超並列V言語とその実行方式

日下部，茂
九州大学システム情報科学研究院知能システム学部門

高橋，英一
九州大学システム情報科学研究院知能システム学部門

谷口，倫一郎
九州大学システム情報科学研究院知能システム学部門

雨宮，真人
九州大学システム情報科学研究院知能システム学部門

出版情報：並列処理シンポジウム，1994年，pp.41-48，1994-06
バージョン：
権利関係：

超並列V言語とその実行方式

日下部茂　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

概要

我々は，本質的に並列性を内在するデータフロー向き関数型言語をベースに超並列V言語の研究を行なっている．状態を持つ並行プロセスの直観的記述のための抽象化単位 agent，さらに agent 間の明示的結合の記述や，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 プログラムは様々な粒度の並列性を内在する．現処理系は，マルチスレッド処理を行ない，lenient な実行もサポートする．対象とする計算機により適する粒度は異なり，AP1000 のように細粒度処理のための特別なハードウェアを持たない計算機での予備評価では静的なコード生成法が性能に大きく影響した．また agent の関係を明示することによる効果も評価し効率向上を確認した．

Abstract

In this paper, we propose a dataflow-based massively parallel programming language, called *V*, which is loosely 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, we can connect agents explicitly and abstract an ensemble of agents on a predefined topology description 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

近年，計算機はパイプラインから超並列計算機まで何らかの並列性を利用しているが，多くのプログラミング言語は依然逐次実行をベースにしている．その場合，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高い手練を要する．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と考える．

関数型言語は：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出来る；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；などの並列処理記述において魅力的な特徴を持つ．そのため，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向き関数型言語 Valid[2] をベースに設計している．

1.1　言語仕様設計方針

関数型言語は高い抽象度を提供し上述のような利点を持つが，操作的な記述や機種に依存した記述は出来ず必ずしも並列処理の記述のすべてを素直に行なえるとは限らない．我々は並列処理記述には，並列に動作する処理体の素直な記述に加え，(i) 並列に動作する各処理体の論理的構造，(ii) それらの間の通信形態，(iii) 物理的マッピング，の3つが重要と考えた．

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 —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，Valid に以下のような拡張を加えた：

- 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

A Dataflow-based Massively Parallel Programming Language, *V*, and Its Implementation
Shigeru Kusakabe, Eiichi Takahashi, Rin-ichiro Taniguchi, Makoto Amamiya
Department of Information Systems, Graduate School of Engineering Sciences, Kyushu University

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ない，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を V 言語自体は持たない．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 で行なう方針を取る．これに関しては別の機会で述べることとする．

1.2 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，次のような実行方式によって lenient な実行をサポートする．

- 関数インスタンス程度の粒度で並列処理を行なう．
- インスタンス内処理を複数のスレッド[1]に分けそれらを並行実行する．遅延を含む処理の要求側は，要求発行後他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，細粒度並列性を内在する 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．次に，グラフからのスレッド抽出を行い，互いにデータ依存関係の無いインスタンスは並列実行，遅延を持つ処理は split-phase 実行とし遅延をオーバラップ出来るよう DVMC をスケジューリングしスレッド化コードを生成する．DVMC を基本とする実行では，動的なインスタンス管理や頻繁な同期処理を行なう必要がある．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，マシンに依存するスレッド実行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，処理系が効率良く実行できるよう試みる．

本稿ではまず，2 節で V 言語の特徴について，3 節でコンパイラについての概要を述べる．4 節で実装例として疎結合並列計算機 AP1000 への実装について述べ，5 節で予備評価を行なう．6 節で関連研究について述べる．

[1] 排他的に実行される命令列．

2 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した．(agent 間/agent 内も含め) 並列性を内在し，実行順はデータ依存関係のみによって決定され，同期は依存関係に従って自動的にとられる．agent 間も依存関係に従って (ストリームを通じ) データフロー的に同期がとられる[2]．

