データフローモデルに基づく超並列V言語とその商用並列計算機上の実装について

日下部 茂[†] 高橋英一[††]
谷口倫一郎[†] 雨宮真人[†]

データフロー実行モデルに基づく超並列V言語を提案し，既存の汎用並列計算機上での実装を通しその実用性を示す．データフロー関数型言語をベースに，その並列処理記述における利点を損なうことなく，状態を持つ並行プロセスを直観的に記述するための抽象化単位agentを導入した．agent間の明示的結合の記述に加え，要素間の論理構造/通信形態を指定してagentの集合を記述できる抽象化単位agent fieldを導入した．V言語はlenientなセマンティクスを持ち細粒度を含む様々な粒度の並列性を内在している．我々は，同期や通信などの非局所処理を削減し計算の局所性を活用できる，マルチスレッド実行を実現するコードを生成することにより特別に細粒度並列処理をサポートしていない汎用並列計算機においても効率の良い実装を可能とした．本稿では，疎結合並列計算機AP 1000を対象にした実装について述べ，予備評価によりその実用性を示す．またagent間の関係記述は記述時だけでなく，効率の良い実装にも役立つよう意図したものであり，予備評価ではその効果も評価し効率向上を確認した．

A Dataflow-based Massively Parallel Programming Language, *V*, and Its Implementation on a Commercially Available Parallel Machine

SHIGERU KUSAKABE,[†] EIICHI TAKAHASHI,[††] RIN-ICHIRO TANIGUCHI [†] and MAKOTO AMAMIYA [†]

In this paper, we propose a dataflow-based massively parallel programming language, called *V*, which is based on a data-flow oriented functional programming language. The language provides a programming unit, called *agent*, to write parallel entities communicating with each other. In addition to connecting agents explicitly, abstraction of ensembles of agents on a predefined topology description is introduced, in order to write a massively parallel program that naturally reflects the structure of a problem. We also present some implementation issues and a preliminary evaluation of our compiler and runtime system developed for the Fujitsu AP1000, a distributed-memory parallel machine with conventional processors.

1. はじめに

逐次実行をベースにした多くの並列処理記述プログラミング言語が存在するが，手続き的な記述に基づく明示的な並列処理の記述や特定のアーキテクチャに依存する記述は習熟を要し，適切な抽象度をもつ記述を支援しているとはいい難い．我々は，計算機が高い並列処理能力を持つ環境では，並列性を内在しコンパイラや実行システムが自動的に並列性を抽出する並列言語が有望であると考える．

データフロー関数型言語は以下のような，並列処理記述において魅力的な特徴を持つ．

- 本質的に並列性を内在し様々な粒度の並列性を統一的に扱うことができる．
- 計算の進行順序を規定するのは依存関係だけで同期はデータフロー概念により自動化されプログラマが明示的に同期を指定する必要はない．
- 形式的な取扱いが容易である．

上記の特徴により，並列処理記述の容易さと処理系での並列処理の扱いやすさを両立できると考え，我々は超並列V言語をデータフロー関数型言語Valid[2]をベースに設計し，その処理系を開発している[16]．関数型言語は実行効率に問題があるとされていたが，データフローモデルに基づく関数型言語でも，大規模数値計算を指向したデータ並列処理に重点をおいたSisalは非データフロー計算機上でも効率良く実行されており，

† 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
†† (株)富士通研究所
Fujitsu Laboratories Ltd.

適切な実装技術により効率の問題は解決可能といえる[6]. V言語はlenientセマンティクスを持っているため,Sisalのようなストリクトなセマンティクスを持つ言語よりは効率の良い実装が比較的困難であると考えられている. しかし, V言語でもコンパイル技法, 実行時システムの工夫によりデータ並列処理を効率良くサポートすることを一つの目標とする. また, V言語では定型的なデータ並列処理だけでなく, 非定型処理もデータフローモデルに基づいて統一的に扱うことを目標としている.

本稿ではまず, 2章で言語仕様設計と処理系実現の方針について述べる. 3章でV言語の特徴について, 4章でコンパイラと疎結合並列計算機AP 1000への実装の概要について述べ, 5章で予備評価を行う. 6章で関連研究について述べる.

2. 言語および処理系の設計方針

2.1 言語仕様設計方針

データフロー指向の関数型言語は高い抽象度を提供し, 並列処理記述における利点を持っているが, 履歴依存処理の記述は困難で, 操作的な記述や機種に依存した記述はできず, 必ずしも並列処理の記述のすべてを素直に行えるとは限らない. 我々は, Validには並列処理記述に重要な以下の三つの点,

- 並列に動作する処理体の素直な記述,
- 並列に動作する各処理体の論理的構造,
- それらの間の通信形態

が欠けていると考える.

並列処理を念頭に対象問題を素直に記述するためには「対象問題を一状態を持つ並列動作可能な処理単位がメッセージ交換によって計算を進めていく系—ととらえることにより, 問題の自然な記述が可能」[1]と考える. データフロー概念をベースにしたメッセージフローモデルの検討により, 上記概念をデータフロー言語にとり入れることは可能である[3]. さらに処理体間の関係記述も必要と考え, 以下の拡張を行う.

- 入出力ストリーム, カプセル化された履歴依存の状態値を持つ計算体の容易な記述のためagentという抽象化単位を導入. (これらagent間の同期も依存関係に従ったデータフロー同期機構によって行い, 同期の明示を避ける.)
- データの生産者/消費者の関係が明らかな問題の構造を明示的に記述するため, agent間のストリームの連結を記述しagentのネットワークが構成可能.
- 要素間の論理的な構造の指定と簡潔な通信形態記述によりagent集合を容易に扱えるfieldという抽象化単位を導入.

ただし, 特定の計算機上での実行効率の向上を意図した計算機依存の並列展開法や物理的マッピングのための記述は, 抽象度とポータビリティを下げることとなるのでV言語自体には入れない. 効率を意識したマシン依存の並列実行の指定等は別途, annotationで行う方針を取る. これに関しては別の機会で述べることとする.

2.2 処理系実現方針

Vプログラムは細粒度の並列性を含め様々な粒度の並列性を内在しており, lenientセマンティクスを持っている. 既存の商用並列計算機上での実装にあたり, 以下の実行方式をとることにする.

- 仮想マシンレベルでの並列処理単位を, ユーザ定義関数のインスタンス程度の粒度とする. 実際に並列に実行するかどうかは各対象計算機ごとにコンパイラが判断する.
- インスタンス内処理を複数のスレッドに分けそれらを並行実行することでインスタンスレベルでのlenient性を実現する. ここでスレッドとは排他的に実行される命令列で, スレッド自体はストリクト, つまり必要なデータがすべて揃わないと実行は開始されず, 実行開始後は終了まで中断することはない.
- 遅延を含む処理はsplit-phase実行とし, 要求発行後ほかの実行可能なスレッドに切替え遅延を隠蔽する.

