計算機アーキテクチャ　105－9
ハイパフォーマンス
コンピューティング　50－9
（1994．3．11）

主記憶共有マルチプロセッサシステム上での マクロデータフロー処理の性能評価

合田 憲人　岩崎 清　松本 健　岡本雅巳　笠原 博徳　成田 誠之助

早稲田大学理工学部

主記憶共有マルチプロセッサシステム上での Fortran プログラムの並列処理では、従来よりマルチタスキング、マイクロタスキングが用いられていた。しかし、マルチタスキングでは、ユーザによる粗粒度タスク間の並列性指定が困難である、スケジューリングオーバーヘッドが大きいという問題がある。また、イタレーション間の複雑なデータ依存や条件分岐によって、マイクロタスキングでは並列化できないループも依然存在する。

本稿では、主記憶共有マルチプロセッサシステム上でのマクロデータフロー処理の性能評価について述べる。本手法では、コンパイラがプログラムの粗粒度タスク (マクロタスク) への分割、マクロタスク間の並列性抽出、スケジューリングコード生成を自動的に行なうことにより、効率良い粗粒度並列処理を行なうことが可能である。

Performance Evaluation of Macro-dataflow Computation on Shared Memory Multi-processor System

Kento AIDA　Kiyoshi IWASAKI　Ken MATSUMOTO　Masami OKAMOTO
Hironori KASAHARA　Seinosuke NARITA

WASEDA University

Parallel processing of Fortran programs on a shared memory multi-processor system has been implemented using multi-tasking and micro-tasking. However, multi-tasking has drawbacks such as difficulty in the extraction of parallelism among coarse grain tasks by users and large scheduling overhead. And, there still exist sequential loop that cannot be concurrentized by micro-tasking because of complicated loop carried dependencies among iterations and conditional branch.

This paper discusses performance evaluation of macro-dataflow computation on a shared memory multi-processor system. The macro-dataflow computation allows us to get efficient parallel processing among coarse grain tasks, because the compiler automatically generates coarse grain tasks(macro-tasks), exploits parallelism among macro-tasks and generates a scheduling routine.

1 はじめに

主記憶共有マルチプロセッサシステム上での Fortran プログラムの並列処理では、従来より、CRAY などに代表されるマルチタスキング、マイクロタスキングが用いられてきた [4, 5]。

マルチタスキングでは、ユーザがソースプログラム中にマルチタスキングのための拡張命令を記述することによって、粗粒度タスク間の並列性を指定し、OS またはランタイムライブラリが実行時にタスクをプロセッサへ割り当てる方式がとられている。しかしこの方式では、ユーザによる粗粒度タスク間の並列性指定は非常に困難であり、一部のエキスパートユーザしか効果的な粗粒度並列性を抽出することができない、OS またはライブラリコールによるダイナミックスケジューリングはオーバーヘッドが大きいといった問題がある。

マイクロタスキングは、従来から最も広く用いられているループ並列化手法であるが、イタレーション間にまたがる複雑なデータ依存やループ外への条件分岐などに起因する並列化できないシーケンシャルループが依然存在する。またマイクロタスキングでは、ループ以外の部分の並列性が有効に利用されていない。

これらに対して著者らは、マクロデータフロー処理手法 [6, 7, 8, 9, 10] を提案してきた。本手法では、コンパイラが、プログラムを粗粒度タスク（マクロタスク）へ分割し、マクロタスク間の制御依存、データ依存関係を解析することにより、マクロタスク間の並列性を実行開始条件 [6] の形で抽出する。次にマクロタスクの実行開始条件をもとにして、実行時にマクロタスクをプロセッサへ割り当てるためのスケジューリングコードを生成する。マクロタスクは、コンパイラの自動生成したスケジューリングコードによってプロセッサへ割り当てられるため、低オーバーヘッドで効率良い並列処理を行うことができる.

