社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－EVA－8　(2)
2004／3／18

PC クラスタを対象とするループレベル並列化機能を有する MIRAI コンパイラにおけるループ再構築部の実装

信原　裕文[†]　峰尾　昌明[†]　上原　哲太郎*　齋藤　彰一[††]　國枝　義敏[††]

概要

我々は，ソフトウェア分散共有メモリ（Software Distributed Shared Memory: Software DSM）システム Fagus を実行時環境とする自動並列化コンパイラ MIRAI を開発している．この両者の組み合わせにより，分散メモリ型並列アーキテクチャにおけるデータの自動分割すなわち初期分散配置の決定および実行時の再配置の自動化を目指す．今回，MIRAI コンパイラに同 DSM 環境を意識したループ再構築部を実装した．さらに，サンプルプログラムを作成し，実装したループ再構築部の有効性を評価，検証した．その結果，ループ再構築部の有用性が証明された．

Implementation and Evaluation of the Loop Restructuring Feature of the Loop-level Parallelizing Compiler MIRAI for PC clusters

Hirofumi Nobuhara[†]　Masaaki Mineo[†]　Tetsutaro Uehara*
Shoichi Saito[††]　and　Yoshitoshi Kunieda[††]

Abstract

An automatic parallelizing compiler MIRAI which is supposed to use software distributed shared memory system Fagus as its runtime support system, is now under developing. Source programs can be automatically parallelized in view of not only tasks but also shared data, namely data distribution/re-distribution, by this approach using MIRAI and Fagus. MIRAI has been newly designed and implemented with loop restructuring feature. Several sample programs were made in order to verify and evaluate the loop restructuring feature. This paper describes both the implementation details of the loop restructuring feature and its evaluation reports.

1. はじめに

現在，我々は PC/WS クラスタをはじめとする分散型を含む幅広い並列アーキテクチャを対象とする自動並列化コンパイラ MIRAI（MultI-gRAIn automatic parallelizing and distributing compiler;以下，MIRAI コンパイラ）を開発している．その実行時システムとしては同研究室にて開発中のソフトウェア分散共有メモリ（Distributed Shared Memory: DSM）システム Fagus[1)]（以下，Fagus）を用いる．

一般に，プログラム全体の実行時間の大部分は，プログラム中のループ部分が占めている．よって，現 MIRAI コンパイラの並列化の対象は，入力されたソースコード内の多重ループ部分とする． さらに，ループ部分の最適化を行うことにより，プログラム全体の実行時間に大きな効果を与えることが可能である．

[†]和歌山大学大学院システム工学研究科
Graduate School of Systems Engineering, Wakayama University
[††]和歌山大学システム工学科
Faculty of Systems Engineering, Wakayama University
*京都大学大学院工学研究科
Graduate School of Engineering, Kyoto University

そこで本論文では，ループ再構築部を実装し，その効果を評価することとした．

2. ソフトウェアDSMシステムFagus

Fagus は，PC/WS クラスタをターゲットアーキテクチャとし，ユーザとコンパイラに cc-COMA (compiler controlled Cache Only Memory Architecture)[2]環境をライブラリとして提供するソフトウェア分散共有メモリシステムである (図1)．

2.1 cc-COMA 実行時環境

一般に COMA とは，全プロセッサエレメント (注1) が一つのメモリアドレス空間を共有し，各 PC のローカルメモリは，この「共有メモリ」のキャッシュとして動作する．すなわち，アドレス空間のどの部分部分が各 PC のローカルメモリに割り当てられるかは動的に変化する．したがって，ある PC において，自身のローカルメモリに存在しないデータへの参照(キャッシュミス)が発生した場合は，データを保持する PC (現所有者)が検索され，その現所有者から最新の値を転送する．また，読み込み参照時にデータ，すなわちメモリ空間のある部分が複数の実行マシン上に複製されて保持されることがある．この場合，データの一貫性制御が必要となる．なお, Fagus ではメモリアドレス空間を4096 バイトの固定サイズ単位(以下これを，OS のページングにならい「ページ」と呼ぶことにする)に分割し，キャッシングの単位としている．

cc-COMA の基本的なアイデアは，上述の COMA 環境をソフトウェアで実現し，共有メモリの一貫性制御を基本的にコンパイラにゆだねようとするものである．本環境を用いることにより，コンパイラがタスクとともにデータを分割して，いずれかの(最適でなくても良い)PC に配置すれば，各タスクで使用されるデータは実行全体を見渡せば最もよく参照する PC が自動的にほぼ占有する．よって，通信量はほぼ最小化されることが期待できる．以上により，プログラマは HPF[3](High Performance Fortran)等と異なり，