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より, 細粒度並列性を持つ, DVMC (Datarol Virtual Machine Code) と呼ぶグラフ表現可能な仮想マシンコードを中間コードとして生成する. 純粋なデータフロー実行方式を対象としたコードでは, 明示的なデータの流れで実行を制御する必要があったが, DVMCが想定する実行方式はDatarol実行モデル[4]に基づいており, 各インスタンスごとにデータを保持する作業用のメモリ領域が割り当てられそのメモリを通して値を受渡しできる. そのため, DVMCは純粋なデータフロー用のコードと比べ, 冗長なデータフローの除去, 共通部分式の除去やコードスケジューリングなどの最適化が容易なコードである[20].

次に, DVMCからの上述の実行方式を考慮したスレッド抽出を行う. 原則として, 遅延を持つ処理はsplit-phase実行とし, 遅延をオーバラップできるようなコードを生成する. DVMCそのままの細粒度並列実行では, 動的なインスタンス管理や同期処理を頻繁に行う

必要があり，既存の商用並列計算機上の実装ではそのオーバヘッドは深刻なものになる可能性がある．抽出すべき粒度は対象計算機によって異なるため，スケジューリングにおいては，マシンに依存する細粒度並列処理のランタイムコストを意識したスケジューリングにより，効率向上を目指す．

また，agentの実現においては，明示された論理的な関係を活用することにより，通信のコスト，物理的配置を考慮し処理系が効率良く実行できるようにする．

3. V言語の特徴

プログラムは基本的に関数から構成されるが，柔軟な通信能力を持ち局所状態をカプセル化できる並行プロセスを容易に記述するためにagentという抽象化単位を導入する．V言語はlenientセマンティクスを持つ言語で，strictな言語が持つ制約：

1. 関数適用の際には，すべての引数が評価済みであること
2. ある構造体が構成される前には，すべての要素が確定していること

を持たない[8]．lazyな言語も，lenientな言語と同様にnon-strictな言語に分類されるが，lazyな言語は“最終結果に寄与しない計算は行われない”という制約を持ち，要求駆動処理で実現されることが多い．一方lenientな言語は“条件分岐時に条件の判定が完了した後，分岐先の実行を行う”という制約だけを持ち，データ駆動的に計算が進み，並列性の抽出が容易である．V言語プログラムの実行順序はデータ依存関係のみによって制約され，agent間だけでなくagent内でも並列性を活用可能である．同期処理はデータフロー方式に基づき依存関係に従って自動的に行われる．

V言語はベースであるValid同様に，数値処理だけでなく記号処理も指向した言語で，再帰的定義を含む関数型言語が基本になっており，原則として自由変数を認めない変数の静的束縛規則を持つ静的に型づけされた言語である．しかしながら，lenientセマンティクスを持ち高並列の処理を指向しており，Validが持っていた高階関数や遅延評価の機構は持っていない．以下に言語の特徴の概要を述べるが，ベースとしたValidと共通の特徴は記述せず，V言語の特徴を中心に述べる．Validの仕様については文献2)を参考にされたい．プログラム・テキストの構成要素は〈式〉，〈変数〉のように角括弧を使った構文変数により記す．カギカッコ[ ]で括られたものは省略可能を表す．

3.1 agent

V言語では，引数を与えてagent定義から入力と出力を持つプロセスをつくり出すことができる．以下が識別子〈agent定義名〉を持つagentの定義である．

agent 〈agent定義名〉 (〈生成時仮引数部〉)
〈ストリームインタフェース部〉
＝〈本体式〉；

〈ストリームインタフェース部〉は

{channel 〈ストリーム引数の並び〉}
[{export 〈ストリーム引数の並び〉}]

のように入力ストリームの変数を指定するchannel変数宣言部と，出力ストリームの変数を指定するexport変数宣言部からなる．channel変数宣言部，export変数宣言部で指定するストリーム用変数の並びの順番は意味を持つ．データを送信したりストリーム同士を結合する際の対象ストリームを指定する時に，疑似変数 ch[k](ex[k]) (k は整定数)を用いることができる．これは，agentが持つ複数の入力(出力)ストリームのうち，宣言部での左から k 番目のストリームを指定する．また，ストリームは要素の型指定がなされ，一種類の型のデータのみを流すことができる．

agentは生成されると〈本体式〉を実行するインスタンスとなり，インスタンス内の実行環境や状態値の初期化は，〈生成時仮引数部〉を通して渡された値を使って行うことができる．〈本体式〉は文法的には任意の式が可能だが，そのagentインスタンスを履歴依存のインスタンスとして記述するには，ストリームの再帰的処理，履歴依存状態値の保持を行う再帰構造を持たせる．

以下がagentインスタンスの生成式で，以降〈変数〉により生成されたインスタンスにアクセス可能である．

〈変数〉＝create(〈agent定義名〉[, 式の並び])

3.2 入力ストリーム操作

入力ストリームに対する基本演算は，head，tail，emptyがある．headは先頭を返し，tailは残りを返す．ストリームはlenientなリストで，tail部が確定していなくてもhead部は参照できる．空channel変数要素への参照はデータ書込みまでサスペンドし，書込まれると待ちが解かれ処理が続けられる．emptyはchannel変数が空なら真，そうでなければ偽を返す．

3.3 通　　信

インスタンス間通信は，メッセージパッシングに相当するものと，問題中の通信構造を反映しインスタンス間でストリームを明示的に結合し通信するものとの

2通りの通信法を提供している．

メッセージパッシング 以下のようにsend式を用いて指定されたagentインスタンスのk番目のchannel変数が表すストリームにデータを送ることができる．(ch[k]@省略時は1番目のchannel変数が選択される)．ひとつのインスタンスから同じストリームへ複数のメッセージを送る場合，受信側で送信順は保存される．

〈変数〉＝send([ch[k]@]
〈agentインスタンス名〉,〈式〉)

上記のような場合，処理結果の返し先を〈変数〉とするメッセージが，指定された〈agentインスタンス名〉のストリームに送られる．受信側でそのメッセージに対する処理を行った後，結果値を返すにはreplyを用いる．メッセージの送信側では，〈変数〉に依存する処理は，メッセージの受け手からの返り値が〈変数〉に束縛されるまで処理が中断する．

結合stream通信 V言語ではagentインスタンス間でデータの生産者/消費者の関係が明らかな場合，出力ストリーム（export変数）を他のインスタンスの入力ストリーム(channel変数)に結合し問題構造を反映したagent間の通信トポロジを明示できる．

link(〈export変数@インスタンス名〉,
〈channel変数@インスタンス名の並び〉)

