含まれるラインの一部しか書き換えない場合，書き換え後にそれを含むラインの全体が最新の値を持つように，必要があればそのラインを最新の値に更新する．あるラインの全体を書き換える場合，もとの値は不要なので更新する必要はない．

書き込み後チェック 書き換えた共有データの値をそのホームにコピーし，書き換えたノードとホームノード以外の持つコピーを無効化する．

なお，これらの各操作の最初に，参照するデータが共

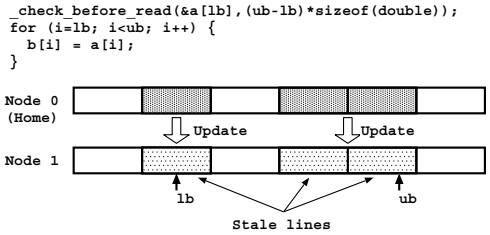

図 5 ライン単位の一貫性制御
Fig. 5 Coherence control based on the DSM lines.

有データか否かをアドレスをもとに判定し，共有データでない場合は何もしないようにする．これは，ポインタ間接参照の場合には参照するデータが共有データか否かがコンパイル時には必ずしも確定できないので，共有データを参照する可能性がある場合にはつねにチェックコードが挿入されるためである．

図 5 を用いて，読み込み前チェックの動作を説明する．この図は変換後のプログラムを示し，`for` 構文でワークシェアされたループの `lb` 番目から `ub-1` 番目までの反復を実行する．`lb` と `ub` の値はスレッドごとに異なる．ここではノード 1 で読み込み前チェックを行う場合を考える．ループの前の関数（実はマクロ）で配列 `a` の `lb` 番目から `ub-lb` 個の要素の読み込み前チェックを一括して行う．`lb` 番目と `ub` 番目の要素がラインの先頭ではない場合，これらを含むライン全体が一貫性制御の対象となる．図では，`lb` から `ub-1` までの要素が 4 つのラインにまたがっている場合を示す．これらのラインは連続しているため，それらの状態のチェックは一括して行う．このうち 1，3，4 番目の 3 つのラインが無効であったとすると，1 番目のラインの更新と，3 番目と 4 番目の 2 つのラインの更新の 2 度の通信が行われる．

読み込み前と書き込み前のチェックでデータの更新が生じた場合には，ホームノードからのリモートコピーが終了するまで後続の処理はブロックされる．それに対して書き込み後のチェックは，書き込みの他のノードへの通知は直後の `flush` 操作までに完了すればよいため，以下のような最適化を行った．

(1) 書き込み直後には書き変えた領域のアドレスとサイズからなる書き込み通知をいったんバッファにため，バッファが一杯になるか `flush` 操作を行うまで他ノードへの通信を遅らせ，バッファ内の書き込み通知を一括して転送する．
(2) バッファに書き込み通知を追加する際には，すでにバッファに同じ領域または連続する領域の書き込み通知が登録されていないかを調べ，登録されている場合にはそれとマージする．

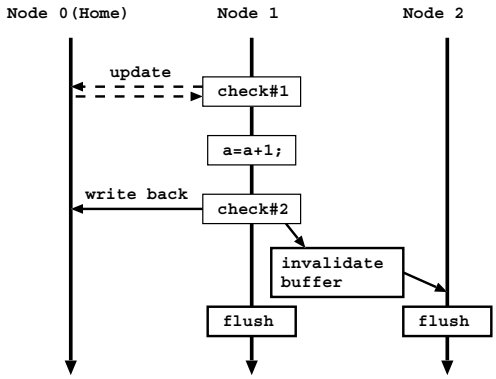

図 6 一貫性制御コードの動作
Fig. 6 Behaviour of the coherence operations.

これらの最適化によって，コンパイル時に最適化できなかった場合でも，共有データの書き込みにともなう通信量を削減する機会が得られる．

図 6 は一貫性制御コードの動作を示す．ノード 1 で共有データ `a` の値を更新する場合，その前と後にチェックコードを実行する．参照前のチェック Check♯1 は，前述の読み込み前チェックと書き込み前チェックの両方を含み，`a` の値が無効であったときにホームから最新の値をコピーする（そのために 1 往復の通信が必要）．参照後のチェック Check♯2 は，前述の書き込み後チェックであり，ホームノードへ最新の値を書き戻すとともに，ノード内の無効化バッファに書き込み通知を登録する．書き込み通知は直後の `flush` までの間に他のノードへ送信され，相手のノードのいずれかのスレッドがポーリングを行ったときに受信され，古くなったコピーが無効化される．

