Vol. 35 No. 4 情報処理学会論文誌 Apr. 1994

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

高橋英一[†] 谷口倫一郎[†] 雨宮真人[†]

本論文では，関数型プログラムを共有メモリ型マシン上で並列実行するためのコンパイル手法，および，実装方式を提案する．用いた言語は *Valid*，ターゲットマシンは Sequent Symmetry S2000 で，関数適用レベルの並列処理を実現する．コンパイラは，まず，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する．グラフは DAG で，ノードで命令を，アークでコントロールフローを表す．次に，グラフを関数適用ノードの子孫と，それ以外の部分に分割し，各部分グラフ毎に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 タイプの方式で行う．共有メモリマシンでは，排他制御が頻繁に生じると処理が逐次化され期待した台数効果を得ることが困難になる．我々は，排他制御の頻度，時間を減らすために，実行タスクキューの複数化，階層化，不可分なマシンコードの使用などの工夫を行った．評価では，Symmetry 上で 16 個のプロセッサを用い，実行速度について，C プログラム，SISAL プログラムとの比較を行った．また，実行効率に対する粒度の影響，*Valid* の拡張による並列制御のオーバヘッドの解決も試み評価した．結果を報告し，本手法の有効性について検討する．

Compiling Technique Based on Dataflow Analysis for Functional Programming Language *Valid*

Eiichi Takahashi,[†] Rin-ichiro Taniguchi[†] and Makoto Amamiya[†]

This paper presents a compiling method to translate a functional programming language *Valid* into an object code, which is executable on a commercially available shared memory multiprocessor, Sequent Symmetry S2000. Since process management overhead in such a machine is very high, our compiling strategy is to 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, are executed in parallel. We report a result of performance evaluation of the compiled *Valid* programs on Sequent S2000 and discuss usefulness of our method by comparing it with C, SISAL compiler.

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，セマンティックスが単純明解であり，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う上で種々の魅力的な性質を持っている[1]．特に，参照透明性の性質より，コンパイラによるプログラムに内在する並列性の抽出が参照透明性がない手続き型言語に比べ容易である．関数型言語は，プログラマに，命令の実行順序や排他制御などのアルゴリズムに現れない並列制御を意識させず，ループなどの特定のプログラムパタン以外のより広い範囲の並列処理を実現する環境を提供する．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実装方式が種々提案されてきている[2)~6)]．

本論文では，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，および，実装方式を提案する．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]．ターゲットマシンは Sequent Symmetry S2000 で，Intel 社製 80486CPU の密結合構成である[8]．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わなければならない．従って，細粒度並列実行はプロセス

† 九州大学大学院総合理工学研究科
Department of Information Systems, Graduate School of Engineering Sciences, Kyushu University

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．粒度を関数適用レベルにすることで，命令の論理的局所性に基づくスタックやレジスタを用いた効率的なコードスケジューリングが可能となる．また，メモリアクセスの局所性が高まるため，キャッシュを活かすことができ，共有メモリマシンでネックとなるバスの飽和を軽減することができる．

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．グラフはDAGで，ノードで命令を，アークでコントロールフローを表す．次に，グラフを関数適用ノードの子孫と，それ以外の部分に分割し，各部分グラフ毎に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タイプの方式で行う．forkでは，関数実行に必要な局所変数の値や実行コードアドレスなどを保持する環境を構築し，キューへセットする．各プロセッサは，キューから環境を取り出し，実行する．joinは関数の実行結果を呼出側へ渡すときに行い，同期カウンタ方式で実現している．同期カウンタは，初期値は1で，fork時に1増やし，join時に1減らす．join時に同期カウンタを0にしたプロセッサが，join以降の命令を実行する．共有メモリマシンでは，排他制御が頻繁に生じると処理が逐次化され期待した台数効果を得ることが困難になる．本方式で必須となる排他制御は，キューへのアクセス，fork，join時の同期カウンタへのアクセスであるが，我々は，排他制御の頻度，時間を減らすために，キューの複数化，階層化，不可分なマシンコードの使用などの工夫を行った．

まず，2章でコンパイラの構成について述べ，3章で並列実行メカニズムについて述べる．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示し，5章で関数型言語の並列実行を提案している他の研究と本研究とを比較し，本手法の有効性を検討する．最後に6章で結論を述べる．

2. 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．フェーズIIでは，各命令の実行順序をグラフを用いて決定し，各命令をターゲットマシンコードに変換する．以下，各フェーズについて説明する．

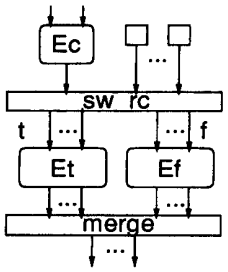

図1 条件式のグラフ
Fig. 1 Conditional expression graph.