本稿では、主記憶共有マルチプロセッサシステム上でのマクロデータフロー処理の性能評価について述べる。まず 2章でマクロデータフロー処理について概説し、3章で主記憶共有マルチプロセッサシステム上でのマクロデータフロー処理の実行方式について述べる。次に 4章で性能評価を行ない、5章で本稿のまとめを行なう。

2 マクロデータフロー処理手法

本章では、Fortran プログラムのマクロデータフロー処理手法について概説する。マクロデータフロー処理は、サブルーチン、ループ、基本ブロック等の粗粒度タスク間の並列処理手法である。

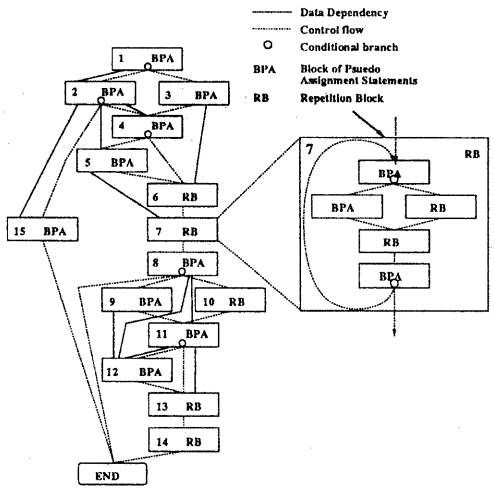

図 1: マクロフローグラフ

2.1 プログラム分割

マクロデータフロー処理では、はじめに Fortran プログラムをマクロタスクと呼ぶ粗い粒度をもつタスクに分割する。マクロタスクは、基本ブロック (BB) あるいは複数の基本ブロックからなるブロック (BPA)、繰り返しブロック (RB)、サブルーチンブロック (SB) から構成される。RB は、DO ループまたはバックワードブランチによって生成されるループであり、より厳密には最外側ナチュラルループを意味する。またサブルーチンに関しては、基本的に可能な限りインライン展開を行なうが、コード長等を考慮して効率良くインライン展開できない場合は、そのサブルーチンを 1 つのマクロタスク (SB) として定義する。

2.2 マクロタスク間並列性抽出

プログラム分割終了後、マクロタスク間の制御フロー、データフローを解析することにより、図 1のようなマクロフローグラフを生成する。図 1において、各ノードはマクロタスクを表し、ノード間の点線エッジ、実線エッジはそれぞれコントロールフロー、データ依存を表す。また、ノード内の円はそのマクロタスク中に条件分岐が存在することを意味している。

次にマクロフローグラフから、マクロタスク間の制御依存、データ依存関係を解析することにより、各マクロ

表 1: 実行開始条件の論理式表現

マクロタスク番号	実行開始条件
1	
2	1_2
3	$(1)_3$
4	$2_4 \vee (1)_3$
5	$(4)_5 \wedge (2_4 \vee (1)_3)$
6	$3 \vee (2)_4$
7	$5 \vee (4)_6$
8	$(2)_4 \vee (1)_3$
9	$(8)_9$
10	$(8)_{10}$
11	$8_9 \vee 8_{10}$
12	$11_{12} \wedge (9 \vee (8)_{10})$
13	$11_{13} \vee 11_{12}$
14	$(8)_9 \vee (8)_{10}$
15	2_{15}

i : MT_i の実行が終了する
$(i)_j$: MT_i が MT_j に分岐する
i_j : MT_i が MT_j に分岐し MT_i の実行が終了する

