プログラミング—言語・基礎・実践— 3－13
（1991. 7. 17）

並列オブジェクト指向言語 A'UM-90
- 並列分散処理方式の評価 -

丸山 勉、 柳田 伸二†、小西 弘一、小長谷 明彦、近山 隆††

日本電気(株) C&C システム研究所 †日本電気技術情報システム開発(株)
††(財) 新世代コンピュータ技術開発機構

本稿では、並列オブジェクト指向言語 A'Um-90 の並列版処理系の概要とその評価について報告する。A'Um は、ストリームと呼ばれる非同期通信機構を基本とする言語である。ストリームを動的に生成、接続することによってオブジェクト間のストリームネットワークを容易に構築することができ、これを用いてオブジェクトの実行制御を効率的に行なうことができる。これまでに A'Um 並列版処理系が完成し評価を行なっている。実装においては、高並列な処理を実現するために、共有メモリの存在は仮定しない方式をとっている。A'Um 並列版処理系は、現在、市販の共有メモリ型並列マシン上で動作しており、通信の局所性等を全く利用せずに全ての通信がプロセッサ間通信となるような負荷分散をとった場合においても、通信の局所性を利用できる 1 プロセッサの場合と較べて、15 プロセッサで約 8~10 倍の性能向上が見られストリーム通信の有効性が確認された。

A Parallel Object Oriented Language A'UM-90
- Preliminary Evaluation of the Distributed Implementation -

Tsutomu Maruyama, Shinji Yanagada †, Kouichi Konishi, Akihiko Konagaya, Takashi Chikayama ††

NEC Corporation, †NEC Scientific Information System Development,
††Institute for New Generation Computer Technology

1-1, Miyazaki 4-Chome Miyamae-ku Kawasaki Kanagawa 213 JAPAN

This paper describes the preliminary evaluation of the distributed implementation of a stream-based parallel object-oriented language A'Um-90. Stream is an asynchronous communication channel between objects. By creating and connecting streams dynamically, the execution order of objects can be controlled easily and efficiently. Parallel A'Um System has been developed on a shared memory parallel machine. No shared memory is assumed in the implementation so that the system will work on distributed memory parallel machines. Under the distribution strategy which creates all objects in different processors, the speed gain is about 8~10 with 15 processors compared with a single processor which can fully optimize local communication.

1 はじめに

オブジェク指向言語では、オブジェクトの処理はメッセージを受信することによって選択され起動される。従って、オブジェクトの実行制御を行なうためには、メッセージの到着順の制御を行なうことが必要となる。逐次的な環境下では、メッセージの到着順の制御は容易であるが、並列環境下では各オブジェクトが並列に動作するために、複数のオブジェクトからあるオブジェクトにメッセージが送信された場合等には、それらのメッセージの到着順を制御するために何らかの機構が必要となる。この機構は、言語の記述性及び処理性能に直接左右するため、様々な試みが行なわれいる [2][6]。

A'Um[1][3] は、オブジェクト間の通信路をストリームをして捉え、ストリームの接続を制御することによって、この機構を実現している。ストリームは自由にストリームおよびオブジェクトと接続することが可能である。接続されるとストリーム中のメッセージは順序を保ったまま接続先に転送されるストリームを動的に生成、接続することによってオブジェクト間のストリームネットワークを容易に構築することができ、これを用いてオブジェクトの実行制御を効率的に行なうことができる。

これまでに A'Um の並列版処理系 (PAS) が完成し、評価を行なっている。実装においては、高並列な処理を実現するために共有メモリの存在は仮定しない方式をとっている。A'Um 並列版処理系は、現在市販の共有メモリ型並列マシン上で動作しており、通信の局所性等を全く利用せずに全ての通信がプロセッサ間通信となるような負荷分散をとった場合においても、通信の局所性を利用できる 1 プロセッサの場合と較べて、15 プロセッサで約 8~10 倍の性能向上が見られストリーム通信の有効性が確認された。