2.1 グラフ生成（フェーズI）

フェーズIでは，まず，*Valid*ソースプログラムに対し，字句解析，構文解析を行い構文木を生成する．次に，構文木に対し，データフロー解析を行い，関数定義毎に完結し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，アークでコントロールフローを表す．ノードは，算術，論理演算，分岐命令など，ソースプログラムに対応した命令を表し，オペランド，デスティネーションを必要な数だけ明示する．アークが表すコントロールフローは，命令間のデータ依存関係に従う．以下，*Valid*の主なプログラム構造である条件式，再帰式，並列式に対するグラフ生成について述べる．

(1) 条件式

図1は条件式

```
if Ec then Et else Ef
```

のグラフである．式Ecが真であれば式Et，偽であれば式Efを評価し，その値が条件式の値となる．条件式のグラフは，式Ec，Et，Efより生成したグラフと，swノード，mergeノード一対より構成する．swノードは，オペランドにEcのデスティネーションをとり，アークtでEtの中で最初に実行可能となる命令を，アークfでEfの中で最初に実行可能となる命令を指す．Et，Efと他の式とのデータ依存関係は，swノードへのアーク，mergeノードからのアークで表す．

(2) 再帰式

一般の再帰式は，再帰式本体を定義とする関数，および，その関数の適用で実現するが，末尾再帰構造の再帰式はループ実行で実現する．**図2**に，以下の再帰式から生成するループ実行グラフを示す．

```
for (v1,..,vn) init (E1,..,En) body
  if Ec then Et else recur(R1,..,Rn)
```

ループ実行グラフは，再帰式本体Expへの入口を意

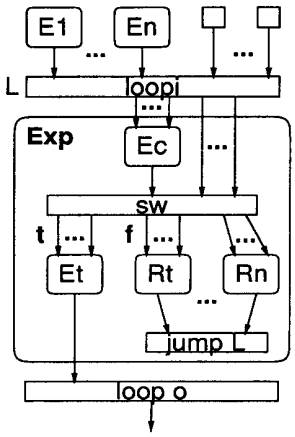

図 2 再帰式のグラフ
Fig. 2 Recursive expression graph.

味する loop_iノード，出口を意味する loop_oノード一対を用いて構成する．loop_i，loop_o両ノードは，後述するコードスケジューリングで使用するためのノードで，ターゲットコードへは変換しない．本体 Exp のグラフでは，再帰呼び出しに相当する recur 式は jump ノードになる．jump ノードは制御が loop_iノードへ移ることを意味する．本体と他の式とのデータ依存関係は，loop_iノードへのアーク，loop_oノードからのアークで表す．

(3) 並列式

*Valid*では，並列式で配列や範囲の各要素に対して同一の演算を施すような互いに依存関係がない処理を簡潔に記述できる[7]．例えば，以下の並列式からは，1 ～ 5 の自乗を要素とする配列を得る．

```
make_array(foreach i in [1..5] body i*i)
```

foreach 以下が並列式である．並列式変数 (i) を in で示す範囲型の式 ([1 .. 5]) で生成される値（1 ～ 5 の各整数）にバインドし，それぞれに対応する body で示す並列式本体 (i*i) を並列に評価することを意味する．並列式の結果値は，上記の例に示したように配列やリスト構造のデータに作り上げられるか，リダクション演算子によりスカラ値になる．コンパイラは，並列式から N_Pだけの並列性を抽出する．すなわち，

$$\frac{\text{範囲式で生成される値の数}}{N_P}.$$

分の並列式本体を逐次的に評価する関数を定義し，範囲型の式で示す範囲に対し，この関数を N_P数，重複がないよう並列に適用するグラフを生成する．現在，N_Pの値は，物理プロセッサ数としている．

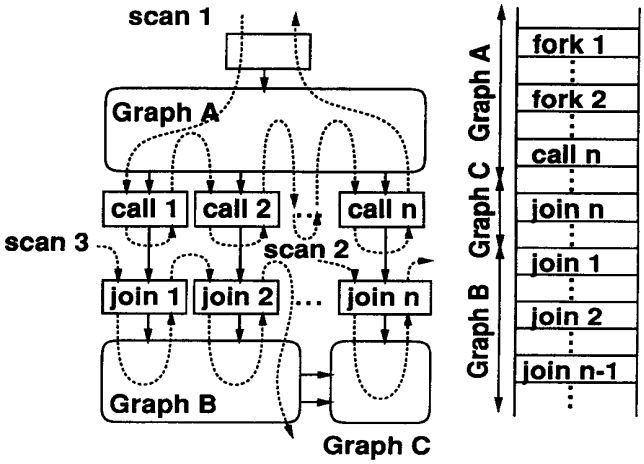

図 3 コードスケジューリング
Fig. 3 Code scheduling.

2.2 コードスケジューリング（フェーズ II）

フェーズ II では，互いにデータ依存関係を持たない関数適用が並列に実行されるようコードの実行順序をスケジュールする．グラフを関数適用とデータ依存関係を持つ部分と持たない部分へ，すなわち関数適用ノードの子孫と，それ以外の部分に分割し，各部分グラフ毎に命令の実行順序を決定する．実際は，アークがデータ依存関係に従っていることから，以下に示す手続きにより，分割とスケジュールを同時に行うことができる．**図 3** にコードスケジューリングの概念図を示す．コンパイラは，まず，関数適用ノード（call_i）と，その子ノードの間に join ノード（join_i）を挿入する．次に，グラフを縦型探索に従って走査し，各命令の実行順序を走査した順序とする．縦型探索を用いることで，命令の実行結果を直ちに次の命令が参照するケースが増えるため，プロセッサのレジスタを有効に利用できる．グラフ走査では，親ノードが全て走査済みである子ノードを任意に選択する．ただし，join ノードへは走査しない（scan 1）．1 回のグラフ走査によって生成されるコード列内の関数適用は，互いに依存関係を持たないので，全て並列実行（fork 実行）にできる．ただし，コード列内での実行順序が最後の関数適用は，現プロセッサが実行しても並列性は失われないので，逐次実行とする．最後の関数適用だけに依存する命令のスケジューリングの後（scan 2），再び，全てのノードをスケジュールするまで，同様の処理を続ける（scan 3）．全ての命令の実行順序関係を静的に決定するためには，条件式，再帰式から生成する