これまで述べた 3 種類のチェックコードは最小限必要な基本的な機能を提供しており，これらのみでも実行時の一貫性制御が行える．しかし，最適化を行う際にはよりオーバヘッドの少ないチェックコードも使用する．それらのチェックコードについては，次章の最適化の説明の中で述べる．

5. OpenMP コンパイラ

5.1 概　　要

我々の OpenMP コンパイラは，OpenMP プログラムを実行時ライブラリの呼び出しを含むマルチスレッドプログラムに変換する．さらに，動的記憶割付けに関する関数の置換と，共有データの一貫性制御のためのコードの挿入と最適化を行う．この節では，SMP 向けのコンパイルと異なる点を中心に OpenMP コン

```
double a, x[N], y[N];
int i;
#pragma omp parallel for
for (i = 0; i < N; i++)
  y[i] += a * x[i];
```

図 7 daxpy プログラムの OpenMP による並列化
Fig. 7 Parallelization of daxpy code with OpenMP.

```
int _dompc_thread(void **args) {
  double *a, *x, *y;
  a = (double *)args[0];
  x = (double *)args[1];
  y = (double *)args[2];
  _barrier();
  _static_schedule_init(0,N,1);
  _get_chunk(&lb, &ub);
  for (i = lb; i < ub; i++) {
    _check_before_read(a, sizeof(double));
    _check_before_read(x+i, sizeof(double));
    _check_before_read(y+i, sizeof(double));
    _check_before_write(y+i, sizeof(double));
    y[i] += (*a) * x[i];
    _check_after_write(y+i, sizeof(double));
  }
  _barrier();
}
```

図 8 daxpy プログラムの変換後のスレッド本体（最適化なし）
Fig. 8 Thread body of the translated daxpy code without optimization.

パイラの機能を説明する．

簡単な OpenMP プログラムの例として，daxpy ループを**図 7** に示す．このプログラムでは `for` 指示文を用いて DOALL 型の並列化をするように指示している．また，変数 `a`，`x`，`y` はスレッド間で共有され，ループ制御変数 `i` はプライベート変数になる．

OpenMP コンパイラは並列領域から各スレッドが実行するための関数を作成する．最適化を行わずに変換した場合のスレッド本体のコードを**図 8** に示す．ここで，下線で始まる関数はコンパイラが生成した関数か実行時ライブラリの関数（あるいはマクロ）を表す．なお，このスレッド本体を呼び出す側の関数では，非共有領域に割り付けられたデータを共有する場合に，共有ヒープに割り付け直す必要がある．

各スレッドは親スレッドから引き継がれる共有データのアドレスの配列を引数として受け取り，共有データを参照するためのポインタを初期化する．ただし，スレッド本体の関数から直接参照可能な共有データ（外部変数など）はポインタを渡す必要はない．並列領域内での共有データの参照には，前節で述べたチェックコードを挿入する．関数 `_check_before_read()` は読み込み前のチェックである．スカラ変数 `a` と配列 `x` の読み込みに対しても読み込み前チェックが挿入される．配列 `y` の要素は読み込んだ直後に書き込まれるため，書き込み前と書き込み後のチェックも生成される．

```
_check_before_read(a,sizeof(double));
_check_before_read(x+lb,(ub-lb)*sizeof(double));
_check_before_read(y+lb,(ub-lb)*sizeof(double));
_check_before_write(y+lb,(ub-lb)*sizeof(double));
for (i = lb; i < ub; i++) {
  y[i] += (*a) + x[i];
}
_check_after_write(y+lb,(ub-lb)*sizeof(double));
```

図 9 同期区間内で最適化した並列ループ
Fig. 9 Optimized parallel loop within synchronization interval.

この例での SMP 向けのコードとの違いは，非共有領域にあるデータを共有する場合に共有ヒープに割り付け直すことと，共有データの参照に対してチェックコードを挿入することである．例に示したコードでは，チェックコードを配列要素の参照ごとに挿入しているため，チェックコードがループの繰返しごとに実行され，個々のチェックのオーバヘッドが小さいとしても，一貫性制御のためのオーバヘッドは性能を大きく低下させる．そこで，コンパイラで一貫性制御コードの最適化を行うことが必須となる．従来のコンパイラ支援ソフトウェア DSM では，メモリ同期点にはさまれた同期区間内でのみ最適化を行っていた．しかし，我々の方法では同期点を越えた最適化を可能にする．以下では，まず従来の最適化手法について述べたのち，同期点を越えた最適化について述べる．