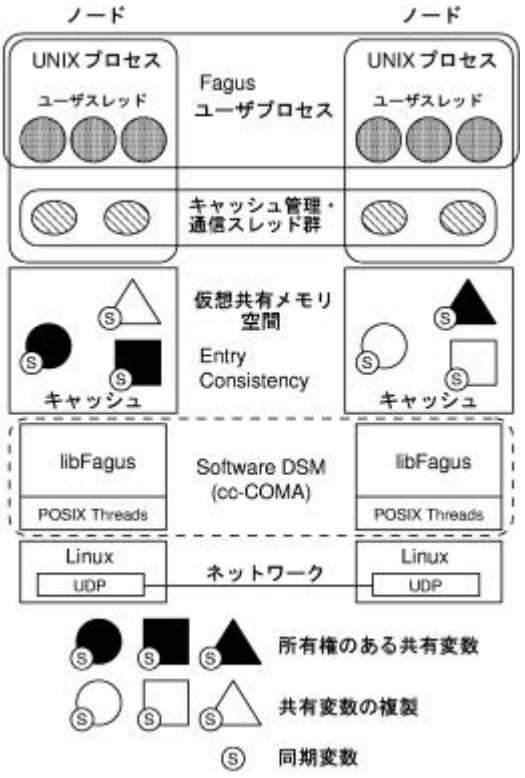

図1　Fagus の全体構成

データ分割や配置をコンパイラに対して指定する必要がない．コンパイラにとってもデータ分散とタスク並列化とを切り離して考えることができ，問題の複雑度を減らすことができる．

しかし，単純な COMA 環境では，PC 間での一貫性制御のためデータ転送が頻発し，結果的に実行性能が低下するおそれがある．そのため，Fagus ではより弱い整合性(Weak Consistency)制御モデル[4]の EC (Entry Consistency)モデル[4]を導入する．したがって，コンパイラは，データ参照パターンを解析して，1)並列タスクに分解し，2)共有変数に対し同期変数を自動的に生成し，関連付けるプリミティブを呼び出す文と，3)同期変数を用いた一貫性制御(共有変数の獲得・解放)を行うプリミティブを呼び出す文を目的コード中の適切な位置に埋め込むだけで並列化できる．コンパイラがデータ参照パターンを詳細に解析することにより，冗長なデータ転送や細切れになったデータ転送を排除し，データ転送と計算をできるたけ並行させながら，自動的かつ単純な COMA より実行効率良く並列化することができる．

注1：本稿では，以下わかりやすいPCクラスタとして説明することとし，単にPCあるいは図1では「ノード」と表記する．
注2：図2の配列要素は4Bの整数型とし，一行の要素数が1024であるため，一行がFagusのページ単位である4096Bとなる．

2.2 配列のデータ分割

配列を共有する場合には，MIRAI コンパイラは図2のように行方向にデータ分割し（注2），行ごとに同期変数一つと関連付ける．Fagus のページの整数倍単位に正しく関連付けを行うためには，MIARI コンパイラは，一行のサイズがページの整数倍となるように，切り上げる．Fagus は，こうしたデータ分割の処理をコンパイラにゆだねている．

通常すべての共有変数アクセスの前後で同期変数の獲得・解放を行い，排他制御を実現する．したがって，図２の二次元配列の場合，行単位に排他制御される．行方向に順にアクセスを行う場合（以後，行方向のアクセスパターンと言う），参照の局所性からキャッシュ(ページ)ミスが発生する頻度が低いのに対し，列方向に順にアクセスを行う場合（以後，縦方向のアクセスパターンと言う），キャッシュ(ページ)ミスが頻発し，並列化の効果は期待できないだけでなく，場合によってはページの取り合い，ピンポン転送，さらにはデッドロックを引き起こす可能性もある．