コード列は，それぞれに対応したグラフの本体ノード（sw，$loop_i$ノードの子孫で，かつ，対応する merge，$loop_o$ノードの先祖）だけで構成する必要がある．2.1節で述べたグラフの構造より，sw，$loop_i$ノードが走査可能となれば，本体ノードだけで，本体ノード全てのノードの実行順序を決定できるため，sw，$loop_i$ノードからは，対応する merge，$loop_o$ノードまでのグラフに，上記手続きを再帰的に適用する．以下，グラフ走査の手続きを示す．

手続き A – グラフの逐次コード化

step 1 グラフの全ノードを未走査に，走査可能ノード集合を

S = {n|n は親ノードを持たないノード }

に初期化する．

step 2 S= ∅ になるまで以下の手続きを繰り返す．

step 2.1 走査継続ノード集合を J = ∅ に初期化して，手続き B を適用する．

step 2.2 step 2.1 でスケジュールされた関数適用が存在する場合，それらを fork 命令へ変更する．ただし，最後の関数適用は，現プロセッサが実行しても並列性は失われないので，逐次実行を意味する call 命令にする．次に走査可能ノード集合を

S ← {n|n は最後の関数適用ノードの子ノード }

とする．関数適用がない場合は S←J とする．

手続き B – グラフの走査

走査可能ノード集合 S が S= ∅ になるまで，S の各要素 n に対し，以下の手続きを繰り返す．

step 1 S ← S - n とする．

step 2 n のノードタイプに応じた以下の手続きを実行する．

[sw ノード] n を走査済にした後，then/else パートグラフそれぞれに対し手続き A を再帰的に適用する．その後，merge ノードを S に加える．

[$loop_i$ノード] n を走査済にした後，再帰式本体のグラフに対し手続き A を再帰的に適用する．その後，$loop_o$ノードを S に加える．

[join ノード] n を走査継続ノード集合 J に加える．

[上記以外のノード] ノード n を走査済にする．n の子ノードで，親ノードが全て走査済であるノードがあれば，それを S に加える．

例として**図 4** に，以下の *Valid*プログラムに対するコード生成を示す．

```
function fibo (n:int) return (int)
 = if n < 2 then 1
   else fibo(n-1) + fibo(n-2);
```

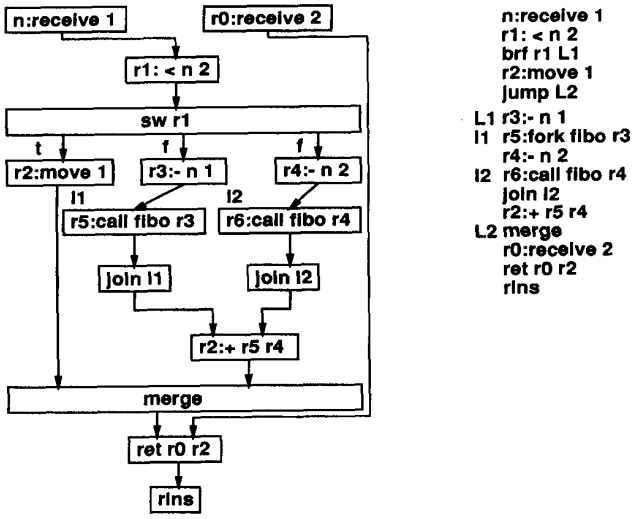

図 4 コードスケジューリングの例
Fig. 4 Example of code scheduling.

フェーズ II の最後では，順序付けられたノードの命令を，ターゲットマシンの対応するコードへ変換する☆．

3. 並列実行メカニズム

ターゲット言語である *Valid*は，原則として自由変数を許さないため，引数と局所的に定義した値のみで関数実行が可能である．従って，UNIX の fork に見られる親プロセスのスタックのコピー等は必要でない．我々は，ターゲットマシン Symmetry の並列 OS である DYNIX 上に，低コストな fork-join 処理を実現するメカニズムを実現し，その上で，生成コードを実行した．ここでは，我々が実現した並列実行メカニズムについて述べる．

関数実行は，Frame と呼ぶ関数の実行環境を保持する共有メモリ上の構造データを用いて行う．**図 5** に Frame の構造を示す☆☆．実行コードアドレスは，全プロセッサに共有される命令コード領域中のアドレスで，実行開始アドレスを表す．呼出側 Frame アドレス，同期カウンタアドレスは，後述する同期（join）操作に用いる．局所変数スロットは，引数や局所的に

☆ 現在は，C 言語を生成し，ターゲットマシンの C コンパイラを用いて，オブジェクトコードを得ている．*Valid*の関数定義に相当する C の手続き定義を生成し，fork，join 処理を行う実行時ライブラリとリンクして実現している．

☆☆ 図では省略したが，Frame は他に排他制御のための lock 変数，待ち行列を構成するための他の Frame へのポインタ，レジスタの退避領域を含む．

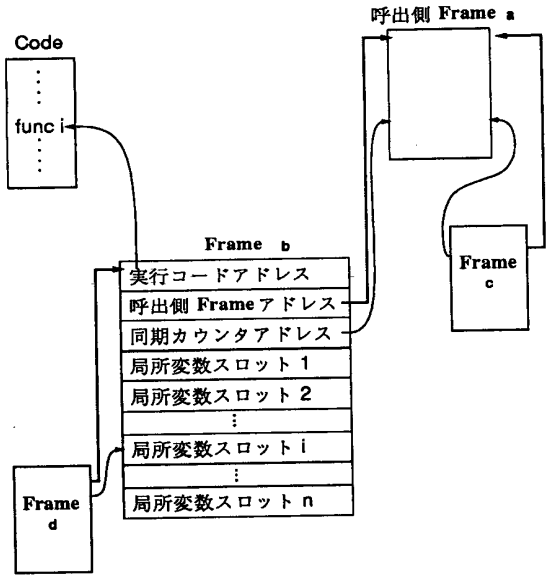

図 5 フレーム
Fig. 5 Frame.

