OpenMP向けコンパイラ支援ソフトウェアDSMにおける最適化コンパイル手法

佐藤茂久[†,☆] 佐藤三久[†,☆☆]

本稿は，コンパイラ支援ソフトウェア分散共有メモリシステム Omni/CSDSM における最適化コンパイル手法を示す．Omni/CSDSM は，OpenMP API を用いて記述した共有メモリ並列プログラムを SMP クラスタ上で透過的に実行することができる．コンパイル時に並列性を考慮しながらプログラム全体にわたる解析を行い，共有データの一貫性制御を効率良く行うための最適化を行うことを特徴とする．本稿では新たな最適化手法として，(1) ネストレベルに応じた手続きクローニング，(2) 非共有データの検出，(3) 一貫性制御プロトコルの最適化，(4) アラインメントの最適化，を提案する．さらに，Omni/CSDSM の最適化に必要なデータフロー解析手法と，その際に用いるプログラムの中間表現を示す．本システムを，Pentium II Xeon プロセッサと Myrinet ネットワークを用いた SMP クラスタ上で実装し，特徴の異なる 4 本の OpenMP C プログラムを用いて最適化の効果を評価した．その結果，最大限最適化を行った場合に，8 ノード × 4 スレッド（32 プロセッサ）で逐次実行時の 7.9 倍から 30.0 倍の性能が得られた．

Compiler Optimization Techniques in a Compiler-directed Software DSM for OpenMP

Shigehisa Satoh[†,☆] and Mitsuhisa Sato[†,☆☆]

In this paper, we present compiler optimization techniques in Omni/CSDSM, a compiler-directed software distributed shared memory system for OpenMP. Omni/CSDSM enables transparent execution of shared address space parallel programs using OpenMP API on a PC-based SMP cluster. The OpenMP compiler analyzes flow of data across all source files taking account of the semantics of OpenMP directives, and performs optimization to reduce coherence overhead. We present new optimization techniques for Omni/CSDSM: procedure cloning based on the nest level of parallel region, non-shared data detection, coherence protocol optimization, and alignment optimization. We also show dataflow analysis techniques that enable compiler optimization for Omni/CSDSM, and internal program representations used by the analyses. We implemented Omni/CSDSM on an SMP cluster comprised of Pentium II Xeon processors and Myrinet network, and evaluated its performance using four benchmark programs that have different characteristics. We obtained execution from 7.9 to 30.0 times faster than serial execution using eight 4-way SMP nodes with all compiler optimization techniques.

1. はじめに

我々は，OpenMP API[1] を用いた共有メモリ並列プログラムを SMP クラスタシステム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システム Omni/CSDSM を提案している[2)~6)]．本研究の目的は，コモディティ・プロセッサと高速なネットワークで構成した SMP クラスタシステム上の並列アプリケーションを，共有メモリモデルに基づく API を用いて容易に記述できるようにすることである．それにより，対称マルチプロセッサ（Symmetric Multi-Processors; SMP）や，CC-NUMA（Cache-Coherent Non-Uniform Memory Access）などのハードウェア分散共有メモリシステム，そしてクラスタのような分散メモリシステムで同一のソースプログラムが利用できるようになり，シームレスな並列プログラ

† 新情報処理開発機構つくば研究センタ
Tsukuba Research Center, Real World Computing Partnership
☆ 現在，株式会社日立製作所システム開発研究所
Presently with Systems Development Laboratory, Hitachi, Ltd.
☆☆ 現在，筑波大学計算物理学研究センター
Presently with Center for Computational Physics, University of Tsukuba

ミング環境を提供することができる．また，コンパイラでプラットホームに応じた最適化を行うことで，性能可搬性を得ることも可能になる．

Omni/CSDSM は**ソフトウェア分散共有メモリ**（Software Distributed Shared Memory; SDSM）システムの一種であり，以下の特徴を持つ[5]．

- ハードウェアや OS の支援なしに，ユーザレベルのソフトウェアで分散共有メモリを実現する．
- クラスタの各ノードで実行するプログラムの仮想アドレス空間の一部を共有アドレス空間とし，ノード間で共有データの一貫性を保証する．
- 共有データ領域はページ単位でホームノードを割り当て，ホーム以外のノードは各ページのコピー（キャッシュ）を持つことができる．
- 共有ページ内のデータは，64 バイト単位で細粒度の一貫性制御を行う．
- 共有データの一貫性を制御するためのコードをコンパイラがプログラムに埋め込む．
- コンパイル時にソースレベルの情報を利用して一貫性制御コードの最適化を行う．

このようにコンパイラによって一貫性制御コードを埋め込み，同時に一貫性制御コードの最適化を行う SDSM を，特に**コンパイラ支援ソフトウェア分散共有メモリ**（Compiler-directed Software Distributed Shared Memory; CSDSM）と呼ぶ．これまでに，Locust[7]，shasta[8]，Sirocco-S[9]，ADSM/UDSM[10),11)] などの CSDSM が発表されている．CSDSM では，コンパイラによる最適化が必須かつ効果的に行える点が，TreadMarks[12)] や SCASH[13)] などの仮想記憶システムに基づく SDSM（以下では，ページベース SDSM と呼ぶ）との最大の相違点である．

本稿では，文献 5) では詳しく説明されていない最適化の実現方法を述べるとともに，以下の新たな最適化コンパイル手法を提案する．

(1) ネストレベルに応じた手続きクローニング
(2) 非共有データの検出
(3) 一貫性制御プロトコルの最適化
(4) アラインメントの最適化

これらの最適化を行うのに必要なデータフロー解析手法と，その際に用いる中間表現も示す．

OpenMP では，並列性が構造的に記述されていることや，緩いメモリコンシステンシモデルを採用していることから，並列性を考慮した解析を精度良くかつ効率良く行うことができる．Omni/CSDSM の OpenMP コンパイラは，これらの利点を生かして，ソースプログラム全体にわたって並列性を考慮したプログラム解析を行い，得られた情報を用いて最適化を行う．

通常の SDSM では，共有データの一貫性の保証は実行時にのみ行われる．しかし，我々の方法では，できるだけコンパイル時に共有データの一貫性の保証に必要なノード間通信を特定し，メッセージパッシングプログラムのような明示的なデータ転送を行うことを特徴とする．コンパイル時に解決できなかった場合にのみ，実行時に一貫性を制御するコードを生成する．さらに，実行時に一貫性の保証を行う場合でも，プロトコルやデータ配置（アラインメント）の最適化により，実行時オーバヘッドを削減する．

本システムを，Pentium II Xeon プロセッサと Myrinet を用いた SMP クラスタ向けに実装し，特徴の異なる 4 本のプログラムで最適化の効果を評価した．その結果，すべての最適化を行うと，局所的な最適化のみ行った場合に比べて，最大 2.96 倍の性能向上が得られた．また，逐次実行時に比べて，8 ノード × 1 スレッド（8 プロセッサ）で 4.79 倍から 7.93 倍，8 ノード × 4 スレッド（32 プロセッサ）では 7.90 倍から 30.0 倍の性能が得られた．実験結果から手続き間解析や並列性を考慮したデータフロー解析に基づく大域的な最適化の有効性が確認できた．

本稿は以下のように構成されている．まず，2 章で前提となる Omni/CSDSM の概要と実行時システムについて述べる．次に，3 章で OpenMP コンパイラの構成を述べる．さらに，4 章で OpenMP プログラムの中間表現を示し，5 章で，一貫性制御の最適化手法とそれに必要な解析手法を述べる．最適化の効果を 6 章で評価する．さらに，7 章で関連研究について述べ，最後に本稿のまとめを記す．

2. Omni/CSDSM の概要

2.1 システム構成

Omni/CSDSM は，OpenMP API を用いて記述した共有メモリ並列プログラムを，ハードウェアでは共有アドレス空間をサポートしないクラスタシステム上で，ソースプログラムの変更なしに透過的に実行することを目的としている．基本設計については，文献 5) で述べているので，この章では本稿で述べる最適化手法の説明に必要な点のみ説明する．

図 1 は，コンパイルと実行の手順を示す．まず，OpenMP を用いて記述されたソースプログラム（ベース言語は C とする）を，OpenMP コンパイラで C プログラムに変換し，さらに既存の C コンパイラでコンパイル，実行時ライブラリとリンクして実行形式を得る．入力プログラムは通常の OpenMP プログラム

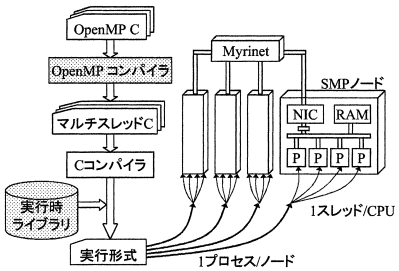

図 1 Omni/CSDSM のシステム構成
Fig. 1 System overview of Omni/CSDSM.

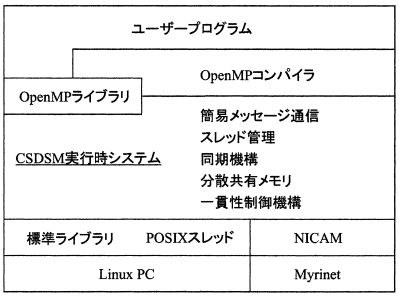

図 2 Omni/CSDSM の実行時システム
Fig. 2 The runtime system of Omni/CSDSM.