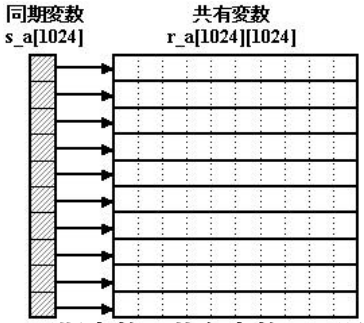

図 2　同期変数と共有変数との関連付け

3. 自動並列化コンパイラ MIRAI

MIRAI コンパイラは Microsoft Windows 2000 上で同社製 Visual C++ 6.0 を用いて開発中であり，Windows9x/Me/NT/2000/XP を動作環境とする．MIRAI コンパイラが生成する並列処理用のC 言語により記述された目的コードは，gcc により再度コンパイルされ，Fagus 上で並列実行される（図 3）．ループレベル並列化のため，各 PC はループボディ単位に処理を分担し，並列実行する．

MIRAI コンパイラはパーザにより Fortran のソースプログラムを読み込み，中間表現である CFG[5]（Control Flow Graph）と抽象構文木[5]（Abstract Syntax Tree: AST）を生成する（図 3）．次に，MIRAI コンパイラでは，並列化対象を多重ループに限定しているため，CFG からループ構造を抽出し，階層タスクグラフ（Hierarchical Task Graph: HTG）[6]に変換する．概念としては，HTG は CFG より上位となるが，プログラミング上，両者を自由に行き来するために，クラスとしては両者を融合している．HTG への変換後，依存解析や最適化の負担を減らすため，まず単純な最適化として，HTG 内の AST を走査し，定数の畳み込み，定数伝播[5]を行う．次に，最適化[5]と依存解析[7]を行い，可能な限り並列化可能なループを特定する．依存解析結果はデータ依存グラフ[7]（Data Dependence Graph: DDG）の形で保持され，後の最適化部（並列化を含む．以下同様）にて使用される．さらに，データフローグラフ（Data Flow Graph: DFG）を生成し，並列化可能なループ内で使用されている変数の特定，配列データへのアクセスパターンなどを調べる．HTG，依存解析結果，DFG から得られた情報を基にループ再構築部にてプログラムの最適化を図る．ループ再構築手法を適用することにより，ループの構造が改変されるため，適用前の依存解析結果や DFG は正確ではなくなる．よって，ループ再構築後，再び依存解析とDFG の再構築を行う．さらなるプログラムの最適化のため依存解析から DFG の再構築までの処理は一般に繰り返し実行する．ループ再構築部が最適なプログラムであると判断する

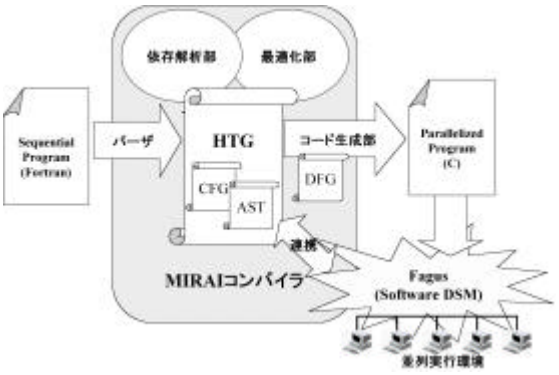

図 3　MIRAI の概要図

まで一連の繰り返し処理を行う．最後に並列化の方針を組み立て，コード生成を行う．

4. ループ再構築部の設計と実装

ループ再構築部では，実行時環境である Fagus の性能を最大限に引き出すため，ループ内部の構造を再構築し，目的コードの最適化を図る．

これまで数多くのループ再構築手法が提案されているが，試作版 MIRAI コンパイラでは，まず第一段階として大きな効果が期待できる Loop Fusion, Loop Interchange[7]を実装することにした．同時に，この両手法の適用のために必要な，Loop Distribution, 計算式の並べ替え[7]の実装を行うことにした．また，Loop Fusion の適用箇所を増大するために Loop Peeling[7]も実装した．