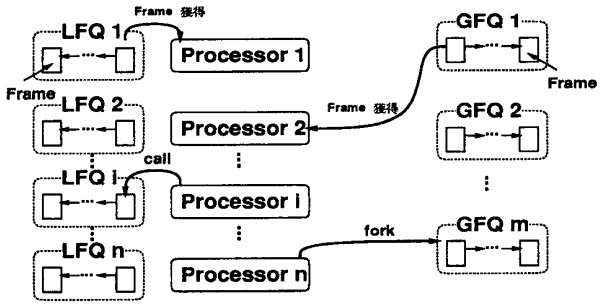

図 6 並列実行メカニズム
Fig. 6 Overview of multi-task monitor.

定義したデータの保持に用い，一部，同期カウンタとしても用いる．Frame は関数適用時に動的に確保，設定され，関数実行終了時まで存在する．

図 6 に，並列実行メカニズムの概要を示す．関数適用（fork/call 実行）は次のようにして行う．まず，Frame のための領域を確保し，次に，実行コードアドレス，呼出側 Frame アドレス（カレント Frame アドレス），同期カウンタアドレス（カレント Frame の局所変数スロットアドレス），実引数をセットし，Frame を実行可能にする．次にそれを fork なら Global Frame Queue (GFQ) へ（図 6 中の Processor n），call なら Local Frame Queue (LFQ) へエントリする（図 6 中の Processor i）．GFQ，LFQ はどちらも実行可能な Frame を保持する LIFO キューであるが，GFQ は並列実行のためのキューで，共有領域上に存在し，全プロセッサからアクセス可能であるのに対し，LFQ は逐次実行のためのキューで，プロセッサ毎のプライベート領域に存在し，対応するプロセッサのみがアクセスできる．各プロセッサは，まず，LFQ を調べ，もし Frame が存在すれば，それを実行する（図 6 中の Processor 1）．LFQ が空であれば，GFQ を調べる．プロセッサは，GFQ 1~m を順に走査し，空でない GFQ を探す．もし，空でない GFQ が見つかり，かつ，lock 操作が成功した場合，先頭 Frame を取り出し，実行コードアドレスへ制御を移し，関数コードを実行する（図 6 中の Processor 2）．

2.2節で述べたコードスケジューリングより，呼出側は，関数適用の結果値を参照する命令実行の前に必ず join 操作を行う．join は，同期カウンタ方式で実現している．同期カウンタは，初期値は 1 で，fork 時に 1 増やし，join 時に 1 減らす．join 操作の結果，同期カウンタが 0 になった場合は，以降の命令を続行するが，そうでない場合，プロセッサは，現 Frame の実行コードアドレスへ次命令アドレスをセットした後，放棄する．その後，新たな実行可能 Frame の獲得，実行を行う．join 操作は，関数実行終了時に呼出側へ結果値を返すときにも行う．同期カウンタアドレスが指す同期カウンタに対し，join 操作を行う．その結果，同期カウンタが 0 になった場合，呼出側 Frame アドレスが指す Frame はサスペンド状態（放棄されている状態）であるので，プロセッサは，現 Frame 解放後，呼出側 Frame の実行を再開する．同期カウンタを 0 にできなかった場合は，新たな実行可能 Frame の獲得，実行を行う．

共有メモリマシンでは，排他制御による処理の逐次化がオーバヘッドになる．本方式で必須となる排他制御は，GFQ へのアクセス，fork/join 時の同期カウンタへのアクセスである．GFQ へのアクセスに対しては，GFQ を複数個用意することで，排他制御による待ちをなくした．各プロセッサは，GFQ の lock 操作に失敗した場合，直ちに他の GFQ の lock 操作を試みる．lock 操作の試みと成功/失敗の判定の手間は，80486 CPU の場合，3 命令である．さらに，排他制御が不要である LFQ を加え，LFQ へのアクセスを GFQ へのアクセスに優先させることで，GFQ アクセスによる排他制御の頻度を減らしている．LFQ の使用は，関数適用の逐次実行を意味するが，並列性を損なわないよう，コンパイル時に決定している（2.2節の手続き A）．同期カウンタの加減算は，80486 CPU の場合，不可分な命令がサポートされているため，1

表 1 生成コードの性能評価
Table 1 Time comparisons of *Valid*, SISAL and C, and speedups and overheads of *Valid*.