5.2 同期区間内の最適化

制御フローが連結な部分プログラムで，途中にメモリ同期点（`flush` 操作）を含まず，入口と出口がメモリ同期点であるようなものを**同期区間**（synchronization interval）と呼ぶ．従来の同期区間内の解析のみに基づいた基本的な最適化には，以下のものがある[9),10)]．

チェックコードのマージ：連続した領域の参照に対するチェックを 1 つのチェックにまとめる．

チェックコードのループ外移動：ループ不変変数のチェックをループ外に出す．

チェックコードの冗長性削除：同期区間内で複数回参照される変数のチェックを 1 つにまとめる．

このうちチェックコードのマージは空間局所性を利用した最適化であり，ループ外移動と冗長性削除は時間局所性を利用した最適化である．

前述の daxpy のプログラムでは並列領域内でメモリ同期が行われないため，共有データの値の一貫性は，並列領域の最初と最後でのみ保証されればよい．そのため，ループ中のチェックをすべてループ外に出して，一括して行うことができる．そうして得られたコードを**図 9** に示す．

上記のような従来の最適化手法は，一貫性制御コードの実行回数や通信量を削減することができるが，メ

モリ同期点ですべての共有データが同期されることは変わらない．そのため，以下のような問題がある．

- 参照されることのないデータの更新や無効化のために余分な通信が生じる．特に，データ配置（ホームノード）が最適化されていない場合にその傾向が顕著になる．
- すでに最新のコピーを持っている場合でもチェックコードが実行されるため，冗長なチェックが生じる．
- データの更新時に読み込み側で通信を起動するため，通信と計算のオーバラップができない．

次節で述べる最適化はこれらの問題を解決することを目的としている．

5.3 同期点を越えた最適化

従来のコンパイラ支援ソフトウェアDSMでは，メモリ同期点を途中に含まない区間でのみ一貫性制御コードの最適化を行っていた．しかし，我々は，スレッド間のデータ依存関係を解析することにより，同期点を越えた最適化を行う．

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**（Parallel Dataflow Analysis; PDA）と呼ぶ．並列データフロー解析では同一スレッド内のデータの流れに加えて，異なるスレッドとの間のデータの流れを考慮した解析を行う．OpenMPプログラムでは緩いメモリコンシステンシモデルを採用しているために，スレッド間にまたがったデータの流れは，書き込み側スレッドと読み込み側スレッドのメモリ同期点（`flush` 操作）を必ず通る．そのため，メモリ同期点でのみスレッド間のデータの流れを考慮すればよいため，効率の良い解析が可能となる．

PDAにより，明示的な `flush` 操作と，暗黙の `flush` 操作に対して，そこで同期の必要となる変数を検出でき，同期点を越えた最適化が可能になる．ただし，変数を指定した `flush` 指示文では，指定された変数を必ず同期し，他の変数に対する同期は行わない．また，`volatile` 属性を持つデータはすべてのプログラム点で同期する必要があるため，最適化の対象としない．

並列データフロー解析によって，以下のような情報を求められる．

- スレッド間の到達定義やDef-Useチェイン
- メモリ同期点で同期の必要な共有データ
- 並列ループ間にまたがるスレッド間データ依存
- 単一のスレッドにしか参照されない共有データ

これらの情報の求め方は，文献13)，14)を参照していただきたい．本稿では，解析法よりも最適化の効果を主に述べる．

上記のような並列データフロー情報を用いることで，一貫性制御コードに以下のような最適化が行える．

不要一貫性制御コードの削除 直前あるいは直後のメモリ同期点で同期の不要な共有データに対する一貫性制御コードを削除する．必要なメモリ同期操作が他のメモリ同期点で行われることが保証されているために一貫性制御コードの削除が可能となる．並列領域内あるいは並列ループ内で書き換えられないデータの読み込みと，特定のスレッドにしか参照されない共有データに対する一貫性制御の削除は特に効果が大きい．

明示的コピーによる更新 共有データを書き換えたスレッドとその値を参照するスレッドが確定している場合に，スレッド間で共有データを直接リモートコピーする．これにより状態フラグを用いた実行時のチェックが削除されるとともに，そのデータを参照しないノードに対する一貫性制御が削減される．さらに，書き込み側スレッドで通信を起動することにより，通信と計算のオーバラップが可能になる．

従来のデータ並列言語のコンパイル手法でも参照パターンに応じた通信の最適化は行われていたが，OpenMPよりも単純な制御構造を対象としていた（たとえば文献15)）．我々の方法は，orphan指示文を含む様々なOpenMP指示文が混在したプログラムにも適用できる．

