プログラミングー言語・基礎・実践ー 13−18
（1993. 8. 20）

関数型プログラムの疎／密結合並列計算機上の実行スケジューリング手法

高橋 英一　谷口 倫一郎　雨宮 真人
九州大学総合理工学研究科

関数型言語は参照透明性という性質を持つため，コンパイラによる並列性の抽出が容易である．本研究では，関数型言語を関数適用レベルで並列実装する手法を提案する．コンパイラは，ソースプログラムからデータフロー解析に基づくコントロールフローグラフを生成し，互いにデータ依存関係のない関数適用が並列実行となるよう，グラフに基づきコードをスケジュールする．我々は既に関数型言語 Valid を密結合並列計算機 Sequent Symmetry へ実装し実行効率を評価した．今回は，疎結合並列計算機 AP1000 への本手法の応用を試みる．

Code Scheduling Technique for Functional Programming Language Valid on Loosely / Tightly Coupled Multiprocessor Systems

Eiichi Takahashi　Rin-ichiro Taniguchi　Makoto Amamiya
Graduate School of Engineering Sciences, Kyushu University

6-1, Kasugakouen, Kasuga-shi, Fukuoka-ken, 816, Japan

In this paper, we present a compiling method to translate a functional programming language Valid into an object code executable on AP1000. Since the cost of process management is very high in such a machine, we exploit coarse-grain parallelism at function application level, and the function application level parallelism is implemented by fork-join mechanism.The compiler translates Valid source programs into controlflow graphs based on dataflow analysis and then serializes instructions within graphs according to flow arcs such that function applications which have no data dependency with each other are executed in parallel. We report results of performance evaluation of the compiled Valid programs on AP1000 and discuss usefulness of our method.

1 はじめに

近年，多様化しているプログラミング言語の中で数学的な関数の概念に基づく関数型言語は，参照透明性の性質により，セマンティックスが単純明解になり，プログラムの機械的な変換や検証，簡潔明瞭なプログラムの記述を行なう上で種々の魅力的な性質を持っている [1]．近年，従来用いられてきたインタプリタ実行による実行効率の悪さを改善するためのコンパイル方式が種々提案されてきている [2, 3, 4, 5, 6]．

本稿では，関数型言語を関数適用レベルで並列実行するための，データフロー解析に基づくコンパイル手法を提案する．用いた言語は Valid で，データフロー計算機をターゲットとした高級言語である [7]．一般に，既存の計算機では，プロセッサはユーザプログラム実行とプロセス管理の両方を行なわなければならない．従って，細粒度並列処理はプロセス生成管理のコストがオーバヘッドとなり実用的でない．また，疎結合並列計算機ではプロセッサ間通信のコストが高いため，細かくプログラムを分割した場合，通信のオーバヘッドのため効率が落ちる．そのため，ここでは粒度が粗い関数適用レベルでの並列実行を考える．コンパイラは，まず，Valid プログラムからデータ依存関係に基づいたコントロールフローグラフを作成する．次に，グラフから互いにデータ依存関係のない関数適用を抽出し，それらが並列実行となるよう各命令の実行順序をスケジュールする．最後に各命令をターゲットマシンのコードへ変換する．関数適用の並列実行は，fork-join タイプのプロセス生成の概念に沿った方式で行なう．既に我々は密結合並列計算機 Sequent Symmetry 上で本手法で生成したコードの実行効率を評価した [9]．今回は疎結合並列計算機 AP1000 へ本手法を応用する．

まず，2 章で並列実行メカニズムについて述べ，3 章でコンパイラについて述べる．次に，4 章で生成コードの実行効率についての評価結果を報告し，最後に 5 章で結論を述べる．

2 並列実行メカニズム

一般に，既存の計算機では，プロセッサはユーザプログラム実行とプロセス管理の両方を行なわなければならない．従って，細粒度並列処理はプロセス生成管理のコストがオーバヘッドとなり実用的でない．そのため，ここでは粒度が粗い関数適用レベルでの並列実行を考える．

