2003－MPS－45 (6)
2003／6／24

自動並列化コンパイラMIRAIにおけるループ再構築部の設計と実現方法

信原 裕文* 峰尾 昌明* 上原 哲太郎† 齋藤 彰一* 國枝 義敏*

概要 自動並列化コンパイラは逐次処理用のプログラム中に存在するループを並列化する際に，データ依存解析を行う必要がある．依存解析により依存があると判定されたループは通常そのままでは並列化不可能である．その中には，ループ再構築手法を適用することで並列化可能となるループもある．本論文では，現在我々が開発を行っている自動並列化コンパイラMIRAIにおけるループ再構築部の設計と実現方法について述べる．また，人手でコンパイルしたループ再構築手法のテストプログラムの予備実験についての評価結果をいくつか示す．

Design and Implementation of the Loop Restructuring Feature for the Parallelizing Compiler, MIRAI

Hirofumi NOBUHARA*, Masaaki MINEO*, Tetsutaro UEHARA† , Shoichi SAITO*, and Yoshitoshi KUNIEDA*

***Abstract*—When loops in sequential programs written in procedural programming languages are parallelized, automatic parallelizing compilers should perform data dependence analysis in order to preserve constraints by data reference order. Although loops with dependences as determined by a dependence analyzer cannot be parallelized as-is, in general, some can be parallelized after applying appropriate loop restructuring optimizations. This paper deals with the design and implementation of the loop restructuring feature of our automatic parallelizing compiler, MIRAI. And also it shows the evaluation results of several pilot studies by hand-compiling some test programs.**

1 はじめに

逐次処理用に記述されたプログラムを並列化する場合, ループ中のデータ参照の順序が変化しないように注意する必要がある．並列実行した際，データ参照の順序が異なると, 逐次実行した際と異なる実行結果になる場合がある．そのため，あらかじめデータ依存解析を行った上で, ループ中のデータの参照順序が保存される場合に並列化可能と判断する必要がある．

しかし, ループ中に依存関係が存在すると判定された場合でもループ再構築を行う最適化手法を適用することで並列化可能となる場合がある．

更に，ループ再構築を行うことで並列度を調節することが可能であり, 並列実行時の速度向上も望める場合がある．また，データの局所化やデータの再配置を行う．

本稿では現在本研究室にて開発中の自動並列化コンパイラMIRAI[1][2]におけるループ再構築部の設計と実現方法を述べる．更に，人手でコンパイルしたループ再構築手法のテストプログラムの評価結果を示し，それらの有効性を検証する．

*和歌山大学大学院システム工学研究科
Wakayama University
†京都大学大学院工学研究科
Kyoto University

2 自動並列化コンパイラMIRAI

現在MIRAIコンパイラ（MultI-gRAIn automatic parallelizing and distributing compiler） [1][2] はMicrosoft Windows2000上でMicrosoft Visual C++ 6.0により開発中である．実行時環境としてはWindows9x/Me/NT/2000を用いる．

図１はMIRAIコンパイラの概要と，ソフトウェア分散共有メモリシステム(DSM)Fagus[2][3]とMIRAIコンパイラとが連係する様子を示した図である．ソフトウェアDSMシステムFagusを用いることにより，コンパイラやプログラマは, 分散メモリアーキテクチャであることを意識することなく, 並列実行を行うことが可能となる．つまり，自動並列化の技術としてPC/WSクラスタをはじめとする分散メモリ型を含む幅広い並列アーキテクチャをターゲットとすることが可能であることを意味している．

図１に示すとおり，MIRAIコンパイラのパーザはフォートラン77で記述されたソースプログラムを読み込み，CFG (Control Flow Graph)，AST (Abstract Syntax Tree) [4]を生成する．現バージョンの並列化対象は多重ループに限定しているので, CFGからループ構造を抽出し, HTG(Hierarchical Task Graph) [5]に変換する．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[4]を行う．こうして生成された中間表現について,最適化と依存解析を行い可能な限り並列化可能なループを特定する．また，DFG(Data Flow Graph) [4]を生成し，並列化可能なループ内に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,DFGから得られた情報を元に並列化の方針を組み立る．最後にコード生成を行い，Fagus制御用のコードを含む，並列処理用のCプログラムを出力する．