このような並列領域全体にわたるデータフロー解析により，コンパイル時に参照パターンの正確に解析できる共有データに対しては，必要最小限のオーバヘッドで一貫性を保証できる．一方，参照パターンの正確に解析できない共有データに対しては，実行時の状態フラグのチェックにより必要な場合にのみ通信を行うことで，実行時のオーバヘッドが過大にならないようにできる．

並列データフロー情報を用いた最適化の例として，**図10** のラプラス方程式を解くプログラム（以下，Laplaceと呼ぶ）を考える．このプログラムはステンシル計算を行っており，共有される行列 `u`，`uu` は，実際には境界部分の要素以外は単一のスレッドにしか参照されない．

従来の最適化手法では，行列 `u`，`uu` に対する連続領域の参照のチェックのマージと，ループ長 `n` のチェックのループ外移動が適用できる．しかし，配列 `u`，`uu` は縁を除く全要素が毎回書き換えられるため，それらの要素への無効化とホームノードへの書き戻しが毎回行われることになる．従来の最適化では，チェックコー

```
double u[N+2][N+2], uu[N+2][N+2];
double err;
int n;
#pragma omp parallel
{
  int i,j,k;
  double err_local, tmp;
  do {
#pragma omp for nowait
    for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
        uu[i][j] = u[i][j];
    err_local = 0.0;
#pragma omp single
    err = 0.0;
#pragma omp for nowait
    for (i=1; i<=n; i++)
      for (j=1; j<=n; j++) {
        u[i][j] = (uu[i-1][j]+uu[i+1][j]+
                   uu[i][j-1]+uu[i][j+1])/4.0;
        tmp = fabs(u[i][j]-uu[i][j]);
        if (tmp>err_local) err_local = tmp;
      }
#pragma omp critical
    if (err_local>err) err=err_local;
#pragma omp barrier
  } while (err>1.0e-5)
}
```

図 10 ラプラス方程式を解く OpenMP プログラム（Laplace）
Fig. 10 Laplace equation solver.

```
for (i=lb; i<=ub; i++)
  for (j=1; j<=n; j++)
    uu[i][j] = u[i][j];

/* update copy on the previous node */
if (_is_first_thread_on_node()
    && _my_node_no > 0) {
  _update(&uu[lb][1],sizeof(double)*n,
          _my_node_no-1);
}
/* update copy on the next node */
if (_is_last_thread_on_node()
    && _my_node_no < _n_nodes-1) {
  _update(&uu[ub][1],sizeof(double)*n,
          _my_node_no+1);
}
_barrier();
```

図 11 参照パターンを利用した最適化を行ったコード
Fig. 11 Translated code with optimizations using sharing-pattern information.

ドの実行回数は削減できるが，参照されるすべての共有データが実行時の一貫性制御の対象となる．

図 11 は，Laplace の第 1 の並列ループに並列データフロー情報を用いて最適化を行って得られたコードを示す．PDA により，以下のように最適化が行われる[13]．まず，ループを分割した各チャンクの境界の反復で参照されるデータのみスレッド間のデータ依存が存在することが分かる．さらに，4.2 節で述べた静的なスレッドの割当てを行うと，チャンク間のデータ依存からスレッド間およびノード間のデータ依存を求められる．よってこの例では，隣接するスレッドが別のノードにあるときに，そのノードのみ値を更新するコードを生成すればよい．複数のスレッドで参照されない配列要素や，並列領域で書き換えられない変数に対しては，一貫性制御は行わない．配列 u に関しては，各要素の値の定義と使用がつねに同じスレッドで行われるため，一貫性制御は行わない．

6. 性能評価

6.1 プラットフォーム

PrentiumPro を用いた SMP クラスタ COMPaS[16] を用いて評価を行った．各ノードは 200 MHz の PentiumPro プロセッサを 4 個持ち，それらが共有バスを介して接続されている．そのようなノードが 8 個，Myrinet を介して接続されている．OS は Solaris 2.5.1 であり，ノード内のマルチスレッドライブラリとして Solaris スレッドを用いた．ノード間の通信は我々の開発した通信ライブラリ NICAM[8] を用いて行う．NICAM はリモートコピーと，ノード間のバリアをサポートしている．

6.2 ベンチマークプログラム

性能評価には，メモリ参照の特徴の異なる 2 つのプログラムを用いた．1 つ目の Laplace は，図 10 に示したプログラムで，ラプラス方程式の陽解法を反復法で解く．行列のサイズは 2048×2048 とし，最外側ループ（図 10 の do ループ）の反復回数は 20 回である．