Valid では原則として自由変数を認めていないので，関数評価に必要な情報は，関数コードエントリ，引数，返値の返し先情報のみである．我々は，上記の性質を利用した低コストの fork-join 処理をサポートする並列実行メカニズムを密結合並列計算機 Sequent Symmetry 上に実現し，その上で，Valid のコンパイルコードを実行した [9]．各プロセッサは，キューから実行可能な関数の Frame を取り出し，実行することを繰り返す．Frame は，関数の局所変数や実行開始アドレス，呼出側関数 Frame へのポインタなど制御のための情報を保持するデータ構造で，fork 時に生成され，キューへエントリされる．Frame もキューも共有メモリ上のオブジェクトである．関数実行終了時には，呼び出し側関数の Frame 内へ直接返値を書き込む．また，同期のためのバリア変数も呼び出し側関数の Frame 内にあり，直接-1 する．その結果，バリア変数が 0 になれば，呼出側関数の Frame をカレント Frame として再起動する．

今回，疎結合並列計算機 AP1000 上で，同様の並列実行メカニズムの実現を試みた．AP1000 の場合，共有メモリを持たないことから，Symmetry でのメカニズムをそのまま適用することはできない．Symmetry では，共有メモリを介して，値や Frame の受け渡しを行うことができたが，AP1000 では，Frame やキューは，セル毎の局所メモリ内で分散して管理しなければならなず，関数の呼び出し，返値のリターン，同期はメッセージ通信で行わねばならない．通信のオーバヘッドを押えるためには，できるだけ通信量を減らす必要がある．そこで，我々は，スレッド並列実行モデルとして提案されている TAM [10] を元に，Symmetry での並列実行メカニズムを修正した．

図 1 に Frame の構造を示す．Frame は関数毎に呼び出し時に動的に生成される．Frame の構造は Symmetry と同様で，ヘッダ部，継続スレッドスタック，ワークエリアよりなる．ヘッダ部は，実行開始スレッド，同期スレッド，呼び出し側関数 Frame へのポインタなどを保持する．実行開始スレッドは，実行開始コードアドレスで，同期スレッドは，現関数の実行終了後，実行可能となる呼出側関数のスレッドコードアドレスである．呼出側関数の Frame へのポインタは，セル ID とセル内の局所メモリアドレスよりなる．継続スレッドスタックは，実行可能なスレッ

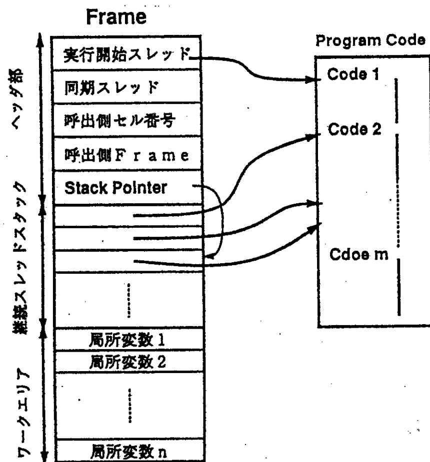


図 1: Frame の構造

ドのコードアドレスを扱い，後述するように1関数内部でのスレッド制御に用いる．ワークエリアは局所変数を保持する．

Frame の実行待ちキューは，各セルがメッセージを保持するために持っているバッファをそのまま流用する．

以下，fork，返値のリターン，join，各セルの動作サイクルについて述べる．

fork (関数適用)

Symmetry では，呼出側が Frame の生成，引数のセットを行っていたが，AP1000 では，通信コストを最小にするために，呼出側は Frame 構築に必要な以下の情報のみからなるメッセージを送信し，Frame 構築は受信側が行う．

- 実行開始スレッド
- 自セル ID
- カレント Frame アドレス
- 同期スレッド
- 引数 $1 \ldots n$

上記のメッセージを受け取った (メッセージバッファから取り出した) セルでは，実行開始スレッドを実行する．実行開始スレッドには，以下に示すコードが，コンパイラによって挿入されている．

1. Frame の領域を確保
2. メッセージから上記の情報を Frame へ転送
3. Frame をカレント Frame とする

返値のリターン

Symmetry では，呼出側関数の Frame へ直接値を書き込んでいたが，AP1000 ではメッセージを使って間接的に行う．関数実行を終了したセルは，呼出側関数の Frame を持つセルへ以下の情報を含むメッセージを送信する．