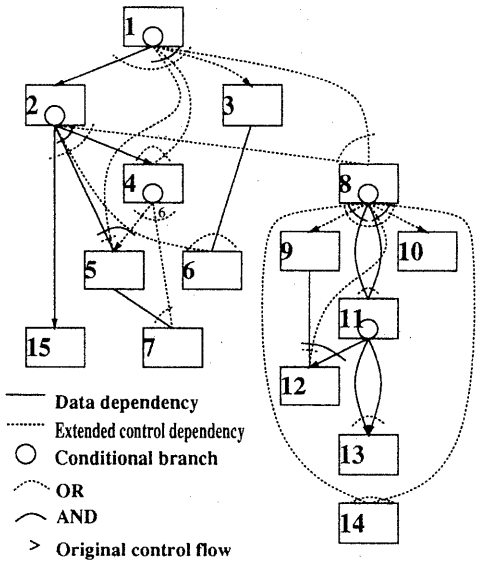

図 2: マクロタスクグラフ

タスクの実行開始条件 [6] を求め、それをグラフ表現したマクロタスクグラフを生成する。表 1に図 1のマクロフローグラフを持つプログラムの実行開始条件を示し、図 2にそのマクロタスクグラフを示す。図 2において、各ノードはマクロタスクを表し、ノード内の円はそのマクロタスク中に条件分岐があることを意味する。ノード間の点線エッジ、実線エッジは、それぞれ拡張制御依存、データ依存を表す。また、ノード内の円を起点とするデータ依存エッジつまり実線エッジは、制御依存とデータ依存の二つを同時に表している。ノードのエッジ入力部、出力部につけられた点線弧、実線弧は、それぞれ弧に含まれるエッジの論理和、論理積をとること意味しており、各マクロタスクの実行開始条件は、ノードの入力エッジであるデータ依存、拡張制御依存を満足する条件の論理式によって表現される。

2.3 マクロタスク分割・融合

マクロタスクグラフ生成後、マクロタスク間の並列性を向上させるため、マクロタスク分割 [8] を行なう。マクロタスク分割では、まず、基本ブロック内部のステートメント間のデータ依存関係を考慮して、一つの基本ブロックを複数の基本ブロックに分割する。また、DOALL ループおよびリダクションループをそのシーケンシャル実行時間と並列処理時間を考慮して、$\lceil n/m \rceil$ (n:イタレーション数、m:プロセッサ数) イタレーションからなる m 個のループに分割する。

次に、データ転送およびスケジューリングオーバーヘッドを最小化するため、マクロタスク融合 [8] を行なう。マクロタスク融合では、マクロタスク間の制御依存およびデータ依存関係、マクロタスクの処理時間を考慮して、並列処理を行なっても処理時間を短縮できないマクロタスク群を融合する。

2.4 スケジューリングコード生成

各マクロタスクは、マクロタスクグラフの情報をもとにして、条件分岐や処理時間の変動に対処するために、Dynamic-CP 法により実行時にプロセッサに割り当てられる [9]。一般的な OS コール等によるダイナミックスケジューリングは実行時のオーバーヘッドが大きいが、本手法では、コンパイラが自動生成する Fortran ソースプログラム専用のスケジューリングコードによって、マクロタスクのダイナミックスケジューリングが行なわれるため、オーバーヘッドを小さく抑えることができる。

2.5 階層型マクロデータフロー処理

プログラム全体が分割不可能な大規模ループで構成されている場合やマクロタスク間に複雑なデータ依存が存在する場合など、マクロタスク分割を行なっても効果的にマクロタスク間の並列性が抽出できない場合がある。この場合、このようなプログラム中の RB、SB 内部に存在するブロック (サブマクロタスク) に対してマクロ

データフロー処理を階層的に適用することで、並列処理による処理時間短縮を行なうことができる [10]。

3 マクロデータフロー処理実行方式

本章では、主記憶共有マルチプロセッサシステム上でのマクロデータフロー処理の実行方式について述べる。

3.1 ターゲットアーキテクチャ

本稿で対象とする主記憶共有マルチプロセッサシステムのアーキテクチャは、4 台程度のプロセッサ (PE) をインターコネクションネットワークを介して共有メモリに接続したものとする。

3.2 スケジューリング方式

スケジューリングコードによるマクロタスクのダイナミックスケジューリング方式としては、スケジューリングコードの実行を単一 PE に集中させる集中スケジューラ方式 [7, 8, 9, 10] と全 PE に分散させる分散スケジューラ方式が考えられる。

分散スケジューラ方式では、複数 PE がスケジューリング管理用データを排他更新するため、メモリアクセス性能の劣化が起こる可能性があるが、集中スケジューラ方式ではこれを押えることが可能である。しかし集中スケジューラ方式では、対象とするマルチプロセッサシステムの PE 台数が少ない場合、スケジューラとして 1PE を占有してしまうために並列処理効果が低下する可能性がある。本稿では、対象とするマルチプロセッサシステムの PE 台数が 4 台程度と少ないので、分散スケジューラ方式を採用する。