もう 1 つのプログラム（以下，Jacobi と呼ぶ）は，係数行列が密行列である連立一次方程式 $Ax = b$ を Jacobi Over-relaxation 法で解く．行列のサイズは 4096×4096，最外側ループの反復回数は 11 回である．

これらはいずれも連立一次方程式の反復解法のプログラムであるが，性質は異なっている．Laplace では 2048×2048 の行列 2 個（u，uu）が最外側ループの反復ごとに毎回書き換えられる．それに対して Jacobi では，4096×4096 の行列 A とベクトル y は並列領域中では書き換えられず，ベクトル x の新旧の値が書き換えられるのみである．

これらのプログラムを前節で述べた 2 通りの最適化を行ったコードに人手で変換し，実行時ライブラリのプロトタイプを用いて性能を評価した．

6.3 実行性能と考察

図 12 は，Laplace に対して，ノード数を 1 から 8，各ノード内のプロセッサ数を 1 から 4 と変化させたときのプログラムの実行時間を示す．たとえば，2×4 は 2 つのノードで 4 スレッドずつ，合計 8 スレッドで実行した場合を意味する．同期区間内の最適化のみ行った場合（PDA なし）は，プロセッサ数を増加してもあまり性能が向上しないのに対して，同期点を越えて最適化した場合（PDA あり）は性能が大幅に改善さ

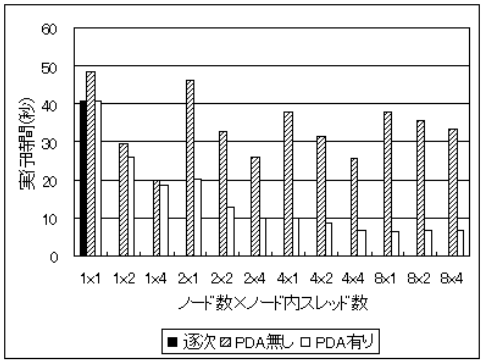

図 12 Laplace の実行時間
Fig. 12 Execution times of the Laplace program.

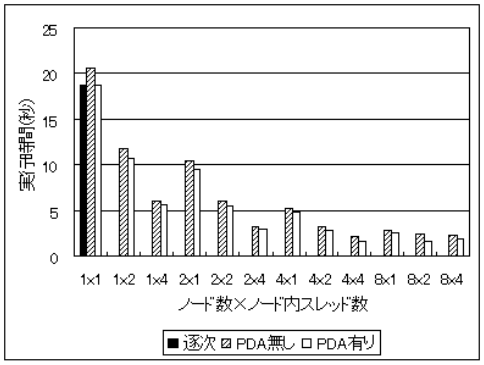

図 13 Jacobi の実行時間
Fig. 13 Execution times of the Jacobi program.

れていることが分かる．

PDA なしの場合，1×1 での実行時間は逐次コンパイルして実行した場合（逐次）よりも 20%も増加している．それに対して，PDA ありの場合は 0.1%しか実行時間が増加していない．

2 ノード以上では，共有データを書き込む際の無効化とホームノードへの書き戻しの時間が実行時間の差として現れている．SMP クラスタではノード内のスレッドどうしでは物理共有メモリを用いることができるため，通信オーバヘッドの大きい PDA なしのグラフを見ると，プロセッサ数が同じならばノード数よりもノード内プロセッサ数を増加する方が良い性能が得られる（たとえば，4×2 より 2×4 の方が実行時間が短い）．しかし，通信オーバヘッドが削減された PDA ありの場合は，特に 8 ノードでノード内の性能向上が見られない．これは `critical` 構文がメッセージ通信を用いたロックで実装されているために，そこで逐次化されることによる．バリアとロックを合わせた同期のオーバヘッドは，8×4 では実行時間の 50%程度になる．また，実行時間は `critical` 構文に用いるロックを取得する順序に依存し，実行時間は 10%以上変動する．ロックの所有者がノード間で移動する場合には通信が発生するが，同一ノードの他のスレッドが所有しているロックを取得する場合には通信は発生しないためである．

Laplace で `critical` 構文を用いていた処理は OpenMP ではサポートされていない最大値を求める演算のリダクション計算である（Fortran 版 OpenMP ではサポートされている）．これを `reduction` 節の実装に用いているロックを使わないアルゴリズムで書き換えると，8×4 では約 60%性能が向上し，実行時間の変動もわずかになった．