MIRAIでは，C++の標準テンプレートライブラリであるSTL(Standard Template Library)[9]を利用している．例えば，MIRAIで採用されている中間表現であるHTG，CFG，ASTおよびDFGはすべて，STLとC++のクラスにより構成されている．

図１中，依存解析部では，GCDテスト，Banerjeeテスト[6][7]，および線形計画法と全探索を用いたテストを実装している．

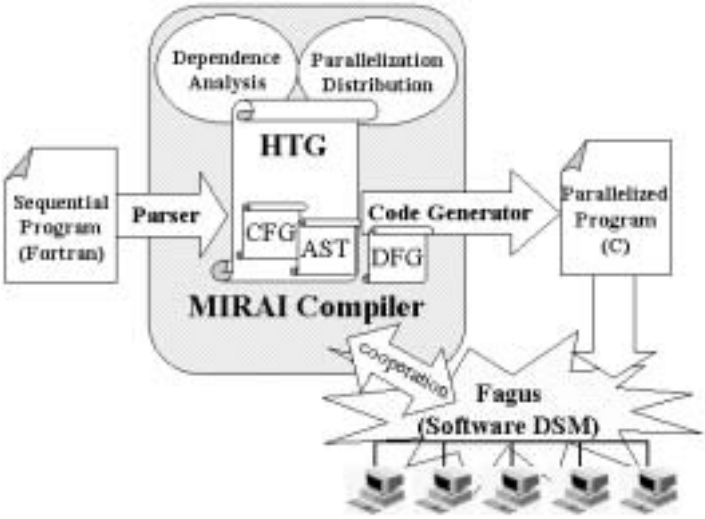

図 1. MIRAIコンパイラの概要図

３　MIRAIにおけるループ再構築に関する最適化

現在MIRAIコンパイラでは，既に提案されている最適化手法[7][8]の中からいくつかを採用し，プログラム中に存在するすべてのループに対し適用可能な手法を可能な限り適用する．

３．１　Loop fusion

Loop fusionは隣接している互いに依存関係の存在しない二つのループを一つのループ[8]へ融合する手法である．Loop fusionを適用することで通常,同期変数の獲得･開放にかかるオーバヘッドを通信のとりまとめにより, 減少させることができる．また,融合後はループボディ内の計算量が増大するため, 各実行ノードの計算量に対し, 通信にかかるオーバヘッドが融合前と比べ，減少する．

表１．Fagus[2][3]の動作環境

PE (Node computer) [up to 16 nodes]	PC-AT compatible CPU: Pentium III 800MHz Main Memory Size: 512MB OS: Linux (kernel 2.4.3)
NIC on each PE (Network Interface Card)	Intel PRO/1000F Server Adapter
Switch	Catalyst 3500 XL
Network	Gigabit ether

例えば，表１の環境下でFagusを用い，16MB(Fagusの4000ページ[3])のサイズの配列を使用し,同期変数の獲得･開放のテストを行った. その結果,7.85秒かかった．１ページ当たり1.9ミリ秒かかることになる. このオーバヘッドを減らすことが実行時間の短縮に繋がる．

並列実行ループについては, Fagus用のコード (同期変数の獲得・開放)をその前後に挿入しなければならない[2][3]．従って，並列化可能なループが隣接して存在し, 互いのループ間にも依存関係が存在しない場合にLoop fusionを適用することで, 同期変数の獲得･開放にかかるオーバヘッドを確実に減らすことが期待できる．

表２ と表３ はLoop fusion適用可能なテストプログラムを人手でコンパイルし, 実行した予備実験結果である. テストプログラム中のループ内では整数型のA,B,Cという３つの二次元配列を用いて計算を行っている. 表２は配列サイズを変更したときに,実行台数を１台～８台へ増やし測定した結果を示す. 表３は同様にループボディ中の代入文の数を変更したときに, 実行台数を１台～８台へ増やし測定した結果を示している．測定結果より，Loop fusionを適用することで, 適用前と比較し，どの場合においても実行時間が20%～50%短縮されている．

３．２　Loop peeling

Loop peelingは通常Loop fusion[8]を適用するため,二つの隣接したループの繰り返し範囲を等しくする手法である．Loop fusionがFagusにとって非常に有効な最適化手法であるため, MIRAIコンパイラにとってLoop peelingは必要不可欠である．