3.3 分散スケジューラ方式

分散スケジューラ方式の実行イメージとスケジューリングコードの基本構成を図 3、図 4にそれぞれ示す。

分散スケジューラ方式では、コンパイラが生成するスケジューリングコードがプログラムコード中に埋め込まれており、各プロセッサは、割り当てられたマクロタスクの実行を終了する度にスケジューリングコードの実行を行う。

スケジューリングコードの機能は、マクロタスクの実行開始条件の管理とマクロタスクの実行起動である。図 3に示すように、各プロセッサはスケジューリングコードを埋め込まれた同一のプログラムコードを実行する。また PE1 は、プログラムコードに加えて初期化処理の

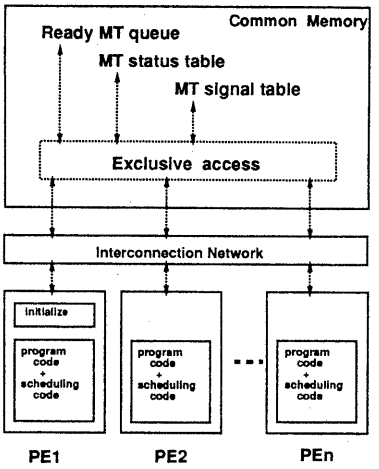

図 3: 分散スケジューラ方式

```
/* variables shared by all PEs */
int     mt_num;   /* number of MT*/
int     mt_stat[mt_num]; /*MT status table*/
int     mt_signal[mt_num]; /*MT signal table*/
int     readyMT_queue;  /*Ready MT queue*/
int     flag_queue, flag_insert;  /*flag for mutal exclusion*/

/* executed by PE1 */
program intialize
{
void init(); /*initialize variables*/

init(mt_stat);
init(mt_signal);
init(readyMT_queue);
}

/* executed by all PEs */
program macro-dataflow
{
int     newMT; /*executable MT*/

void lock(); /*lock routine for mutal exclusion*/
void unlock();   /*unlock routine for mutal exclusion*/
void get_newMT();   /*get executable MT from readyMT_queue*/
void chek_ex_cond;   /*check if executable conditon of MT has satisfied*/

label_1:    if("program has finished") then goto label_end;
            lock(flag_queue);
            get_newMT(newMT);
            unlock(flag_queue);
            if(not("executable MTi exist")) then goto label_1;
            goto label_MTi;

label_MT1:  execution of MT1
            if("MT1 has branched") then mt_signal[MT1.branch] = 1;
                    /*MT1.branch : number which indicate MT1 has branched*/
            execution of MT1
            if("execution of MT1 has finished") then mt_signal[MT1.finish] = 1;
                    /*MT1.finish : number which indicate MT1 has hinised*/
            check_ex_cond(MT1);
            if("executalbe MTj has found") then  {
                mt_stat[MTj] = 1;  /*MTj : number of MTj*/
                lock(flag_inset);
                insert_queue(readyMT_queue, MTj);  }
                unlock(flag_insert);
            goto label_1

label_MT2:
        :
label MTi:
        :
label MTn:
}
```

Program Code

図 4: スケジューリングコード

ためのコードを実行する。マクロタスクのスケジューリング管理用データ（MT実行管理変数）としては、マクロタスクの状態（非実行可能，実行可能）を管理するマクロタスク状態テーブル、マクロタスクグラフのエッジに相当するマクロタスクの終了・分岐情報(信号)を管理するマクロタスク信号管理テーブル、実行可能なマクロタスクが登録されるレディーMTキューを用意する。これらのMT実行管理変数は、各プロセッサにより共有され、その更新は排他的に行われる。

スケジューリングコードがマクロタスクを実行する手順を以下に示す。

1. プログラムの実行開始時：PE1が初期化処理（MT実行管理変数の初期化、プログラム開始時点での実行可能マクロタスクのレディーMTキュー登録）を行う。

2. プロセッサがレディーMTキューから実行可能マクロタスク（MTi）を取り出し、MTiの実行を開始する。

3. MTi実行中：MTiからの他のマクロタスクへの分岐があれば、それをマクロタスク信号管理テーブルへ通知する。

4. MTi実行終了後：MTiの実行終了をマクロタスク信号管理テーブルへ通知し、次にまだ実行可能でないマクロタスクのうち、その実行開始条件中にMTiに関する条件(MTiが終了する、またはMTiが他のマクロタスクへ分岐する)が含まれるマクロタスクの実行開始条件を調べる。ここで実行可能なマクロタスク（MTj）が存在すれば、マクロタスク状態テーブル上のMTjの状態を実行可能に変更し、MTjをレディーMTキューへ登録する。

