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
九州大学大学院総合理工学研究科情報システム学専攻

谷口, 倫一郎
九州大学大学院総合理工学研究科情報システム学専攻

雨宮, 真人
九州大学大学院総合理工学研究科情報システム学専攻

https://doi.org/10.15017/17290

出版情報：九州大学大学院総合理工学報告. 14 (4), pp.403-410, 1993-03-01. 九州大学大学院総合理工学研究科
バージョン：
権利関係：

九州大学大学院総合理工学研究科報告
第14巻　第4号403-410頁 平成5年3月

Engineering Sciences Reports, Kyushu University
(KYUSHU DAIGAKU SOGORIKOGAKU KENKYUKA HOKOKU)
Vol. 14, No. 4 pp. 403-410 Mar. 1993

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**

(平成4年11月30日　受理)

Compiling Technique based on Dataflow Analysis for Functional Programming Language Valid

Eiichi TAKAHASHI, Rin-ichiro TANIGUCHI and Makoto AMAMIYA

In this paper, we present a compiling method to translate a functional programming language Valid into an object code executable on a commercially available shared memory multiprocessor (a Sequent Symmetry S2000). 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 serialized 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 Sequent S2000 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う上で種々の魅力的な性質を持っている[1]．近年，スーパコンビネータ[2]，G-マシン[3]，シリアルコンビネータ[4]，Process-Oriented Dataflow System (PODS)[5] など，従来用いられてきたインタプリタ実行方式による実行効率の悪さを改善する方式が提案されてきている．スーパコンビネータは非効率的な関数型プログラムのリダクション操作を改善する手法である．G-マシン，シリアルコンビネータはスーパコンビネータを効率的に実行するグラフリダクション操作をノイマン型マシンコードに変換する．G-マシンでのグラフのノード処理を並列処理に拡張した〈ν, G〉-マシン[6]は密結合マシン上に実現されている．シリアルコンビネータは疎結合マシン上で実現され，コードをプロセッサ間の通信コストに基づく最適な粒度でプロセッサへ写像する．PODS はノイマン型の疎結合マシンをターゲットにし，thread レベルのデータフローグラフをノイマン型マシンのコードへ変換する．

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]．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．一般に，密結合マシンでは，プロセス管理のコストが高く，単一バス結合のためデータ転送率が低い．従って，プロセス数が多く，同期などの通信が頻繁に起こる細粒度の並列処理では実行効率は上がらない．そのため，粒度が粗い関数適用レベルでの並列実行をここでは考える．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

*情報システム学専攻博士後期課程
**情報システム学専攻

た方式で行う．Symmetry の OS である DYNIX が提供する fork-join は，関数適用レベルの並列処理では不要である親プロセススタックイメージの子プロセス環境へのコピーを行うためオーバヘッドになる．そこで我々はコピーを行わない低コストの fork-join 処理を DYNIX 上に実現した．

まず，2章で関数型言語 Valid と本手法での並列実行方式について述べる．次に3章でコンパイラの構成について述べ，4章で生成コードの実行効率についての評価結果を示す．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. 関数型言語 Valid と並列実行方式

はじめに Valid を簡単に紹介し，次に関数適用レベルでの並列実行方式について述べる．

2.1 Valid

関数型言語 Valid は，タイプ付きの関数型言語に基づき，Algol 系のシンタックスを持つ，プログラムは全て関数定義と式で構成する．式にはブロック式，条件式，再帰式，並列式，関数適用などがある[7]．例1に Valid で記述した Fibonacci 関数 fibo の定義を示す．

例1

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

[並列式]

Valid では，並列式で配列やリスト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．

foreach elem-vars **in** set-exprs **do** parallel-body

並列式は並列実行可能な処理単位を fork-join 概念に基づき陽に記述する．上記形式において set-exprs は range 型の式の並びであり，elem-vars は set-exprs で生成される集合のインデックスを示す変数の並びである．並列式の本体 parallel-body は集合の各要素に対して並列に活性化（fork）される処理単位である．各並列式本体から生成される値は，（暗黙の）join 操作により，配列またはリスト構造のデータに作り上げられる．