さらに, MIRAIコンパイラでは隣接した二つの並列可能なループだけでなく, 間に他の命令が挟まっている場合でも, 適用を試みる. これは, Loop peeingの目的がLoop fusionの適用箇所を増加させるだけでなく，Fagusのオーナーの決定機能[2]を利用する

機会を増やすためでもある．Loop peelingの適用により，二つの対象ループの繰り返し範囲(初期値，終値)は等しくなる．さらに，二つのループ間にループ繰り越し依存関係が存在する場合，Loop peelingを適用し，存在する依存関係の依存距離と等しい回数分ループボディを剥ぎ取ることで，同一の配列名の添え字式内が共通式となる．この結果，二つのループの前後に同期変数の獲得･開放を行うコードを挿入するのみで並列実行可能となり，それぞれのループ毎に同期変数の獲得･開放を行う必要がないばかりでなく，バリア同期をとる必要もなくなる．

表２. Loop fusion適用前後の実行時間比較（１）

Size of one dimension	# of PEs	Execution time (sec)	
		Before	After
1024	1	0.85	0.62
	2	0.55	0.39
	4	0.35	0.25
	8	0.17	0.12
2048	1	3.39	2.47
	2	2.04	1.48
	4	1.19	0.86
	8	0.63	0.47
4096	1	13.69	9.93
	2	9.54	7.88
	4	6.28	5.07
	8	3.12	2.36

The execution environment is shown in Table１.
The number of assign statements in the pair of loops before loop fusion are two and one respectively.

表３. Loop fusion適用前後の実行時間比較（２）

# of assign statements	# of PEs	Execution time (sec)	
		Before	After
2 statements + 3 statements	1	1.33	0.73
	2	0.85	0.46
	4	0.57	0.31
	8	0.27	0.14
10 statements + 5 statements	1	1.46	0.78
	2	0.93	0.50
	4	0.59	0.29
	8	0.29	0.16
10 statements + 10 statements	1	1.57	0.90
	2	0.99	0.57
	4	0.65	0.37
	8	0.32	0.18

The execution environment is shown in Table１.
The size of each arrays is 1024 x 1024 and each element is an integer.

表４. 二次元配列(INTEGER*4[1024, 1024]:1024ページ)を16台で初期化したときの実行時間

Access pattern and data-partitioning pattern	Execution time (sec)
Both row-wise	0.286
Column-wise and row-wise, respectively	83.009
Same as above, with transpose before/after loop	1.861

The execution environment is shown in Table１.

３．３　Loop interchangeと配列の転置

MIRAIコンパイラは通常，列毎に多次元の配列を

表５. 配列の転置適用前後の実行時間比較

# of arrays	# of PEs	Execution time (sec)	
		Before	After
1	1	1.01	0.0895
	2	224.25	0.0449
	4	186.95	0.0225
	8	69.70	0.0113
2	1	2.50	0.1190
	2	843.50	0.0665
	4	428.23	0.0300
	8	225.96	0.0151
3	1	3.89	0.1511
	2	1022.8	0.1030
	4	659.30	0.0378
	8	364.04	0.0192

The execution environment is shown in Table１.
The size of each arrays is 1024 x 1024 and each element is an integer.
Each execution time of “after” is including the one for transpose.
Each element of each array is added by one in the target loop.

ページの整数倍単位に分割する．したがって，配列へのアクセス方向がデータ分割の方向と異なる場合はアクセスするデータの転送にかかるオーバヘッドが非常に大きくなってしまう(表４参照)．MIRAIコンパイラはこのような場合にLoop interchangeを適用し，多次元配列のアクセスの方向をデータ分割の方向と合わせる．Loop interchangeは依存関係が変化せず，ループが完全な入れ子状になっている場合に適用可能である．単純にLoop interchangeを適用できない場合，MIRAIコンパイラは，実行時に配列の転置を行う．表５に配列の転置を含んだテストプログラムの実行時間を示す．配列の転置適用前のプログラムは，配列へのアクセス方向がその配列の分割方向と異なるため，並列実行を行うと逐次実行を行った場合に比べ非常に遅くなっている．一方，配列の転置を適用した後，並列実行を行った場合は，表５のように台数の増加に伴い，順調に実行時間が短縮している．

３．４　Loop distribution

MIRAIコンパイラは，Loop distributionを適用する

ことで完全入れ子ループを作成する. 完全入れ子ループにすることで, 前節までに述べた各種ループ再構築手法の適用が可能となる.

