プログラミング環境 EULASH におけるコンパイラの実装

鬼頭 宏幸, 美辺 央希, 山本 淳二, 亀井 貴之, 藤原 崇, 天野 英晴

慶應義塾大学理工学部

本論文では高速なローカルメモリとアクセスに大きなレイテンシを伴う共有メモリの双方を持つマルチプロセッサを最適に使用するための環境 EULASH のひとつとして、コンパイラを実装する。

EULASH においてソースプログラムは共有変数のみを用いる軽量なスレッドを用いて記述される。このプログラムをコンパイラが両メモリシステムを効率良く使用するように再構成する。コンパイラによる最適化の結果 Jacobi 法では 90% の共有変数が共有メモリを必要としない型に変換される。また、評価の結果、最適化を行うことでスイッチ結合型マルチプロセッサ上で実行時間が 9% 短縮された。

Design and Implementation of compiler for the EULASH: an environment for efficient use of multiprocessor

H. Kitoh, O. Minabe, J. Yamamoto, T. Kamei, T. Fujiwara, H.Amano

Faculty of Science and Technology, Keio University

In this paper, we describe design and implementation of a compiler for the EULASH programming environment which makes the best use of a multiprocessor with high speed local memory and shared memory with a large latency.

On the EULASH, a program is written with light weight threads using only shared variables, and restructured by the compiler in order to use the both of memory systems efficiently. By the optimization of the compiler, 90% of shared variables in a problem solving simultaneous equations with the Jacobi's method can be converted so as not to use the shared memory. Through the executing evaluation on a switch connected multiprocessor, it is shown that the execution time with the optimization is 9% better than that without optimization.

1 まえがき

EULASH (Environment for Using Local And SHared memory) [1][2] は 2 種類の異なったアクセス速度のメモリシステムを持つマルチプロセッサを対象とした並列ソフトウェア開発 / 実行環境である。バス結合型に代表される単一の共有メモリを主記憶とするマルチプロセッサは、主記憶の競合の問題から大規模化が困難である。このため、多くの大規模マルチプロセッサでは、高速にアクセス可能なローカルメモリと、アクセスに時間のかかるグローバルメモリの 2 種類のメモリシステムを持つ場合が多い。複雑なディレクトリキャッシュを持たない単純な NUMA 型システムや、NYU Ultracomputer[3] や BBN Butterfly TC2000[4] のようなスイッチ結合型マルチプロセッサがこの代表例である。

これらの計算機上で高速に実行を行おうとするならば、高速なローカルメモリを効率的に使用することが不可欠である。もしプログラマがプロセスと変数の割当を注意深く行うならばアクセスに時間のかかる共有メモリの使用を最小限にとどめることができる。しかしながら、このことはプログラマに取って大きな負担となる。

そこで、EULASH 環境では、ユーザに単純なメモリモデル (UMA) を想定したスレッドを使ったプログラミングモデルを提供し、ユーザによる最適化なしでローカルメモリと共有メモリを効率良く利用できることを目標としている。EULASH コンパイラはスレッドを仮想プロセッサに割り当て、共有メモリによるオーバヘッドが最小になるようにプログラムを再構成する。また、プログラム中の変数を分類し、ローカルメモリや共有メモリに配置する。しかし、このことを実現するためには、EULASH がスレッドと共有変数の関係を把握する必要がある。

我々は、スレッドと共有変数の関係を実際よりも小規模なデータで「実行前試行」することによって、各変数、各スレッドのアクセス情報を計測し、プログラムの再構成を行なうコンパイラを提案している [2]。しかし、小規模データによる「実行前試行」と本実行では動作が異なる場合が多く、必ずしも最適な変換が出来なかった。また、あるイタレーションになって始めてアクセスされるような変数を持っているプログラムの場合、小規模データによる「実行前試行」では誤った解析 (コンパイル) をする場合がある。

そこで、本稿では、「実行前試行」を行なわず、データ配置、スレッド配置を静的に解析するコンパイラを EULASH システムに実装し、評価を行なう。

2 プログラミングモデル

EULASH 環境では、プログラマはアプリケーションプログラムを Mach や UNIX 上の C-threads ライブラリ等のマルチスレッドライブラリと似た記法で記述する。この時、プログラマは EULASH ライブラリで用意された関数で並列性を記述する。

その際に、データの配置等は考慮する必要はなく、ユーザはアルゴリズムの記述に専念することが出来る。この場合、EULASH システムはスレッドのプロセッサへの割り当て、データの配置場所を自動的に決定する。一方、ユーザがデータの配置場所、スレッドの割り当て法、スレッドのコンテキストスイッチのポリシーについて決定を行なうことも出来る。