次に Jacobi の実行時間を**図 13** に示す．Jacobi では共有データの参照の大半が読み込みであり，PDA なしの場合でも一貫性制御のための通信はわずかしか生じない．そのため PDA なしと PDA ありとの差は Laplace ほど大きくはない．これは，PDA なしの場合でも通信はほとんど発生していなかったため，PDA ありの場合にチェックコードの命令実行オーバヘッドは削減されるが，通信オーバヘッドの削減の効果は小さいためである．

Jacobi で 8×4 のときに性能が低下しているのは，Laplace と同様にバリアとロックのオーバヘッドが実行時間の約 60%と大きいためである．

以上から，本稿で述べたコンパイラ支援ソフトウェア DSM により，規則的なデータ参照パターンを持つプログラムでは，良好な性能を得られることが分かる．また，共有データの読み込みの多いプログラムでは従来のコンパイラ支援ソフトウェア DSM で行われた同期区間内の最適化のみでも良い性能が得られるが，書き込みの多いプログラムでは並列データフロー解析に基づいた一貫性制御コードの最適化が有効であることが分かる．

一方，不規則なデータ参照を行うプログラムでは，本稿に述べた方法だけで良好な性能を得るのは難しい．そのようなプログラムでは，連続領域あるいは同一データの参照などをコンパイル時に検出するのが困難なため，チェックコードの最適化が行いにくい．そのため，命令実行オーバヘッドが増加するとともに，細粒度のデータ転送が頻発する．ただし，並列領域内で書き換えられないデータは一貫性制御を削除することができる．また，不規則な参照でも読み込みが中心であれば通信はわずかしか発生しないため，性能への影響は少ない．不規則な書き込みの多いプログラムでは，インスペクタ/エグゼキュータ法などの動的な手法が有効と思われる．

今回用いた実行時ライブラリでは，すべてのノードが共有データ全体のコピーを持つため，参照したノードのみがコピーを持つ方式と比べると，無効化のオーバヘッドが大きい．そのため，ホームノードの最適化や，ディレクトリを用いたプロトコルによって参照したページのみコピーを持つように変更することでも性能を改善できる．しかし，そのような最適化を行っても，動的な一貫性制御のみを行った場合はチェックコードの実行のオーバヘッドが生じるため，チェックコード自体を削減できる本方式と同等の性能は得られないと考えられる．また，ロックとバリアのアルゴリズムについては，今後改善していく．

7. 関連研究

ソフトウェア DSM と OpenMP の実装について，本研究と関連する研究と比較する．

7.1 ソフトウェア DSM

分散メモリシステム上で共有メモリプログラミングモデルを実装する方法には，ソフトウェア DSM のような実行時に一貫性制御を行う動的な方法のほかに，HPF[4] のような静的な方法がある．

HPF ではコンパイル時に共有アドレス空間への参照を各ノードの局所アドレス空間への参照に変換し，メッセージパッシングプログラムを生成する．しかしこの方法では，コンパイル時にメモリアクセスパターンの特定できない場合に，オーバヘッドが大きくなる．インスペクタ/エクゼキュータ法によってオーバヘッドを削減できる場合もあるが，適用範囲は限られている．

TreadMarks[17] や SCASH[18] などの仮想記憶システムを利用したソフトウェア DSM では，局所アドレス空間の一部を共有アドレス空間と見なし，実行時に各ノードの共有アドレス空間の一貫性をソフトウェアによって維持する．それによってクラスタ上で共有メモリプログラムを実行することが可能になるが，ハードウェア DSM に比べて一貫性制御のオーバヘッドが大きく，データ配置などの最適化を行わないと良好な性能は得られにくい．また，粗粒度（ページ単位）で一貫性制御を行うため，フォルスシェアリングが生じやすい．

ソフトウェア DSM の別の実現法として，コンパイラによって一貫性制御コードを挿入する方法も研究されている[9),10),19)~23)]．これらの方法では，コンパイラがプログラムを解析して一貫性制御コードの最適化を行うことで，一貫性制御のオーバヘッドを削減することができる．しかし，これまでの研究では一貫性制御コードの最適化を同期区間内でしか行っていなかった．

本稿で述べたコンパイラ支援ソフトウェア DSM では，他のソフトウェア DSM のように動的な一貫性制御を行うだけでなく，データの参照パターンがコンパイル時に正確に分かるものについては，必要な通信を明示的に行うメッセージパッシング的なコード生成を行うことができ，動的な一貫性制御と静的な一貫性制御を併用しているといえる．さらに，並列データフロー解析を行うことによって，従来のコンパイラ支援ソフトウェア DSM よりも積極的な最適化が可能となる．