このように結合されたストリームに対してはput(〈式〉,〈export変数〉)でexport変数側に〈式〉の値を出力することにより結合先のchannel変数に値が渡される．

また，formではインスタンスの生成と結合様体の指定を同時に行うことができる．

form{〈結合インスタンス生成式を含む
局所定義の並び〉 in 〈式〉}

ここで〈結合インスタンス生成式〉は，

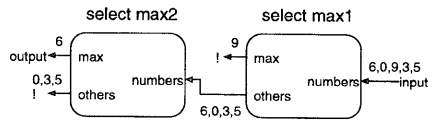
〈create式〉〈channel引数指定部〉〈export宣言部〉

で，〈channel引数指定部〉および〈export宣言部〉の指定は〈create〉式の引数となっているagentのストリームインタフェース部と個数，型とも一致している必要がある．〈export宣言部〉で宣言される識別子はform内で有効であり，その識別子を参照することにより生成するagentの出力ストリームを参照することができる．〈channel引数指定部〉では生成するagentインスタンスのchannel引数に与える入力ストリームを指定する．〈channel引数指定部〉にはform式内およびその外側のスコープで定義されたストリームとリスト，〈export宣言部〉で宣言された識別子が指定できる．inのあとの〈式〉の結果がformの結果となる．**図1**にformを用いた例題を示す．この例は，入力に整数ストリームをとり，最大値とそれ以外の整数のストリームを返すselectmaxのインスタンスを結合し整数ストリームの要素中2番目に大きな値を返すものである．selectmaxの〈本体式〉は再帰のためのfor式となっており，

for 〈再帰変数の並び〉
init 〈再帰変数の初期値の並び〉
body 〈再帰本体〉

という構文をしておりValid[2]と同様のものである．〈再帰変数の並び〉の各変数は〈再帰本体〉の各再帰インスタンスへの入力値を示す変数で，初期値が〈再帰変数の初期値の並び〉で指定され〈再帰本体〉内で局所的に使われる．〈再帰本体〉は通常再帰の停止条件を調べる条件式と再帰のためのrecur式を含む．

max1は標準入力std_inから入力ストリームを受けとり二番めのexport変数を通し入力ストリームから最大値を取り除いたものをstrm1とするselectmaxインスタンスである．max2はmax1が生成する




```
program secondmax
 agent selectmax()
  {channel numbers:integer}
  {export max,others:integer}
 = for (tmp:integer;nums:stream) init
       (head(numbers),tail(numbers))
    body
       if nums=nil
        then put(max,cons(tmp,nil))
              after put(others,nil)
        elsif tmp<head(nums)
          then recur(head(nums),tail(nums))
                after put(others,tmp)
        else recur(tmp,tail(nums))
              after put(others,head(nums));

={form
   max1=create(selectmax){std_in}{!,strm1},
   max2=create(selectmax){strm1},{result,!}
  in result};
```

図1 整数ストリームの要素中2番目に大きな値を返す例
Fig. 1 An example of selecting the second maximum value from an integer stream.

strm1を受けとり，最大値をresultとして返すselectmaxインスタンスである．"!"は返される値を捨てることを表す．

3.4 集合体

V言語では同一定義を持つagentインスタンスの集合体であるfieldインスタンスの生成ができる．field定義は

field 〈field定義名〉(〈agent定義引数〉)
(〈生成時仮引数部〉)
[ストリームインタフェース部]
[on 〈構造テンプレート〉] [do 〈設定式〉]
=〈本体式〉

によってなされ，fieldインスタンスの生成は

〈変数〉=organize(〈field定義名〉,
〈agent定義名〉[, 式の並び])

によってなされる．我々は並列処理の効果が最も得られるのはagentインスタンスの集合を規則的な構造に構成できる場合と考え，構造テンプレートと設定式によりagentインスタンスの集合が規則性を持った論理的構造と通信形態とを持つ場合には簡潔に記述できるようにする．構造テンプレートを指定しなければ，論理的な構造を持たない単なるインデクスが付けられたagentインスタンスの集まりとして扱われる．ここでfieldインスタンスもagentインスタンス同様，入出力ストリームを持ち，〈ストリームインタフェース部〉でagentと同様の記法で記述される，〈設定式〉では，各要素インスタンスの初期設定やストリームの結合を記述する．構造テンプレートと設定式で明示可能な規則性は，記述時だけでなく実装時にも利用される．

例として，**図2**のように構造テンプレートで指定されたtorusの構造に配置され，4近傍と通信するagentであるcomm4のインスタンスから構成される集合体Fをとりあげる．comm4は，四つの入力実数ストリームと出力実数ストリームを持ち，自身の値と入力ストリームからの値を用いて別途定義された関数funcで自身の値を更新し，更新後の値を出力ストリームに渡すことを指定回数だけ繰り返す．Fおよびcomm4の定義は以下の通りで，

```
agent comm4(val:real)
{channel in1,in2,in3,in4: real}
{export out: real}
=for (v:real;count:integer;
      i1,i2,i3,i4:stream of real)
   init (val,0,in1,in2,in3,in4) body
   if count>=M then v
   else
   { slet put(v,out),
         next=func(head(i1),head(i2),
                   head(i3),head(i4))
     in recur(next,count+1,tail(i1),tail(i2),
                           tail(i3),tail(i4))};
field F(com4)(vals:array of real; n:integer)
 on torus do connect4(n,n,self) ;
= foreach (i,j:integer) in ([1..n],[1..n])
    body create(com4,vals[i][j]) ;
```

fieldFのインスタンスは

Fi=organize(F, comm4, An_array, size)

により生成される．ここでAn_array, sizeはそれぞれ別途定義された実数の配列，整定数である．

Fの定義の設定式におけるconnect4は，4近傍通信を明示するためのものである．torus構造での4近傍をup, down, right, leftで指定した場合，以下のように表される結合を簡潔に指定できる．この通信パターン明示は実装時のマッピングの際に利用することで効率の良い実行を可能とすることを意図している．

```
function connect4(m,n:integer;f:field on torus)
 foreach (i,j:integer) in ([1..m],[1..n])
   body link(ex[1]@f[i,j],ch[1]@up,ch[2]@right,
                          ch[3]@down,ch[4]@left) ;
```

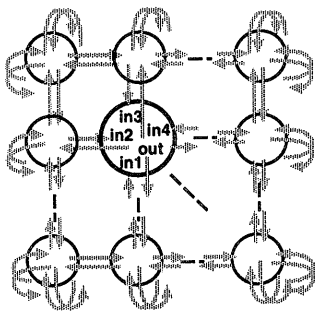


図2 構造テンプレートtorus上で4近傍通信をするagentの集合体

Fig. 2 An agent-field whose agent instances communicate with 4 neighbors on torus template.