以下に V 言語の特徴の概要を述べる．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．

2.1 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．以下がその定義とインスタンスの生成である．

```
agent ⟨agent 定義名⟩(⟨生成時仮引数部⟩)
⟨ストリームインターフェース部⟩
 = ⟨本体式⟩ ;
⟨変数⟩=create(⟨agent 定義名⟩[, 式の並び])
```

ストリームインターフェース部は

```
{ 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 変数部と，出力ストリームの変数を指定する export 部からなる．

本体の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オブジェクトとして記述するには，履歴依存状態値の保持，ストリームの処理，を行なう再帰構造を持たせる．インスタンス内の環境や状態値の初期化は，生成時仮引数を通して渡された値により行なう．新しい agent インスタンスの生成や，出力ストリームに対する操作 (メッセージ送信) なども行える．

2.2 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

[2] agent により好ましくない非決定性が生じる可能性がある場合，処理の一貫性を保つために明示的に実行順を指定することも可能．

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true，そうでなければfalseを返す．

2.3 通信

インスタンス間通信においては，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2通りある．

メッセージパッシング その名前を知っているagentインスタンスには以下のように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@は省略時は1番目のchannel変数が選択される)．送られたデータはそのストリームに暗黙のうちにマージされる．以下のような場合，

```
⟨変数⟩=send([ch_k@]⟨agent インスタンス名⟩,⟨式⟩)
```

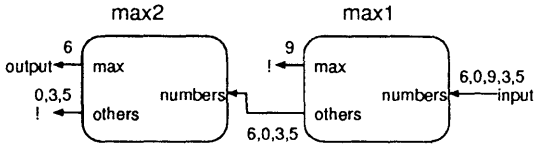
⟨変数⟩が処理結果の返し先となるメッセージが，指定された⟨agent インスタンス名⟩のストリームに送られ，送られた側でそのメッセージに対する処理の後，**reply** を行なえば結果データはストリームを経由せず関数呼び出しの結果値のように直接⟨変数⟩に依存する処理を起動する．

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通信を明示できる．

```
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．

図1にインスタンスの生成時にストリーム結合を行なう join を用いた例題を示す．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例を以下に示す．'eos はストリームの終端を示し，数字つきの"$"はストリームを結合するための一時変数でこのjoin内でのみ有効なものである．"!" はストリームをsinkする．




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

= {join
    max1=create(selectmax){std_in}{!,$1},
    max2=create(selectmax){$1},{std_out,!}} ;
```

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

2.4 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⟩)[ストリームインターフェース部]
 [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])
```

我々は並列処理の効果が最も得られるのは規則的な構造にagentインスタンスの集合を構成できる場合と考え，構造テンプレートによりagent集合の論理的構造と通信形態をを指定して扱えるようにした．構造テンプレートは現在 **mesh**, **torus**, **tree**, **hypercube** がある．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．インデクス指定部はインデクス変数とサイズを指定する．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，⟨ストリームインターフェース部⟩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．

例題として4近傍の画素とデータを交換，その平均を自分の値とすることを何度か繰り返すaverageを用いて，図2のような通信パターンでagent集合体Fをtorus構造に生成する例を示す．torus構造ではup, down, right, leftで4近傍が指定できる．

```
agent average(val:real)
{channel in1,in2,in3,in4: real}
{export out: real}
=for (v:real;count:integer;i1,i2,i3,i4:stream)
   init (val,0,in1,in2,in3,in4) body
   if count>=M then v
   else
   { slet put(v,out),
         next=(head(i1)+head(i2)
               +head(i3)+head(i4))/4
     in recur(next,count+1,tail(i1),tail(i2),
                           tail(i3),tail(i4))};
```