本稿では A'Um 並列版処理系 (PAS) の概要と、その評価について述べる。次章では、A'Um の概要について述べる。第 3 章では、並列版処理系 (PAS) の概要についてのべる。第 4 章では、並列版処理系の評価結果について述べる。

2 A'Um 概要

A'Um の主な特徴は以下の 3 点である。

- 並列オブジェクト指向
- 並列性を基本とした記述
- ストリームによる通信

A'Um 言語モデルの構成要素は、オブジェクトとストリームとメッセージである。オブジェクトはメッセージを受けとって、そのメッセージによって選択されるメソッドを実行する。各オブジェクトは原則として並行に動作する。

更に、A'Um にはメソッド中の各アクションに実行順序を記述する構文はなく、各アクションは並行に実行することが可能である。各アクション間での実行順序は、値の依存関係、メッセージの発信と受信の前後関係、およびメッセージの到着順序のみによって決まる (但し、後で述べるように PAS においては、各アクションは実際には並列に実行されない)。

以下、A'Um の最も重要な特徴であるストリームの基本操作について述べる。

2.1 ストリーム通信

ストリームはメッセージキューとしての機能を持つ。ストリームは生成時にはどこにも接続されていない。複数のストリームを接続、合流させることにより、オブジェクト間のネットワークを自由に作ることが可能である。このようにしてネットワークを動的に作ることによって、各オブジェクトへのメッセージの到着順を容易に制御することができる。

以下ストリームの基本機能について述べる。

ストリームの生成

ストリームは動的に生成することができる。ストリームには、頭部 (inlet) と尾部 (outlet) がある。ストリームに対するメッセージ送信は、outlet に対して行なわれる。ストリームが生成された時点ではそのストリームはどこにも接続されていない。このような未接続のストリームに対してもメッセージ送信を行なうことが可能である。未接続のストリームに対して送信されたメッセージは、ストリーム中に留まる。

ストリームの接続

ストリームの inlet は、他のストリームの outlet およびオブジェクトと接続することができる。ストリームが接続されると、そのストリーム中に蓄えられていたメッセージは接続先に送信される。このとき、各ストリーム内でのメッセージ順序が保証されるばかりでなく、ストリーム間での順序も保証される。例えば、ストリーム A の outlet とストリーム B の inlet が接続された場合には、ストリーム B 中のメッセージは、ストリーム A 中のメッセージがオブジェクトに到着した後に到着することが保証される。

ストリームの閉鎖

それ以上参照されないストリームは閉鎖することができる (言語処理系ではそれ以上参照されない場合に自動的に閉鎖される)。ストリームの閉鎖は、ストリームの接続先に伝搬される。ストリームの接続先がオブジェクトでり、その入力ストリームが全て閉鎖された場合にはオブジェクトは消滅する。

2.2 A'Um プログラム

A'Um プログラムはおおよそ図 1のような形をとる。以下、A'Um プログラムにおける代表的な処理について説明する。

ストリーム

ストリームはプログラム中で、大文字で始まる変数で表される。ストリームの inlet は ~Stream のように

```
class class_name.          % class 定義
   int inlet_slots....     % inlet slot の宣言
   out outlet_slots...     % outlet slot の宣言
:method(args...) ->        % method 名と引数
   actions...              %   method 内処理
...
end.                       % class 定義の終り
```

図 1: A'Um プログラムの概略

変数の頭に '^' をつけて表され、outlet は何もつけずに単に Stream のように表される。ストリームは自由に生成することができ、プログラム中に新たな変数を書けば、自動的にストリームが生成される。

ストリーム変数のスコープは、メソッド内部のみである。従って、メソッドの実行が終了すると不必要となったストリームは自動的に回収される。プログラム上で特にストリームの回収に関して記述する必要はない。