4. 処理系

4.1 コンパイラ

図3に現在作成中のコンパイラの概要を示す．コンパイラは大きく分けて，機種依存/非依存処理のフェーズに切り分けられる．現在コード生成の簡単化/可搬性のため，コンパイラの機種依存部の最終フェーズではターゲットシステム用Cコードを生成しており，低レベルの最適化は各システムのCコンパイラに依存している．以下に，各部の概要について述べる．

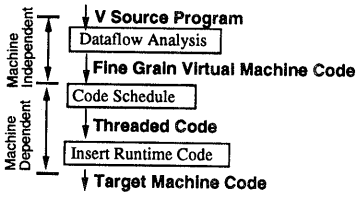


図3 コンパイラの概要
Fig. 3 Overview of our compiler.

4.1.1 機種非依存処理

コンパイラの機種非依存処理部では，Vプログラムから，字句解析，構文解析を行った後，データ依存解析を行い，各対象マシンのコードを生成するための中間コードとして細粒度並列性を持つDVMC[16]を生成する．V言語はデータフロー実行モデルを持ち，Datarol-II[12]のような細粒度並列マシンのコードはDVMCから比較的容易に生成できる．我々は，既存の並列計算機用を対象にする場合にも，逐次的なコードから並列性を取り出すより，本質的に並列なコードをスケジューリングする方が良いと考え，DVMCを中間言語として，Symmetry[17]，AP 1000[10]などへのコード生成を行っている．

ここでは以降の説明に必要なDVMCの特徴を述べる．DVMCはグラフ表現可能なコードで，グラフの各ノードは各スレッドを表し，ノード間のアークはスレッド間の制御やデータの流れを表す．インスタンス間のデータの受渡しは明示的であるが，インスタンス内のスレッド間のデータの受渡しは，各インスタンスごとに割り当てられるデータ保持用のメモリ領域を通して行われる．DVMCは，遠隔メモリ参照やインスタンスの生成，インスタンス間通信など，遅延を伴う操作はsplit-phase実行としている．スレッドが発火した場合，スレッドを構成するコード列はサスペンドすることなく排他的に実行される．各コードは演算やデータの授受を行い型付のオペランドを持つ．

4.1.2 機種依存処理

ここではAP 1000を対象とした機種依存処理部の概要を述べる．AP 1000は細粒度並列処理のための特別なハードウェアはもっておらず，V言語を実装するための細粒度並列処理をソフトウェアで実現する必要があるため，コード量，オーバヘッドともに増大してしまう．また，AP 1000の要素プロセッサは時間的/空間的局所性を利用する従来型の高性能プロセッサであるため，細粒度並列処理のような局所性の少ない計算は得意でない．そのため，以下に述べるようなコード生成・最適化方針をとる．

前処理 スレッドを構成する各演算コードについて対象マシンでのアセンブラコードに置き換え，各スレッド内の計算コストを算出する．またAP 1000のような疎結合計算機では，他のプロセッサとのデータやコントロールの授受はメッセージ通信で行うため，インスタンス生成，データ授受，リモート変数アクセス操作においては通信コストの見積りを行う．通信コストには通信機構のハードウェアの遅延時間に加えメッセージ構築などに必要なソフトウェアによるコストも考慮する．AP 1000の通信時間は文献19)に記載されている値を用いた．キャッシュのヒット/ミスヒットは静的には予測できないので，コスト算出時にはキャッシュはヒットするものと仮定している．キャッシュ評価ツールを用いた予備実験で，かなり高いキャッシュヒット率が出ているのを確認し，そのような仮定を立てている．

STEP 1 仮想マシンレベルでは，原則として各引数データは個別に送るという方式をとっている．しかし実際のコードでは，あるインスタンスの同一スレッドから同一インスタンスへ複数のデータを送る場合は，以下の解析を行い，可能ならば複数のデータをまとめて送るようにする．ここで考慮されるのは，インスタンス内に閉じた依存解析で，引数データを生成する部分と引数データを送る部分を同一スレッド内におくことが可能な，同一インスタンスへの複数の通信である．同一インスタンスへの複数の通信でもこれ以外は：lenientセマンティクスの実現のために個別に行う必要があり得る，または，個別に実行することによりcallerとcallee間での計算のオーバラップが期待できる可能性がある，のどちらかだと判断し個別に通信するコードのままとする．n個のデータ$D_1 \cdots D_n$を送る場合，あるデータD_iを送るための，メッセージ構築，ネットワークへの送信，ネットワークからの受信，データ取り出しの見積りコストの合計を$C_{com}(D_i)$とし，以下の条件が成り立てばメッセージデータを結合して通信が一度で済むようなコードを生成する．

$$(C_{com}(D_1)+\cdots+C_{com}(D_n))+(C_{fr}+C_{th}+C_{sy})*n) > (C_{com}(D_1|\cdots|D_n))+C_{fr}+C_{th}+C_{sy}*i)$$

ここで$D_1|\cdots|D_n$はn個のデータを結合したデータを表し，C_{fr}, C_{th}, C_{sy}はそれぞれ，フレーム切替え，スレッド切替え，同期のためのコストである．iはコード変換後の対応する部分での同期操作数である．実行環境については後述するが，あるインスタンスに引数データを送るごとに，対象スレッドが属するインスタンスフレームへの実行環境切替え，データ

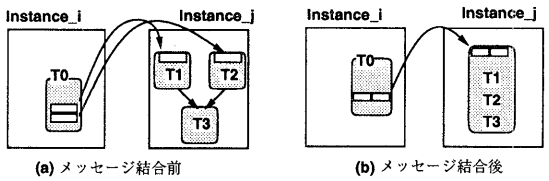


図4 メッセージ結合
Fig. 4 Message combining.