で，指示文の制限や拡張は行っていない．OpenMP コンパイラが生成する C プログラムは，Omni/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である．

Omni/CSDSM のターゲットは，Myricom 社の Myrinet によって接続された PC ベースの SMP クラスタである．実行形式を SMP クラスタの各ノードでそれぞれ異なるプロセスとして同時に起動し，各プロセスはノード内のプロセッサ数に応じて既存のスレッドライブラリを用いてスレッド（物理スレッドと呼ぶ）を起動する．各物理スレッドは起動されると，OpenMP のスレッド（論理スレッドと呼ぶ）の割当てを待ち，割り当てられたらそれを実行する．

2.2 実行時システムの概要

Omni/CSDSM の実行時システムは，SMP クラスタ上の細粒度ソフトウェア分散共有メモリシステムと，その上に構築された OpenMP 実行時ライブラリからなる．**図 2** は，実行時システムの構成要素を示す．

オペレーティングシステムとして Linux を用い，ノード内の並列処理には POSIX スレッドライブラリ，ノード間の通信には Myrinet 用のリモート通信ライブラリ NICAM[14] を用いる．これらの既存のシステムソフトウェアの上に，Omni/CSDSM の実行時システムを構築した．

まず，NICAM はリモート書き込み/読み込みと全ノードでのバリア同期しかサポートしないため，リモート書き込みを用いて簡単なメッセージ通信機能を実装した．次いで，リモート書き込みとメッセージ通信を用いて，SMP クラスタ上でのマルチスレッド機構，同期機構（バリア・ロック），分散共有メモリの管理機構と一貫性制御機構を実装した．さらに，その上に OpenMP の実行時ライブラリと指示文の機能を実現するためのライブラリを実装した．実行時システムの詳細については，文献 5) を参照していただきたい．ただし，最適化手法の説明に必要な，分散共有メモリと一貫性制御機構の詳細については，この章の残りで述べる．また，本稿で用いる実装は，文献 5) で示したものよりもバリア性能が改善されるとともに，次節で述べるようにファーストタッチ方式によるホームノードの決定方式がサポートされている．

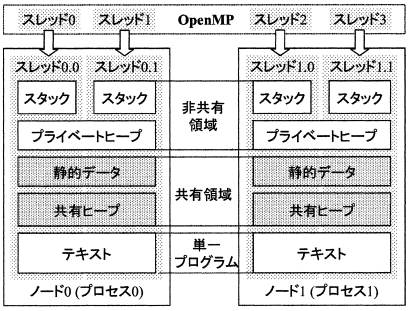

図 3 Omni/CSDSM の仮想アドレス空間
Fig. 3 Virtual address spaces of Omni/CSDSM.

2.3 共有アドレス空間

Omni/CSDSM の実行形式はクラスタの各ノードでそれぞれ独立したプロセスとして起動される（図 1 参照）．そのため，各プロセスは独自の仮想アドレス空間を持っている．そ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，ノード間の一貫性を保証することで，分散共有メモリを実現する．ただし，SMP クラスタの場合，同一ノード内のスレッド間では物理共有メモリを用いてデータを共有する．

仮想アドレス空間は，用途ごとにいくつかのセグメントに分割される．**図 3** は Omni/CSDSM のアドレス空間の構成を示す．逐次プログラムの場合に存在するセグメントのほかに新たに**共有ヒープ**を設け，共有ヒープと静的データをノード間で共有可能な共有アドレス空間とする．従来のヒープはノード内でのみ参照されるデータを配置し，**プライベートヒープ**と呼ぶ．OpenMP では動的に割り付けるデータは共有データと見なされるため，プライベートヒープはライブラリ内部で使用するデータや，ノード間で共有されないことが保証できる動的データのために使用される．

共有アドレス空間は 4 KB の**ページ**に分割され，各ページはホームノードが割り当てられる．**ホームノード**とは，そのページに配置されたデータの実体を持つノードであり，他のノードはそのページのデータのコピーを持つことができる．**ページディレクトリ**は各ページについて，ホームノード・コピーを持つノードの集合・ページの一貫性制御プロトコルなどのデータを保持する．すべてのノードがページディレクトリ全体のコピーを持っている．

共有ページのホームノードの決定方式として，固定ホーム方式とファーストタッチ方式をサポートする．固定ホーム方式では，全ページのホームはノード 0 であり，全ノードがコピーを持つ．ファーストタッチ方式では，プログラムの実行開始後にそのページのデータを最初に参照したスレッドの実行されているノードをホームとする．また，ページ内のデータを参照したノードのみがコピー集合に加えられる．一度コピー集合に登録されたノードは，プログラム終了までコピー集合から取り除かれることはない．

2.4 一貫性制御プロトコル

共有データの一貫性制御に用いるプロトコルとして，無効化プロトコルと更新プロトコルをサポートする．どちらのプロトコルを用いるかはデータごとに選択することができ，さらに同一データでも参照点ごとにプロトコルを変更することもできる．デフォルトは無効化プロトコルであり，更新プロトコルはライブラリ内部のデータや最適化で選択されたときに用いられる．このように共有データの参照パターンに応じた柔軟な一貫性制御が可能になることが，SDSM の利点の 1 つである．

まず，無効化プロトコルを用いて一貫性を保証する場合の動作を説明する．共有アドレス空間内のページは，さらに 64 バイトごとの**ライン**と呼ぶ単位に分割され，このラインを単位に一貫性制御を行う．各ノードが，共有データの最新のコピーを持つか否を示すためにラインごとの**状態フラグ**を設ける．状態フラグは有効か無効かを示す 1 ビットだけからなる．

無効化プロトコルの共有データの場合，状態フラグを元に必要に応じてライン内のデータを最新のコピーに更新するとともに，内容の古くなったラインを無効化する必要がある．また，データのホームノードの持つ値が最新であることを保証する必要もある．そのために，次の 3 種類の基本操作（チェックコード）を用いる．

読み込み前チェック　参照する範囲を含むラインの状態フラグを調べ，無効なラインがあればその内容をホームノードからコピーし，更新されたラインの状態フラグを有効にする．

書き込み前チェック　読み込み前チェックと同様であるが，全体を書き換えるラインは元の値を参照しないため，ラインの更新はせず，フラグのみ有効にする．

書き込み後チェック　書き換えたデータの値をホームノードにコピーし，他のノードの書き換えたデータを含むラインを無効化する．ただし，ホームに書き戻すデータは書き換えたライン全体ではなく，ライン中の書き換えた部分だけである．

これらのコードに共通して，参照する領域が共有領域か否かのチェックが必要である．また，汎用のチェックコードでは複数のラインにまたがるデータの処理も考慮している．そこで，コンパイル時に必ず共有領域を参照すると保証できる場合や，データサイズがコンパイル時定数の場合のために特化したコードも用意している．それらのコードの中から，コンパイル時に利用可能な情報を元に，最も実行時オーバヘッドの小さなものをコンパイラが挿入する．

図 4 は，無効化プロトコルの際の一貫性制御コードの動作を示す．図の左側は共有データの読み込み，右側は書き込みを示す．読み込み前と書き込み前のチェックコードは，最適化を行わない場合は共有データの参照の直前に挿入する．しかし，データのアドレスやサイズの式中に現れる変数のデータ依存関係を壊さない範囲で，直前のフラッシュ操作の直後までの間で移動できる．同様に書き込み後のチェックコードは，書き込みと直後のフラッシュ操作までの間に挿入する．

次に更新プロトコルの場合の一貫性の制御方法を説明する．更新プロトコルでは状態フラグは用いず，書き換えた後に全ノードのコピーを更新する．また，参照するノードが特定できるときには，そのノードのコピーのみ更新することもできる（選択的更新と呼ぶ）．無効化プロトコルに比べると，状態フラグの管理が不

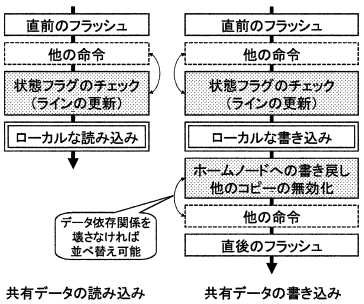

図 4 無効化プロトコルの動作
Fig. 4 Operations of the invalidation-based protocol.

要であることと，データ転送が書き込み側で行われるために通信と計算がオーバラップできるという利点がある．しかし，転送したデータが使用されない場合，無駄なデータ転送を行うことになる．そのため，転送したデータが使用されることがコンパイル時に検出できるときのみ，更新プロトコルを用いる．

更新プロトコル用の一貫性制御操作には以下がある．

(1) 特定ノードへの書き込み
(2) ホームノードへの書き戻し
(3) ノード 0 への書き込み
(4) コピーを持つ全ノードの更新
(5) 全ノードの更新

これらはいずれも共有データの書き込み後の操作であり，読み込み前・書き込み前の操作は不要である．これらの操作は共有データの参照パターンを元にコンパイラが効率の良いものを選択して挿入する．その具体的な条件は以後の章で述べる．

3. OpenMP コンパイラの概要

この章では，Omni/CSDSM の OpenMP コンパイラの構成と，最適化機能について述べる．

3.1 コンパイラの構成

Omni/CSDSM の OpenMP コンパイラは，一貫性制御の最適化を行うほかに以下の特徴を持つ．

(1) 1 つ以上のファイルからなるプログラム全体を対象に手続き間解析・手続き間最適化を行う．
(2) フローセンシティブかつコンテクストセンシティブなポインタ解析を行う．
(3) 動的に割り付けられたデータは，割付け関数の呼び出し文脈に応じて，異なるデータオブジェクトとして区別する（ヒープ解析）．
(4) 配列の参照範囲を矩形で表したデータフロー解析を行う（配列セクション解析）．
(5) 配列の添字式に変数を含んだまま配列セクション解析を行う（シンボリック解析）．
(6) OpenMP の指示文の意味を考慮した手続き間データフロー解析を行う．
(7) スレッド間のデータ依存関係を解析する（並列データフロー解析）．
(8) C コンパイラとして GNU コンパイラ（GCC）を仮定し，GCC の拡張機能を利用する．

これらの特徴のうち，逐次プログラムの最適化コンパイルと共通の技術については説明を省略し，Omni/CSDSM 特有の問題について本稿で説明する．なお，OpenMP コンパイラ自体は Java 言語とパーサージェネレータ JavaCC を用いて記述している．

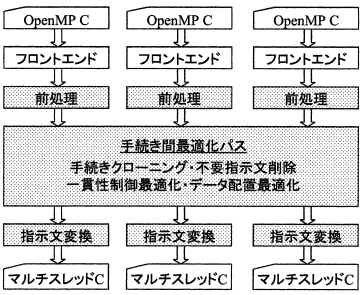

図 5 OpenMP コンパイラの概要
Fig. 5 Overview of the OpenMP compiler.