Program	Time (s)					Speedup	Overhead
	Valid 16cpus	C 1cpu		SISAL 16cpus			
sum(1, 10^6)	0.0241	0.241	(10.0)	0.0200	(0.830)	15.0	7.69%
matrix(128)	0.987	4.49	(4.55)	0.323	(0.327)	12.8	37.5%
matrix(256)	9.07	38.4	(4.23)	3.11	(0.343)	13.3	23.4%
nqueen(10)	4.21	25.3	(6.01)	65.2	(15.5)	7.48	15.5%
qsort(10^4)	3.21	12.1	(3.77)	8.09	(2.52)	5.07	65.8%

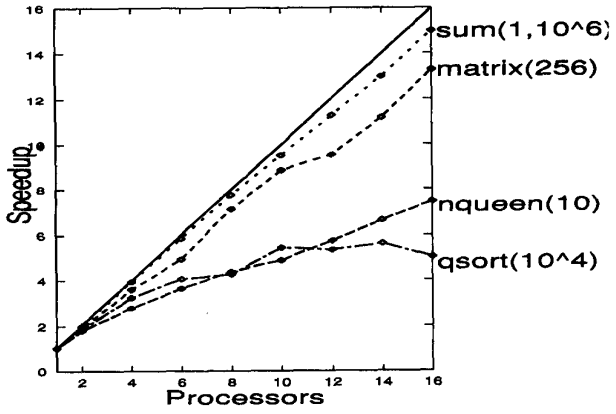

図 7 スピードアップ
Fig. 7 Speedup graphs for four benchmark *Valid* programs.

命令で行うことができる．そのため，命令レベルでの待ちは生じない．

4. 生成コードの評価

生成コードの評価は，Sequent Symmetry S2000 上で 16 個のプロセッサを用いて行った．評価は，C プログラムと SISAL プログラムとの実行時間の比較，スピードアップ（プロセッサ数 1 ～ 16）について行った．SISAL は数値計算の並列処理をターゲットとした関数型言語で，SISAL コンパイラはループに対し並列化を行う[9]．比較対象とする C，SISAL プログラムは，*Valid* プログラムと同一のアルゴリズムを用いるものとする．C プログラムは逐次で最適化なし，SISAL プログラムはコンパイラ（OSC version 12.9）のデフォルト設定での並列化および最適化でコンパイルした．C の逐次プログラムとの比較は，並列処理の主目的の一つである高速処理の達成度を実用的な見地から評価するためである．SISAL プログラムとの比較は，SISAL では並列化はループに限られるものの，完成した並列言語処理系に対する本処理系の性能を評価するためである．スピードアップは，排他制御やバスの飽和など並列処理に起因するオーバヘッドを評価するためである．C，SISAL プログラムとの比較および生成コードの性能分析結果を**表 1** に示す．スピードアップの評価結果を**図 7** に示す．表 1 について，Time は *Valid*，C，SISAL プログラムの実行時間を秒単位でそれぞれ示す．値は *Valid*，C プログラムについては，システムコールを使って計測した user time，SISAL プログラムについては，SISAL の性能評価ツール speedups を用いて求めたものである．括弧内の数字は *Valid* とのスピード比で次式で求めた値である．

$$\text{スピード比} = \frac{\text{C または SISAL での実行時間}}{\textit{Valid}\text{の実行時間}}$$

Speedup は 1 プロセッサでのスピードに対する 16 プロセッサでのスピードの比，Overhead は，16 プロセッサでのプログラム実行時間中に，システムが占める時間の割合を示す．システム時間には，fork/call 処理，join 処理，Frame 獲得が含まれる．値は DYNIX が提供する profiler を用いて求めた．図 7 について，グラフの横軸はプロセッサ数，縦軸はスピードアップを示し，斜めの直線は理想的なスピードアップを示す．

sum(x,y) は $\sum_{i=x}^{y} i$ を求めるプログラムで，*Valid*，SISAL ではリダクション演算を用いたループ（*Valid* では並列式）で実現している．スピードは C の 10.0 倍，SISAL の 0.83 倍，スピードアップは 15.0 倍であった．SISAL とは，並列実行メカニズムと最適化の差異がスピードの差として現れたが，並列式，リダクション演算に関しては，十分な性能を達成できたといえる．

matrix(n) は n × n 行列同士の積を求めるプログラムである．スピードは n=256 のとき，C の 4.23 倍，SISAL の 0.343 倍であった．ちなみに，C のナイー

ブな並列プログラム☆は，0.422秒（n=128），2.85秒（n=256）で，スピードを比較すると，*Valid*はn=256のとき0.314倍である．SISAL，Cの並列プログラムとのスピードの差は配列の実装の差異によるものである．C，SISALでは，配列は同一型の集まりとしているが，*Valid*では，型は必ずしも一様である必要はない．そのため，配列の実装がC，SISALの実装よりも複雑にならざるを得なかった．*Valid*でもC，SISAL同様，配列は同一型の集まりとすれば，効率的な実装が可能となり，SISAL，Cの並列プログラムと同等のスピードを達成できると考える．

nqueen(n) はn-Queenパズルの全解探索プログラムで，Cは本システムのリスト処理ライブラリを用い，SISALはストリームを用いた．スピードはCの6.01倍，SISALの15.5倍であった．nqueenでは，SISALのコードは並列化されず，逐次実行となった．理由は，nqueenの場合，解の探索が終了するまで，解のリスト（SISALの場合はストリーム）のサイズが決定せず，SISALではサイズが動的に決まるストリーム（または配列）の定義ループは並列化の対象外であるためである．一方，本方式（*Valid*）では，並列化の対象が関数適用であるため，並列化でき，7.48倍のスピードアップを達成できた．

qsort(n) はn要素のリストのQuick Sortプログラムである．C，SISALの実装は前述したnqueen同様である．スピードはCの3.77倍，SISALの2.52倍であった．SISALのコードはnqueenと同様の理由で並列化されなかったにも関わらず，*Valid*はnqueenと異なり16プロセッサを用いたわりに速くない．原因は，実行時間の62%がFrame獲得のためのGFQのアクセスであったことから，リスト生成と関数適用のオーバラップができなかったためであると考えられる．ストリーム並列処理を実現すれば解決できるが，要素単位の同期が必要となるため，排他制御，タスクスイッチの頻度の増加がオーバヘッドになると考えられる．ストリーム並列処理は現在，実装中である．