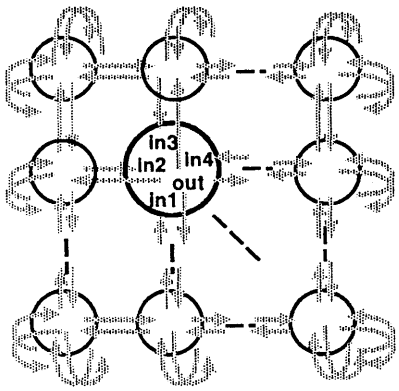


図 2: torus構造をしたagent集合体

```
field F(averag)(val:array of real) on torus
 do foreach (i:integer) in ([1..n]) body
     foreach (j:integer) in ([1..n]) body
      link(ex[1],ch[1]@up,ch[2]@right,
                 ch[3]@down,ch[4]@left) ;
= foreach (i:integer) in ([1..n]) body
    foreach (j:integer) in ([1..n]) body
      create(averag,val[i][j]) ;
```

3 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．機種依存/非依存処理を切分けており，非依存処理部の出力はDVMCと呼ぶ抽象マシンコードである．以下に，各部の概要について述べる．

3.1 機種非依存処理

コンパイラは，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ない，

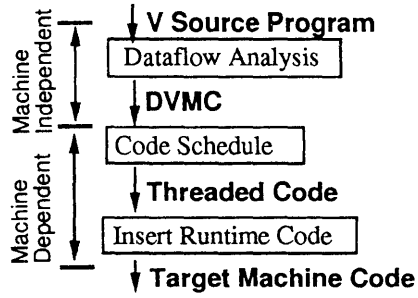


図 3: コンパイラの概要

Symmetry[10], AP1000[6], Datarol Machine[8]など，各マシンの並列コードを生成するための中間コードとしてDVMCプログラムを生成する．DVMCは，命令レベルから，関数レベルまでの幅広い並列性の抽出，および，抽出する並列性のレベルに応じた最適化に対応することを狙ったもので，ソースプログラムの構造を表現するパートSST(Source level Structure Tree)と，細粒度のグラフCDDG(Control and Data Dependency Graph)を表すパートに分かれる．CDDGはDatarolコード[11]をベースにした抽象マシンコードで，コンパイラはCDDGだけを用いて並列コードを生成することができるが，SSTを用いることで，スレッド合成などの粒度最適化，スレッドスケジューリングが容易になる．

3.2 機種依存処理

ここでは，各計算機の特性，並列処理のためのランタイムのコストなどを考慮して，DVMCからのスレッド抽出，およびスケジューリングを行う．以下その概要を述べる．

STEP 1 DVMCへターゲットマシン情報を付加．
例えば疎結合計算機では，他セルとのデータやコントロールの授受は，全て遅延を伴う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は全てsplit-phase操作となる．

STEP 2 スレッド抽出．
以下の原則を守って，グラフよりスレッドを抽出する[3]．

規則 1 split-phaseアークの親と子は別スレッドに属する．

規則 2 複数の親を持つノードはどの親とも別スレッドに所属．

STEP 3 コードスケジューリング．
各スレッド内の命令を，逐次コード化する．ま

[3] 親子関係の用語は[11]に準じている

た，スレッド間の実行制御のため，図 4 に示すパタンに応じたコードをコンパイラは挿入する．(a),(b) はそれぞれ，スレッドの最後，途

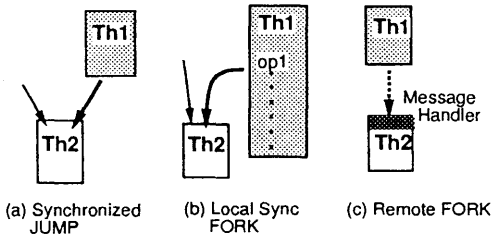


図 4: スレッド間実行制御

中から他の局所スレッド起動を試みる場合に Synchronized JUMP 命令，Local Sync FORK 命令を挿入している．各命令の動作は 4.3で述べる．(c) は，split-phase 操作の場合で，T2 の先頭に起動 (返答) メッセージを適切に解釈するコード Message Handler がコンパイラによって加えられる．