図 5 は OpenMP コンパイラの処理の概要を示す．コンパイラのフロントエンドは，各ソースファイル（プログラム単位）をファイルごとに字句解析・構文解析を行い中間語を生成する．バックエンド（図 5 の網掛け部）では，プログラム単位ごとにプログラム構造の正規化やループ情報の解析などの前処理を行った後，プログラム全体に対して手続き間解析・最適化を行う．その段階でどのような一貫性制御コードを挿入するかを決定し，さらにプログラム単位ごとに OpenMP 指示文の変換と一貫性制御コードの生成を行い目的コード（C プログラム）を生成する．

Omni/CSDSM の OpenMP コンパイラで行う最適化は，それに必要なデータフロー解析を元に，以下のように分類できる．

(1) 局所データフロー解析に基づく最適化
  (a) チェックコードのマージ
  (b) チェックコードの冗長性削除
  (c) チェックコードのループ外移動
(2) 手続き間解析に基づく最適化
  (a) ネストレベル解析と手続きクローニング
  (b) 不要指示文の削除
  (c) 非共有データ（静的・動的）の検出
  (d) 動的データのプライベート化
  (e) 一貫性制御プロトコルの最適化
  (f) アラインメントの最適化
(3) 並列データフロー解析に基づく最適化
  (a) 明示的コピーの生成

このうち (1) と (3) については文献 5) でも述べているが，本稿でも説明を加える．また，(2) の各最適化は本稿で新たに提案するものである．これらの最適化はいずれも直接あるいは間接に一貫性制御の実行時オーバヘッドを削減することを目的としている．

以下の章では，これらの解析・最適化で用いる中間表現を説明した後，データフロー解析のレベルごとに最適化とそれに必要な解析について述べる．

4. 中間表現

解析・最適化を行う際のプログラムの中間表現として，コールグラフ・並列フローグラフ・同期グラフの3つの中間表現を用いる．また，最適化に先駆けて行われるプログラム構造の正規化にも触れる．

4.1 コールグラフ

コールグラフは手続き間の呼び出し関係を表す．逐次プログラムのコールグラフは，手続きをノードとし，手続きの呼び出し関係をエッジで表す．我々のコールグラフでは，手続きだけでなく並列領域もコールグラフのノードとして表す．並列領域をコールグラフのノードとすることにより，そのノードに並列領域ごとの情報を持たせることができるとともに，コールグラフ上で逐次部と並列部を容易に区別できる．

コールグラフのエッジは，手続き呼び出しのコールサイトごとに作成する．そのため，一般にコールグラフは多重有向グラフとなる（すなわち，同一ノード間の同方向のエッジが複数ありうる）．これにより，呼び出し文脈を区別した解析が容易に行える．`parallel` 指示文などによる並列領域の生成（フォーク）は，その並列領域を含む手続きのノードから並列領域を表すノードへのエッジを張ることで表す．また，`parallel` 構文を構成するブロック内で直接呼び出される手続きは，その `parallel` 構文を含む手続きのノードではなく，その `parallel` 構文で生成される並列領域を表すノードからエッジを張る．

コールグラフの例は，5.2.2 項で示す．

4.2 並列フローグラフ

並列フローグラフ（Parallel Flow Graph; PFG）は，コールグラフのノード（手続きまたは並列領域）ごとに作成され，ノードの表す手続きまたは並列領域内の指示文と実行文を表現する．

並列フローグラフ $PFG = (N, E)$ はノードの集合 N とエッジの集合 E からなる有向グラフである．PFG のノードは次のように分類できる．

プログラムの始点と終点： PFG の表す手続きあるいは並列領域の入口と出口をそれぞれノード s と e で表す．

逐次ノード： OpenMP 指示文を含まない実行文からなる基本ブロックを表す．

指示文ノード： プログラム中の個々の OpenMP 指示文を表す．ただし，指示文とその作用を受けるブロックが組になって構文（Construct）として使用される指示文では，その構文の入口と出口をそれぞれ1つの指示文ノードで表す．

PFG のエッジは，逐次ノード間およびノード s，e と逐次ノードの間は，逐次プログラムの制御フローグラフと同様に，制御の流れに沿ってエッジを張る．OpenMP 指示文に対しても基本的には同様に，先行するノードから指示文ノード（構文の場合はその入口ノード），指示文ノード（構文の場合はその出口ノード）から後続のノードへエッジを張る．

以下の指示文に対しては，上記と異なる方法でノード・エッジを作成する．

- `for` 構文では，構文の入口と出口のノードの間に，並列実行するループ全体ではなく，ループ本体のみを入れ，構文の入口から出口へのエッジを張る．並列ループは PFG でサイクルを構成しない．
- `sections` 構文では，構文の入口から各セクションの入口，各セクションの出口から構文の出口へのエッジを張り，セクション間のエッジは張らない．
- `parallel` 構文などの並列領域を生成する指示文では単独のノードを作成する．同時に，コールグラフのその並列領域を表すノードに対して，並列領域の内部を表す PFG が作成される．

また，各指示文のノードには，指示文に付加された指示節（`private` など）の情報や，ループ範囲の情報が付加される．

基本ブロックを表す通常のノードには，実行文の中間語が付加される．実行文は，解析を行いやすくするために正規化されている．たとえば，手続き呼び出しは単独の式文か代入文の右辺にのみ現れる，共有データの参照は式文でしか行われない，などである．

図 6 は，3 種類の指示文について，いずれも，文 `S1` と `S2` の間のプログラム片の PFG での表現を示す．単独のノードで表される指示文はその指示文名（図中の `barrier` など）でノードを示し，構文の入口と出口にノードが作成される指示文は，指示文名にそれぞれ `enter` と `exit` を付けたノード名を付ける．ただし，`for` 構文の入口は，`for(i=0,n,1)` のようにルー

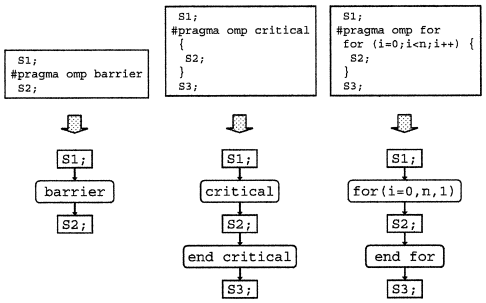

図 6 OpenMP 指示文の並列フローグラフでの表現
Fig. 6 Representations of OpenMP directives in PFG.

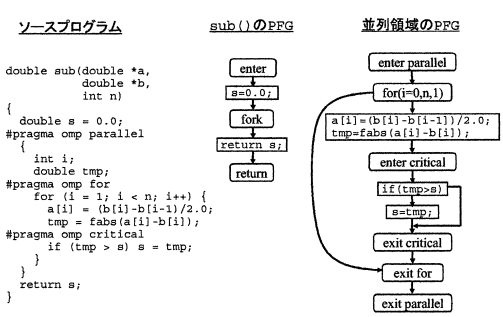

図 7 並列領域を含む手続きの並列フローグラフ
Fig. 7 PFGs for a procedure with a parallel region.

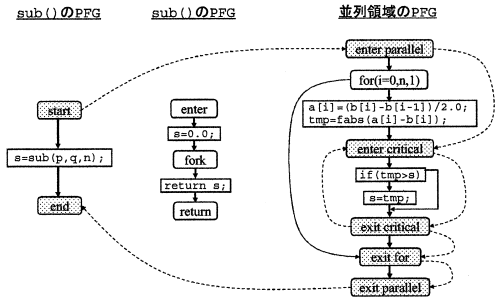

図 8 並列フローグラフと同期グラフ
Fig. 8 PFG and synchronization graph.