- 同期スレッド
- Frame アドレス
- 返値

呼出側関数の Frame を持つセルは，上記のメッセージを受け取る (メッセージバッファから取り出す) と，同期スレッドを実行する．同期スレッドは，以下に示すコードでコンパイラが生成する．

1. Frame アドレスが示す Frame のをカレント Frame とする．
2. メッセージから返値を Frame へ転送
3. join 処理

join 処理

join は Symmetry 同様，バリア同期で実現している．バリア変数は同期の対象とするスレッドの数に初期化されており，join 時に-1 される．join 時，もし，バリア変数が 0 になれば，次スレッドを継続スレッドスタックへ push する．例えば，図 2において，スレッド J はスレッド T1,T2, 終了後，スレッド T3 を起動する join を行うスレッドで，対応するバリア変数は 2 に初期化しておく．J は T1,T2 終了の度起動され，バリア変数を-1 する．T1,T2 両スレッド終了時，バリア変数は 0 となるので，T3 を継続スレッドスタックへ push する．

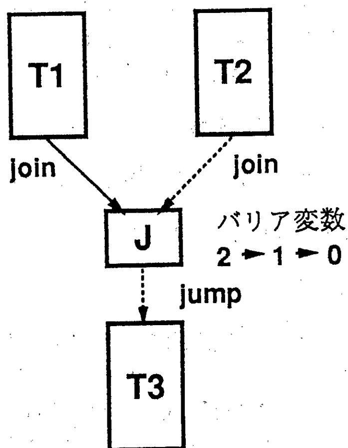


図 2: join 処理

各セルの動作サイクル

AP1000 での各セルは以下の動作を繰り返す.
Main ループ

step 1 メッセージの受信.

step 2 プログラム終了メッセージであれば終了.

step 3 メッセージの先頭より，実行コードアドレスを取り出し，実行.

step 4 もし，プログラムが終了すれば，プログラム終了メッセージを全セルへ送信.

step 5 現 Frame の継続スレッドスタックが空になるまで以下を繰り返す.

step 5.1 継続スレッドスタックからスレッドコードアドレスを pop する.

step 5.2 スレッドを実行.

step 5.3 もし，プログラムが終了すれば，プログラム終了メッセージを全セルへ送信

3 コンパイラ

コンパイルは2つのフェーズに分かれる．フェーズIでは Valid ソースプログラムからデータ依存関係に基づいたコントロールフローグラフを生成する．グラフは DAG で，ノードで命令を，アークでデータ依存に基づいたコントロールフローを表す [9]．フェーズ II ではグラフを逐次コード化しターゲットマシンコードに変換する．最後のターゲットマシンコードへの変換を除き，Symmetry と AP1000 は共通である．ここでは，フェーズ II でのコードスケジューリングについて述べる．

3.1 フェーズ II – コードスケジューリング

コードスケジューリングは以下の手順で行う.

1. Frame の割り当て
2. グラフの分割
3. 逐次コード化

以下，各手順について述べる.

Frame の割り当て

1 関数 (グラフ)1Frame として割り当てる.

グラフの分割

グラフを 1 スレッドとなる部分グラフへ分割する．分割は以下に示すアークで行う (図 3).

- 関数適用の結果を参照する命令ノードとその親ノードとの間のアーク
  完全に分離できない場合は，次の逐次コード化で分離する.
- 条件分岐グラフの merge ノードへのアーク
- ループグラフの $loop_i$ ノードへのアーク

逐次コード化

各部分グラフ毎に命令ノードをアークに従いソートし，コードブロックを生成する (図 [?])．生成手続きを以下に示す.

1. 部分グラフのノード集合を N とする．N = ∅ になるまで，以下を繰り返す.
   (a) N の全ノードに対し，トポロジカル·ソーティングを行う.
   (b) ソートされたノード集合 N'(∈ N) をコードブロックとする.