7.2 OpenMP の実装

OpenMP はこれまで主に SMP と CC-NUMA で用いられているが，クラスタ向けの実装についての研究もいくつかある．Hu ら[24] と Scherer ら[25] は，TreadMarks を用いた OpenMP の実装について報告している．Sato ら[26] は SCASH 上で実装した．これらはいずれも仮想記憶システムに基づいたソフトウェア DSM であり，局所性の高いアプリケーションでは良好な性能が得られるようである．これらの方法と，我々のコンパイラ支援ソフトウェア DSM との性能比較は今後の課題である．

DSM 向けの OpenMP の実装で良好な性能を得るには，データ配置の最適化が重要である．そのため，OpenMP にデータ配置あるいはデータ分散の指示文を追加することも提案・実装されている．Bircsak ら[27] は DSM 向けの指示文の拡張について提案している．SGI MIPSpro コンパイラではデータ配置指示文をサポートしている．これらのデータ配置指示が有効であることは明らかであるが，汎用的な仕様を策定できるか否かはまだ明らかではない．また，本稿で述べたような共有パターンの情報を利用して，データ配置の最適化を行うことも考えられる．

8. まとめと今後の課題

並列処理の普及のためには，様々なプラットフォーム上でシームレスに利用できる並列プログラミング環境が必要である．そのためのプログラミングモデルとしては共有メモリモデルが望ましい．SMP クラスタ上で共有メモリモデルを実現する一手法として，OpenMP プログラムを対象としたコンパイラ支援ソフトウェア DSM を提案した．

コンパイラ支援ソフトウェア DSM では，ソースプログラムを解析することでアプリケーション固有の特徴を検出し，プラットフォームに応じた最適化を行うことが可能である．さらに OpenMP では構造的な並列性記述と緩いメモリコンシステンシモデルのために，効果的な最適化を効率良く行うことができる．プログ

ラマは特定のプラットフォームを仮定せずに（ただし，局所性などは考慮して）プログラムの記述を行い，コンパイラでプラットフォームに応じた最適化を行うようにすることで，性能可搬性のあ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．

今後は，開発中のコンパイラを用いてより多くのプログラムに対する詳細な性能評価を行っていくとともに，性能改善の手法を研究していく．本稿ではコンパイラによる一貫性制御コードの最適化に重点をおいたが，実行時ライブラリの改良による性能向上の余地も大きい．また，SMP，CC-NUMA，ページベースソフトウェア DSM などの他のプラットフォーム向けの OpenMP の実装での，コンパイラによる最適化も検討していく．

謝辞 日頃，ご指導・ご討論していただいている，新情報処理開発機構つくば研究センタの皆様，および電子技術総合研究所・筑波大学・東京工業大学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．さらに，有益なコメントをいただいた査読者の方々に感謝いたします．