5. プログラム終了でなければ2へ。

4 性能評価

本章では、主記憶共有マルチプロセッサシステムであるAlliant FX/4およびTITAN 3000V上でのマクロデータフロー処理の性能評価について述べる。

4.1 評価用マルチプロセッサシステム

4.1.1 Alliant FX/4

図5に性能評価に用いるAlliant FX/4のアーキテクチャを示す。本マルチプロセッサシステムは、ベクトル、スカラユニットを持つコンピューテーショナルエレメント(CE)4台を256KBキャッシュ、メモリバスを介して32MBの共有メモリに接続した構成であり、さらに各CEはコンカレンシーコントロールバスにより接続されている。ピーク性能は、40MIPS、47.2MFLOPSである。

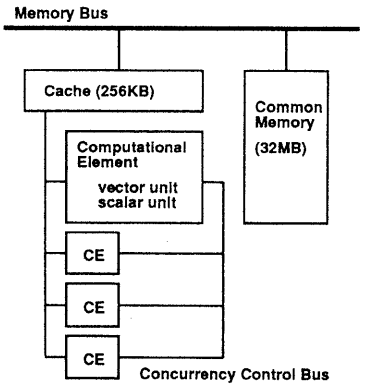

図5: Alliant FX/4

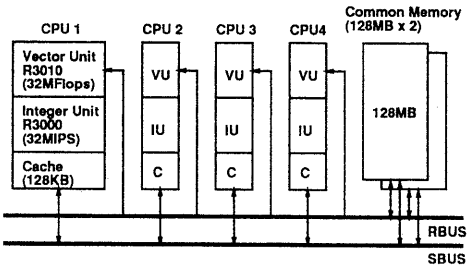

図6: TITAN 3000V

4.1.2 TITAN 3000V

図6に性能評価に用いるTITAN 3000Vのアーキテクチャを示す。本マルチプロセッサシステムは、ベクトルユニット(R3010)、スカラユニット(R3000)および128KBキャッシュを持つPE4台を2本のバスを介して256MBの共有メモリに接続した構成である。ピーク性能は、128MIPS、128MFLOPSである。

4.2 評価方法

Alliant FX/4およびTITAN 3000V上でマクロデータフロー処理の性能評価を行なうために、スケジューリングコードを挿入したプログラム(マクロデータフロー処理コード)を作成し、実行する。

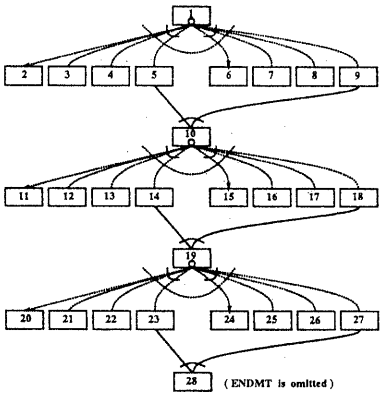

図 7: テストプログラムのマクロタスクグラフ

性能評価に用いるプログラムとして、オーバーヘッド評価のためのテストプログラムおよび対象帯状マトリクス係数行列を持つ連立方程式の解法である CG(Conjugate Gradient) 法プログラムを用いる。

4.3 テストプログラムによる性能評価

4.3.1 テストプログラム

オーバーヘッド評価のためのテストプログラムのマクロタスクグラフを 図7に示す。図7において、マクロタスク {2,3,4,5} (MT {2,3,4,5}) は、ループボディが 6 ステートメントからなる処理時間の等しい DO ループであり、並列実行可能である。また、MT {6,7,8,9}、MT {11,12,13,14} 、MT {15,16,17,18} 、MT {20,21,22,23} 、MT {24,25,26,27} についても同様である。MT {1,10,19,28} は基本ブロックである。