プ制御変数とその範囲を示す．

図 7 は，並列領域を含む手続きの並列フローグラフを示す．この例では，手続き `sub()` とその中の並列領域に対してコールグラフのノードが作成され，それぞれの並列フローグラフが作成される．図中で，楕円で表されたノードは手続きの入口・出口と指示文に対するノード，その他のノードは実行文（基本ブロック）を表すノードであり，実線の矢印が PFG のエッジである．手続き `sub()` の PFG 内のノード `fork` は並列領域の生成を表し，その並列領域の本体が図の右側の PFG である．

4.3 同期グラフ

同期グラフは，メモリ同期点（OpenMP のフラッシュ操作）の間の実行順序の制約を表す有向グラフである．前節で述べた PFG は手続きおよび並列領域ごとにそれぞれ作成したが，同期グラフはプログラム全体で 1 つ作成する．同期グラフのノードの集合は，フラッシュ操作をともなう PFG ノードをプログラム全体で集めたものである．そのノードの集合に，新たに**同期エッジ**と呼ぶエッジを張る．ノード n からノード m へ同期エッジが張られる場合，あるスレッドがノード n を実行した後，同一スレッドまたは異なるスレッドが他の同期点を実行する前にノード m を実行する可能性があることを示す．

図 8 は，図 7 の例で，他に並列領域がないとした場合の並列フローグラフと同期グラフを示す．図では新たに手続き `sub()` を呼び出す手続き `main()` の PFG も示している．図中で網かけのノードで示されているのが同期グラフのノードであり，点線の矢印が同期グラフのエッジである．手続き `main()` の入口 `start` と出口 `end` はプログラムの始点と終点を表し，これらもメモリ同期点と見なす．すると，`start` の直後に実行されうるメモリ同期点は並列領域の入口（`enter parallel`）であるため，それへの同期エッジが張られる．ループ中のクリティカルセクションの入口と出口もメモリ同期点であるが，クリティカルセクションは複数のスレッドで任意の順序で実行されうるため，その入口から出口および出口から入口への同期エッジが張られる．ただし，入口から出口への同期エッジは同一スレッドでの実行のみ示し，一方，出口から入口へのエッジは異なるスレッドの実行を示している（グラフ上ではこれらは区別されない）．

同期グラフはプログラム全体にまたがって作成されるが，ノード数はプログラム中のメモリ同期点数に比例し，エッジ数は最悪でその自乗に比例する．しかし，現在の OpenMP プログラムではメモリ同期点の多くが明示的あるいは暗黙のバリア同期であるため，メモリ同期点間の実行順序の制約が多く，エッジ数もノード数よりそれほど多くはならないと思われる．したがって，同期グラフを用いたプログラム全体にわたる解析を行っても，一般の手続き間解析よりも効率良く行える．

5. 一貫性制御の最適化手法

5.1 局所データフロー解析に基づく最適化

5.1.1 同期区間内の解析

同期区間内では，スレッド間の相互作用を考慮することなく解析できるため，PFG を用いて，逐次プログラムと同様のデータフロー問題を解くことができる．ただし，同期区間内にはメモリ同期点とならない OpenMP の指示文も含まれるため，それらの扱い方は注意が必要である．

ここで求めるデータフロー情報は，一般に用いられる露出使用と到達定義をさらに詳細に分類したものである．共有データは，同期区間内で最初の参照（読み込みまたは書き込み）の前と，最後の書き込みの前に同期されることが保証されればその間の参照点で同期する必要はない．そのため，同期区間内で最初の参照点と最後の定義点を求め，それらの参照点のみ同期

を行うことで，冗長なメモリ同期を除外することができる．

まず，以下の集合を定義する．

直接露出使用点集合： あるプログラム点 p に対して，変数 v の使用点 u が p の直接露出使用であるとは，メモリ同期点・v の定義・u の参照式中の変数の定義を通らずに p から u まで達するパスが存在することをいう．プログラム点 p に対する直接露出使用点全体の集合を p の直接露出使用点集合と呼ぶ．

直接露出定義点集合： あるプログラム点 p に対して，変数 v の定義点 d が p の直接露出定義であるとは，メモリ同期点・v の定義・d の参照式中の変数の定義を通らずに p から d まで達するパスが存在することをいう．プログラム点 p に対する直接露出定義点全体の集合を p の直接露出定義点集合と呼ぶ．

直接到達定義点集合： あるプログラム点 p に対して，変数 v の定義点 d が p の到達定義であるとは，メモリ同期点・v の定義・d の参照式中の変数の定義を通らずに d から p まで達するパスが存在することをいう．プログラム点 p に対する到達定義点全体の集合を p の直接到達定義点集合と呼ぶ．

ここで対象とする変数は，共有データである可能性のある変数のみである．また，変数には静的に名前の付けられた変数だけでなく，配列セクションやヒープ領域などを表す仮想的な変数も含む．露出使用点集合・露出定義点集合は後方（制御フローの逆向き）への伝搬，到達定義点集合は前方への伝搬により求められる．

メモリ同期をともなわない OpenMP 指示文のノードでは，以下のように扱う．並列ループ本体の使用点・定義点の情報は，その参照式に並列ループのループ制御変数が含まれていない場合，並列ループの外へ伝搬できる．ループ制御変数が含まれている場合には，ループ本体の入口または出口より先へは伝搬できない．逐次ループの場合，参照式にループ制御変数が含まれているときは，参照式を配列セクションの参照に置換して，ループ外へ伝搬できる．指示文に `private` などのデータ属性指示節が付加されている場合は，属性の変換された変数どうし（共有版とプライベート版）は別の変数として扱う．それ以外の，メモリ同期をともなわない指示文ノードはデータフロー情報を変化させずに伝搬できる．

上記の 3 つの集合を元に，さらに以下の集合を定義する．

最上方直接露出使用変数集合： プログラム点 p の直接露出使用点集合に変数 v の使用点 u が含まれるとき，p にメモリ同期点を通らずに到達可能なプログラム点 q で，q の直接露出使用点集合で v を含まないものがあるとき，v は p の最上方直接露出使用変数であるといい，そのような変数全体の集合を p の最上方直接露出使用変数集合と呼ぶ．

最上方直接露出定義変数集合： プログラム点 p の直接露出定義点集合に変数 v の定義点 d が含まれるとき，p にメモリ同期点を通らずに到達可能なプログラム点 q で，q の直接露出定義点集合で v を含まないものがあるとき，v は p の最上方直接露出定義変数であるといい，そのような変数全体の集合を p の最上方直接露出定義変数集合と呼ぶ．

最下方直接到達定義変数集合： プログラム点 p の直接到達定義点集合に変数 v の定義点 d が含まれるとき，p からメモリ同期点を通らずに到達可能なプログラム点 q で，q の直接到達定義点集合で v を含まないものがあるとき，v は p の最下方直接到達定義変数であるといい，そのような変数全体の集合を p の最下方直接到達定義変数集合と呼ぶ．

あるメモリ同期点 s から，他のメモリ同期点を通らずに変数 v の使用点 u に到達可能な場合，s から u までの間のプログラム点 p で，最上方直接露出使用変数集合に v を含むものが必ず 1 つ以上存在する．定義点と最下方到達定義変数集合についても同様の関係がある．

ノード n に対する上記の集合を以下のように表す．

- 最上方直接露出使用変数集合：$UpEUse(n)$
- 最上方直接露出定義変数集合：$UpEDef(n)$
- 最下方直接到達定義変数集合：$DnRDef(n)$

ここで，PFG や同期グラフでプログラム点はノードの境界またはノード中の文の境界に存在する．しかし，以下では説明を簡単にするために，データフロー情報を保持する必要のあるプログラム点はノード境界にのみ現れるものとする．$UpEUse(n)$ と $UpEDef(n)$ は，ノード n の直後のプログラム点に対するそれぞれの情報を表し，$DnRDef(n)$ はノード n の直前のプログラム点に対する情報を表す．

図 9 は，同期区間内の解析で得られる情報の例を示す．この例では，前後をバリア同期ではさまれた並列ループについて，各ノードの最上方露出使用・最上方露出定義・最下方到達定義の情報を示している．ループ中の配列 `a` の定義は，添字式に並列ループのルー

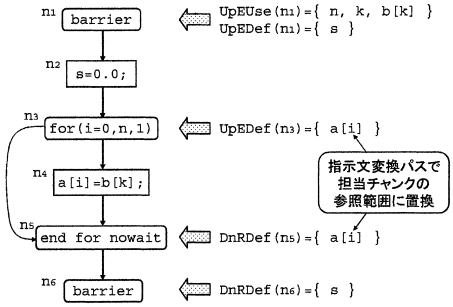

図 9 局所データフロー情報の例
Fig. 9 An example of local dataflow information.

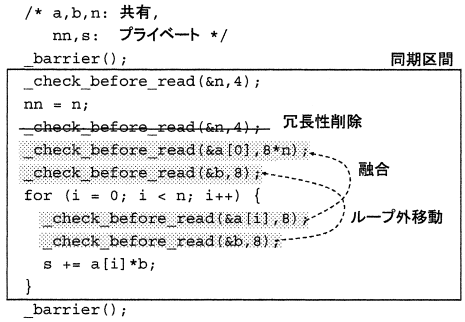

図 10 局所最適化
Fig. 10 Local optimization.