スレッドと変数の関係は静的に解析される。プログラマによって記述されたスレッドはプロセスにまとめられる。 EULASH ではプロセスとは仮想プロセッサであり、これは静的にマッピングされる。大抵は実プロセッサに 1 対 1 に割り当てられるが、パーティションの変更やシステムのエラーによってターゲットマシンのプロセッサ数が変更になった場合などは実プロセッサ 1 つに複数のプロセスが割り当てられることがある。

結果として、単純なプログラミングモデル (図 1(a)) は図 1(b) のように再構成される。

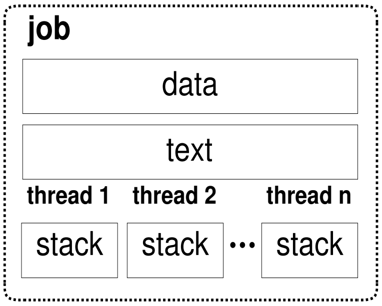
3 EULASH の概要

EULASH の目的はバックエンドマシン上でシングルジョブを高速に実行することである。

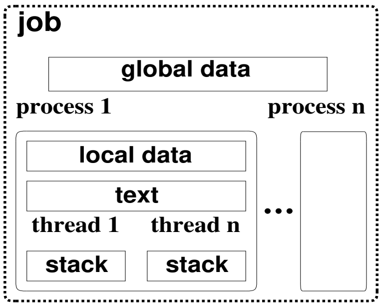
EULASH 環境では次の部分から構成される。

- 変数とスレッドの関係の解析と、スレッドのグルーピングを行なうコンパイラ
- スレッド・プロセスの管理と仮想記憶を行なうカーネル
- ライブラリ

カーネルではユーザモードでは実行できない処理だけを行なう。例えば、ジョブの管理、割り込み処理、仮想記憶である。他の処理についてはライブラリとして提供し、ユーザモード・スーパバイザモード間の移動を出来るだけなくすことで処理のオーバヘッドを最小化する。



(a) model for user



(b) restructured model

図 1: プログラミングモデル

4 EULASH コンパイラ

本コンパイラはユーザが記述したプログラムの共有変数のデータ配置、スレッドの配置を静的に解析し、カーネルの実行モデル (図 1(b)) に変換する。

同一プロセッサにあるスレッドはローカルデータ (ローカルメモリ上におかれる) を共有することが出来るので、効率良く実行するためには関係の密なスレッドを同一プロセスに配置することが重要となる。

4.1 コンパイルの流れ

1. 字句解析・構文解析
   中間言語作成, 各種テーブル (識別子, 定数, 文字列) 作成, プログラムツリー作成
2. 最適化 (一般的な各種プログラム変換)
   誘導変数 (induction variable) の検出, 代入, 定数伝搬 (constant propagation), スカラ前向き代入
3. 最適化 (EULASH 用の最適化)
   スレッドのグループ化, 共有変数へのアクセス予測テーブルの作成, 解析, 共有変数アクセスの文の変更, EULASH のコード、データ転送コードの挿入,
4. コード生成

スレッドのグループ化の後、共有変数へのアクセス予測テーブルをプロセス毎に生成する。まず、並列実行部全体のアクセス予測をするため予測テーブルを作る。次に、さらに細かい解析をするために、各バリア同期ステートメントに注目してアクセス予測テーブルを作る。さらに、そのアクセス予測テーブルによって、共有変数の挙動を決定する。

プログラム実行中、共有変数は大きく分けて 3 つの挙動をすることになる。

共有型: 共有メモリに配置され、読み込み、書き込みが行なわれる。

ローカル型: 実行中、ローカルメモリに配置され、読み込み、書き込みが行なわれる。

移動型: 共有変数をローカルメモリにコピーし、読み込み、書き込みを行なう。また、バリア同期の直前に共有メモリにコピーされる。

4.2 アクセス予想テーブルの作成, 解析

スレッドのグループ化の後、共有変数へのアクセスを予測する。なお、この時プログラムは誘導変数の検出、代入、定数伝搬、スカラ前向き代入など一般的なプログラム変換をした結果、プログラム中で共有変数へのアクセスが完全に予測できるもののみを対象にする。現在、C 言語における `goto` 文、`switch` 文、ポインタ変数は考慮にいれていない。また、アクセス予測するステートメントは並列実行されるものに限る。なお、アクセス予測テーブルはすべて実行中生成されるプロセス分作成する。