を受け取るスレッドへの切替えなどが必要となる．それらの細粒度並列処理のための操作はソフトウェアで実現されており，ここでのコストの見積りもそれらをアセンブラに換算して求めている．

図4に単純な例を示す．図4では，インスタンスInstance_iのT_0からインスタンスInstance_jの二つのスレッドT_1，T_2へ，二つのメッセージが送られていたのが(a)，一つにまとめてInstance_jへ送るように変換される(b)．それにともないInstance_j内のスレッドT_1, T_2, T_3が一つにまとめられ，起こり得るインスタンスやスレッドの切替数が減り，T_1およびT_2とT_3との間の同期処理も不要になっている．

STEP 2 インスタンス生成の際，通信コストとローカル/リモートの計算のオーバラップのトレードオフを評価し，リモートに生成するのかどうかの判断を行う．この解析のため，まずインスタンスの呼出し関係の木を求め，葉の方から各インスタンスの実行コストを見積もる．静的に判断できないものに対してはヒューリスティックを用いる．再帰や繰り返しの回数は定数とし，条件分岐においては最も高いコストを持つインスタンス呼出しを選ぶ．コストの見積り後，呼出し木の葉の方から再帰的に以下の手続きを行う．あるcalleeインスタンスをcaller側ローカルに実行した場合のcallerの実行時間 C_L とそうでない場合の実行時間 C_R を求め，$C_L < C_R$ と判定される時は，そのcalleeインスタンスはcaller側でローカルに実行するものとする．caller側で複数のcalleeインスタンスをローカルに実行可能な場合，最も早くcaller側で結果値が必要とされるものを選ぶ．

STEP 3 後述するようにインスタンスごとにフレームという作業領域を割り付ける実行方針をとるが，フレームは比較的管理コストが高い．スタックを用いてもデッドロックにつながるサスペンドをする可能性がないインスタンス呼び出しに対しては，フレームを用いずスタックを用いる．その解析のため，文献21)に提案されているスケジューリング法を応用した手法でコードをトレースしながら，以下のような条件を調べる：(条件A) そのインスタンス呼出しのための通信を単一のスレッドから送信可能，(条件B)呼出先のインスタンス内のスレッドを一本のスレッドにスケジューリング可能．呼出先のインスタンスに，さらにインスタンス呼出しがある場合には，再帰的に解析を行う．あるインスタンス内に複数のインスタンス呼出しがある場合は，すべてが条件A，条件Bともに満たし，それらの呼出しの間に順序づけが可能でなければならない．また，プロセッサ間通信により起動されるスレッドを含んでいる場合には，スタックは使用できないとする．

理想的な場合，lenient実行方式は並列性を最大限に抽出することにより性能向上に寄与するが，実際の並列計算機上ではかなりのオーバヘッドが生じ，lenient実行方式に固執することは得策でないことが多い．上記**STEP 1**では，部分計算の効果が期待できない先行実行回数を削減することにより通信回数や同期処理回数，コンテクスト切替数を削減しオーバヘッドを減少させる．次に**STEP 2**では，効果の期待できない遠隔プロセス生成を削減し，プロセッサ間通信のオーバヘッドを減少させる．**STEP 3**では管理コストの高いフレームの使用頻度を削減することで効率向上を図る．本方式ではlenient実行方式を前提としたコードから始め，有用なlenient実行部分だけを残し，他は対象計算機が得意とする方式で実行することになる．

4.2 並列実行環境

以下，AP 1000上の並列実行方式について述べる．関数適用やagent生成により動的に生成されるインスタンス程度の粒度での並列実行を基本とする．

4.2.1 AP1000

今回実装の対象とするAP 1000は，25 MHzのクロック周波数，16 MBのメモリを持つSparc 64台を要素プロセッサとして持つ．各ノードのメッセージコントローラMSCのバッファレシーブの機能により到着したノード間メッセージをCPUに割り込みをかけずにリングバッファに受けとることができる．

4.2.2 フレーム

並列に実行されるインスタンスごとにフレームというデータ構造が作業領域として動的に生成され，インスタンス実行終了まで存在する．**図5**に示すように，フレームは局所変数保持のためのフレーム変数スロット，インスタンス内部のスレッド実行制御に用いる継続スレッドスタックとその先頭を指すポインタCSPよりなる．現実装では，フレームサイズはコンパイル時に決定し，フレームはある程度まとめて生成しフリ

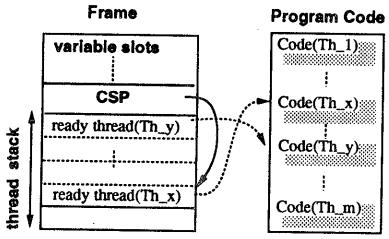


図5 フレームの構造
Fig. 5 Overview of a frame.

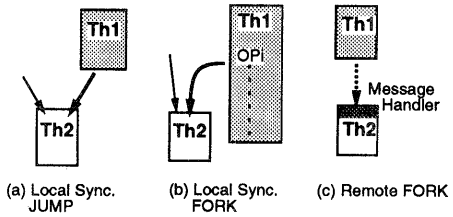


図6 スレッド間実行制御
Fig. 6 Execution controls between threads.

ーリストで管理している．

4.2.3 スレッド実行制御

インスタンス内には一つのフレームを共有する複数のスレッドが存在し，各スレッドは同期変数とフレームの継続スレッドスタックによって動的にスケジューリングされる．AP 1000 はスレッドレベルの並列処理をサポートしていないため，コンパイラは図6に示すパタンに応じ，スレッド間の実行制御のためのコードを挿入する．図6(a)のようにスレッドの最後から他の局所スレッド起動を試みる場合に Local Synchronized JUMP 命令（以下 LSJ 命令と呼ぶ）を挿入する．LSJ 命令はバリア同期つきジャンプ命令であり，図6(a)の場合，Th 2 を継続とする他のスレッドが終了している時のみ，Th 1 から Th 2 へ直ちに制御が移る．図6(b)のようにあるスレッドの途中から，他のスレッド Th 2 へアークが存在する場合は，バリア同期を行い同期成立時に対応するスレッドを実行可能にする Local Synchronized FORK 命令（以下 LSF 命令と呼ぶ）を挿入する．図6(c)は，split-phase 操作の場合で，リモートのスレッドからの返答メッセージにより Th 2 が起動される．コンパイラによって，Th 2 の先頭に返答メッセージを適切に解釈する Message Handler コードが挿入される．

複数スレッドの継続スレッドであるスレッドの起動では，同期成立チェックを行う．同期変数は，同期ポイントごとにフレーム変数スロットを割り当てて実現する．フレーム取得時に同期操作を行うスレッドの数

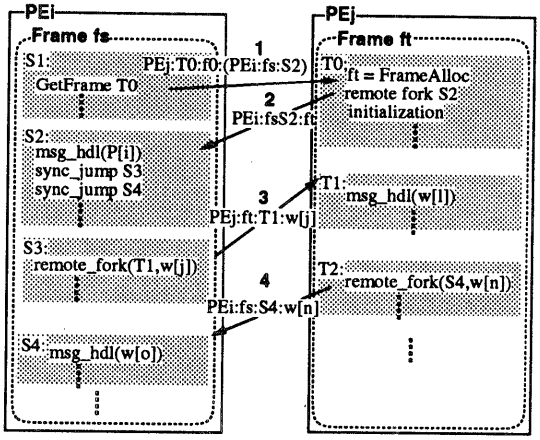


図7 インスタンス生成とデータの授受
Fig. 7 Creation of an instance and parameter passing between instances.