プ制御変数が含まれているため，露出定義・到達定義はループの外側へは伝搬されない．しかし，ループ中で配列 b の定義は，添字式がループ独立な式であるため，直前のメモリ同期点（barrier 指示文）まで伝搬される．

局所最適化で用いるデータフロー情報では，必ず露出/到達するものだけを対象とした，確定情報（Must 情報）のみを考え，露出/到達する可能性のあるものすべてを含む不確定情報（May 情報）は考えていない．しかし，後述の到達定義解析やメモリ同期点解析では最適化では不確定情報も解析し，明示的コピーの生成に用いる．

5.1.2 同期区間内の最適化

同期区間内の一貫性制御コード（チェックコード）の最適化（局所最適化）は，上記の情報を用いて行う．まず，同期区間内で行える最適化には，以下がある．

チェックコードのマージ： 連続領域に対する同一種類のチェックコードを 1 つにまとめる．

チェックコードの冗長性削除： 同一データ・同一種類のチェックコードが複数実行されるとき，最初の 1 つ以外を削除する．

チェックコードのループ外移動： ループ内にある参照式がループ不変なチェックコードをループ外へ出す．

図 10 は，これらの最適化の例を示している．ここに現れる一貫性制御コードの引数は，第 1 引数がデータのアドレス，第 2 引数がデータサイズ（単位はバイト）である．変数 n は同期区間内で 2 度読み込まれるが，間にメモリ同期点がないため，2 度目の読み込み前チェックは冗長であり，削除できる．配列 a は，ループ中で要素を順次読み込まれるため，それらの連続領域の読み込み前チェックをループの前に一括して行う．変数 b は，ループ中で繰り返し読み込まれるため，ループの前に 1 度だけ読み込み前チェックを実行すればよい．この例では一貫性制御プロトコルはデフォルトの無効化プロトコルとしている．しかし，次節以降の最適化と局所最適化を併用する場合，更新プロトコルのデータに対しても同様の最適化が行える．

上記の最適化の分類は，最適化なしで一貫性制御コードを挿入したときとの比較に基づいており，最適化を行う際には実際に不要な一貫性制御コードの削除や移動を行うのではない．その代わり，前記の同期区間内のデータフロー解析によって，最小限必要な一貫性制御コードの挿入位置を求め，最適化後のコードを直接挿入する．そのためには，前述の最上方露出使用・最上方露出定義・最下方到達定義の情報があれば十分である．

無効化プロトコルの場合，以下のように一貫性制御コードを挿入する．最上方露出使用変数集合が空でないプログラム点に対して，その集合に属する変数に対する読み込み前チェックを挿入する．同様に，最上方露出定義変数集合から書き込み前チェック，最下方到達定義変数集合から書き込み後チェックを挿入する．ただし，並列ループ本体の入口と出口に一貫性制御コードを挿入する際には，ループ制御変数を担当するチャンクの範囲に置き換えて，融合したコードをループの前または後ろに挿入する．図 9 の例の場合，a[i] に対する一貫性制御コードをループ本体に挿入する代わりに，チャンクの範囲を lb から ub とすると，a[lb:ub] に対する一貫性制御コードを挿入する．

更新プロトコルの場合には，最下方到達定義を元に，一貫性制御コードを生成すればよい．

5.2 手続き間解析に基づく最適化

5.2.1 手続き間解析

我々の OpenMP コンパイラは，フローセンシティブかつコンテクストセンシティブな手続き間データフロー解析を行う．ただし，この節ではスレッド間のデータ依存関係までは求めず，別名関係と，各並列領

域単位での共有データの定義/使用，特定指示文での定義/使用のみ解析する．ここで行う手続き間解析には，以下がある．

ネストレベル解析： コールグラフの各ノード（手続きまたは並列領域）の呼び出されうるネストレベルを解析する．

ヒープ解析： 動的に割り付けられるデータについて，割付け関数の呼び出し文脈ごとに別のデータとして区別した名前付けを行う．

ポインタ解析： プログラム中のポインタ（配列型引数を含む）について，その指示先および別名関係を解析する．

参照変数解析： 各手続きおよび並列領域について，その中で実名または別名で定義・使用される変数を解析する．

ポインタ解析・ヒープ解析では，主として逐次部分で代入されるポインタの指示先と，配列型引数についての別名関係を求める．並列領域内で代入のあるポインタについては，ポインタが共有データの場合，代入の順序関係を正確に解析しないため，精度が低くなる．

参照変数解析でもスレッド間の相互作用を考慮せずに，逐次プログラムと同様の手法で解析する．そのため，並列領域内の定義・使用の順序関係は正確に反映されず，並列領域内で定義・使用があるか否かのみが分かる．しかし，それだけでも最適化に有用な情報が得られる．

5.2.2 ネストレベルに応じた手続きクローニング

コールグラフの作成時に，並列領域のネストレベルの解析を行う．ネストレベルとは，逐次領域をレベル 0 とし，逐次領域から生成される並列領域をレベル 1，以後，レベル n の並列領域から生成される並列領域をレベル $n+1$ とする．ただし，OpenMP ではネスト並列を実装するかは処理系依存であるため，Omni/CSDSM ではネスト並列をサポートせず，ネストレベルは 0 と 1 のみ考慮すればよい．

同一手続きが異なるネストレベル，すなわち逐次領域と並列領域，で呼び出される可能性がある場合は，手続きクローニングを行い，ネストレベルごとに異なる手続きが呼び出されるように変換する．したがって，コールグラフの各ノードは単一のネストレベルでのみ呼び出されることが保証される．各ノードのネストレベルが確定することにより，以下の最適化が可能になる．

不要指示文削除 逐次領域でのみ実行される，並列領域の生成以外の指示文は不要なため，削除する．

ライブラリ関数の評価 OpenMP の実行時ライブラリのうち，`omp_in_parallel()` などの関数の値を評価し，関数呼び出しを返却値に置き換える．分岐条件の評価などの他の最適化も同時に行う．

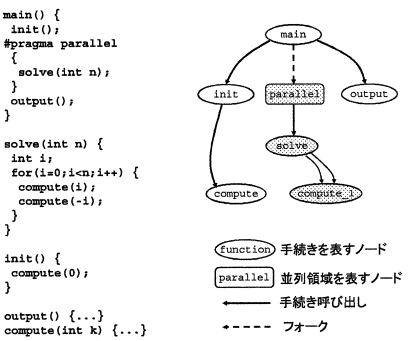

図 11 OpenMP プログラムとクローニング後のコールグラフ
Fig. 11 An OpenMP program and its call graph after procedure cloning.

ただし，これらは本稿の評価に用いたプログラムには効果がない．これらの最適化よりも，各手続きが逐次領域と並列領域の一方でしか実行されないことが保証されるために，一方の場合のみ考慮した一貫性制御コードを生成することができることが有用である．

図 11 は，簡単な OpenMP プログラムとそのコールグラフを示す．プログラム中には 5 つの手続きと 1 つの並列領域がある．このうち手続き `compute()` は，手続き `init()` から 1 回，手続き `solve()` から 2 回呼ばれるが，それぞれネストレベルが異なるため，クローン手続き `compute_1()` が作成される．そのため，手続きのノードが 6 個，並列領域のノードが 1 個作成される．図中で網かけされているノードは並列領域（ネストレベル 1），その他は逐次領域（ネストレベル 0）で実行される．

5.2.3 非共有データの検出

手続き間解析によって，プログラム中の並列領域が検出され，変数の別名関係，プログラム全体にわたる変数の使用方法が検出される．それによって，スレッド間（ノード間）で共有される可能性のあるデータが限定され，そうでないデータに対する一貫性制御を削除することができる．具体的には，以下の場合に一貫性制御が不要となる．

逐次領域でのみ参照される共有データ： 並列領域で参照されることのない共有データは，単一ノード（ノード 0）でしか参照されないため，ノード間の一貫性は保証しなくてよい．また，並列領域内でも `master` 構文内でしか参照されなければ，逐次領域のみの参照と見なせる．

プライベートな配列型引数： 手続き内解析のみ行う

ソースプログラム

```
main() {
  int i,n; double a[N];
  n = ...;
  for (i=0;i<n;i++) a[i]=...;
#pragma omp parallel shared(a,n)
  {
    double *b = (double *)malloc(sizeof(double)*n);
    sub(a,b,n);
  }
}
void sub(double *p, double *q, int m) {
  int j;
  for (j=0;j<m;j++) q[j]=p[i];
}
```

最適化結果

main()	a[0:n]	共有，更新プロトコル
	b[0:n]	非共有，プライベートヒープに割付
	n	共有，更新プロトコル
	i	プライベート
sub()	p,q,j	プライベート
	p[]	共有，更新プロトコル，a[]と別名
	q[]	非共有，b[]と別名

図 12 手続き間最適化の例
Fig. 12 An example of interprocedural optimization.

場合，配列型引数は共有データである可能性があるため，一貫性制御が必要である．しかし，実引数を解析することにより，共有データである可能性がないことが分かれば，一貫性制御を削除できる．

プライベートな動的データ： ヒープ解析・ポインタ解析で，動的に確保されたデータが共有ポインタから指示されることがないことが検出されたとき，そのデータは共有ヒープではなくプライベートヒープに割り付け，一貫性制御も削除する．

図 12 は，手続き間最適化の例を示す．ここで，ポインタ b を通して参照されるデータは動的に確保されるためデフォルトでは共有データとなるが，この b と仮引数 q からしか指示されないため，割り付けたスレッド以外が参照することはできない．そこで，このデータはプライベートヒープに割り付け，b と q を通した参照の一貫性制御を削除することができる．

5.2.4 一貫性制御プロトコルの最適化

Omni/CSDSM では，データごとや参照点ごとに一貫性制御プロトコルを変更できる．そこで，デフォルトは無効化プロトコルであるが，更新プロトコルの方が効率の良い場合に，コンパイル時に更新プロトコルを選択する．

異なるプロトコルの混在を容易にするために，まず共有データを以下のように分類する．前述のように共有領域は静的データ領域と共有ヒープ領域からなる．このうち，静的データ領域をさらに以下の部分領域に分類する．