STEP 4 メモリ資源割り当て．

次節で述べる並列実行メカニズム上のレジスタ，スタックフレーム，フレーム変数スロット，ヒープなどの記憶階層を容量，速度などを考慮して割付けるコードにする．

4 並列実行方式

以下，AP1000 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．疎結合型計算機ではプロセッサ間通信コストが高く，リモートアクセスのような双方向通信は，アクセス要求側に返答待ちが生じ，効率が落ちる．そこで，インスタンス内を複数のスレッドに分け，アクセス要求側は，要求後他の実行可能なスレッドに実行を切替え遅延をオーバラップすることを試みる．インスタンス内が複数スレッドに分かれており，lenient な関数呼出も実行できる．

4.1 フレーム

インスタンス毎にフレームというデータ構造が動的に生成され，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ーリストで管理している．

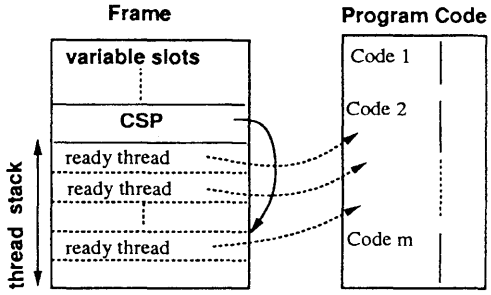


図 5: フレームの構造

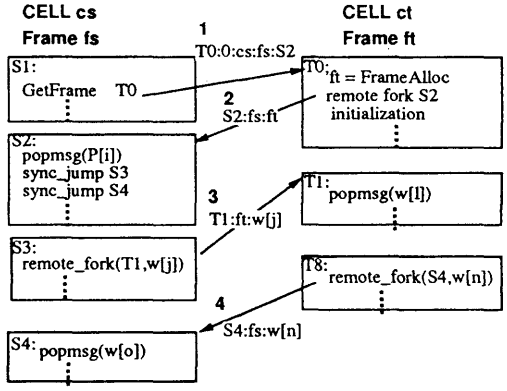


図 6: インスタンス生成とデータの授受

4.2 インスタンス生成

図 6にインスタンス生成とデータの授受の概略を示す．

1. セル ct 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，継続点 (セル ID, フレームポインタ, スレッド ID) をセットしておく．
2. セル ct では，新規フレーム (=ft) を取得し，ft とともに要求側の継続点を起動するメッセージを返す．
3. セル ID，フレームポインタ，スレッド ID で特定されるリモートのスレッドにデータを渡し起動する．
4. 返値渡しについても返し先の継続点へ返値を渡し起動することで行う．

4.3 スレッド実行制御

インスタンス内には1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．図 4(a),(b) に示すように，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，バリア同期を行う．バリア変数は，同期ポイント毎にフレーム変数スロットを割り当てて実現する．現フレーム取得時に同期操作を行うスレッドの数へ初期化しておき，Synchronized JUMP，Local Sync FORK 命令で，1減じていく．もし，バリア変数を 0 にした時，Synchronized JUMP 命令であれば，直ちに継続スレッドへ制御を移し，Local Sync FORK 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．

5 予備評価

5.1 細粒度並列処理

我々の実行モデルは TAM に類似しており，実行時のフレーム内でのスレッドの動的グループ化 [5] 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 AP1000 上での実装では，インスタンスを構成するスレッド数が少なくその長さも短い場合には通信のオーバヘッドが非常に大きく実行効率を下げていた [12]．データフロー計算モデルに基づくような遠隔呼び出しプロトコルで，細粒度で通信と計算のオーバラップを試みるのは，EM4 のような通信が早いマシンの場合は有効である [16]．しかし，AP1000 のように細粒度並列処理を特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ランタイムのコストを考慮したコード生成が必要である．

AP1000 の隣接 PE 間での 1 回あたりの送受信時間は pingpong ベンチマークによれば $31 + 0.04x$ μ秒である (ここで x はメッセージ長 [byte]，1clock は 40[ns]，AP1000/C 送受信ライブラリ関数含む)．また，現段階の我々の AP1000 上の細粒度並列実装の基本操作コストは表 1の通りである[4]．