へ初期化しておき，LSJ 命令，LSF 命令によって同期変数を減じていく．もし，同期変数を 0 にしたのが LSJ 命令であれば，直ちに継続スレッドへ制御を移す．同期変数を 0 にしたのが LSF 命令であれば，継続スレッド ID を現フレームの継続スレッドスタックへプッシュする．プロセッサは，フレームに対し，継続スレッドスタックが空になるまで，

1. スレッド ID をポップ．
2. スレッド ID が指すスレッドへ制御を移す．

を繰り返す．スレッド実行中は現フレームは他からブロックされる．

4.2.4 インスタンス生成

図7にインスタンス生成とデータ授受の概略を示す．

1. プロセッサ PEi から PEj へ新規フレーム取得要求メッセージを送信する．メッセージには，caller 側のセル ID，フレームポインタ，スレッド ID を継続点としてセットしておく．
2. PEj では，新規フレーム ft を取得し，データとしての取得フレーム ID を含み，caller 側の継続点を起動するメッセージを PEi に返す．
3. 引数データをプロセッサ PEj，フレーム ft，スレッド T 1 で特定される callee 側のスレッドにデータを渡し処理を起動する．
4. 返値渡しの場合も，caller 側の継続点にデータを渡し，返値に依存するスレッドを起動する．

5. 評　　価

5.1 細粒度並列処理

まず，細粒度並列処理の評価を示す．ここでは，4.1.2

項で述べたような細粒度並列処理のオーバヘッド削減のためのコード生成戦略の効果を段階的に評価する．例としてナイーブな解法でのN-queen全解探索の例題にそのコード生成戦略を適用した場合の台数効果改善の結果を図8に示す．ここでの台数効果の基準はV言語プログラムと同一アルゴリズムを用いたC言語プログラムを1セルで実行したものを用いており，N=8で実行時間は0.726秒である．

- **Opt0**： DVMCそのままに近い形態での実行．インスタンス呼出しなどは完全にlenientに実行する．我々の実行方式でも，TAM（Threaded Abstract Machine）同様，実行時のフレーム内でのスレッドの動的グループ化[18]により効率の良い細粒度並列処理が実現できると期待していた．しかしながら細粒度並列処理のオーバヘッドが大きくほとんど台数効果が出ていない．
- **Opt1**： 4.1.2項で述べた**STEP 1**によりインスタンス間通信を削減するスケジューリングを行ったコードの実行結果．この場合，lenient性は減少するが，通信頻度の削減により通信操作自体に加え，同期処理，コンテクストの切替えの回数も減り**Opt0**に比べ効率が向上している．
- **Opt2**： さらに**STEP 2**を適用し，ローカルなインスタンス呼出しを積極的に用いることでノードメッセージ通信回数が減少し，効率が向上している．**Opt1**と**Opt2**を比べるとかなり効率が向上している．これは，**Opt1**では通信のオーバヘッドが正味の計算より大きく効果が期待できない場合にもリモートにインスタンスを生成していたのを，**Opt2**では効果が期待できるものだけに限定したためである．
- **Opt3**： さらに**STEP 3**を適用し，可能な場合にはフレームを用いたインスタンス呼出しでなく，スタックを用いた呼出しを行うことで，管理コストの高いframeの使用頻度が減少し効率が向上している．

データフロー計算モデルに直接基づくような細粒度の遠隔呼出しプロトコルで，通信と計算のオーバラップを試みるのは，コンテクストの切替えや通信のコストが低いマシンの場合は有効である[24]．しかし，AP 1000のように細粒度並列処理を特別にサポートしていない計算機の場合には現実的ではなく，効率向上のためにはランタイムのコストを考慮した前述のようなコード生成が有効である．

5.2 データ並列処理

データ並列処理の評価として，行列積の問題に対する台数効果を図9に示す．ここでもCプログラムをプロセッサ台数1で実行した場合の時間（512×512の実数行列に対し約559秒）を基準にしている．文献14）では行列積の結果に関してはプロセッサ台数16までの速度向上比しか記載されておらず，実行時間や問題サイズも明記していないため正確な比較はできないが，AP 1000上の他の言語としてVPP Fortranと比較する．VPP FortranのSPREAD MOVE構文を用いた例では約96%の台数効果が得られている．我々の場合もプロセッサ台数16で約97%の台数効果が得られており，遜色のない性能である．細粒度並列性を持つ言語のデータ並列処理でもAP 1000のような商用並列計算機上に効率良く実装可能といえる．

5.3 agentとfieldの処理

まず，1整数データを含むメッセージで通信相手の処理を起動する場合のコストを，メッセージパッシング通信を用いる場合と，結合ストリーム通信を用いる場合とで比較する．送信側でメッセージを組み立てて送信，受信側でストリームにバッファリング，ストリームからメッセージを取り出してスレッドを起動する

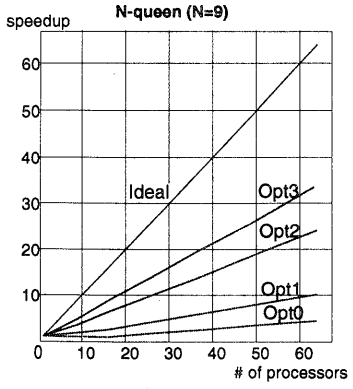


図8 N-queen全解探索問題での速度向上比の改善
Fig. 8 Speed-up improvement in N-queen exhaustive program.

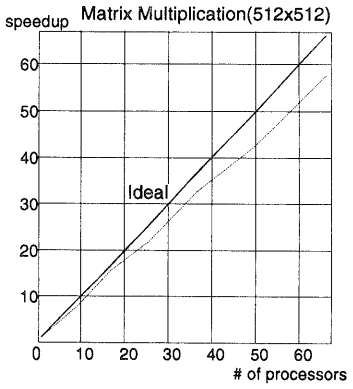


図9 行列積（512×512）の台数効果
Fig. 9 Speed-up in matrix multiplication program (512×512).