３．５　Strip mining

MIRAIコンパイラでは一次元の配列を共有変数として扱わないために, これまで一次元の配列にアクセスするループは並列化不可能としていた. しかし，配列要素数が充分に多い場合は,次元数を拡張するScalar expansion[8]と同じ考え方で二次元配列として扱うことが可能である. 変換前の一次元配列をページの整数倍に分割し, ページ毎に行とする二次元配列として扱う. 二次元に拡張された配列を参照するループにはStrip miningを適用し，ループを二重化する. すなわち, 元のループの外側に二次元化された配列要素を縦方向(各行毎)にアクセスするループが生成される.

表６はStrip mining適用前と後のサンプルプログラムの実行時間を比較したものである. 既に述べたが，Strip mining適用前は並列化不可能である．適用後のプログラムを並列実行すると, 実行台数の増加に伴い実行時間が順調に短縮されていることがわかる.

３．６　Loop tiling

Loop tilingは前節で述べたStrip miningを拡張し, 多次元一般化した手法である. 通常, 主にデータの局所性を高めるために使用される．したがって, Fagus上での実行を考えると，この手法は常に有効であると予想される.

表６. Strip mining適用前後の実行時間比較

	Execution time (sec)			
# of PEs	1	2	4	8
Before	0.180	-	-	-
After	0.238	0.140	0.060	0.030

The execution environment is shown in Table 1 .
The size of each arrays is 1024 x 1024 and each element is an integer.
In the loop body, each element is multiplied and divided by 5, repeatedly five times.

Loop tilingを適用すると，ループ内で使用される共有変数は，この場合に限りブロック毎[10]に分割される.

４　おわりに

現在, MIRAIコンパイラでは各種ループ再構築手法が関数として実装されつつある．コンパイラは, これら複数の最適化手法を組み合わせて適用した後の実行時間を推測し, どの手法を, どう組み合わせて適用するのが最適なのか戦略を立てる.

今回実装を行ったループ再構築手法の他にも多くの手法が既に提案されている. これらの手法に関してもMIRAIコンパイラに取捨選択しつつ実装していく予定である.

参考文献

[1] T. Uehara, T. Nakanishi, M. Mineo, S. Saito, K. Joe, A. Fukuda, and Y. Kunieda, “MIRAI:Automatic Parallelizing and Distributing Compiler Based on cc-COMA Approach,” in *2001 Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, vol. III, Hamid R. Arabnia, Ed. CSREA Press, pp. 1193–1199.

[2] M. Mineo, S. Yokote, T. Uehara, S. Saito, and Y. Kunieda, “An Automatic Parallelizing Compiler MIRAI with Data Distribution Function and its Runtime Support System Fagus for Distributed Memory Architecture,” in *2002 Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, vol. III, Hamid R. Arabnia, Ed. CSREA Press, pp. 1451–1457.

[3] S. Saito, S. Yokote, T. Uehara, and Y. Kunieda, “The Implementation of a Compiler Controlled Software Distributed Shared Memory System `Fagus’ as a Runtime Support System for Automatic Parallelizing Compilers,” in *2001 Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, vol. III, Hamid R. Arabnia, Ed. CSREA Press, pp. 1186–1192.

[4] A. V. Aho, and J. D. Ullman, *Principles of Compiler Design*. Addison-Wesley Pub, 1977.

[5] 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, Oct. 1994.

[6] U. Banerjee, *Dependence Analysis.* Kluwer Academic Pub., 1997.

[7] H. Zima, and B. Chapman, *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

[8] D. F. Bacon, S. L. Graham, and O. J. Sharp, “Compiler Transformations for High-Performance Computing,” in *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[9] D. R. Musser, and A. Saini, *STL Tutorial & Reference Guide: C++ programming with the standard template library*. Addison Wesley Longman Inc., 1996.

[10] High Performance Fortran Forum, Rice University, *High Performance Fortran Language Specification, Version 2.0.* Springer, 1997.