[4] なお，このコストは AP1000/C で記述されたランタイム処理をアセンブラにコンパイル後，アセンブラ命令数をカウントしたものである．現段階ではアセンブラレベルでの最適化は行なっていない．また AP1000 の通信ライブラリのコストは含んでいない

表 1: 細粒度並列処理操作コスト

操作			コスト [命令]
message	書込		$10 + 3x$
	取出		$3x$
thread 切替	frame 間	message 取出後	54
		local queue	59
	frame 内	jump	24
		syncjump	30
		through	2
同期チェック			6

(x: メッセージの要素数)

そこで我々は，このコストを考慮して以下のように何段階かのコード生成戦略の再適用を行ないオーバヘッドの削減を試みた．例として N-queen 全解探索の例題にこれらを適用した場合の台数効果改善の結果を図 7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いる．

- **Opt0:** インスタンス内のみでコードスケジューリング，特に冗長に分割されたスレッドの静的併合を行なう．インスタンス呼出などは完全に lenient に実行する．
- **Opt1:** インスタンス間通信も対象としたスケジューリングを行なう．Opt0 に加え，同一スレッド内からの同一フレームへのメッセージの結合，リモートインスタンス生成要求メッセージと継続点渡しメッセージの結合，リモートインスタンス生成要求メッセージと引数渡しメッセージの結合などを行なう．lenient 性は減少する．
- **Opt2:** Opt1 に加え local なインスタンス呼出を積極的に用いる．
- **Opt3:** Opt2 に加え可能な場合，frame を用いたインスタンス呼出でなく，ユーザスタックを用いた呼出を行なう．

Opt1 では通信と同期の回数が減少し効率が向上した．Opt2 では静的に自セル内での呼出を決定しセル間通信が減少したため効率が向上した．Opt3 では一部ユーザスタックを用いた呼出を用いたためフレーム管理のコストが削減されている．

5.2 agent

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までの命令数は，メッセージパッシン

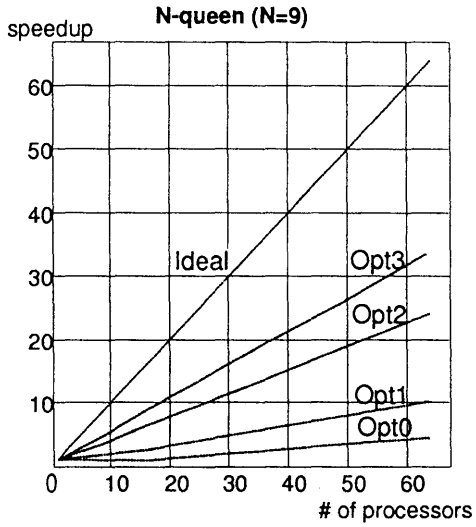


図 7: N-queen 全解探索問題での速度向上比の改善

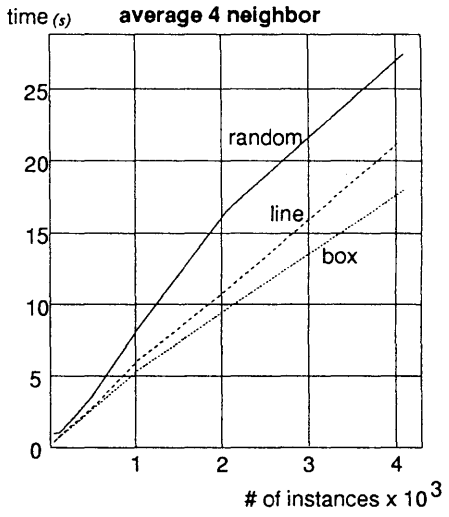


図 8: 4 近傍平均を計算する field でのインスタンス配置の影響

グで最低約 130 命令，結合ストリーム通信で最低約 110 命令であった[5]．結合ストリーム通信では，結合された相手にしかメッセージを送らないためメッセージ中に含ませる情報が少ないため差が生じている．また，現段階ではそのコストを評価していないが，送信に当たっては論理インスタンス ID を物理 PE にマッピング後のインスタンス ID に変換する必要がある．その変換テーブルを (分散) 実装した場合，2 つの通信法のコスト差はさらに大きくなると予想される (特定の相手との通信なら毎回テーブルを引かなくとも良い)．また，結合ストリーム通信を用いる場合，物理的に近くにマッピングしやすい，特殊なケースで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 [14] のような効率良い実装が可能，などが期待できる．