スロット

スロット変数は、オブジェクトに固有の変数である。各メソッドからアクセスすることが可能であるため、メソッド間での共有変数として用いることができる。スロット変数には、inlet スロットと outlet スロットの 2 種類がある。inlet スロット変数は、ストリームの inlet を保持するのに用いられ、outlet スロット変数は、ストリームの outlet を保持するのに用いられる。

ストリームの接続とスロットへの代入

A'Um プログラムにおいては、演算子 '=' は 2 つの意味を持ち、以下のルールによって、使い分けられている。

1. ストリームの接続
   X = ^Y のように、'=' の左辺が outlet であり、右辺が inlet である場合は、ストリームの接続が行なわれる。このとき、メッセージは右辺のストリームから、左辺のストリームへと流れ、左辺のストリーム中のメッセージが、右辺のストリーム中のメッセージより先に到着することが保証される。
2. スロット変数への代入
   スロット変数のスコープは、クラス定義内である。従って、メソッド間に跨るデータの共有はスロット変数と用いて行なうことができる。inlet slot は頭に @ をつけて用いられ、outlet slot は ! をつけて用いられる。
   (a) '=' の左辺が inlet slot である場合
   @inlet_slot = ^X のように '=' の左辺が inlet slot である場合には、右辺の inlet が inlet slot に代入される。
   (b) '=' の右辺が outlet slot である場合
   X = !outlet_slot のように '=' の右辺が outlet slot である場合には、左辺の outlet が outlet slot に代入される。

以上のストリームの接続と代入においては、以下のような記述も可能である。

X = ^Y = ^Z

これは、

X = ^Y, X = ^Z

と等価である。このとき、ストリーム Y と Z に蓄えられていたメッセージは、ストリーム X 中のメッセージより後に到着するが、Y と Z 中のメッセージの到着順は保証されない。

オブジェクトの生成と消去

オブジェクトの生成は、#class_name と書くことによって実行される。このとき、#class_name の返却値は生成されたオブジェクトへのストリームである。一般には、

#class_name = ^X

のように用いられ (生成されたオブジェクトとストリーム X の inlet が接続される)、X に送信されたメッセージは、生成されたオブジェクトに到着する。

どのストリームからも参照されなくなったオブジェクトは自動的に回収される。このとき、オブジェクトのスロット変数によって参照されているストリームも必要に応じて自動的に回収される。

メッセージ送信

メッセージ送信は、

Stream :message(args..)

のように書くことによって実現される。送信先は outlet slot であることもある。このメッセージ送信は式の値を持ち、ストリームの outlet を返す。従って、

X :message(args) = ^Stream,

のように書くこともできる。この場合 :message は Stream に対して送られたメッセージより先に到着することが保証される。また、

X :m(args) :n(args) = ^Stream,

のような記述も可能である。この時は、メッセージ :m が先に到着し、:n はその後で到着することが保証される。さらに、オブジェクトを生成した場合にも、その返却値はオブジェクトへのストリームとなるので、以下のような記述も可能である。

#class_name :message(args...) = ^Stream

宛先ストリームを指定しないメッセージ送信は自分自身へのメッセージ送信となる。このメッセージは、他のオブジェクトからのメッセージより優先的に実行される。

条件判断

条件判断は、基本的にはオブジェクトに値を問うメッセージを送信し、その返答を受けるという形をとる。以下に示すのがその記法である。ストリーム X に値が 1 であるかを問うメッセージを送信し、その結果がメッセージ名 ':true もしくは ':false というメッセージによって返答される。

```
(X == 1)? (
   :'true ->
      ....
   :'false ->
```

```
class aum.
  :go(^Argv,^Environment) ->
    #mesh :build(256, X, X) = ^X.
end.

class mesh.
  out objA, objB, objC.
:build(^N, ^Right, ^Down)->
  (N > 1) ? (
    :'true ->
       #mesh:build(N/4,AtoB,  AtoC) =!objA,
       #mesh:build(N/4,RightA,BtoD) =^AtoB=!objB,
       #mesh:build(N/4,CtoD,  DownA)=^AtoC=!objC,
       #mesh:build(N/4,RightC,DownB)=^BtoD=^CtoD,
       Right:v_edges(^RightA, ^RightC),
       Down :h_edges(^DownA,  ^DownB);
    :'false ->
       ....
  ).
:v_edges(X, Y) ->
  !objA = ^X,
  !objC = ^Y.
:h_edges(X, Y)->
  !objA = ^X,
  !objB = ^Y.
end.
```