までの命令数は，メッセージパッシング方式に比べ結合ストリーム通信だと約84%に削減することができた．これは，send式を用いた通信に必要である送信側での継続点付加，受信側での継続点取り出しのための処理が，結合ストリーム通信の場合不要なためである．

また，結合ストリーム通信の場合，結合情報を利用することにより論理的に関係を持つインスタンス同士を，物理的に近い位置にマッピングしやすいという利点を持つ．さらに，場合によっては，メッセージをバッファリングせずGHCのメッセージ指向実装法[22]のような効率の良い実装を行うことも可能と考えられる．

次に，3章の図2の例題を選び，記述レベルで明示された問題の論理構造情報を処理系が活用した場合，効率の良い実行ができるか否かに焦点をおいて評価した．論理的な配置の規則性，および通信の規則性を処理系が活用できた場合とそうでない場合を比較するため，averageインスタンスを，AP 1000の64セルにランダムに配置(random)，1次元ブロックのインスタンスをグループ化して配置(line)，可能な限り2次元正方ブロックのインスタンスをグループにして配置した場合(box)の3通りについて評価した結果を**図10**に示す．

実行時間はrandom＞line＞boxで，その実行時間の差はインスタンス数の増大とともに大きくなっている．これより，問題中の論理構造を利用して4近傍通信に適したようにインスタンスを配置した方が効率の良い実行ができていることがわかる．

上記のように問題が持つ論理的な構造と物理的な計算機の構造が一致する場合は実行時の効率向上も得やすいが，両者が一致しない場合については，処理系で自動的に最適なマッピングを行うことは容易ではない．現在ランダムマッピングと実行時間を比較する予備評価を行っているが，必ずしも実行時間が短縮されるわけではない．しかしながら記述時に論理的なagent間の結合トポロジーを明示し，多数のagentから構成される問題を抽象化することは有用と考えており，実装時の効率の良いマッピング法についてはannotationによる処理系支援を検討している．

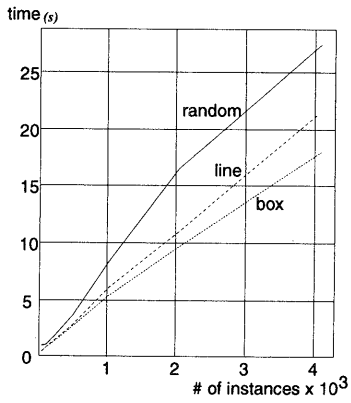


図10 4近傍通信を行うfieldでのインスタンス配置の影響
Fig. 10 Effect of instance mapping of a field whose agent instances communicate with four neighbors on a torus template.

6. 関連研究

Vのagent間メッセージパッシングは非同期でデータフロー同期機構により依存関係にしたがって自動的に同期がとられるため，手続き的な言語に見られるような過去型，未来型，現在型などの区別[23]は必要ない．V言語はオブジェクトベース言語ともいえるが，現仕様では継承など文献15)で論じられているようなオブジェクト指向の機能は持っていない．しかしその重要性は認識しており，導入については検討中である．

fieldに類似した概念を持つ言語にpC＋＋やOCore[13]がある．pC＋＋[9]はC＋＋にcollectionというクラスを導入し，collectionの中のオブジェクトに対する並列操作が可能であるが，データ並列的な処理を指向しておりコントロール並列処理は支援していない．実プロセッサへのマッピングにはHPF的なtemplate，alignmentの指定法を用いている．OCoreはオブジェクトの集合の構造化，およびオブジェクトにはないメッセージ処理の並列性の提供を目的として，共同体を導入している．オブジェクトによるコントロール並列処理と，共同体によるデータ並列処理を実現する．OCoreでは共同体におけるリダクションやバリア同期などの大域的操作を積極的に支援している．一方，V言語でもデータ並列処理は記述できるがバリア同期操作の直接的支援はしていない．これはV言語がデータの流れと同期を同一視し暗黙の並列性を前提としているためである．実プロセッサへのマッピングに関して，OCoreではデフォルト以外のマッピングはメタレベルで行うが，我々はannotationで行うことを検討している．

ストリームを利用した通信を行う点に関して，$\mathcal{A}'\mathcal{UM}$[25]と類似している．$\mathcal{A}'\mathcal{UM}$はストリームを用いて対象問題をとらえ，並列性を最大限に抽出する．ストリームに対するmerge joint，append jointなどV言語では直接支援していない操作を持つ．またストリームを通したストリームの委譲など，ストリームインタフェースが定義時に決まるV言語のagentでは実

現できないストリーム操作を支援している．一方，V言語は関数呼出しやメッセージ交換なども備えており，単一パラダイムの$\mathcal{A'UM}$とアプローチが異なる．V言語は関数型言語がベースのlenientセマンティクスを持つ言語でデータフロー原理に基づいて並列性を最大限に抽出する．

結合されたストリーム通信を最適化し，メッセージをバッファリングすることなく，実行する方式は，GHCのメッセージ指向並列実行方式[22]と類似しているが，GHCのメッセージ指向実行方式はレスポンスを重視している．GHCのメッセージ指向実行方式では共有メモリ上でゴールを共有する実現方式をとっており，大規模・高並列処理には適していない．V言語の密結合マシン上への実現ではタスクプールを分割しアクセス競合を避ける手法[21]を用いることが可能である．

Vはagent内にも細粒度並列性を持ち，オブジェクト内並列性を持たないもの[24]と比較すると，細粒度並列実行を低オーバヘッドで実現できる超並列計算機上では，スレッドの多重実行により優れた耐遅延性を発揮し高い性能を期待できると思われる．既存並列計算機上の実装において，ランタイムコストを意識し効率を上げようとしているのはConcert[11]と同様であるが，我々が対象としているのはデータフローモデルに基づく言語であり，さらに粒度の小さな並列性を対象にしている．我々の実行方式はTAM[18]同様の背景を持つが我々の中間コードはデータの流れとコントロールの流れを統一したDatarol[5]に基づいた仮想マシンのコードである．実際のコードは文献20)で述べられているDatarolコードを基礎にしており，データは原則としてインスタンスフレーム（作業用メモリ）を用いて受け渡されるという方針（by-referenceメカニズム）のもとにコード生成がなされる．そのため文献20)中でも評価されているように冗長なノードやアークは取り除かれている．一方，文献18)ではTAMへのコンパイルのため，データと制御の流れを明示的に分けたDualGraphという中間表現を用いており，データの流れが明示的でないTAMの実行方式との乖離が見られる．また，“条件分岐式では条件判定が確定するまで分岐先の計算は先走らない”というlenient言語のセマンティクスの制約ため不要であるはずの，冗長なswitchノードやmergeノードが生成されている．我々の中間コードではそのような冗長なノードは生成されず適切な抽象度を持つと考える．

7. おわりに

データフロー向き関数型言語をベースに記述側からも処理側からも扱いやすい超並列記述言語を目指した超並列V言語の主な特徴，処理系の概要と予備評価について述べた．

agentという抽象化単位や，agent間の関係を反映した通信記述，要素間の論理的構造/通信形態を指定してagentの集合を取扱う抽象化単位fieldを言語仕様に導入した．

Vプログラムは細粒度の並列性を内在するが，現処理系はマルチスレッド実行方式を基本としている．関数適用インスタンス程度の粒度で並列に実行し，インスタンス内は複数のスレッドに分割しそれらを並行に実行することでlenientな実行を実現し，遅延の隠蔽を試みている．AP 1000のように細粒度並列処理のための特別なハードウェアを持たない計算機上のでも，適切なコード生成戦略により細粒度並列を内在する言語を高い実行効率で実装できることがわかった．