1. 各ステートメントについて、共有変数の読み込み、書き込みのリスト (配列の場合はインデックスリスト) を作る。
2. 全ステートメントの共有変数アクセスリストを集計して、共有変数への読み込み、書き込みの数を計算する。
3. さらに、ステートメント中の各バリア同期間に生じる共有変数へのアクセス予測テーブルを作る。これは、まずステートメント中の各バリア同期ステートメントと自然ループの入口に注目し、プログラムフローグラフを図 2のように変換し、変換後の各エッジの共有変数へのアクセスを計算する。

4.2.1 プログラムフローグラフの変換

ユーザはプログラムの並列性を並列実行する関数を指定することによって記述する。ここでは、その

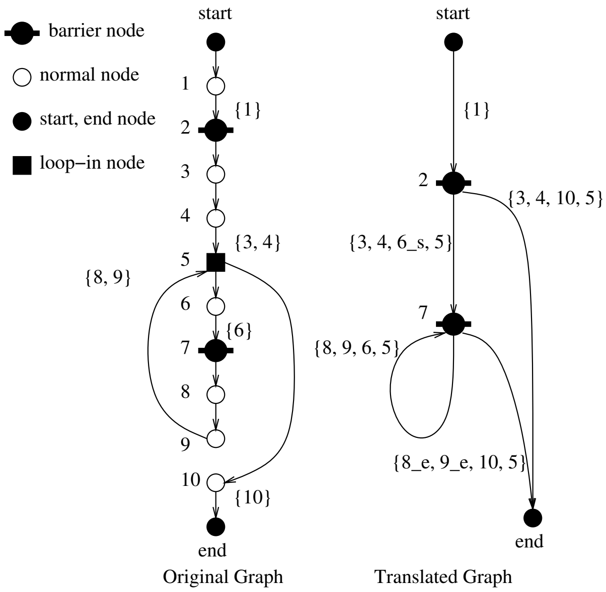


図 2: プログラムフローグラフの変換

関数のプログラムフローをバリア同期ステートメント、ループの入口に注目して解析、変換する。

図 2のようなプログラムフローグラフ *G(N,E,start)* について考える。バリア同期の集合 *SYNC*、ループの入口の集合 *LOOPIN* はそれぞれ以下のようになる。ここでは、*SYNC*、*LOOPIN*、スタートノード (*START*)、エンドノード (*END*) 以外のノードはノーマルノード (*NORMAL*) とする。

$$NORMAL = \{\ 1,\ 3,\ 4,\ 6,\ 8,\ 9,\ 10\ \},$$
$$SYNC = \{\ 2,\ 7\ \},\ LOOPIN = \{\ 5\ \}$$

まず、後述するアルゴリズム (4.2.3) に従い、バリア同期ステートメント、ループの入口について、そこからもっとも近いバリア同期、ループの入口までに到達する可能性のあるステートメントの集合 *BS* を求める。

$E := (e1, e2)$ とする。$e1, e2 \in \{SYNC \cup LOOPIN \cup START \cup END\}$

$$BS(E) := \{N : N \in NORMAL\}$$

図 2では中括弧でくくってあるものが *BS* に該当する。

$$BS((start, 2)) := \{1\}, BS((2, 5)) := \{3, 4\},$$
$$BS((7, 5)) := \{8, 9\}, BS((5, 7)) := \{6\},$$
$$BS((5, end)) := \{10\}$$

次に、*BS(E)* を用いてバリア同期間で実行されるステートメントの集合 *BBS(E)* を求める。

$E := (e1, e2)$ とする。$e1, e2 \in \{SYNC \cup START \cup END\}$

$$BBS(E) := \{N : N \in \{NORMAL \cup LOOPIN\}\}$$

図 2では、ループの直前のバリア同期 (2) からループ内の最初のバリア同期 (7) まで通過するステートメントの集合は $\{3, 4, 5, 6_s\}$ である。ステートメント 6 はそのループ変数の初期値を持ったステートメント 6_s としてステートメントとして扱う。同様にループ内の最後のバリア同期 (7) からループの直後のバリア同期 (この場合存在しないので end) までに通過するステートメントの集合は $\{8_e, 9_e, 5, 10\}$ である。$\{8_e, 9_e\}$ はループ変数の終了値をもったステートメントである。

$$BBS((start, 2)) := \{1\},$$
$$BBS((2, 7)) := \{3, 4, 6_s, 5\},$$
$$BBS((2, e)) := \{3, 4, 10, 5\},$$
$$BBS((7, 7)) := \{8, 9, 6, 5\},$$
$$BBS((7, end)) := \{8_e, 9_e, 10, 5\}$$