図 2: メッシュを生成するプログラム (Mesh)

```
            ....
    )
```

2.3 ネットワークプログラミング

以下、プログラム例を用いてその実行過程を追いながら、A'Um プログラム及びストリームの機能について説明する。

メッシュ構造を作るプログラム例を図 2に示す。このプログラムは、図 3に示したようなトーラスな構造を作るものである。メッシュは階層的に作られていく。最初の層のオブジェクトの数は 1であり、次第に下の層に行くに従って、オブジェクトの数は log オーダーで増えていく。例に示したプログラムは、説明を簡単にするために 1,4,16,64, …とオブジェクトの数が増えて行く場合のものである。最下層につくられる構造が目指すメッシュ構造である。

処理系が起動されると、class aum に :go というメッセージが送信される。#mesh によって、class mesh のオブジェクトが生成され、次いで、そのオブジェクトにメッセージ :build(256,X,X) が送信される。オブジェクトにメッセージを送信した後の値は、そのオブジェクトへのストリームである。引数にこの値を渡すことによってトーラス構造が作られる。

プログラムの動作の概要について簡単に説明する。ある階層のオブジェクトは、以下のようにして一つ下の階層に 4つのオブジェクトを作る (図 4)。

1. 同じ階層の右側のオブジェクト (Right) と下側のオブジェクト (Down) へのストリームを引数として受けとる (メッセージ :build の第 2,3 引数)。
2. 4つのオブジェクトを生成する。これらが 1階層下のオブジェクトとなる。これらの内で、他のオブジェクトによって生成されたオブジェクトと接続する必要があるオブジェクトへのストリームをスロットに代入する (objA,objB,objC)。
3. 4つのオブジェクトに渡したストリームのなかで、右側のオブジェクトが作る 4つのオブジェクトの内の 2つと接続されるべきストリームの inlet(^RightA, ^RightB) を右側のオブジェクトに送信し (メッセージ :v_edges)、接続してもらう。
4. 4つのオブジェクトに渡したストリームのなかで、下側のオブジェクトが作る 4つのオブジェクトの内の 2つと接続されるべきストリームの inlet(^DownA, ^DownB) を下側のオブジェクトに送信し (メッセージ :h_edges)、接続してもらう。
5. このオブジェクト自身も、その上側のオブジェクトおよび左側のオブジェクトから来た接続要求 (メッセージ :v_edges, :h_edges) に対して、先ほど作ったオブジェクトを接続する。

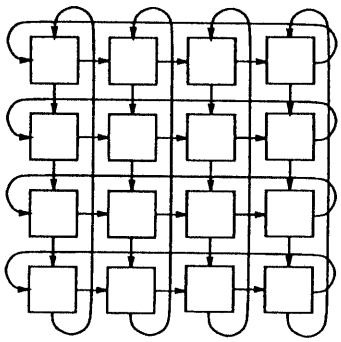

図 3: メッシュの構造

以上の処理を繰り返すことによって、メッシュ構造を作ることができる。

このようなプログラムにおいて、ストリームを使うことの利点は各オブジェクトが自分の 1階層下に作った 4つのオブジェクトを他のオブジェクトによって 1階層下に作られたオブジェクトと接続するのではなく、メッセージ :v_egdes および:h_edges を用いて、ストリームの inlet を送るというかたちで、他のオブジェクトに接続してもらうことができる点にある。即ち、各オブジェクトは自分の作った 4つのオブジェクトのアドレス以外は知る必要はない。このため、各オブジェクトは次々と自分の下の階層のオブジェクトを生成することができる。