関数適用レベルでの並列処理は，ループの並列化よりも多くの並列性を抽出できる反面，Fibonacci関数のように，関数本体のコストが小さく，関数適用の回数が多い場合，並列制御の処理がオーバヘッドになる．関数本体のコストおよび関数適用頻度に対する並列制御のオーバヘッドを評価した．用いたプログラムsum'(x,y,i) は，x ～ yまでの数の総和を求めるプログラムである．

☆ 文献10) のGuide to Parallel Programming, Page 5–27に掲載されているプログラムに対し，整数型の配列とする，時間計測の命令を挿入する等の修正を行ったプログラムである．最適化を指定してコンパイルした．

表2 性能に対する関数本体のコストの影響
Table 2 Effect of the cost of the function **sum'(1,10^6,i)** on performance.

Parameter i	Time(s)	Speedup	Overhead
$i = 1$	6.75	6.23	63.9%
$i = 10$	0.935	6.85	61.9%
$i = 10^2$	0.155	9.79	44.6%
$i = 10^3$	0.0648	14.0	11.5%

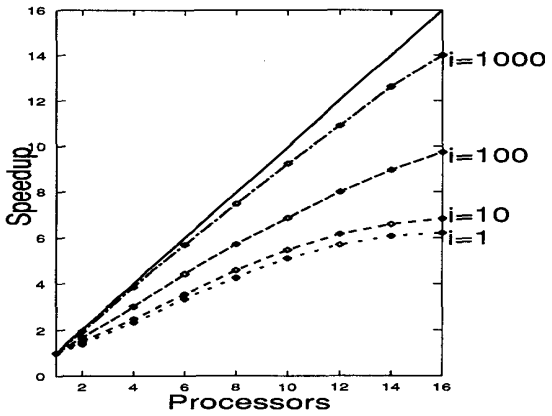

図8 sum'(1,10^6,i) のスピードアップ
Fig. 8 Speedup graphs for sum'(1,10^6,i).

```
function sum'(x,y,i:int) return(int)
 = if y-x < i then
     for (s,j:int) init(0,x) body
       if j > y then s else recur(s+j, j+1)
   else {let m = (y - x) / 2,
         in sum'(x, m) + sum'(m+1, y)};
```

iで分割統治法による総和からループ実行による逐次的な総和へアルゴリズムを切替える範囲を指定する（i = 1のときは全て分割統治法による総和となる）．従って，iの値が大きいほど関数本体のコストは大きくなり，関数適用の頻度は低くなる．x=1，y=10^6，i = 1，10，100，1000についての評価結果（16プロセッサ使用）を**表2，図8**に示す．表2のTime，Speedup，Overheadの値は，表1と同じ方法で求めた．実行時間の差は関数適用のオーバヘッドを反映し，スピードアップの差は排他制御による処理の逐次化，バス飽和によるオーバヘッドを反映している．i<100では，システムのオーバヘッドが50%を越えている．オーバヘッドの内容は，Frame獲得とコンテクストスイッチが全体処理時間の約40%を占めていたことか

ら，GFQ アクセスの競合とバスの飽和であると考えられる．sum' の再帰式部分はループ実行のコードにコンパイルされ，内容は，以下の通りである．

- データ転送（メモリ→レジスタ）5 命令
- 比較（メモリとレジスタ）1 命令
- 整数加算（レジスタとレジスタ）2 命令
- 条件付きジャンプ 1 命令
- 無条件ジャンプ 1 命令

本実装で実用的な効率を期待するには，上記内容のコード 100 回分以上に相当する関数本体のコストが必要である．

fork 処理のオーバヘッドに対する解決方法として，コンパイル時に関数のコストを見積り，低コストの関数は fork しない，インライン展開することが考えられる．しかし，再帰呼び出しを含む関数では静的に正確なコストの見積りをすることはできない．現仕様の *Valid*では，sum' で行ったように，アルゴリズム設計段階で解決する以外になく，これはプログラムの portability を損ねる．*Valid*に限らず，関数型言語は，逐次，並列実行を意識せずに記述できる反面，上述したプログラムの最適化が困難である欠点を持つ．例えば，関数の逐次実行，並列実行の選択，疎結合マシンの場合，各プロセッサ上への関数，構造データの配置についての最適化を，たとえプログラマが知っていてもプログラムに反映することができない．関数型言語の利点を損なわずに，上記の問題を解決する方法として，Yale 大学の ParAlfl[11),12)]等，annotation などの metalinguistic device を関数型言語に組み入れる手法が提案されている．関数の実行方法，関数，構造データのプロセッサへの写像はプログラムの意味とは独立であるため，metalinguistic device の組入れ拡張を *Valid*に施すことで，sum' で指摘した問題を解決することができると考える．そこで，*Valid*に関数実行方式の最適化のための拡張を施すことを試みた．

GFQ へのアクセスによる排他制御およびバス飽和の原因である fork 処理は 2.2節で述べたように，コンパイラが決定していたが，これをプログラマが制御できるよう関数適用を拡張した．

関数適用 ::= 関数名（実引数並び）$[論理式]

$[ ] 内の論理式が真であれば fork するが，偽であれば fork せず，現プロセッサが継続実行することを意味する．**表 3**，**図 9** の fib8(30) は，上記の拡張を fibo(30) に施したものである．表 3 の Time，Speedup，Overhead の値は，表 1 と同じ方法で求めた．fib8 は，fork

表 3 fork 処理の制御 annotation 導入による効果
Table 3 Effect of introducing annotation to control fork operation.