例2

```
a: array=foreach u in [1..5] do fact(u)
```

例2は1から5までの階乗を求める式である．これを評価すると配列 [1, 2, 6, 24, 120] を得る．もし，a の型がリストであればリストを得る．

2.2 並列実行方式

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う．例1の一部

$$\underset{(a)}{\text{fibo}(n-1)} + \underset{(b)}{\text{fibo}(n-2)}$$

において，2つの関数適用 (a), (b) の間にはデータ依存関係はないので，(a), (b) を実行する2つの子プロセスを生成（fork）し並列に実行する．＋演算は関数適用 (a), (b) に依存しているので，同期をとった後（join）実行する．同期はバリア同期で実現する．

並列式は，そのまま fork-join スタイルで実現できるが，一般に生成するプロセス数は物理プロセッサ数に比べ非常に大きく，プロセス生成がオーバヘッドになり効率を落す．そこで，本方式では，並列式で生成するプロセスは仮想プロセスとみなし，物理プロセッサ数だけ生成したプロセスに均等配分して実行する†．各プロセスは割り当てられた仮想プロセスを逐次的に処理する．例えば，仮想プロセス数が1000で物理プロセッサ数が10の場合，各プロセスは100個の仮想プロセスを逐次的に処理することになる．

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．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．以下，各フェーズについて説明する．

3.1 フェーズⅠ-グラフ生成

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

†物理プロセッサ数は，プログラム実行開始時に Sequent C が提供するライブラリ関数を呼び出して求めている．

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関係を表す．グラフは関数定義毎に完結している．**Fig. 1** に例1の Fibonacci 関数のグラフを示す．

以下，Valid の主なプログラム構造である条件式，再帰式，並列式について説明する．

(1) 条件式

Fig. 2 (a) は，p(x) が真の時 qt(y) を，偽の時 qf(z) を評価し，結果を値とする条件式のグラフを示す．sw ノードは，アーク t, f で，真の時実行可能となる命令，偽の時実行可能となる命令をそれぞれ指す．条件式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．条件式の一般的な構造を **Fig. 2 (b)** に示す．

(2) 再帰式

Fig. 3 は末尾再帰構造の再帰式のグラフである．末尾再帰構造からはループ実行のグラフを生成する．**Fig. 3 (a)** では，まず，x, y をそれぞれ $f_1(a)$, $f_2(b)$ に初期化し再帰式本体を評価する．p(x) が真の時，return 式を評価し，結果 (g(x, y, z) の評価値) を再帰式の値とする．p(x) が偽の時は recur 式を評価し，結果 ($h_1(x)$, $h_2(y)$ のそれぞれの評価値) を新たに x, y とし，再び再帰式本体の評価を行う．末尾再帰構造の再帰式では recur 式は jump ノードになる．jump ノードはオペランドに $loop_i$ ノードのラベルをとり，制御が $loop_i$ ノードへ移ることを意味する．再帰式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．**Fig. 3 (b)** に一般的な末尾再帰式のグラフの構造を示す．末尾再帰構造でない再帰式では，再帰式本体を無名の関数として定義し，関数適用のグラフを生成する．

(3) 並列式

並列式はプロセスを物理プロセッサ数だけ生成し，それらに仮想プロセスを均等配分するコードで実現する．並列式

a: array＝foreach u in [l..h] do Exp

に対し，コンパイラは，まず，並列式本体 Exp を含む関数 _f01 を新たに定義する．関数 _f01 はあらかじめ割り当てた配列領域の $(h-l+1)/N_{pe}$ 個の要素を $(h-l+1)/N_{pe}$ 回繰り返し Exp を計算し定義する関数である．ただし，N_{pe} は物理プロセッサ数を表す定数