これに対して、ストリーム機能を用いない場合には、各オブジェクトは、右側および下側のオブジェクトがそれぞれ作った 4つのオブジェクト中で自分の作ったオブジェクトと接続しなければならないオブジェクトのアドレスを受けとる必要がある。さらに、これらを受信するまでは、1階層したのオブジェクトに

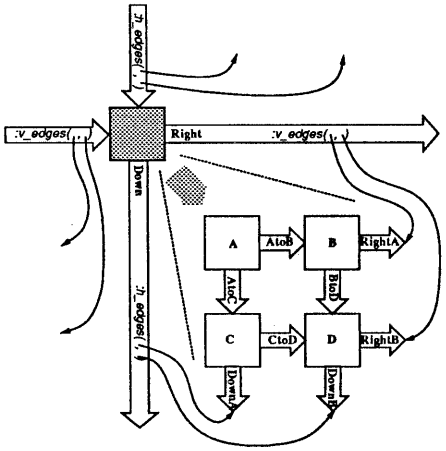

図 4: メッシュの生成

:build メッセージを送信することはできない (RightA, RightB, DownA, DownB が確定するまで :build メッセージを送れない)。このため、ここで待ちが必要となる。従って、ストリームを用いない場合には、プログラマは、2に加えて、オブジェクトのアドレスを受信するためのメソッドを2種類と、さらに、その2種類のメソッドが実行されたことを確認してから、:build メッセージを送信するという部分を記述しなければならないことになる。

以上述べたように、ストリーム機能を用いることによって、処理の記述が簡潔になるばかりでなく、待ちが不要になる分、処理速度の向上も期待できる。

3 並列版処理系 (PAS)

並列版処理系 (PAS: Parallel A'Um System) は現在、市販の共有メモリ並列マシン Symmetry 上に実装されている。Symmetry は共有メモリを持つマシンではあるが、PAS の実装においては、非共有メモリ型のより多くのプロセッサを持つマシンでも動作するように、共有メモリを仮定しない実装方式をとっている。現在、メッセージの送受信部にソケットを用いた分散処理系についてもデバッグ中である。以下、PAS について述べる。

3.1 概要

A'Um プログラムはまず、A'Um コンパイラによって抽象機械語にコンパイルされ、次いで 並列版処理系 PAS によって実行される。プログラム Mesh に対するコンパイル結果の一部を図 3.1に示す。抽象機械語のレベルはかなり高く、A'Um の基本機能であるメッセージ送信、ストリームの接続等とほぼ1対1に対応している。

A'Um の言語モデルでは、各アクションは並列に実行可能である。従って、あるメソッドの実行中に、他のメソッドの実行を開始することも可能である。しかし、汎用プロセッサベースの並列マシンへの実装を考えた場合には、各アクションのような細粒度の処理を並列に実行しても処理速度の向上は期待できない。このため、並列版処理系 (PAS) における並列処理の単位はオブジェクトであり、各メソッド内の処理は逐次的に行なわれる。また、あるメソッドを実行中に他のメソッドの処理を開始することもない。

オブジェクト内の処理はこのように逐次的ではあるが、A'Um コンパイラは、各メソッド内のアクションの実行順序は任意であることを利用して、処理の最適化を行なっている。

3.2 データ構造

A'Um の基本データ構造は、オブジェクトとメッセージとストリームである。

オブジェクトは、メッセージキュー, スロット変数領域, クラステンプレートへのポインタ, スケジューラ待ち用のポインタ等から構成される。オブジェクトに送られたメッセージは、このキューにつながれる。メッセージを有するオブジェクトは、スケジューラの待ち行列につながれ、順番が来るとオブジェクトに接続されていたメッセージが実行される。