4.1 各手法の適用に関する考察と適用条件

分散メモリ型並列アーキテクチャをターゲットとするために，各ループ再構築手法を適切に選択し，各手法の適用場面を限定する必要がある．以下，考察した手法ごとにまとめる．

4.1.1 Loop Fusion

Loop Fusion の効果は二つの要素により決定付けられると考えられる．一つは共有変数の配列サイズであり，もう一つはループボディ内の計算量である．そこで，上記二項目について何らかの関係を見出すべく予備実験を行った．

実験内容は，適用を行う二ループのループボディ内の計算量をそれぞれ 5個と 10 個の代入文と固定し，配列サイズを1024 から4096 まで変化させた場合の実行時間の測定である（図 4）．この実験で，どの配列サイズでも，Loop Fusion 適用後は適用前の60%程度の実行時間となった．つまり，Loop Fusion の効果に配列サイズはおそらく一般に無関係であることがわかった．よって, Loop Fusion の効果は，ループの繰り返し処理のオーバヘッドとループボディ内の計算量の比率により決定される．これとは別に，ループボディ内 の計算量（粒度）がより多い方が並列実行の効果も期待できるため，ループボディ内の

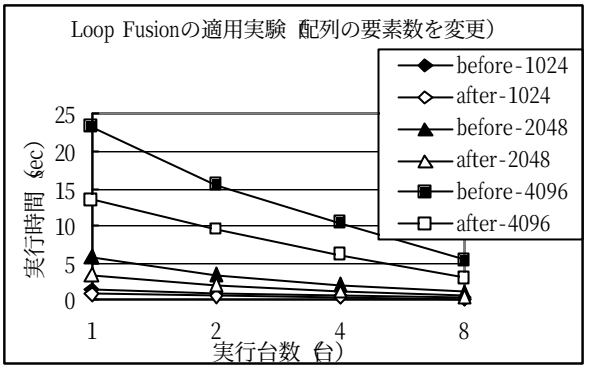

図 4　Loop Fusion の適用実験（配列の要素数を変更）

計算量を増大する Loop Fusion は特に適用場面を考慮する必要なく, 積極的に適用すべきであると考えられる．

4.1.2 Loop Peeling

Loop Peeling は，本来 Loop Fusion の適用場面を増大するための手法であり，処理内容もループの繰り返し数を分割するだけであり, 特に Loop Fusion が適用できるならば，並列実行効率を下げる危険性は一般にないと言える．よって，適用可能な場合は積極的に行う．

4.1.3 Loop Distribution

通常 Loop Distribution は多重ループ内のループを全て完全入れ子ループにし，他の手法の適用場面を増大させるために用いられる．しかし，分散メモリ型並列アーキテクチャでは，ループボディ内の計算量が少ない場合には並列実行することで逆に実行時間が増大する恐れがある．予備実験として，一つのループボディ内に 10 個の代入文があるサンプ

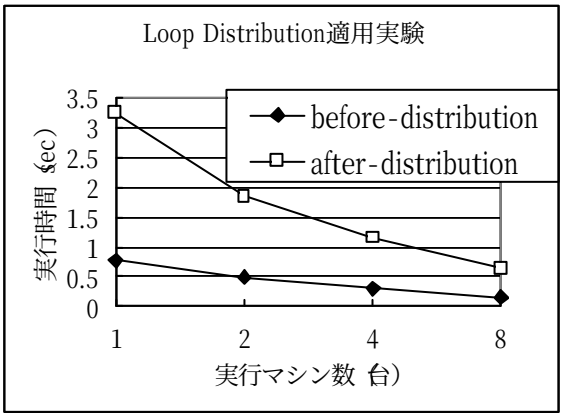

図 5　Loop Distribution 適用実験結果

ルプログラムを用意し，このプログラムに同手法を適用して，10 個のループに分割した．図 5 は，この変換前後のプログラムの実行時間を比較した図である．図 5 では，同手法を適用することで実行時間が約 4 倍に増大している．よって，試作版 MIRAI コンパイラでは，同手法を完全入れ子ループ作成時のみに適用を行うことにした．