Program	Time(s)	Speedup	Overhead
fibo(30)	7.61	6.24	73.5%
fib8(30)	2.84	15.4	70.6%

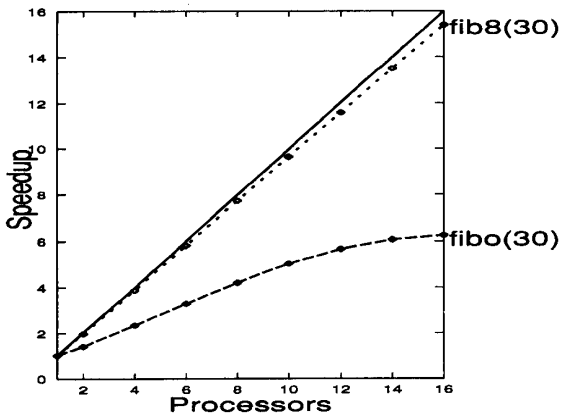

図 9 fibo(30), fib8(30) のスピードアップ
Fig. 9 Speedup graphs for fibo(30) and fib8(30).

制御を仮引数 n を用いて行う．n≥8 であれば fork し，そうでなければ fork しない．fib8 の *Valid*プログラムを以下に示す．

```
function fib8(n:int)return(int)
 = if n<2 then 1
   else fib8(n-1)$[n>=8]+fib8(n-2)$[False];
```

関数適用の拡張により，スピードアップについて，拡張無しの場合の 6.24 倍から 15.4 倍へと大幅に改善することができた．fork 処理回数の減少に伴い，GFQ へのアクセス回数が減ったことによる．しかし，Overhead については，Frame 生成のコストが，依然として大きいため，大きな改善は見られなかった．解決方法として，関数の現プロセッサによる継続実行の場合，各プロセッサが持つスタック上で C と同様のスタイルの関数呼出しを行う，あるいは，Frame 内にスタックエリアを設け，Frame 上で C と同様のスタイルの関数適用を行うことが考えられる．前者の方法は，関数実行環境がスタック上に固定してしまうので，並列実行メカニズム自体を変更しなければならない．我々は，既に文献 13) でこの方法を試み評価した．その結果，fib8(16 プロセッサ使用) で C プログラムの 5 倍のスピード（0.899 秒）を達成できた．しかし，関数実行環境がスタック上に固定されるため，lenient な実行ではデッドロックを起こす．後者は，再帰呼出しを含む関数に対し，あらかじめ，十分大きな Frame を確保する必要があるので，メモリ効率が悪くなる欠点が

ある．しかし，本手法と組み合わせて，関数実行方法（fork/call）に応じ，使用するFrameのスタックの有無を切替える方法をとれば解決できると考える．

今回取り入れたfork制御は，ターゲットマシンを疎結合マシンにした場合，関数適用と物理プロセッサのマッピングへ容易に拡張できる．$[ ]内に論理式でなく整数値をとる式を許し，値をプロセッサIDと解釈しマッピングするコードを生成すれば，ParAlflのmapped expressionの機能を実現できる．

5. 関連研究との比較

関数型プログラムを既存の並列マシン上で並列実行する研究にThomas Johnssonらの$<\nu, G>$-マシン[4]，CullerらのTAM (Threaded Abstract Machine)[6]などがある．

$<\nu, G>$-マシンはスーパコンビネータ[2]を効率良く実行するグラフリダクションマシンである．$<\nu, G>$-マシンの実行コードは，グラフリダクションマシンの振舞いを忠実に反映したコードで，関数型言語Lazy MLをスーパコンビネータの定義式に変換し，それをコンパイルして得る．$<\nu, G>$-マシンでは，命令の実行順序は実行時に決まるため，プログラム実行に必要な情報は全てグラフの形のプログラム断片として共有メモリ上に存在する．そのため，プロセッサ数が多くなるとバスが飽和し効率が低下する．本方式でも，GFQアクセスによってバス飽和が生じる場合がある．しかし，命令の実行順序を静的に決定するため，関数をプロセッサ毎にローカルなスタックとレジスタを用いて実行し，共有メモリ上に存在するFrameへは引数，局所変数に対するアクセスのみに限定することができる．従って，本方式の方が，バス飽和を避けるためのキャッシュの有効利用が可能である点で有利である．ただし，命令の実行順序を静的に決定する本方式では，高速な遅延評価，高階関数を実現することは困難で，特にLispでのeval関数実行に相当する関数定義自体を実行時に合成する関数の実行は，インタプリタの組み込みなど特別な機構が必要である．