メッセージは、オブジェクトのメッセージキューに繋ぐためのポインタ、メッセージ ID および引数領域からなる。

ストリームは、ジョイントとして実現される。ジョイントには2種類がある。一つは、ストリームの順序を指定しない併合を表すものであり、マージジョイントと呼ばれる。他の一つは、併合の順番を陽に指定する場合に用いられるものであり、アペンドジョイントと呼ばれる。各ジョイントは、接続先が未確定の場合には、メッセージを蓄えるキューとして実現され、接続先が確定した場合には、単なるポインタとして機能する。マージジョイントの構造を図6に示す。

3.3 アドレス管理

A'Um 並列版処理系 (PAS) では、プロセッサ間の共有メモリを仮定していない。アドレス空間はプロセッサ間で異なるため、プロセッサ間でのアドレス管理が必要となる。このために、アドレス輸入表とアドレス輸出表を用いている。

アドレスには、グローバルアドレスとローカルアドレスの2種類がある。ローカルアドレスは、各プロセッサ内のアドレス空間そのものである。ローカルアドレスが他プロセッサに輸出される場合には、そのローカルアドレスは、アドレス輸出表に登録され、グローバルアドレスに変換される。グローバルアドレスは、PE 番号と PE 内でのユニークな番号から構成され、システム内で一意に定まるものである。アドレス輸出表への登録にはハッシュを用いており、同一のローカルアドレスには、必ず同一のグローバルアドレスが割り振られる。

輸入されたグローバルアドレスは、輸入表に登録され、ローカルアドレスに変換される。輸入されたグローバルアドレスを保持するために用いられるデータ構造は、マージジョイントそのものである。輸入表への登録においては、ハッシュを用いており同一のアドレスは、必ず同一のエントリに登録される。

```
% class 定義の開始
   .class #mesh;
% 参照クラス
   .classref #if_then_else;
% メッセージ定義
   .pid 'v_edges/--', "v_edges", 2, --;
   .pid 'h_edges/--', "h_edges", 2, --;
   ...;
% スロット定義
   .outlet objA;
   .outlet objB;
   .outlet objC;
% メソッド定義
   .method 'build/+++', "build", 3, 0x7;
'build/+++':
   create_integer R3, 1;
   if_gt R0, R3, ''true_v1m1mesh','false_v1m1mesh';
   ...;
''false_v1m1mesh': % ラベル
   create_instance R3, #'mesh';
   create_integer R4, 4;
   div R0, R4, R5;
   create_mjoint R4;
   create_mjoint R6;
   send R3, 'build/+++', R5, R4, R6;
   set_outlet R3, 'objA';
   create_instance R3, #'mesh';
   split_n R3, 1;
   ...
   send R3, 'build/+++', R7, R0, R6;
   connect R3, R8;
   connect R3, R4;
   send R1, 'v_edges/--', R5, R0;
   close R1;
   send R2, 'h_edges/--', R9, R6;
   close R2;
   descend;
   ...;
   .end #mesh;
```

図 5: コンパイル結果

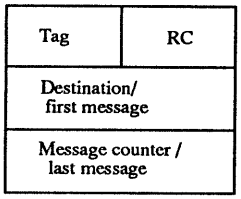

図 6: ジョイントの構造

上記のように、輸出表と輸入表を用いて、ローカルアドレスとグローバルアドレスの変換を行なうことによって、プロセッサ間での参照カウント操作のためのメッセージ送信回数を最低限に抑えている。

3.4 ガーベジコレクション

並列環境下でのリアルタイムガーベジコレクションを実現するために、重みつき参照カウント方式を用いている。一般に、参照カウント操作命令としては、参照カウントの加算と減算があるが、加算はストリームの分散実装を行なうための制御メッセージと重複が可能なため、PASでは、減算メッセージのみがある。