- 逐次参照領域：逐次領域または並列領域中の master 構文内でのみ参照される静的データを配置する．
- 並列読み込み領域：並列領域内で，まったくあるいはマスタースレッド以外には書き換えられることのない静的データを配置する．
- 並列書き込み領域：並列領域内で書き換えられる可能性のある静的データを配置する（デフォルト）．

ページ内のライン単位で状態フラグを持つ．

これらの部分領域は，その境界がページの境界になるように配置し，各ページはいずれか 1 つの部分領域に属するようにする．

手続き間解析によって，並列領域で使用・定義されうる共有データが検出されている．そこで，その情報を元に，静的データを上記のいずれかの部分領域に配置する．逐次参照領域に配置されたデータは一貫性制御を行わない．並列読み込み領域に配置されたデータは，プログラム全体にわたって更新プロトコルで一貫性制御を行う．並列書き込み領域に配置されたデータは，デフォルトでは無効化プロトコルで一貫性制御を行うが，プログラムの一部分だけ更新プロトコルを用いてもよい．

部分領域の指定には，GCC の拡張機能を用いており，初期値付データしか領域を指定できない．そのため，実行形式が増大するのを避けるため，逐次参照領域と並列読み込み領域に配置するデータは，スカラ変数とページサイズ（4 KB）以下の小さな配列のみとした．並列書き込み領域に配置されたデータでも更新プロトコルを用いることができるため，この制限は一貫性制御コードの生成には影響ない．

図 12 の例では，スカラ変数 n と配列 a は並列領域内で読み込みのみされるため，更新プロトコルを選択する．これらの変数は自動変数であるため通常は並列領域の入口で共有領域に割り付け直されるが，再帰呼び出しのない手続きでは，関数スコープの静的変数に宣言を変えることで，再割付けを不要にしている．さらに，仮引数 p はその実引数が配列 a のみであり，つねに更新プロトコルのデータであるため，読み込みに対して一貫性制御コードは生成しない．もし，p の実引数が呼び出し元によって無効化プロトコルの変数と更新プロトコルの変数の両方の場合があるときは，p の参照に対して無効化プロトコルの一貫性制御コードを生成する．そのとき，更新プロトコルのデータを参照すると，本来不要な状態フラグの検査を行うことになる．

5.2.5 アラインメントの最適化

SMP や DSM ではフォルスシェアリングにより一貫性制御のオーバヘッドが増大することがあることがよく知られている．特に無効化プロトコルを用いる場合に，ライン粒度でのフォルスシェアリングによる余分な無効化の発生と，ページ粒度でのフォルスシェアリングによるホームノード割当ての悪化が考えられる．更新プロトコルの場合には，フォルスシェアリングは

問題にならない．

そこで，無効化プロトコルを用いる共有データについて，データのラインまたはページ境界へのアラインメントを行った．この場合も GCC の拡張機能を用いており，初期値付データしかアラインメントを指定できず，アラインメントを指定するとファイルサイズが増大するため，スカラ変数とページサイズ（4 KB）以下の小さな配列のみをアラインメントの対象とした．

5.3 並列データフローに基づく最適化

通信最適化は，スレッド間のデータ依存関係を元に，メッセージパッシングプログラムのような明示的なデータ転送を行うコードを生成する最適化である．前節の手続き間解析を利用した各最適化は，変数単位でしか最適化を行えなかったが，通信最適化では配列の要素単位でスレッド間のデータ依存関係を解析し，共有される要素のみを転送することができる．

5.3.1 並列データフロー解析

通信最適化を行うには，スレッド間のデータ依存関係を調べる必要がある．そのための，スレッド間のデータの流れ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．並列データフロー解析については，すでに文献 2)，3) でも述べているため，概要のみ説明する．

並列データフロー解析により，以下の情報が求められる．

- 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

並列フローグラフと同期グラフを用いることにより，同一スレッド内と異なるスレッド間の双方のデータの流れを含めた解析が行える．ただし，それだけでは特定のスレッド間のデータの流れまでは識別できない．そこで，並列ループ間にまたがった Def-Use チェインのあるものについて，より詳細なデータ依存解析を行う．これにより，規則的な参照を行う配列については，要素単位でスレッド間のデータ依存関係を求めることができる．

5.3.2 明示的コピーの生成

並列データフロー解析によって，スレッド間のデータ依存関係が検出できると，正しく並列実行するために必要なノード間通信を特定できる．その結果を用いて，書き込み側スレッドから読み込み側スレッドへ，明示的にデータを転送するコードを挿入する．また，同一ノード内のスレッド間でデータ依存が生じる場合は，物理共有メモリを介してデータを共有できるため，ソフトウェアによる一貫性制御は不要となる．

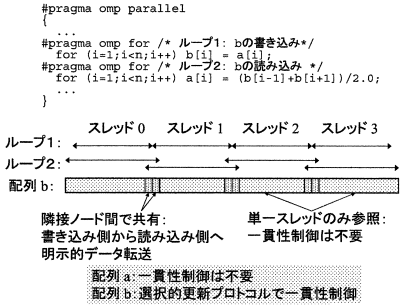

図 13 並列データフロー解析に基づく最適化
Fig. 13 PDA optimization.

図 13 を用いて，並列データフロー解析に基づく最適化の効果を説明する．ここでは 2 つの並列ループで配列 `a` と `b` を参照している．配列 `b` について考えると，到達定義解析を元に，第 1 のループの定義と第 2 のループの使用との間にデータ依存関係が生じることが分かる．そこでさらに，ループ制御変数やスケジューリング，配列参照の添字を元に要素単位のデータの流れを調べる．その結果，第 1 のループで書き換えた要素のうち，チャンクの先頭と最後の反復で書き換えた要素が，それぞれ 1 つ前のスレッドと 1 つ後のスレッドで読み込まれることが分かる．そこで，チャンク境界の要素についてのみ，書き込み後に隣接ノードの持つコピーの値を更新する，選択的更新プロトコルによって一貫性制御を行う．チャンク境界の要素以外は単一スレッドにしか参照されないため，ノード間の一貫性制御は不要となる．同様に，配列 `a` については，スレッド間のデータ依存はまったく生じないため，一貫性制御は不要となる．

6. 最適化の効果

6.1 SMP クラスタ COMPaS II

本稿の評価に用いる SMP クラスタは，4 ウェイ SMP をノードとし，それを 8 ノード，Myricom 社の Myrinet ネットワークインタフェースを介して接続した SMP クラスタ COMPaS II である．**表 1** に COMPaS II の構成を示す．Myrinet 用の通信ライブラリとして NICAM [14] を用いる．各ノードはメモリは 4 ウェイにインタリーブされており，そうでない場合よりもバスバンド幅は向上している．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．GNU C コンパイラは，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として利用され，最適化オプ

表 1 COMPaS II の構成
Table 1 Configuration of COMPaS II.

プロセッサ	Intel Pentium II Xeon 450 MHz
チップセット	Intel 440NX
キャッシュ	L1: 16 KB/16 KB, L2: 1 MB
メモリ	各ノード 1 GB 以上
ネットワーク	100Base-TX, Myrinet
OS	Red Hat Linux 6.2（kernel 2.2.16）
C コンパイラ	GNU C コンパイラ (egcs 1.1.2)

表 2 ベンチマークプログラム
Table 2 Benchmark programs.

名前	説明	データサイズ
EP	NAS 並列ベンチマークの EP	1 MB
JOR	連立一次方程式の直接解法	128 MB
Laplace	Laplace 方程式の陽解法	67 MB
CG	NAS 並列ベンチマークの CG	62 MB

ションは `-O3 -funroll-loops -malign-double` とした．

6.2 ベンチマークプログラム

性能評価には，**表 2** に示す 4 本の OpenMP C プログラムを用いた．これらのプログラムはそれぞれ異なる特徴を持つ．

JOR（Jacobi Over-Relaxation）は，連立一次方程式 $Ax = b$ を反復法の一種である JOR 法で解く．解ベクトル x の値を繰り返し改良していくため，並列領域内では行列 A とベクトル b は読み込みのみ行われ，x の近似解のみが書き換えられる．行列サイズは 4096×4096 とし，11 回の反復で収束する．

Laplace は，二次元正方格子上のラプラス方程式の境界値問題を Jacobi 法で解く．数学的には JOR と同様の連立一次方程式の反復解法であるが，データ構造が異なる．Laplace は二次元格子で隣接する 4 点の平均を繰り返し求める，ステンシル計算を行う．そのため，新旧の値を持つ二次元配列の更新が毎回行われる．行列サイズは 2048×2048 とし，21 回の反復を行った．

EP と CG は NAS 並列ベンチマークの OpenMP C 版であり，NPB2.3 の逐次 Fortran 版を元に作成した．EP は並列領域の最後で配列リダクション計算を行う以外は同期のない，ほぼ線形な速度向上が見込める単純な並列プログラムである．CG は共役勾配法のカーネルプログラムで，疎行列の参照で配列を添字とする間接参照によって，不規則なデータ参照が行われる．この間接参照をそのままコンパイルすると最適化が効果的に行えないため，評価に用いたプログラムでは，共有配列のプライベートコピーに対して不規則参照を行うように書き換えた．ただし，このプライベート配列は引数として渡している．なお，EP はクラス W，CG はクラス A のデータを用いた．

表 3 逐次版と 1 スレッド版の実行時間（単位：秒）
Table 3 Execution times of serial and single thread executions.