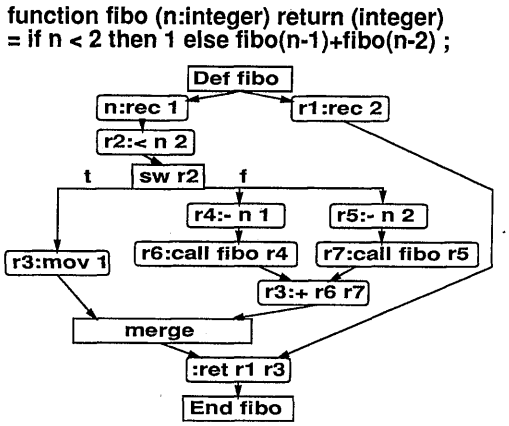

Fig. 1 Example of controlflow graph.

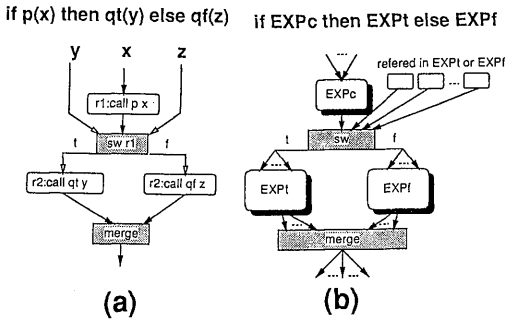

Fig. 2 Controlflow graph of conditinal expression.
(a) Example of graph.
(b) Structure of graph.

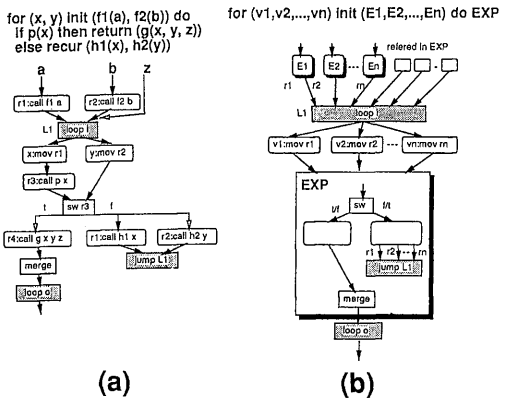

Fig. 3 Controlflow graph of recurrence expression.
(a) Example of graph.
(b) Structure of graph.

である．次に並列式を _f01 の関数適用を N_{pe} 回 fork する再帰式に変換する．変換後，再帰式（末尾再帰構造）のグラフを生成する．以下に，上記並列式の変換後のコードを示す．

```
a= {let _f01=function (s, e: integer; v: array)
    return (array)
      =for (u, w) init (s, v) do
        if u>e then return w
        else recur (u+Npe, write (w, u, Exp));
  in for (p, v) init (0, create_vector (h−1+1)) do
      if p<Npe then return v
      else {let ! =fork (_f01, p+1, h, v) in recur
      (p+1, v)}
```

ただし，上記コードは正確には Valid ではなく，手続き的な中間コードである．配列 v は create_vector 手続きによりヒープ上に割り当てられ，配列の各要素は関数 _f01 を実行する N_{pe} 個の子プロセスによりセットされる．要素の値をセットする手続き wirte (w, u, Exp) は Exp の評価値を配列 w の要素 u にセットするという意味である．関数との配列の授受は先頭ポインタを用い，変数 a の定義は子プロセス全てが終了した後である．

3.2 フェーズⅡ-コード生成

フェーズⅡでは，まず，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（逐次化）する．次に各命令をターゲットマシンの対応するコードへ変換する．ターゲットマシンコードへの変換は単純な置き換えにすぎないので，ここでは，命令のスケジューリングについて述べる．

命令スケジューリングは同期ポイントの決定，グラフの逐次コード化の順で行う．

[同期ポイントの決定]

グラフがデータ依存関係に基づいているため，関数適用ノードの子ノードは，

case 1　関数適用の結果を参照する命令ノード
case 2　sw ノード，$loop_i$ ノード
case 3　merge ノード，$loop_0$ ノード