4.1.4 計算式の並べ替え

一般に依存方向が下向きであれば，同期を挟みながら並列実行が可能（いわゆる DO-ACROSS 型ループ）である．しかし，これをFagus で実現すると，逐次実行時よりも実行速度が遅くなってしまう．試作版 MIRAI コンパイラでは安全のため，「並列実行による実行時間の減少」＞「並列実行のための通信時間の増加」を保証できなければ並列実行を行わない．よって，計算式の並べ替えは他手法の適用を可能とするためにのみ適用する．

4.2 ループ再構築手法の適用順序

試作版 MIRAI コンパイラでは，Loop Peeling，Loop Distribution，計算式の並べ替えの各手法は，4.1 で記した結果と考察より，Loop Fusion や Loop Interchange の前処理として適用する．

同期変数の獲得・解放の回数を減少させることが可能な Loop Fusion に対し，Loop Interchange は同じく同期変数の獲得・解放の回数を減少させるだけでなく，ループボディ内の縦方向のアクセスパターンを横方向（2.2 参照）へと矯正できるため効果が大きい．さらに，この両手法を適用可能な場合，どちらの手法を優先しても最終的には同じ結果に収束する．しかし，場合によっては Loop Interchange を一回適用するのみで最適なプログラムに変換される場合がある．よって，Loop Interchange とLoop Fusion に関しては，Loop Interchange を優先して適用することにした．

以上の考慮点から，試作版 MIRAI コンパイラではループ再構築部を今回図 6 のように設計した．

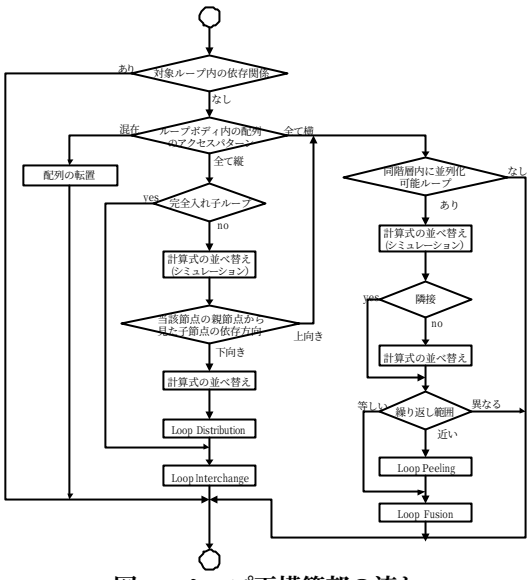

図 6　ループ再構築部の流れ

5. 評価と考察

本研究の評価方法として，実装した各種ループ再構築手法を適用可能なテストプログラムを作成する．作成したプログラムに対してループ再構築手法を適用した場合と適用しなかった場合の実行時間を計測し，比較する方式を採った．以下，4 章で予備的に行った実験で明らかとなった以外の事項について，まとめ報告する．

5.1 Loop Fusion の適用実験

ループボディサイズが 5 と10 の 2 つのループからなるテストプログラムに対し，Loop Fusion を適用した場合と適用しなかった場合の実行時間を測定した．測定結果を図 7 に示す．

Loop Fusion の適用により，ループの繰り返しによるオーバヘッドがほぼ半減され，一台での実行時でも，効果が期待できる．図 7 にはこの様子が顕著に表れている．

並列化ループの効率向上のためには，ループの繰り返しによるオーバヘッドを軽減し，ループボディ内の計算量（粒度）を増加させることが必要である．Loop Fusion はこれら両要項を満たす有効な手法である．本研究では Loop Fusion が与える効果的な特長を MIRAI コンパイラ上で実現できたと言える．