4.3.2 評価結果

Alliant FX/4 上でのテストプログラムの実行時間 (スカラ実行時間) を図 8に示す。図中縦軸の seq. はシーケンシャル実行時間、4PE MT は 4PE 上でマルチタスキングを適用した場合の実行時間、4PE MDF は 4PE 上でマクロデータフロー処理を適用した場合の実行時間を表している。また、4PE MT および 4PE MDF において、斜線部分はテストプログラムのオーバーヘッドを考慮しない理想的な並列処理時間であり、黒く塗りつぶされた部分はオーバーヘッドを表している。図 8より、マクロデータフロー処理のオーバーヘッドがマルチタスキングより小さいことがわかる。

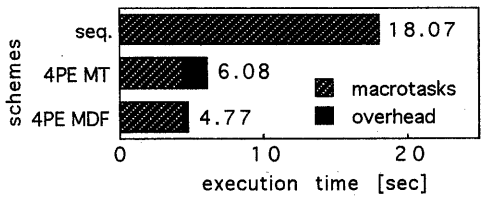

図 8: テストプログラム実行時間

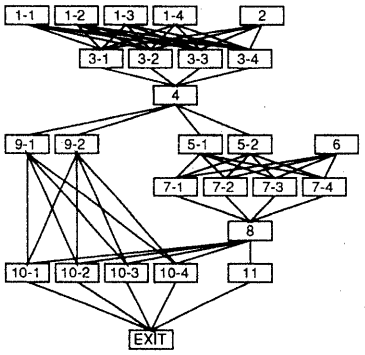

図 9: CG 法マクロタスクグラフ (収束計算ループ内)

4.4 CG 法プログラムによる性能評価

4.4.1 CG 法プログラム

CG 法プログラムは、初期化部分と実際に連立方程式の係数行列を解く収束計算ループから構成されている。このうち、プログラム実行時間のほとんどが後者の収束計算ループによって占められ、また、収束計算ループ内でブロック (サブマクロタスク) 間の並列性が得られるため、収束計算ループ内のサブマクロタスクに対して、マクロデータフロー処理を適用する。さらに、サブマクロタスク間の並列性を向上させるため、サブマクロタスクに対してマクロタスク分割を適用する。CG 法収束計算ループ内のサブマクロタスクのマクロタスクグラフを図 9に示す。

4.4.2 Alliant FX/4 上での評価結果

図 10から図 13に Alliant FX/4 上での CG 法プログラムの実行時間 (スカラ実行時間、ベクトル実行時間) を示す。マクロタスクの処理時間と並列処理効果の関係を示すため、図 10から図 13は、連立方程式の係数行列のサイズを 240x240、128x128、64x64、32x32 とした 4 種類のプログラムの実行結果を示している。

図 10: FX/4 CG 法実行時間 (240x240)

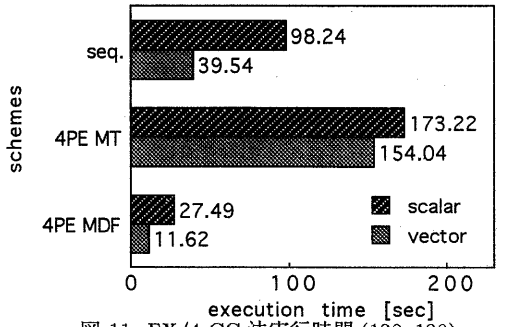

図 11: FX/4 CG 法実行時間 (128x128)

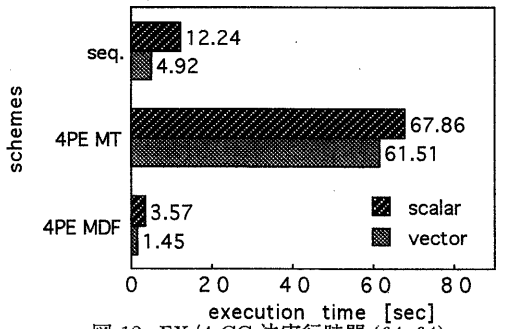

図 12: FX/4 CG 法実行時間 (64x64)