である．case 1 の場合，同期ポイントは関数適用の直後である．case 2 の場合，後述する逐次化でプログラムの意味を保存するために，同期ポイントはプログラム実行が制御構造に入る前になければならない．従って，関数適用ノードの直後である．case 3 の場合，条件式，または，再帰式の評価終了後に同期が必要となるので，子孫へのパス上で最初に出現する case 1, 2 で示したノードの直前である．

同期ポイントには join ノードを挿入する．join ノードのオペランドは関数適用ノードのラベルで，fork-join の対応関係を保持する．

[グラフの逐次化]

グラフの逐次化は縦型探索に基づきグラフを走査し，走査した順を各命令の実行順序とすることで行う．グラフ走査では，親ノードが全て走査済である子ノードを任意に選択する．ただし，join ノードへは走査可能なノードがなくなった後，走査を開始する．また，sw ノード，$loop_i$ ノードからは，プログラムの意味を保存するために，それぞれ対応する merge ノード，$loop_0$ ノードまでに走査手続きを再帰的に適用する．**Fig. 4** にグラフ走査の概念図を示す．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

1．グラフの全コードを未走査に，走査開始ノード集合を S= {n | n は親ノードを持たないノード} に初期化する．

2．S=Ø になるまで以下の手続きを繰り返す．

(a)　走査継続ノード集合を J=Ø に初期化して，S の各要素 n に対し手続き B を適用する．

(b)　S←J とする．

手続きB-グラフの走査

走査可能なノード n がなくなるまで以下を繰り返す．

1．もし，n が sw ノードであるなら 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 n にして手続き B を繰り返す．

2．もし，n が $loop_i$ ノードであるなら 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，$loop_0$ ノードを n にして手続き B を繰り返す．

3．n が join ノードであるなら，n を走査継続ノード集合 J に加え，バックトラックする．

4．ノード n を走査済にする．親ノードが全て走

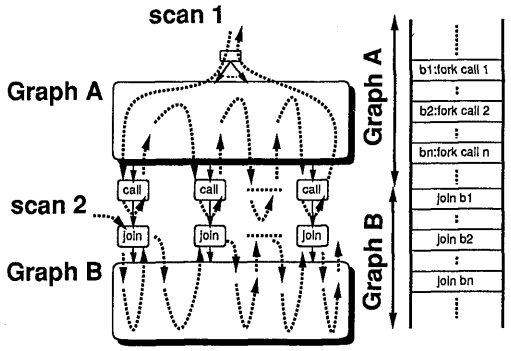

Fig. 4 Code scheduling.

査済である子ノードがあれば，それを n として手続きBを繰り返す．なければ，バックトラックする．

4. 生成コードの評価

生成コードの評価は，Sequent Symmetry S2000 上で1～16個の CPU を用いて行った．Symmetry の OS である DYNIX は fork-join を提供する．しかし，DYNIX の fork が行う親プロセススタックイメージの子プロセス環境へのコピー[9]は，関数適用レベルの並列処理では必要でなく，オーバヘッドになる．関数適用レベルの並列処理では，子プロセス実行に必要な情報は，関数のコードエントリ，引数，返り値の格納アドレス，同期で用いるバリア変数へのポインタのみである．また，子プロセス実行中，親プロセスとの通信はない．そこで，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環境を DYNIX 上に実現し，その上で，本手法で生成したコードを実行した．はじめに並列実行環境の構成について述べ，次に生成コードの評価結果を報告する．