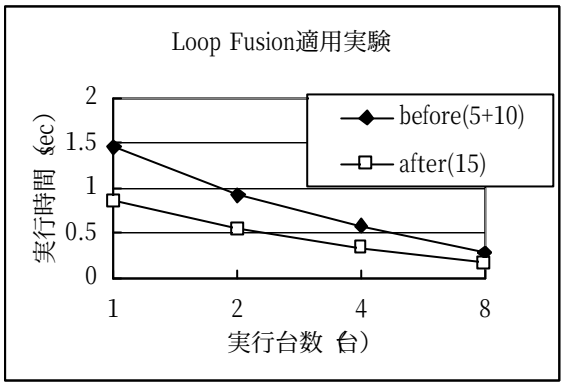

図 7　Loop Fusion 適用実験結果

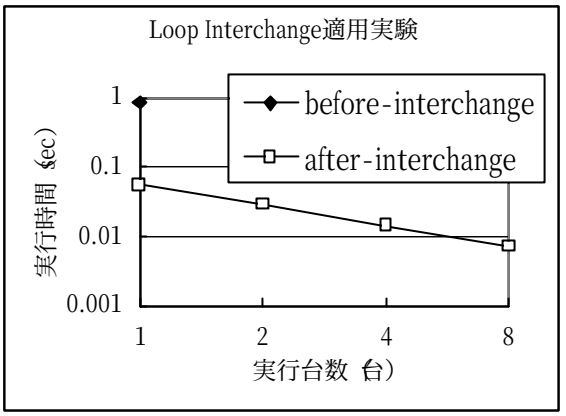

図 8　Loop Interchange 適用実験結果

5.2 Loop Peeling の適用実験

試作版 MIRAI コンパイラに Loop Peeling を実装したことにより，実装前は Loop Fusion を適用できなかったプログラムを，Loop Fusion 適用可能な形へ変換できた．繰り返し範囲の異なるループは，一般的に珍しくはないため，Loop Peeling の実装は Loop Fusion の適用上，非常に効果的であると言える．よって，本研究にて行った Loop Peeling の実装は MIRAI コンパイラにおいて，有効であると考える．

5.3 Loop Interchange の適用実験

ループ中の配列へのアクセスパターンが縦方向であるテストプログラムに対し，Loop Interchange を適用した場合としなかった場合の実行時間を測定した．測定結果を図 8 に片対数グラフの形で示す．手法適用前のプログラムでは，配列へのアクセスパターンが縦方向であるため，同期変数の獲得・解放が頻発する（2.2 参照）．そのため一台から八台まで台数を増やして測定を試みたが，処理が終了せず，逐次実行時の結果のみ表示した．これに対し，同手法適用後はアクセスパターンが横方向に最適化されているため，実行台数の増加に伴い，実行時間が減少している．図 8 ではこの様子がよく表れている．

6. おわりに

実施した評価実験から，今回実装を行ったループ再構築部は有効であったと考える．また，単に各手法を独立して適用するだけではなく，適用対象ループの解析を行い，個々のループに対して異なる最適化手順を作成する「賢い最適化部」に近いレベルまで視野に入ってきたと考える．

参考文献

1) 横手 聡，林 章仁，齋藤 彰一，上原 哲太郎，國枝 義敏，“高速通信ライブラリ Wind を用いたソフトウェア分散共有メモリシステム Fagus の性能評価，”情報処理学会研究会報告 2001-OS-88, Vol. 2001, No. 78, pp.35-42 (2001.7).

2) Saito, S., Uehara, T., Joe, K. and Kunieda, Y., “cc-COMA: the compiler-controlled COMA as a framework for parallel computing,”Innovative Architecture for Future Generation High Performance Processors and Systems '98, pp.114-119 (1999).

3) High Performance Fortran Forum，”High Performance Fortran 2.0 公式マニュアル，”シュプリンガー・フェアラーク東京(1999).

4) Ａ・Ｓ・タネンバウム著 水野忠則，鈴木健二，宮西洋太郎，佐藤文明訳，“分散オペレーティングシステム，”プレンティスホール(1996).

5) 中田 育男，“コンパイラの構成と最適化，”朝倉書店(1999).

6) M.Girkar and C.D.Polychronopoulos.，”The hierarchical task graph as a universal intermediate representation,”International Journal of Parallel Programming, 22(5), pp.519-551 (October 1994).

7) Hans Zima/Barbara Chapman 共著 村岡 洋一 訳，“スーパーコンパイラ，”(オーム社, 1995 年).