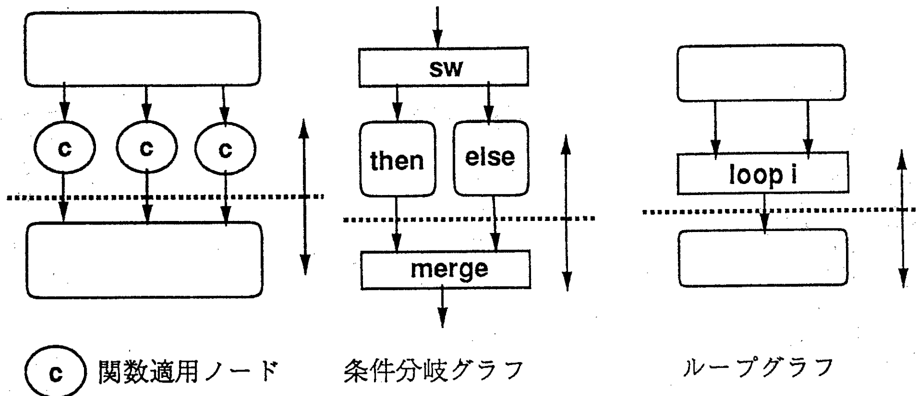


図 3: グラフの分割

(c) N ← N - N'

複数の親ブロックをもち，先頭が merge ノード以外のノードの命令であるコードブロックの先頭に join 処理コードを挿入する．AP1000 の場合，メッセージを扱うためのメッセージハンドラのコードを以下に示すコードブロックの先頭に挿入する．

- 親ブロックがない (関数内で最初に実行されるコードブロック)
- 親ブロックの最後の命令が fork である．

各コードブロックに名前をつけ，コードブロック間の依存関係を

JUMP コードブロック名

で表す．

4 生成コードの評価

AP1000 上で 64 台のセルを用いて生成コードを評価した．ターゲットマシンコードを AP1000 用 C 言語とし，1 コードブロックを 1 関数とした．実行は，0 番セルが最初に実行する関数を起動する CALL メッセージを送信することで始まり，プログラム終了メッセージが全セルへ放送されるまで続く．ホストは起動と終了処理のみ行う．

表 ??にセルを 1 台，64 台で実行した時の各例題の評価結果を示す．計測時間内におけるタスク，ライブラリの実行時間の平均値，計測時間に対する割合，スピードアップを表す．計測時間は，セルが Main ループに入ってから出るまでである．また，タスク実行時間は，例題プログラムと Main ループの処理時間からなり，ライブラリ実行時間は，AP1000 用 C 言語が提供するメッセージ通信ライブラリの実行時間を意味する．

pi(n) は $4/(1+x^2)$ を 0 ∼ 1 の範囲で積分することでπを求めるプログラムである．n は 0 ∼ 1 の範囲の分割数を意味し，大きい程よい近似が得られる．評価に用いた Valid プログラムを以下に示す．

```
program pi;
 function pi (n:int) return (real)
  = {let w = 1.0 / n
     in w * + foreach i in [0..n-1] body
                {let x = w * (i + 0.5)
                 in 4.0 / (1.0+x*x)
                }
     };
 = pi(1000000);
```

ここで，+は引数にタプルをとり，全要素の総和を求めるリダクション演算子である．上記のプログラムは，意味的には並列式の本体

```
{ let x = w * (i + 0.5)
  in 4.0 / (1.0+x*x)
```