参 考 文 献

1) Protic, J., Tomasevic, M. and Milutinovic, V.: Distributed Shared Memory: Concepts and Systems, *IEEE Parallel & Distributed Technology*, Vol.4, No.2, pp.63–79 (1996).
2) E. Culler, D. and Singh, J.P.: *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann (1999).
3) OpenMP ARB: *OpenMP: Simple, Portable, Scalable SMP Programming*. `http://www.openmp.org/`.
4) Koelbel, C.H., Loveman, D.B., Schreiber, R.S., Steele Jr., G.L. and Zosel, M.E.: *The High Performance Fortran Handbook*, The MIT Press (1994).
5) OpenMP ARB: *OpenMP C and C++ Application Program Interface Version 1.0* (1998).
6) OpenMP ARB: *OpenMP Fortran Application Program Interface Version 1.1* (1999).
7) Adve, S.V. and Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial, *IEEE Comput.*, Vol.29, No.12, pp.66–76 (1996).
8) 松田元彦，田中良夫，久保田和人，佐藤三久：SMP クラスタ向けネットワーク・インタフェース AM 通信，情報処理学会論文誌，Vol.40, No.5, pp.2225–2234 (1999).
9) Scales, D.J., Gharachorloo, K. and Thekkath, C.A.: Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, *Proc. 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.174–185 (1996).
10) 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最適化コンパイル技法の評価，情報処理学会論文誌，Vol.39, No.6, pp.1729–1737 (1998).
11) Real World Computing Partnership: *Barrier Collection*. `http://www.rwcp.or.jp/lab/mpperf/barrier-collection/`.
12) Lee, D.: *A Memory Allocator*. `http://gee.cs.oswego.edu/dl/html/malloc.html/`.
13) 佐藤茂久，草野和寛，佐藤三久：OpenMP コンパイラにおける一貫性制御の最適化，2000 年記念並列処理シンポジウム（JSPP2000）論文集，pp.221–228 (2000).
14) 佐藤茂久，草野和寛，佐藤三久：OpenMP 並列プログラムのデータフロー解析手法，情報処理学会研究報告，00-HPC-82, pp.71–76 (2000).
15) Pugh, W. and Rosser, E.: Iteration Space Slicing and Its Application to Communication Optimization, *Proc. 1997 Int'l Conf. on Supercomputing*, pp.221–228 (1997).
16) Sato, M.: COMPaS: A PC-based SMP cluster, *IEEE Concurrency*, Vol.7, No.1, pp.82–86 (1999).
17) Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W. and Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Comput.*, Vol.29, No.2, pp.18–28 (1996).
18) 原田 浩，手塚宏史，堀 敦史，住元真司，高橋俊行，石川 裕：ソフトウェア分散共有メモリシステムにおけるページ転送方式の比較，情報処理学会論文誌，Vol.41, No.5, pp.1410–1419 (2000).
19) Scales, D.J. and Gharachorloo, K.: Design and Performance of the Shasta Distributed Shared Memory Protocol, *Proc. 1997 Int'l Conf. on Supercomputing* (1997).
20) Scales, D.J., Gharachorloo, K. and Aggarwal, A.: Fine-Grain Software Distributed Shared Memory on SMP Clusters, *Proc. 4th Int'l Symp. on High-Performance Computer Architecture* (1998).
21) Chiueh, T. and Verma, M.: A Compiler-Directed Distributed Shared Memory System, *Proc. 9th ACM Int'l Conf. on Supercomputing*, pp.77–86 (1995).
22) Schoinas, I., Falsafi, B., Hill, M.D., Larus, J.R. and Wood, D.A.: Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory, *Proc. Int'l Conf. on Parallel Architecture and Compilation*

Techniques (1998).
23) Inagaki, T., Niwa, J., Matsumono, T. and Hiraki, K.: Supporting Software Distributed Shared Memory with an Optimizing Compiler, *Proc. 1998 Int'l Conf. on Parallel Processing* (1998).
24) Hu, Y.C., Lu, H., L.Cox, A. and Zwaenepoel, W.: OpenMP for Networks of SMPs, *Proc. 13th Int'l Parallel Processing Symp. and 10th Symp. on Parallel and Distributed Processing* (1999).
25) Scherer, A., Lu, H., Gross, T. and Zwaenepoel, W.: Transparent Adaptive Parallelism on NOWs using OpenMP, *Proc. 7th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pp.96–106 (1999).
26) Sato, M., Harada, H. and Ishikawa, Y.: OpenMP Compiler for a Software Distributed Shared Memory System SCASH, *Proc. Workshop on OpenMP Applications and Tools* (2000).
27) Bircsak, J., Craig, P., Crowell, R., Harris, J., Nelson, C. and Offner, C.D.: Extending OpenMP for NUMA Architectures, *Proc. Workshop on OpenMP Applications and Tools* (2000).

(平成 12 年 9 月 13 日受付)
(平成 13 年 3 月 9 日採録)

佐藤 茂久（正会員）

昭和 41 年生．平成元年東京理科大学理学部応用数学科卒業．平成 3 年東京理科大学大学院理学研究科数学専攻修士課程修了．同年（株）日立製作所入社．システム開発研究所にて最適化/並列化コンパイラの研究開発に従事．平成 10 年より新情報処理開発機構に出向．OpenMP コンパイラ，ソフトウェア DSM，SMP クラスタを用いた高性能計算の研究に従事．コンパイラ（プログラム解析，コード最適化），並列処理，マイクロプロセッサ・アーキテクチャ等に興味を持つ．IEEE，ACM 各会員．

草野 和寛（正会員）

昭和 40 年生．平成元年九州大学工学部情報工学科卒業．平成 3 年九州大学大学院総合理工学研究科情報システム工学専攻修士課程修了．同年，日本電気（株）入社．平成 9 年より新情報処理開発機構つくば研究センタに出向．並列化コンパイラ，並列化支援ツール等の研究に従事．

佐藤 三久（正会員）

昭和 34 年生．昭和 57 年東京大学理学部情報科学科卒業．昭和 61 年東京大学大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．平成 3 年，通産省電子技術総合研究所入所．平成 8 年より，新情報処理開発機構つくば研究センタに出向．現在，同機構並列分散システムパフォーマンス研究室室長．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術，グローバルコンピューテイング等の研究に従事．日本応用数理学会会員．