次に，2節の図 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の情報を処理系が活用した場合，効率の良い実行が出来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，AP1000 の 64 セルにランダムに配置 (ramdon)，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 8に示す．

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．

[5] 命令数のカウント法は表 1と同じ．データ転送時間，AP1000 通信ライブラリ含まず．

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 [15] は必要ない．言語仕様において，継承など [9] で論じられているようなオブジェクト指向の機能の導入は今後の課題である．

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 [14] 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 では共有メモリ上でのゴール共有方式実現をとっており，大規模・高並列処理には適していない．我々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ことを試みている [13]．

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 [16] と比較すると，スレッドのオーバラップ実行により細粒度並列実行を効率良く実現できる超並列計算機上では耐遅延性に優れると思われる．我々の実行方式は TAM[5] 同様の背景を持つが我々の中間コードは Datarol[3] を拡張したもので，TAM の中間コードである Dual Graph とくらべ構文情報を積極的に利用可能にす

るなどしてスケジューリングを容易にしている．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[7]と同様であるが，我々はremote/localの使い分けに加え，粒度を変化させることによる実行効率向上を試みている．

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agent 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，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．AP1000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実現では，静的なコード生成戦略が実行効率に大きな影響を与えた．

また，agentの実現では，問題の論理構造をプログラムに反映可能なことにより，処理系もその明示された情報を活かしたマッピングを行ない，効率の良い実行が出来た．

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の実現する．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AP1000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] G.A.Agha "Actors: A Model of Concurrent Computation in Distributed Systems", MIT Press, 1986.

[2] M. Amamiya, et al. "*Valid*: A High-Level Functional Programming Language for Data Flow Machine", Rev. ECL, Vol.32, No.5, p.p.793-802, NTT, 1984.

[3] M.Amamiya, and R.Taniguchi : "Datarol: A Massively Parallel Architecture for Functional Language", Proc. IEEE 2nd SPDP, p.p.726-735, 1990.

[4] A.A Chien "Concurrent Aggregates" The MIT Press, 1993.

[5] K.E.Schauser et al. "Compiler-Controlled Multithreading for Lenient Parallel Languages" Proc. of 5th FPCA, 1991.

[6] H.Ishihata et al. "An architecture of highly parallel computer AP1000", IEEE Pacific Rim Conference on Communication, Computers and Signal Processing, p.p.13-16, 1991.

[7] V.Karamcheti et al. "Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Language on Stock Hardware", Proc. of Supercomputing'93, 1993.

[8] 川野, 他 "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 –", JSPP'94(発表予定).

[9] B.Meyer "Object-Oriented Software Construction", Interactive Software Engineering, 1988.

[10] Sequent Computer Systems,Inc.: *System Summary Manuals* (1990).

[11] 立花, 他 "データフロー解析による関数型言語Validのコンパイル法", 情処論 Vol.30, No.12, p.p.1628-1638, 1989.

[12] 高橋, 他 "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", 情処研究報告 PRG93-73-3, pp.137-144, 1993.

[13] E.Takahashi et al. "Compiling Technique Based on Dataflow Analysis for Functional Programming Language *Valid*", Proc. of SISAL'93, p.p.49-58, 1993.

[14] K.Ueda et al. "Message-Oriented Parallel Implementation of Moded Flat GHC" Proc. of Int. Conf on FGCS'92, p.p.799–808, 1992.

[15] A.Yonezawa, editor "ABCL: An Object-Oriented Concurrent System – Theory, Language, Programming, Implementation and Application" The MIT Press, 1990.

[16] A.Yonezawa et al. "Implementing Concurrent Object-Oriented Languages on Multicomputers", IEEE Parallel & Distributed technology, p.p.49-61, 1993.