4.1 並列実行環境

Fig. 5 に並列実行環境の構成を示す．図中，Worker はスタート時に DYNIX が 1CPU 対応で生成する並列プロセスで，仮想的なプロセッサとして振舞う．Process Queue はリングバッファで fork 時に Worker により Process Descriptor がセットされる．Process Descriptor は，関数コードエントリ，引数，返り値の格納アドレス，バリア変数へのポインタより構成される．Process Queue は各 Worker に相手 Worker の数割り当てられる．各 Process Queue には Process Descriptor をセットできる Worker が一対一に対応付けられている．Process Stack はプロセス実行時の作業領域，サスペンドプロセスの待ち行列に用いる．Worker は Process Queue から Process Descriptor を Process Stack へロードし実行する．fork 時は自分に対応した相手 Worker の Process Queue へ Process Descriptor をセットする．Process Queue に空きがない場合は，fork せず，関数適用をCプログラムの関数呼び出しと同様のスタイルで逐次的に実行する．Process Queue が空である，または，実行中の関数が終了した場合，Worker は Process Stack 上にサスペンドプロセスがあれば再起動を試みる．再起動できない，または，実行中の関数がサスペンドした場合，Process Queue から新たに Process Descriptor を Process Stack へロードし実行する．

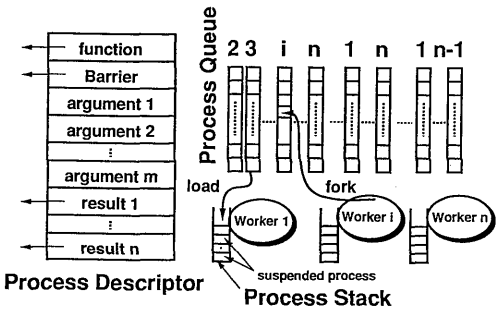

Fig. 5 Paralled function application support system.

4.2 評価結果

評価は，Cプログラムとの実行時間の比較，スピードアップ（CPU 数1～16）について行った．比較対象とするCプログラムは，Valid プログラムと同一のアルゴリズム，データ構造を用い，職人的な最適化は行っていない．評価データには DYNIX のシステムコールを使って計測した user time を用いた．Cプログラムとの比較を **Table 1** に，スピードアップを **Fig. 6** に示す．**Table 1** について，サイズは各プログラムの実行パラメータ，実行時間は16プロセッサ時の実行時間（秒），C program はCプログラムの実行時間（秒），スピード比は次式で求めた値である．

$$\text{スピード比}=\frac{\text{Cでの実行時間}}{\text{16CUP 時の実行時間}}$$

Fig. 6 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す．点線は理想的なスピードアップ，水平線はCプログラムの性能を示す．スピードアップの基準が2プロセッサ時であるのは，使用した

Table 1 Execution time in seconds of the Valid program and the C program for the nine benchmark programs.

プログラム	サイズ	実行時間(sec.)	C program(sec.)	スピード比
mult	128	1.037	10.68	10.30
mult	256	9.613	80.99	8.425
nqueen	10	4.695	47.84	10.19
qsort	10^4	4.451	25.72	5.778
sum	$10^6/1$	5.279	7.619	1.443
sum	$10^6/10$	0.848	1.487	1.754
sum	$10^6/100$	0.121	0.677	5.595
sum	$10^6/1000$	0.047	0.569	12.11
fibo	30	6.694	4.805	0.718