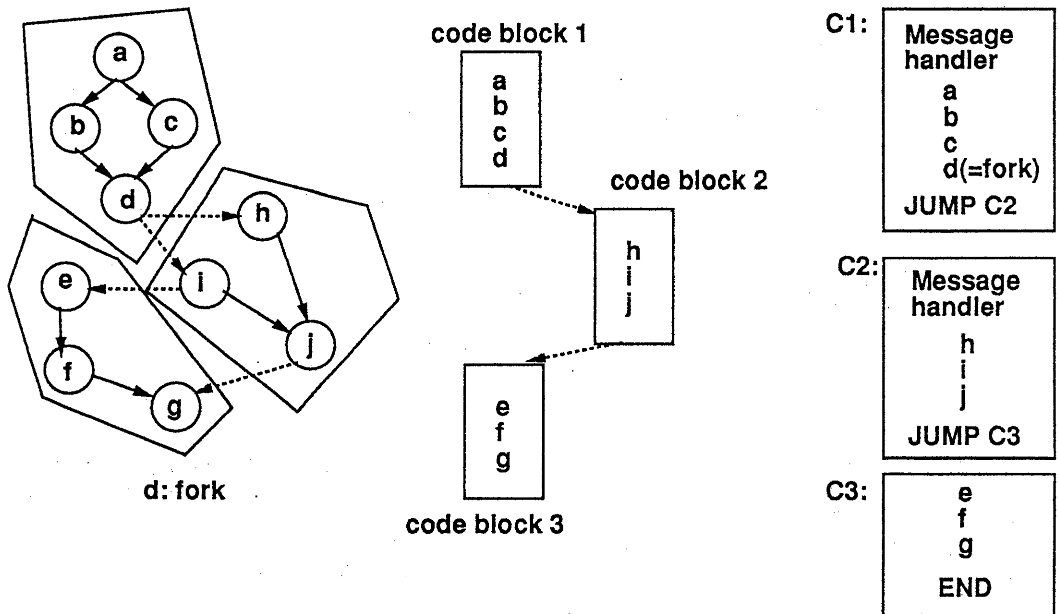


図 4: 逐次コード化

表 1: 実行結果

プログラム	台数	実行時間 ($\times 10^{-6}$ sec.)				スピードアップ
		タスク		ライブラリ		
pi(10^6)	1	6447500	(100%)	712	(0.00%)	–
pi(10^6)	64	100982	(80.4%)	24443	(0.14%)	51.3
fibo(20)	1	2078308	(75.7%)	666140	(24.3%)	–
fibo(20)	64	33857	(30.0%)	57342	(62.8%)	30.0
queen(8)	1	1139265	(54.6%)	946479	(45.4%)	–
queen(8)	64	59411	(34.9%)	110853	(65.1%)	12.2

```
}
```

がn個並列に評価され，各結果値の総和が求められる．実際は，n個を全セル(N_c)へ均等に配分し，各セルはn / N_c個の並列式本体を逐次的に実行する．リダクション演算子により，各セルは自分の担当部分の総和を求めて，呼出側へ結果を送る．呼出側は返値の総和を求めてリダクション演算子の結果値とする．現在，並列式のコンパイルコードでは，並列度が使用セル数以上であれば全セルへ前述したように均等配分し，セル数より小さければ，ランダムに割り当てている．結果はセル64台で，約1.3秒，セル1台の時の約51倍のスピードアップを達成できた．分割数が10^6と大きく，プログラムの振舞も，並列式本体のコストが一様であるなど単純であったため，高い実行効率を得ることができた．今後は，並列式に対し，さまざまなマッピング手法を導入して，より一般的な問題に対処していく予定である．

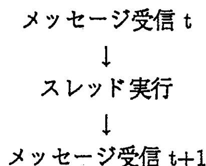
fibo(n)はnaiiveなfibonacciプログラムで，粒度制御のためのannotationを用いている．評価プログラムを以下に示す．

```
program fibo;
 function fibo (n:int) reutrn (int)
  = if n < 2 then 1
    else fibo(n-1) $[n>4]
         + fibo(n-1) $[n>4];
= fibo(20);
```

上記プログラム中，$[n>4]がannotationである．annotation

関数適用 $[論理式]

は，論理式が真であれば関数適用は他のセルで行い，偽であれば自セルで行うことを意味する．上プログラムでは，fibo(4)まで並列に展開されて実行される．並列展開された関数は，現在，ランダムにセルへ割り当てている．結果はセル64台で，約0.091秒，セル1台の時の約30倍のスピードアップを達成できた．セル64台で通信のための時間が全体の実行時間の62.8%を占める．タスク実行時間に限れば，約61倍のスピードアップであるのに対し，通信ライブラリの実行時間は約12倍である．さらに通信ライブラリの時間の内訳を調べると，送信ライブラリは約49倍のスピードアップであるのに対し，受信ライブラリは約3.3倍のスピードアップであった．明らかに受信ライブラリがオーバヘッドになっていることがわかる．理由は，fibo(n)では，関数のコスト自体が小さいため，