また，agentの実現では，処理系が記述レベルで反映された問題の論理構造の情報を活かしたマッピングを行い，効率の良い実行ができた．

今後は，基本操作を中心に改良を加えさらに低コストのランタイム処理系を実現する．また，コード生成を洗練化しさらなる効率向上を目指す．

謝辞 日頃ご討論頂く雨宮研究室の諸氏，とりわけ山下欣宏，永井拓の両氏の多大な協力に感謝します．また，並列計算機AP 1000の実行環境を提供頂きました富士通研究所(株)に感謝の意を表します．

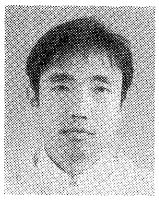
本研究の一部は文部省科学研究費補助金（重点領域研究(1)課題番号04235104「超並列記述系・処理系に関する研究」）による．

参考文献

1) Agha, G. A.: *Actors: A Model of Concurrent Computation in Distributed Systems*, the MIT Press (1986).
2) 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. J 71-D, No. 8, pp. 1532-1539 (1988).
3) 雨宮真人，長谷川隆三：並列協調システムにおけるメッセージ処理機構，日本ソフトウェア科学会第4回大会論文集，pp. 315-318 (1987).
4) 雨宮真人：超多重並列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).
5) Amamiya, M. and Taniguchi, R.: Datarol: A Massively Parallel Architecture for Functional

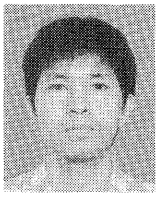
Language, *Proc. 2nd IEEE Symp. Parallel and Distributed Processing*, pp. 726-735 (1990).
6) Cann, D. C.: Retire Fortran? A Debate Rekindled, *Comm. ACM*, Vol. 35, No. 8, pp. 81-89 (1992).
7) Chien, A. A.: *Concurrent Aggregates*, the MIT Press (1993).
8) Ekanadham, K.: "A Perspective on ID" in Chapter 6 of Frontier Series, *Parallel Functional Languages and Compilers*, Szymanski, B. K. (ed.), ACM Press (1991).
9) Ganno, D. and Lee, J. K.: Object Oriented Parallelism: pC++ Ideas and Experiments, *Proc. 1991 Joint Symp. Parallel Processing*, pp. 13-23 (1991).
10) Ishihata, H., Horie, T., Inano, S., Shimizu, T. and Kato, S.: An Architecture of Highly Parallel Computer AP 1000, *IEEE Pacific Rim Conf. Communication, Computers and Signal Processing*, pp. 13-16 (1991).
11) Karamcheti, V. and Chien, A.: Concert—Efficient Runtime Support for Concurrent Object-Oriented Programming Language on Stock Hardware, *Proc. Supercomputing'93*, pp. 598-607 (1993).
12) 川野哲生, 日下部茂, 谷口倫一郎, 雨宮真人：細粒度スレッド処理のためのコンテクストスイッチ機構—並列計算機 Datarol-II の階層メモリシステム—, 並列処理シンポジウム JSPP'94 予稿集, pp. 81-88 (1994).
13) 小中裕喜, 石川 裕, 前田宗則, 友清孝志, 堀 敦史：超並列オブジェクトベース言語 Ocore の商用並列計算機上での実装, 並列処理シンポジウム JSPP' 94 予稿集, pp. 113-120 (1994).
14) 岩下英俊, 進藤達也, 岡田 信：VPP Fortran：分散メモリ型並列計算機言語, 並列処理シンポジウム JSPP' 94 予稿集, pp. 153-160 (1994).
15) Meyer, B.: Object-Oriented Software Construction, Interactive Software Engineering (1988).
16) 文部省重点領域研究,「超並列原理に基づく情報処理基本体系」第4回シンポジウム予稿集, pp. 143-190 (1994).
17) Sequent Computer Systems, Inc.: System Summary Manuals (1990).
18) Schauser, K. E., Culler, D. E. and Eicken, T.: Compiler-Controlled Multithreading for Lenient Parallel Languages, *Proc. 5th ACM Conf. Functional Languages and Computer Architecture*, pp. 50-72 (1991).
19) 清水俊幸, 堀江健志, 石畑宏明：AP 1000 の性能評価—メッセージハンドリング, 放送, バリア同期の効果—, SWoPP' 92 情報処理学会研究報告, 92-ARC-95-12 (1992).
20) 立花 徹, 谷口倫一郎, 雨宮真人：データフロー解析による関数型言語 Valid のコンパイル法, 情報処理学会論文誌, Vol. 30, No. 12, pp. 1628-1638 (1989).
21) Takahashi, E., Taniguchi, R. and Amamiya, M.: Compiling Technique Based on Dataflow Analysis for Functional Programming Language *Valid, Proc. SISAL'93*, pp. 49-58 (1993).
22) Ueda, K. and Morita, M. : Message-Oriented Parallel Implementation of Moded Flat GHC, *Proc. Int. Conf. Fifth Generation Computing Systems 1992*, pp. 799-808 (1992).
23) Yonezawa, A. ed.: *ABCL : An Object-Oriented Concurrent System—Theory, Language, Programming, Implementation and Application*, The MIT Press (1990).
24) Yonezawa, A., Matsuoka, S., Yasugi, M. and Taura, K. : Implementing Concurrent Object-Oriented Languages on Multicomputers, *IEEE Parallel & Distributed Technology*, Vol. 1, No. 2, pp. 49-61 (1993).
25) Yoshida, K. and Chikayama, T. : *A'UM*—A Steam-Based Concurrent Object-Oriented Language, *New Generation Computing*, Vol. 7, No. 2, pp. 127-157 (1990).

(平成6年9月19日受付)
(平成7年5月12日採録)



日下部 茂（正会員）

1966年生. 1989年九州大学工学部情報工学科卒業. 1991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. 同年より同専攻助手. 並列処理記述言語, 関数型言語, データフローモデルに基づく細粒度並列処理の研究に従事.



高橋 英一（正会員）

昭和41年生. 平成6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博士課程単位取得退学. 同年, (株)富士通研究所入社. 以来, マイクロプロセッサ, コンパイラ, およびキャッシュの研究に従事. ソフトウェア科学会会員.

谷口倫一郎（正会員）

昭和55年九州大学大学院工学研究科修士課程修了．同年同大学院総合理工学研究科助手．平成元年同助教授．工学博士．画像理解，並列処理，画像処理システムに関する研究に従事．電子情報通信学会篠原記念学術奨励賞，本会論文賞，本会坂井記念特別賞受賞．人工知能学会，電子情報通信学会各会員．現在，本学会コンピュータビジョン研究会幹事．

雨宮　真人（正会員）

1942年生．1967年九州大学工学部電子工学科卒業．1969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，ACM各会員．