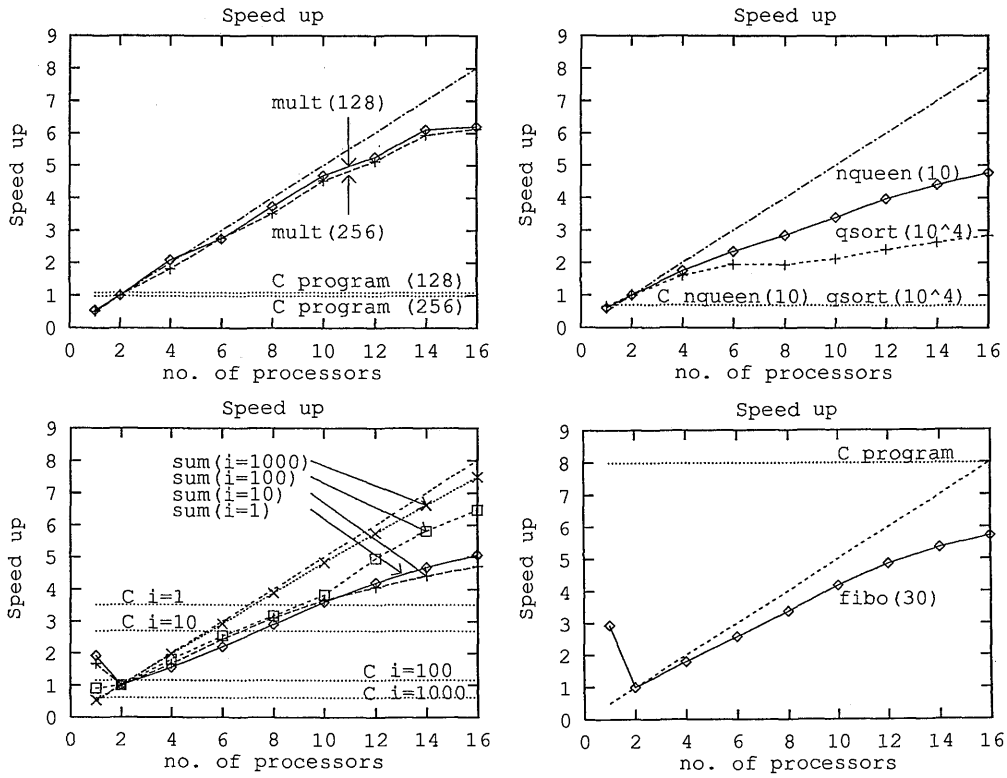

Fig. 6 Speedup graphs for nine benchmark programs.

並列実行環境では，1プロセッサ時のforkは，必ずCプログラムと同様のスタイルの関数適用となり，プロセッサ数2以上の場合と実行方式が異なるからである．スピードアップ評価の目的は並列処理に起因するオーバヘッドの評価であるので，並列処理を行わない1プロセッサ時の結果を基準にすることは不適当である．

行列積multでは，128×128行列同士の積mult(128)，256×256行列同士の積mult(256)について行った．それぞれ，Cプログラムの10倍，8倍のスピードを達成できた．スピードアップは，どちらも16プロセッサで理想の約76%のスピードアップを達成できた．

n-Queenパズルの全解探索nqueenでは，10-Queen nqueen(10)について，Quick Sort qsortでは，10000要素のソートqsort(10000)について行った．スピードは，それぞれCプログラムの10倍，5.8倍で，スピードアップは16プロセッサでそれぞれの理想の60%，35%であった．qsortの並列効果がmult, nqueenに比べて小さい理由は，現在，ストリーム並列処理をサポートしていないため，リスト生成と関数適用のオーバラップができないためである．ストリーム並列処理の実現は今後の課題であるが，データ参照時のプロセス同期のためのプロセス間通信が必要となるため，プロセス管理が複雑になりオーバヘッドが増加すると考えられる．

例題sumでは，プロセスの粒度と数に対する実行効率の変化を調べた．sum(l, h, i)のiは分割統治法による総和からループによる逐次的な総和へアルゴリズムを切替える最大の範囲を意味する．つまり，i=100のとき，ℓ～hの範囲が100以下であれば，逐次的なループで総和を求める．従って，iの値が大きいほどプロセス粒度は大きくなる（i=1のときは全て分割統治法による総和となる）．sum(l, h, i)では，1～10^6までの総和を，i=1, 10, 100, 1000についてそれぞれ行った．スピードはそれぞれCプログラムの1.4倍，1.8倍，5.6倍，12.1倍で，スピードアップは理想の63%，59%，81%，94%であった．スピードアップがどのケースも理想の約60%以上であるのに対し，スピードはi=1000以外実用的でない．理由は，iの値が小さい（1～100）場合，fork処理のコストが関数実行のコストに比べ比較的大きく，処理回数も多いため，fork処理がオーバヘッドになり，その結果，2プロセッサ時のスピードが極端に遅くなったためである．