このサイクルが速くなる．受信，実行サイクルのスピードに比べ，メッセージの到来頻度が低いと待ち時間が生じ，これがオーバヘッドになったためだと考えられる．この問題を解決するためには，粒度をさらに粗く指定する必要がある．しかし，前述したように，ランダムに関数をセルへ割り当てているため，粒度が粗くなると負荷の不均衡が生じ，効率が低下する．Yale大学のParAlflのMapped expressionの機能をannotationに採り入れることで解決できると考える．

queen(n)はnQueenパズルの全解探索プログラムである．全ての解を表示すると，それ自体がほとんどの時間を占めてしまうため，評価プログラムでは，単に解の数をカウントするだけにした．プログラムを以下に示す．

```
program queen;
 function queen0(n,c:int, col,lef,rig:list)
 return(int)
 = if c > n then 1
   else + foreach r in [1..n] body
    if member(r,col) or
       member(c-r, lef) or
       member(c+r, rig)
    then 0
    else
     queen0(n,
            c+1,
            cons(r,col),
            cons(c-r,lef),
            cons(c+r,rig));
function queen(n) return (int)
```

```
 = queen0(n,1,nil,nil,nil);
= queen(8);
```

結果はセル64台で，約0.17秒，セル1台の時の約12倍のスピードアップを達成できた．nqueen(n)は台数効果が他の2プログラムと比較してかなり低い．理由は，通信ライブラリの実行時間全体に占める割合が大きいことから，現在，構造データの分散管理は行っておらず，引数としてメッセージに含ませているため，通信量が多くなり，オーバヘッドになったためと考えられる．構造データの分散管理メカニズムを実現する必要がある．

5 まとめ

本稿では，関数型言語を商用並列マシン上で並列実行するためのコードスケジューリング方法，および，並列実行メカニズムについて述べた．この手法は関数型言語をデータフロー解析することで求めたグラフを用いて，関数適用レベルの並列処理を実現する．実際に生成したコードをAP1000上で実行し，効率を実行速度，スピードアップの点で評価した．結果として，並列式を用いるごく単純な問題についてはかなりの効率を達成できたが，関数本体のコストが小さい場合と構造データの扱いがネックであることがわかった．現時点ではまだ，naiive なインプリメントで，改良すべき点が多く残っている．今後は，並列実行メカニズムの改良とともに，分散データ管理，関数とセルとの様々なマッピングのメカニズムを実現していくと同時に，より大きく複雑なプログラムでの評価を行っていく予定である．

参考文献

[1] 雨宮真人，超多重並行処理のためのプロセッサ・アーキテクチャ，情報処理学会「コンピュータアーキテクチャ」シンポジウム，99 (1988).

[2] Simon L.Peyton Jones, The Implementation of Functional Programming Language, PRENTICE-HALL INTERNATIONAL (1987).

[3] Thomas Johnsson, Compiling Lazy Functionlll Language, Chalmers University of Technology DEPARTMENT OF COMPUTER SCIENCES (1987).

[4] Lennart Augustsson, Thomas Johnsson, Parallel Graph Reduction with the$< \nu, G >$-machine. ACM Proc.4th International Conference on Functional Programming Languages and Computer Architecture, 202 (1989).

[5] Paul Hudak, Distributed Execution of Functional Programs Using Serial Combinators, IEEE TRANSACTIONS ON COMPUTES, Vol.C-34, No.10, 881 (1985).

[6] Lubomir Bic, A Process-Oriented Model for Efficient Execution of Dataflow Programs. JURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 8, 42 (1990).

[7] 長谷川隆三，雨宮真人，データフローマシン用関数型言語 Valid. 電子情報通信学会論文誌 D Vol.J71-D No.8 1532 (1988).

[8] Paul Hudak, Para-Functional Programming. IEEE Computer 19,8,60(1986).

[9] 高橋英一，谷口倫一郎，雨宮真人，データフロー解析に基づく関数型言語 Valid の並列化コンパイラ，並列処理シンポジウム JSPP'93, pp.127-134.

[10] David E.Culler, Anurag Sah, Klaus Erik Schauser,Thorsten von Eicken, John Wawrzynek.Fine-grain Parallelism with Munimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 1991