4.2.2 アルゴリズム: BS を求める

まず、バリア同期とループの入口から各ステートメントに到達する可能性のあるステートメントの集合 *FS(n)* を *suc(n)* から求める。さらに、*FS(n)* をもとに *BS(n)* を求める。

プログラムグラフ $G(N, E)$ で $n \in N$ とし、直接後続ノードの集合を $suc(n)$ とする [6]。

begin
for every $s \in (NORMAL \cup LOOPIN \cup START \cup END)$ **do** $FS(s) := \phi$ **end for**;
repeat
  $stable$:=**true**;
  for every $s \in N$ **do**
    $new := \phi$;
    if $s \in SYNC \quad || \quad s \in LOOPIN$
    then $N' := \phi$ **else** $N' := suc(s)$**fi**
    for every $n \in N'$ **do**
      $new := new \cup FS(n) \cup n$
    end for
    if $new \neq FS(s)$ **then** $stable$:=**false fi**
    $FS(s) := new$
  end for;
until $stable$;
end

begin
for every $i \in (SYNC \cup LOOPIN \cup START)$ **do**
  for every $j \in suc(i)$
    $k = ((SYNC \cup LOOPIN \cup END) \cap FS(j));$
    $BS(i,k) := (FS(j) \cup j) - k$
  end for
end for
end

4.2.3 アルゴリズム: BBS を求める

BS(E) を用いてバリア同期間で実行されるステートメントの集合 *BBS(E)* を求める。

begin
for every $(e0s, e0d) \in \{E : BS(E) \neq \phi\}$ **do**
  if $e0s \in LOOPIN$
  then
    for every $(e1s, e1d) \in \{E : BS(E) \neq \phi\}$ **do**
      if $e0d == e1s$
        then $BBS(e0s, e1d) := BS(e0s, e0d) \cup$
          $BS(e1s, e1d) \cup e0d$
      else if $e0s \in START \,||\, e0s \in SYNC$
        $BBS(e0s, e0d) := BS(e0s, e0d)$
      fi
    end for
end

4.3 アクセス予測テーブルの解析, データ転送コードの挿入

まず、並列実行される全ステートメントにおける予測テーブルから、一つのプロセスからだけアクセスされる変数、プロセス生成後読み込みだけ行なわれる変数を検出し、このデータをローカル型、それ以外を共有型として取り扱う。

バリア同期間で同じ共有変数をある程度以上アクセスする場合は一旦ローカルメモリにコピーし、ローカルメモリにある共有データをアクセスしたほうが、効率良く実行できる場合がある。これには共有変数をローカルメモリへコピーし共有メモリへ書き戻すコストと、何度か共有メモリにアクセスするコストとのトレードオフである。これはターゲットとなるマシンによって閾値を決め決定することが出来る。このように、共有変数が共有メモリ、ローカルメモリと移動するようなものを移動型として扱う。

実際には、あるバリア同期間で閾値をこえる回数の同じ共有変数への書き込み, 読み込みがあった場合、バリア同期の直後にその変数をローカルメモリにコピーするコードを挿入する。さらに書き込みの場合はその後のバリア同期の直前に共有メモリに書き戻すコードを挿入する。

5 評価

5.1 ターゲットマシン

EULASH のターゲットとしているマシンは、コヒーレントキャッシュを持たない大規模 NUMA やスイッチ結合による UMA である。現在、EULASH はスイッチ結合型マルチプロセッサ SNAIL[5] で稼働している。

SNAIL は EULASH のターゲットの一つであるスイッチ結合型のマルチプロセッサである。SNAIL (図 3) では 16 プロセッサと 16 共有メモリモジュールが SSS-MIN(Simple Serial Synchronized-Multistage Interconnection Network) と呼ばれる高速な MIN によって結合されている。プロセッサボードそれぞれに 4 つの CPU (MC68040) とローカルメモリが搭載されている。共有メモリシステムでは Fetch & Decrement と Test & Set のような同期機構がサポートされている。表 1ではメモリシステムに対するアクセス時間を示している。この表からわかるように SNAIL では共有メモリのアクセス時間がローカルメモリに対するそれより 5 倍程大きい。また、MIN が混雑をすると共有メモリのアクセス時間はこの表の値より大きくなり得る。