TAMはBerkeley大学のCullerらによって提案された抽象計算モデルである．分散メモリマシンを想定し，メッセージ通信メカニズムでthreadレベルの並列処理を実現する．TAMの実行コードであるthread記述言語TL0は，現在，データフロー言語Id90をコンパイルして生成される．まず，Id90プログラムからグラフを生成する．次に，グラフをプログラムの意味を保存する規則に基づき分割，併合し，いくつかの部分グラフ（partition）へ変換する．最後に，各partition中の命令を逐次コード化し，thread定義を生成する[6]．本方式とTAMとの主な相違点は，TAMではthread内に条件分岐，ループを許さないが，本方式では許している，TAMではthread内部ではコンテクストスイッチを起こさないが，本方式では，コンテクストスイッチを起こすjoin操作を内部にもつ，TAMでの関数実行はnon-strictであるが，本方式はstrictである，をあげることができる．これより，本方式はTAMでの静的スケジューリングの範囲を，より広くしたものと見なすことができる．つまり，TAMでは複数のthreadとなり，動的に実行制御を行う部分を，本方式では静的にスケジューリングして一つのthreadとしている．従って，TAMはプログラムのlenient性を利用できる分，本方式に比べ，プロセッサ稼働率を高く保持できる点で有利である．ところが，共有メモリマシンでの実装を考えると，TAMでは，効率化のため，メッセージ通信によるfork操作を，ターゲットFrameへの直接的なアクセスで実現することになる．その結果，Frame内threadの動的スケジューリング，同期には複数のプロセッサが関与することになり，排他制御が必要である．TAMでは，複数のpartitionをlenient性を壊さない規則に従い一つのpartitionへ併合することで，上記のオーバヘッドを減らすことができるが，本方式のコードは，更に，条件分岐，ループに対して併合を行った場合に相当するため，本方式が排他制御の頻度がより低い分有利である．コード生成については，本方式はTAMよりも分割の制限が弱いため，TAMでのpartition併合と同様の効果を2.2節で述べたアルゴリズムで実現でき，コンパイル手法はより単純である．また，前述したようにメッセージ通信を用いない共有メモリ型マシンを想定している本方式では，TAMでのMessage handlerに相当するコードが不要となり，生成コードはよりコンパクトになる．プログラムのlenient性の利用と，それに伴う排他制御のオーバヘッドの増加についての評価，および，分散メモリマシンへの実装は今後の課題である．

6. おわりに

本論文では，関数型言語を共有メモリマシン上で並列実行するためのコンパイル手法，および，実装方式を提案した．この手法は関数型言語 *Valid* をデータフロー解析することで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 Sequent Symmetry S2000 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，fork 回数が極端に多い場合，排他制御による処理の逐次化，バスの飽和によって効率が落ちること，その場合，fork 制御の annotation を導入することで解決できることがわかった．また，関数本体のコストが極端に小さい場合の関数起動のコストと構造データの扱いがネックであることがわかった．今後は，上記の問題を解決するために，ストリーム並列処理の実現，関数実行，構造データの扱いに対する metalinguistic device の *Valid* への組み込み拡張を行い，また，プログラムの lenient 性の利用，分散メモリ型マシンへの実装も行う予定である．

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, pp. 99–108 (1988).
2) Jones, S. L.: *The Implementation of Functional Programming Language*, PRENTICE-HALL INTERNATIONAL (1987).
3) Johnsson, T.: *Compiling Lazy Functional Language*, PhD Thesis, Chalmers University of Technology, Sweden (1987).
4) Augustsson, L. and Johnsson, T.: Parallel Graph Reduction with the$< \nu, G >$-machine, *ACM Proc. 4th International Conference on Functional Programming Languages and Computer Architecture*, pp. 202–213 (1989).
5) Hudak, P.: Distributed Execution of Functional Programs Using Serial Combinators, *IEEE Trans. Comput.*, Vol. C-34, No. 10, p. 881 (1985).
6) Schauser, K. E., Culler, D. E. and von Eicken, T.: Compiler-Controlled Multithreading for Lenient Parallel Languages, *Proc. of FPCA'91 Conference on Functional Programming Languages and Computer Architecture*, Springer-Verlag (1991).
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, p. 1532 (1988).
8) Sequent Computer Systems, Inc.: *System Summary Manuals* (1990).
9) Lawrence Livermore National Laboratory: *SISAL: Streams and Iteration in a Single-Assignment Language: Reference Manual*, Livermore, Calif., Ver.1.2 edition (1985). Manual M-146, Rev.1.
10) Sequent Computer Systems, Inc.: *Language Tools Manuals* (1987).
11) Hudak, P.: Para-Functional Programming, *IEEE Computer*, Vol. 19, No. 8, p. 60 (1986).
12) Hudak, P.: Exploring Parafunctional Programming: Separating the What from the How, *IEEE Software*, Vol. 5, No. 1, pp. 54–61 (1988).
13) 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, 信学技報, Vol. 92, No. 493, pp. 29–37 (1993).

(平成 5 年 9 月 17 日受付)
(平成 5 年 12 月 9 日採録)

高橋 英一 (正会員)

昭和 41 年生．平成 3 年九州大学大学院総合理工学研究科情報システム学専攻修士課程修了．現在，同専攻博士後期課程在学中．関数型プログラミング言語を用いた並列処理の研究に従事．ソフトウェア科学会会員．

谷口 倫一郎 (正会員)

昭和 30 年生．昭和 53 年九州大学工学部情報工学科卒業．昭和 55 年同大学院工学研究科修士課程修了．同年九州大学大学院総合理工学研究科情報システム学専攻助手．平成元年より同助教授．工学博士．画像理解，画像処理システム，並列処理システムの研究に従事．電子情報通信学会，人工知能学会各会員．

雨宮 真人 (正会員)

昭和 17 年生．昭和 42 年九州大学工学部電子工学科卒業．昭和 44 年同大学院工学研究科修士課程修了．同年日本電信電話公社武蔵野電気通信研究所入所．以来，プログラミング言語・処理系，自然言語理解，データフロー・アーキテクチャ，並列処理，関数型／論理型言語，知能処理アーキテクチャ，等の研究に従事．現在九州大学大学院総合理工学研究科情報システム学専攻教授．工学博士．電子情報通信学会，ソフトウェア科学会，人工知能学会，IEEE，AAAI 各会員．