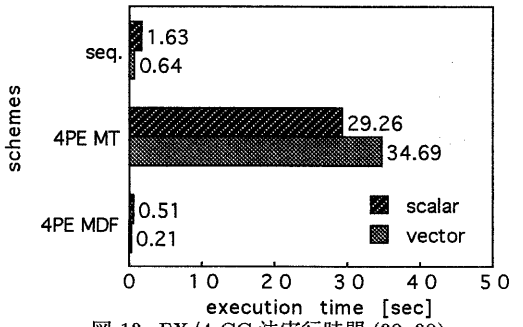

図 13: FX/4 CG 法実行時間 (32x32)

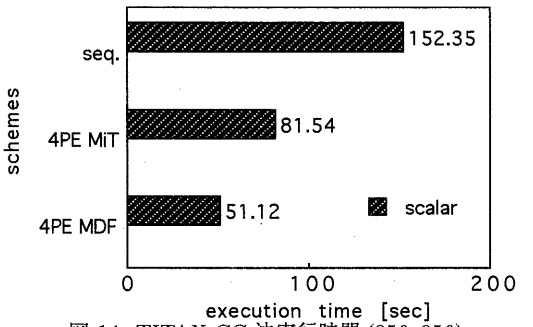

図 14: TITAN CG 法実行時間 (256x256)

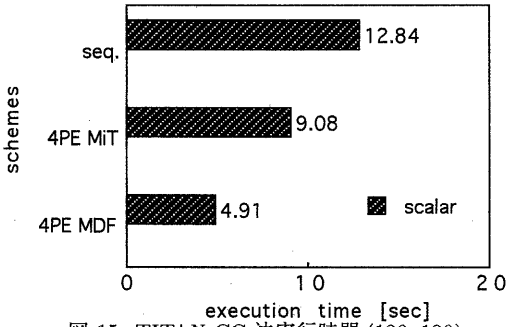

図 15: TITAN CG 法実行時間 (128x128)

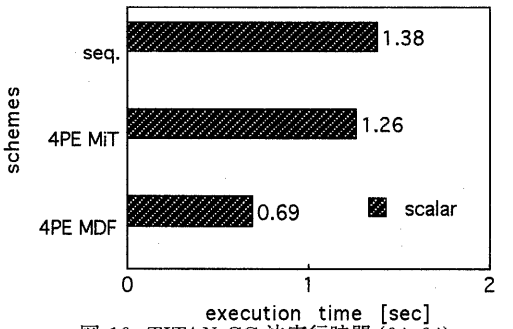

図 16: TITAN CG 法実行時間 (64x64)

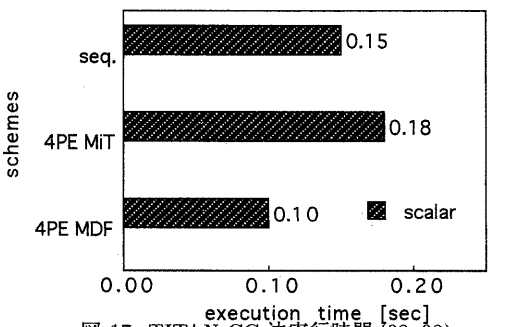

図 17: TITAN CG 法実行時間 (32x32)

図10から図13より、マクロデータフロー処理を適用した場合の実行時間がシーケンシャル実行時間に比べて短縮されていることがわかる。連立方程式の係数行列サイズが240x240の場合、スカラ実行時間は約1/3.7、ベクトル実行時間は約1/3.4に短縮されている。また、係数行列サイズが32x32の場合でも、スカラ実行時間が約1/3.2、ベクトル実行時間が約1/3.0に短縮されている。ここで、係数行列サイズつまりマクロタスクの処理時間が小さい場合は、スケジューリングオーバーヘッドがマクロタスクの処理時間に対して相対的に大きくなるため、マクロタスクの処理時間が大きい場合に比べて並列処理効果がやや下がることがわかる。

一方、マルチタスキングを適用した場合の実行時間は、スケジューリングオーバーヘッドが大きいため、係数行列サイズ240x240の場合のスカラ実行時間がわずか(シーケンシャル実行時間の約1/1.02)に短縮されている他は、シーケンシャル実行時間より長くなってしまっている。