プログラム	逐次版	1 スレッド版
EP	65.77	65.13 (0.99)
JOR	13.46	10.48 (0.78)
Laplace	14.74	14.38 (0.98)
CG	39.27	45.53 (1.16)

6.3 実行性能と考察

それぞれのプログラムについて，最適化レベルやスレッド数（ノード数とノード内スレッド数）を変化させて，実行性能を評価した．その際，以下のような条件で行った．

- 並列領域で実行する論理スレッドと同数の物理スレッドのみを起動する．
- ループのスケジューリングは，すべて `static` スケジューリングとする．
- 論理スレッドと物理スレッドの対応は静的に決定する（同一ノード内のスレッドは連続したスレッド番号を持つ）．
- ノード数を 1，2，4，8，各ノード内のスレッド数を 1，2，4 と変化させ，それぞれ実行時間を測定する．ただし，ノード内スレッド数は全ノードで同じにする．そのため，プロセッサ数はノード数 × ノード内スレッド数となる．
- ホームノードはファーストタッチ方式で決定する．
- 実行時間は初期化や結果の検証を含まない，カーネル部分のみを測定する．

実行性能は，OpenMP 指示文を無視して逐次コンパイルしたものとの実行時間の比を速度向上率として求める．**表 3** は，逐次版と，並列版を 1 スレッドで実行した場合の実行時間とその比を示す．表から分かるように，最大 22%の性能差が見られる．並列版は一貫性制御の最適化を最大限に行った場合であり，一貫性制御コードのオーバヘッドはごくわずかで，1 スレッドのため通信も発生しない．そのため，性能差はキャッシュや TLB のミスなど，一貫性制御以外の要因と思われる．以下の評価では，この点に注意していただきたい．

図 14，**15**，**16**，**17** は，最大限最適化を行ったときの 4 つのプログラムの速度向上率を示す．図から分かるように，EP と JOR では良好な性能が得られているが，CG と Laplace では特にノード内の性能向上率が低い．以下，さらに詳しく分析する．

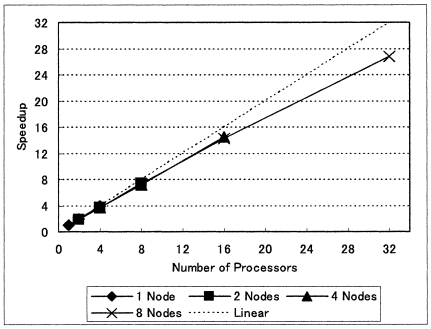

図 14 EP の速度向上率
Fig. 14 Speedup of EP.

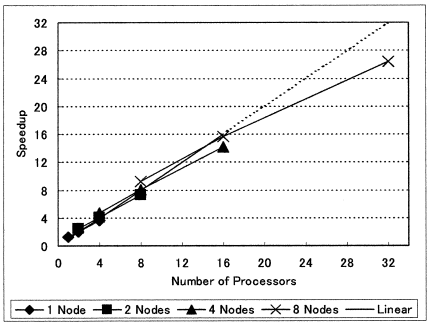

図 15 JOR の速度向上率
Fig. 15 Speedup of JOR.

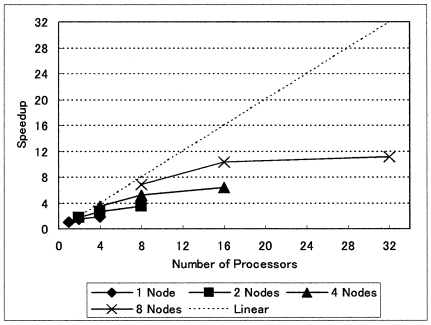

図 16 Laplace の速度向上率
Fig. 16 Speedup of Laplace.

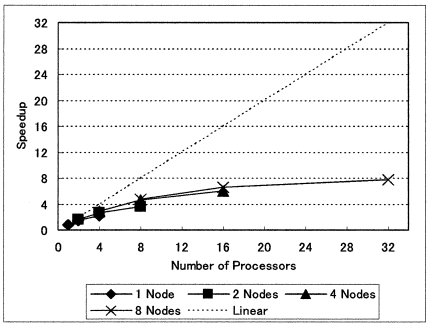

図 17 CG の速度向上率
Fig. 17 Speedup of CG.

表 4 プログラムごとの最適化の効果
Table 4 Effectiveness of optimizations for each program.

最適化	EP	JOR	Laplace	CG
局所最適化	○	○	○	○
手続き間最適化	△	△	△	○
プロトコル最適化	×	△	△	△
アラインメント最適化	△	△	△	△
通信最適化	×	○	○	○

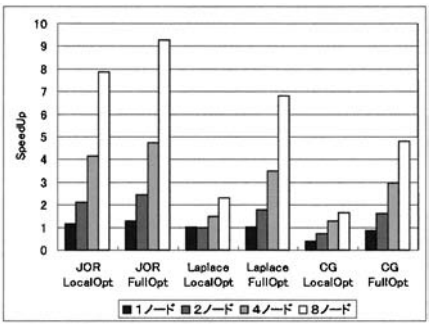

図 18 局所最適化のみと全最適化の速度向上率の比較
Fig. 18 Comparison of the speedup of local and full optimization.

表 4 は，各最適化の効果の有無をプログラムごとに示している．表で○は最適化が適用でき，性能への効果も高かったものを示す．△は最適化を適用できたが，性能への効果は少なかったもの，×は最適化が適用できなかったものを示す．**図 18** は，EP 以外に対して，局所最適化のみ行った場合（LocalOpt）と，最大限最適化を行った場合（FullOpt）について，ノード内スレッド数を 1 とし，ノード数のみ変えた場合の速度向上率を示す．なお，局所最適化を行わない場合の性能は非常に低いため，測定しなかった．

まず，各最適化の効果を検証する．局所最適化は，いずれのプログラムでも効果がある．これは，いずれのプログラムも配列の規則的参照を含むためである．手続き間最適化は，並列領域内で読み込みのみ行うデータの一貫性制御コードを削除できるが，性能への効果は小さい．これは，一貫性制御コードの実行オーバヘッドは削減できるが，通信量は削減されないためである．CG の場合，手続き間ポインタ解析によって，不規則に参照する配列が局所データであることを検出したことにより，融合などの最適化の効かなかった一貫性制御コードを削減した効果が大きい．プロトコル最適化とアラインメント最適化は，いずれもスカラデータか小さな配列にしか適用できなかったため，性能への効果は少ない．CG では大きな配列でページ単位のフォルスシェアリングが生じており，ページ単位

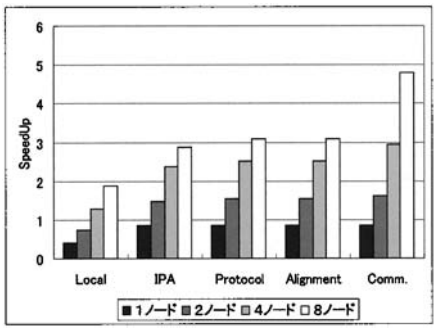

図 19 CG の最適化レベルごとの速度向上率
Fig. 19 Speedup of different optimization levels for CG.

でのアラインメントによる性能改善の余地があることが分かっている．通信最適化は，CG や Laplace のように，並列領域内で規則的に書き換えられる配列の多い場合に効果が高い．

次に，プログラムごとに効果を比較する．まず，EP では共有データの参照が少ないため，局所最適化のみで十分であった．JOR では，共有データの読み込みが中心のため，元々一貫性制御のための通信が少なく，局所最適化で一貫性制御コードの実行オーバヘッドを削減するだけで高い性能が得られた．なお，JOR ではスーパーリニアな性能向上が得られているように見えるが，これは逐次版よりも並列版の方が 1 スレッドでの実行性能が良いためである．Laplace では，共有データの書き換えが頻繁に生じるため，規則的に書き換えられる配列のスレッド間のデータ依存関係を解析できたことによる通信最適化の効果が大きい．CG では，手続き間ポインタ解析によるプライベートデータの検出と，通信最適化が効果が高かった．CG の最適化内容を変えたときの速度向上率を，**図 19** に示す．最終的に，EP と JOR では良好な性能が得られたが，CG と Laplace では特にノード内の性能向上率が低い．CG と Laplace は，SMP 用にコンパイルした場合の性能と Omni/CSDSM の 1 ノード時の性能はほぼ同じである．そのため，性能低下の原因は一貫性制御のオーバヘッドによるものではなく，データの書き込みが多いために共有バスのバンド幅が不足するためと思われる．また CG の場合は，計算の粒度が小さく，制御の同期のオーバヘッドが相対的に大きいことも性能低下の要因である．

7. 関 連 研 究

7.1 OpenMP のクラスタ向け実装

OpenMP はこれまで主に SMP と CC-NUMA で用いられているが，SDSM に基づく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[15] と Scherer ら[16] は，TreadMarks を用いた OpenMP の実装について報告している．Omni/SCASH[17] は，SCASH 上で実装した．これらはいずれもページベース SDSM を用いており，コンパイラによる最適化は行っていない．

CC-NUMA や SDSM で重要になるデータ配置（データ分散）の扱い方はさかんに議論されている．Bircsak ら[18] は，NUMA 向けのデータ配置指示文を提案している．Omni/SCASH[17] は，データ分散とスケジューリングの指示文を拡張している．一方，Nikolopoulos ら[19] は，データ分散指示は不要であり，ハードウェアと実行時システムだけで解決できると述べている．また，山中ら[20] は OpenMP プログラムを分散メモリ並列計算機で利用するための，OpenMP の拡張仕様を提案している．