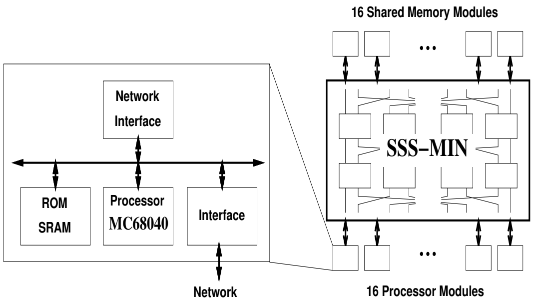


図 3: SNAIL のブロック図

表 1: メモリのアクセス時間 (SNAIL)

memory system	access type	time(ns)
local memory	read/write	250
shared memory	read	1375
	write	250

5.2 最適化の効果

EULASH コンパイラによる最適化の効果の評価は **Jacobi** 法のプログラムによって行なった。このプログラムでは、問題のサイズを 50 元の連立 1 次方程式とし、一つのスレッドが一つの式を計算するようにしたため、50 スレッドが生成される。1 回の反復でバリア同期が 2 回行なわれる。

解析の結果、係数行列と定数ベクトルは値が変化せず、読み込み専用変数として扱えるため、それらはすべてローカル型としてコンパイルされた。変更の生じる解ベクトルでは、全てのスレッドから参照されるため、ローカルに保持できない。また、スレッドのグループ化によってデータ型は変化しなかった。その結果、共有変数の 96%の変数がローカル型としてローカルメモリに配置され、その他は共有型として共有メモリに配置された。

図 4は SNAIL において **Jacobi** 法の最適化の効果を示している。**Unoptimized** は共有変数をすべて共有型として扱い、**Optimized** では共有変数をローカル型と共有型とに分けて扱っている。最適化の結果、9%高速になることがわかる。これは、比較的小さい 50 元の行列について計算したが、それよりも大きいデータサイズで計算を行なえば、より最適化の効果が出て来ると思われる。

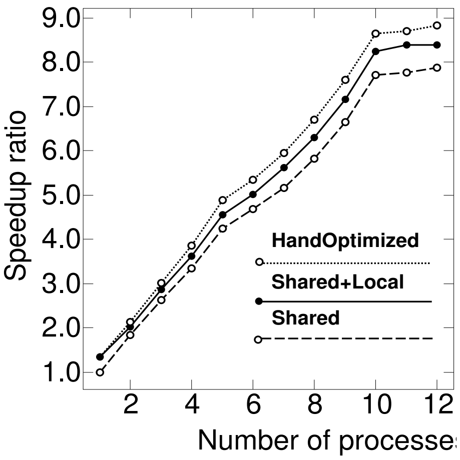


図 4: スピードアップ率 (Jacobi)

6 結論

高速なローカルメモリと低速な共有メモリをもつ並列計算機を効率良く使用するソフトウェア環境として EULASH が提案されている。本稿ではその一部であるコンパイラを提案、実装した。

Jacobi 法においてはコンパイラの最適化によって 96% の共有変数をローカルメモリに配置し、約 9 % の高速化がみられた。

参考文献

[1] J. Yamamoto, D. Hattori, J. Yamato, T. Tokuyoshi, Y. Yamaguchi, H. Amano, "A preprocessing system of the EULASH: an environment for efficient use of multiprocessors with local memory," Proceedings of the Seventh IASTED/ISMM International Conference, pp.68-71.

[2] 山本 淳二, 鬼頭 宏幸, 山口 喜弘, 亀井 貴之, 藤原 崇, 米田 卓司, 天野 英晴, "プログラミング環境 EULASH の SNAIL における評価," CPSY96-43 May. 1996.

[3] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. Mcauliffe, L. Rudolf, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," IEEE Transaction on Computer Systems, Vol. c-32, No.2, 1983.

[4] BBN Laboratories, "Butterfly (TM) Parallel Processor Overview," BBN Computer Company, Cambridge, MA, 1st edition, June 1985.

[5] M. Sasahara et al. "SNAIL: A Multiprocessor Based on the Simple Serial Synchronized Multistage Interconnection Network Architecture," Proceedings of the 1994 International Conference on Parallel Processing, Vol.I, pp.I-117–I-120, Aug. 1994.

[6] Hans Zima, Barbara Chapman, "Supercompilers for Parallel and Vector Computers," Ohmsha, Jul. 1995.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "Compilers Principles, Techniques, and Tools" SAIENSU-SHA, 1990.