また、ベクトル実行時間の並列処理効果がスカラ実行時間よりわずかに低くなっているは、並列性向上のためにマクロタスク分割を適用されたDOループのベクトル長がシーケンシャル実行の場合より短くなり、ベクトル効果が下がったためと考えられる。

4.4.3 TITAN 3000V上での評価結果

図14から図17にTITAN 3000V上でのCG法プログラムの実行時間(スカラ実行時間)を示す。Alliant FX/4の場合と同様に、図14から図17は、係数行列サイズを256x256、128x128、64x64、32x32とした4種類のプログラムの実行結果を示している。図中縦軸の4PE MiTは4PE上でマイクロタスキングを適用した場合の実行時間を表している。

図14から図17より、マクロデータフロー処理を適用した場合の実行時間がシーケンシャル実行時間より短縮されていることがわかる。スカラ実行時間は、係数行列サイズ256x256の場合で約1/3.0、係数行列サイズ32x32の場合で約1/1.5と短縮されていることがわかる。ここでもマクロタスクの処理時間が小さくなるに従って、マクロタスクの処理時間に対するスケジューリングオーバーヘッドが相対的に大きくなり、並列処理効果が下がることがわかる。

また、TITAN 3000V上でループ並列処理であるマイクロタスキングを適用した場合の実行時間は、係数行列32x32の場合を除いてシーケンシャル実行時間より短縮されているが、係数行列256x256のスカラ実行の場合で約1/1.9であり、並列処理効果はマクロデータフロー処理を適用した場合に比べて低いことがわかる。

5 むすび

本稿では、主記憶共有マルチプロセッサシステム上でのマクロデータフロー処理の性能評価について述べた。マクロデータフロー処理の実行方式として、ターゲットアーキテクチャのPE数が比較的少ない場合に有効な分散スケジューラ方式を採用し、Alliant FX/4およびTITAN 3000V上でマクロデータフロー処理の性能評価を行なった結果、マクロデータフロー処理が、従来手法であるマルチタスキングおよびマイクロタスキングより、プログラムの実行時間を短縮できることが確認された。今後は、マクロデータフローコンパイラを開発するとともに、ベクトル効果を考慮したマクロタスク分割手法をインプリメントする予定である。

参考文献

[1] A.V.Aho, R.Sethi, J.D.Ullman : Compilers Principles, Technique, and Tools, Addison-Wesley, (1988)

[2] D.Gajski, D.Kuck, D.Lawrie, A.Sameh : Cedar - A Large Scale Multiprocessor, Proc. 1983 International Conference on Parallel Processing, pp524-529, (Aug.1983)

[3] D.A.Padua, M.J.Wolfe : Advanced Compiler Optimizations for Supercomputers, Comm. ACM, Vol.29 No.12, pp1184-1201, (Dec.1986)

[4] M.D.Guzzi,D.A.Padua,J.P.Hoeflinger, D.H.Lawrie : Cedar Fortran and Other Vector and Parallel Fortran Dialects, Proc. Supercomputing '88, pp 114-121, (Mar.1988)

[5] A.H.Karp, R.G.Babb II : A Comparison of 12 Parallel Fortran Dialects, IEEE Sftware Vol.5 no.5, pp52-67, (Sep.1988)

[6] 本多, 岩田, 笠原：Fortranプログラム粗粒度タスク間の並列性検出手法, 信学論 Vol.J73-D-I, No.12, pp951-960, (1990-12)

[7] H.Kasahara, H.Honda, K.Aida, M.Okamoto, S.Narita : OSCAR FORTRAN COMPILER, Proc. Workshop on Compilation of Languages for Parallel Computers, (Oct.1991)

[8] 笠原, 合田, 吉田, 岡本, 本多：Fortran マクロデータフロー処理のマクロタスク生成手法, 信学論 VolJ75-D-I No.8, pp511-525, (1992-08)

[9] 本多, 合田, 岡本, 笠原： Fortranプログラム粗粒度タスクのOSCARにおける並列実行方式, 信学論 VolJ75-D-I No.8, pp526-535,(1992-08)

[10] 岡本, 合田, 宮沢, 本多, 笠原：OSCARマルチグレインコンパイラにおける階層型マクロデータフロー処理手法, JSPP '93, pp95-102, (1993-05)