Fibonacci関数fiboでは，fibo(30)について行った．スピードは，Cプログラムの0.718倍，スピードアップは16プロセッサで理想の72%であった．関数fiboの実行コストは小さく，fork回数が多いため，前述したsum同様，fork処理のオーバヘッドが顕著に現れた．

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等[8]，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, fiboで指摘した問題点を解決することができると考える．

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〈ν, G〉-マシン[6]，Lubomir Bicの Process-Oriented Dataflow System (PODS)[5]などがある．

〈ν, G〉-マシンはスーパコンビネータを効率良く実行するグラフリダクションマシンで，高速な遅延評価や高階関数の機能を実現できる．〈ν, G〉-マシンの実行コードである〈ν, G〉-マシンコードは，関数型言語Lazy MLをラムダリフティングと呼ぶプログラム変換手法を用いて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．コード実行につ

いてみると，〈ν, G〉-マシンは計算モデルであるグラフリダクションマシンの振舞いを忠実に反映するため，命令の実行順序は実行時に決まる．一方，本方式では式と式の間の実行順序を静的に決定し，従来型プログラムのコンパイルコードと同様なスタイルで実行する．〈ν, G〉-マシンでは実行時にリダクションを行うための頻繁なヒープ操作を必要とし，さらに十分なヒープ領域がとれない場合，ガベージコレクションを必要とする．実行中，グラフの形のプログラム断片が共有メモリ上にあるため，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．本方式では，構造データの操作以外ヒープ操作は必要でない．プログラム実行は共有メモリに設けたスタック上で行うが，他のプロセッサからのアクセスはバリア変数操作と結果値の書き込みに限られるため，アクセスの局所性が高く，キャッシュ効果により，バスの飽和を避けることができる．プログラムから引き出せる並列性は本方式の方が低いが，プロセッサ数が数個から数十個の密結合マシンの場合，プログラムが持つ並列性を全て引き出す必要はない．むしろ，粒度を粗くとり，密結合マシンでオーバヘッドとなるバス飽和を避けることが全体の実行効率をあげることになるので，上記の計算機での実行では本方式が有利である．しかし，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，特に実行時に合成され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．

PODS は，粒度を命令レベルから thread レベルに粗くしたデータフローモデルをノイマン型の疎結合マシンで実現したものである．まず，データフロー言語 Id をデータフローグラフへコンパイルする．次に，グラフをデータ依存関係に基づき Subcompact Process (SP) とよぶ逐次処理ブロックに変換する．最後に SP に分散実行のためのプリミティブを埋め込んで実行する．本手法は PODS に大変近いが，粒度が thread レベルではなく関数レベルであることとターゲットアーキテクチャが疎結合でなく密結合である点で異なっている．PODS を密結合マシン上で実現すると，本方式より粒度が小さいため，プログラムの並列性をより多く引き出せる．しかし，プロセス同士の通信が必要であるため，同期ポイントが増え，サスペンドや再起動処理が本方式より頻繁に起こる．その結果，プロセス管理コストのオーバヘッドが顕著となり効率が落ちる．従って，密結合マシン上で PODS の手法を用いて高い実行効率を達成するためには，粒度を関数レベルまで粗くしなければならず，結果的に本手法と同じ手法になる．

6. 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ンパイル手法を提案した．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，効率を実行速度の点で評価した．結果として，関数本体の実行コストがプロセス管理コストに対し極端に小さいものを除けば実用的な効率を達成できること，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，評価，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，99 (1988).

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

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

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

5) Lubomir Bic, A Process-Oriented Model for Efficient Execution of Dataflow Programs. JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **8**, 42 (1990).

6) Lennart Augustsson, Thomas Johnsson, Parallel Graph Reduction with the 〈ν, G〉-machine. ACM Proc. 4th International Conference on Functional Programming Languages and Computer Architecture, 202 (1989).

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) Sequent Computer Systems, Inc., Symmetry System Summary.