7.2 DSM 向けコンパイル手法

CSDSM に関する研究には以下のものがある．Locust[7] はデータ並列プログラムのデータ参照パターンを解析し，参照パターンに応じて効率の良い一貫性制御方法を選択する．Shasta[8] と Sirocco-S[9] は，同期区間内で一貫性制御コードの最適化を行う．これらは機械語レベルで一貫性制御コードの挿入を行うため，最適化に利用できる情報は限られる．ADSM/UDSM[10),11)] は，手続き間ポインタ解析を行い共有データの参照を検出するとともに，ソースレベルの情報を用いて同期区間内で一貫性制御コードの最適化を行う．

集中共有メモリと局所メモリを持つ並列計算機向けのコンパイラでも，DSM 向けのコンパイラと同様に，リモートデータの参照を削減するための手法が研究されている．吉田ら[21] は逐次プログラムの自動並列化コンパイラで，集中共有メモリを介したデータ転送を削減するタスクスケジューリング法を提案した．山本ら[22] は並列プログラムのコンパイラで，同期区間内で共有データを局所メモリにコピーして使用する方法を提案した．

7.3 並列プログラムの解析と最適化

共有メモリ並列プログラムの解析・最適化には，以下の研究がある．Chow ら[23] は，同期がなく共有変数のある `cobegin/coend` を用いた並列プログラムの解析の枠組みを示した．Grunwald ら[24] は，`cobegin/coend`・並列セクション・イベント同期を持つ並列プログラムで到達定義の解析法を提案し，Ferrante ら[25] がそれを `doall` と配列セクション解析を扱えるように拡張した．Srinivasan ら[26] は，並列性を扱える SSA 形式を提案した．Krishnamurthy ら[27]

は，共有変数を持つ SPMD プログラムのデータフロー解析を行い通信の最適化に利用した．Knoop ら[28] は，`cobegin/coend` のある並列プログラムでのコード移動の方法を示した．Novillo ら[29] と Lee ら[30] は，Concurrent SSA を用いて共有変数を持つ並列プログラムのスカラ最適化の方法を示した．Collard[31] は，並列プログラムに対する配列 SSA 形式を提案した．

Java ではつねにマルチスレッド実行されるため，最適化の際には並列性を考慮しなければならない．たとえば，Choi ら[32] などのエスケープ解析の研究では，プライベートデータの検出や不要な同期の削除を行っている．しかし，メモリモデルの不備[33] などから，これまでの Java コンパイラの多くは最適化の際に並列性を正しく考慮していなかった．

以上のような従来の共有メモリ並列プログラムの解析・最適化の研究には OpenMP を対象としたものはなく，並列化構文やメモリコンシステンシモデルなどに違いがあるため，そのままでは OpenMP に適用することはできない．特に，緩いメモリコンシステンシモデルと構造的な並列性記述を有効利用した方法は，本研究以外に見当らない．

汎用的なマルチスレッドプログラムでなく，HPF のようなデータ並列言語に対する最適化の研究は数多く行われている．たとえば，Pugh ら[34] は，同一並列ループの反復間の依存関係を求め，通信の最適化に利用した．また，Chandra ら[35] は，細粒度 SDSM 向けの HPF コンパイラで，コンパイル時に一貫性制御の最適化を行った．我々の最適化も発想はよく似ているが，これらよりも一般的な並列プログラムに適用できる．

8. ま と め

本稿では，OpenMP プログラムの並列性を考慮したデータフロー解析手法と，得られた結果を利用した Omni/CSDSM の一貫性制御コードの最適化手法を示した．OpenMP を用いた並列プログラムは，構造的な並列性記述と緩いメモリモデルを利用することにより，他の共有アドレス空間並列プログラムよりも容易に最適化が行える．それらの利点を生かした，OpenMP プログラムに対する手続き間解析や並列データフロー解析の方法を示した．得られた情報を用いて，一貫性制御コード，一貫性制御プロトコル，データ配置（アラインメント）などの最適化を行った．これらの方法を Pentium II Xeon プロセッサと Myrinet で構成した SMP クラスタ向けに実装し，最適化の効果を確認した．8 ノード × 4 スレッドで逐次実行の 7.9 倍から 30.0 倍の性能が得られた．本研究により，SMP クラスタにおいても共有メモリモデルを用いた並列プログラミングが有効であることが示された．

今後の研究課題として，より広範囲のプログラムを用いた性能評価や，OpenMP プログラムの自動データ配置，CSDSM 以外のプラットホームでのコンパイル時最適化などを考えている．

謝辞 本研究についてご支援・ご討論していただいた Omni OpenMP プロジェクトの皆様および TEA グループの皆様に感謝いたします．また，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) OpenMP ARB: *OpenMP: Simple, Portable, Scalable SMP Programming.* `http://www.openmp.org/`
2) 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).
3) 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 2000-HPC-82, pp.71–76 (2000).
4) Satoh, S., Kusano, K. and Sato, M.: Compiler Optimization Techniques for OpenMP Programs, *Proc. 2nd European Workshop on OpenMP* (2000).
5) 佐藤茂久，草野和寛，佐藤三久：OpenMP 向けコンパイラ支援ソフトウェア DSM，情報処理学会論文誌，Vol.42, No.4, pp.788–801 (2001).
6) 佐藤茂久，草野和寛，佐藤三久：OpenMP 向けコンパイラ支援ソフトウェア DSM の性能評価，情報処理学会研究報告 2001-ARC-141, pp.7–12 (2001).
7) Chiueh, T. and Verma, M.: A Compiler-Directed Distributed Shared Memory System, *Proc. 9th ACM Int'l Conf. on Supercomputing*, pp.77–86 (1995).
8) 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).
9) 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).
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) 丹羽純平，松本 尚，平木 敬：コンパイラが支援するソフトウェア DSM 機構：ADSM と UDSM の性能評価，情報処理学会研究報告 99-HPC-77 (1999).
12) 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 Computer*, Vol.29, No.2, pp.18–28 (1996).
13) 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).
14) 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・インターフェース AM 通信，情報処理学会論文誌，Vol.40, No.5, pp.2225–2234 (1999).
15) Hu, Y.C., Lu, H., Cox, A.L. and Zwaenepoel, W.: OpenMP for Networks of SMPs, *Proc. 13th Int'l Parallel Processing Symp. and 10th Symp. on Parallel and Distributed Processing* (1999).
16) 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).
17) 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).
18) 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).
19) Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J. and Ayguade, E.: Is Data Distribution Necessary in OpenMP, *Proc. Supercomputing 2000* (2000).
20) 山中栄次，金子正教，堀田耕一郎：分散メモリ並列計算機向け OpenMP 拡張仕様，情報処理学会研究報告 2001-ARC-141, pp.13–16 (2001).
21) 吉田明正，越塚健一，岡本雅巳，笠原博憲：階層型粗粒度並列処理における同一階層内ループ間データローカライゼーション手法，情報処理学会論文誌，Vol.40, No.5, pp.2054–2063 (1999).
22) 山本淳二，鬼頭宏幸，美辺央希，山口喜弘，天野英晴：ローカルメモリをもつマルチプロセッサのソフトウェア環境 EULASH のプリコンパイラの評価，電子情報通信学会論文誌，Vol.J81-D-1, No.10, pp.1130–1140 (1998).
23) Chow, J.-H. and III, W.L.H.: Compile-time Analysis of Parallel Programs that Share Memory, *Proc. 19th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pp.130–141 (1992).
24) Grunwald, D. and Srinivasan, H.: Data Flow Equations for Explicitly Parallel Programs, *Proc. 4th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pp.159–168 (1993).
25) Ferrante, J., Grunwald, D. and Srinivasan, H.: Computing Communication Sets for Control Parallel Programs, *Proc. 7th Int'l Workshop on Languages and Compilers for Parallel Computing*, pp.316–330 (1994).
26) Srinivasan, H., Hook, J. and Wolfe, M.: Static Single Assignment for Explicitly Parallel Programs, *Proc. 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pp.260–272 (1993).
27) Krishnamurthy, A. and Yelick, K.: Optimizing Parallel Programs with Explicit Synchronization, *Proc. 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.196–204 (1995).
28) Knoop, J. and Steffen, B.: Code Motion for Explicitly Parallel Programs, *Proc. 7th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pp.13–24 (1999).
29) Novillo, D., Unrau, R.C. and Schaeffer, J.: Concurrent SSA Form in the Presence of Mutual Exclusion, *Proc. 1998 Int'l Conf. on Parallel Processing* (1998).
30) Lee, J., Padua, D.A. and Midkiff, S.P.: Basic Compiler Algorithms for Parallel Programs, *Proc. 7th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pp.1–12 (1999).
31) Collard, J.-F.: Array SSA for Explicitly Parallel Programs, *Proc. 5th European Conf. on Parallel Processing* (1999).
32) Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V.C. and Midkiff, S.: Escape Analysis for Java, *Proc. 1999 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pp.1–19 (1999).
33) Pugh, W.: The Java Memory Model. `http://www.cs.umd.edu/~pugh/java/memoryModel/`
34) 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).
35) Chandra, S. and Larus, J.R.: Optimizing Communication in HPF Programs on Fine-Grain Distributed Shared Memory, *Proc.*

6th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming, pp.100–111 (1997).

(平成 13 年 5 月 8 日受付)
(平成 13 年 8 月 22 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 年同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 13 年まで新情報処理開発機構つくば研究センタに出向．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．

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 年同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．同機構並列分散システムパフォーマンスつくば研究室室長．平成 13 年より，筑波大学電子・情報工学系教授．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等の研究に従事．日本応用数理学会会員．