3.5 制御メッセージ

既に述べたようにストリームはジョイントと呼ばれるデータ構造を用いて実現されている。プロセッサ間でのストリームの接続等の処理は、相手のストリームまたはオブジェクトのアドレスが即座にわかり、それらを直接操作することが可能であるが、プロセッサ間の処理ではこれらの操作に制御メッセージが必要となる。

このような制御メッセージには、原則として以下のものがある。

- Create
  オブジェクト生成要求メッセージ
- WhereAreYou
  オブジェクトの位置問い合わせメッセージ
- IamHere
  WhereAreYou メッセージに対する返答
- Close
  参照カウント減算メッセージ

以下、制御メッセージが用いられる処理について説明する。

オブジェクトの生成

他プロセッサ中にオブジェクトを生成は以下のように行なわれる。

1. 自 PE 中に未接続のジョイントをつくる。
2. 他 PE にオブジェクト生成要求を送る。返答先は上記のジョイントとする。
3. オブジェクト生成要求を受信した PE 中はオブジェクトを作る。
4. オブジェクトのアドレスを引数として IamHere メッセージを返答する。
5. IamHere を受信することによって、接続先が確定する。この間に、このジョイントにメッセージが送信されていれば (このジョイント中に蓄えられている)、これらをオブジェクトに対して送信する。

上記のようにジョイントを一旦生成することによって、オブジェクト生成要求に対する返答を待たずに、即座にそれ以降の処理を行なうことを可能としている。

ストリームの接続

同一PE内で、ストリームの接続が行なわれた場合には、ジョイントの連鎖が作られ、ストリームに対してメッセージ送信が行なわれた時などに、必要に応じてこの連鎖のデレファレンスが行なわれる。

しかし、並列環境下でのストリームの実装では、次のような問題点が生じる。異なるPE上にある複数のジョイントが次々と接続された場合には、接続を行なわれた回数分だけ、PE間でのメッセージの転送が行なわれてしまい(異なるPE間でのポインタのデレファレンスはメッセージの到着順等の保証を考えると非常に非効率である)、メッセージの転送回数を不必要に増大させる。また並列環境下ではストリームに対する操作が、異なるPE間で全く非同期に行なわれるためメッセージの到着順を正しく保証するために何らかの手段が必要となる。

従って、分散環境下でのストリームの実装においては、以下の2点を実現することが重要である。

- PE間でのメッセージ送信回数の最小化
- (参照カウント操作メッセージを含む)効率的なメッセージ到着順の保証

上記の問題点を解決するために、PE間にまたがるストリーム操作に対しては、以下の基本手順をとることにする。

1. ストリーム同士の接続が行なわれた場合には、接続先ストリームにWhereAreYouメッセージを送信する。
2. ストリームの接続先は未確定にする。従って、それ以降このストリームに送信されたメッセージはこのストリーム内に保持され、接続先のストリームには送られない。
3. WhereAreYouメッセージに対する返答であるIamHereメッセージの受信を待つ。
4. WhereAreYouを受信したオブジェクトは自分自身の位置を引数としてIamHereを返送する。
5. IamHereを受信したストリームは、接続先をIamHereの引数とする。
6. 新たな接続先に対してメッセージ送信等を行なう。

この手順の特徴は、一つ先のストリームのみに対してWhereAreYouを送信し、WhereAreYouがそれより先のストリームに転送されないことにある。一つ先のストリーム経由でWhereAreYouを送信することによって、一つ先のストリーム中のメッセージがオブジェクトに受理された後でWhereAreYouがオブジェクトに到着しIamHereが返送されるため、メッセージの正しい到着順を保証できると同時に、一つ先のストリームしか経由しないため、メッセージ転送回数を最小化することができる。

図7を例にとり、この手順を具体的に説明する。図7において、まずストリームBとストリームCが接続されたとする。この時、C中のメッセージをBに転送せずに、Bに対してWhereAreYouを送信する。このWhereAreYouの引数はCの位置である。次いで、ストリームAとストリームBの接続が行なわれると、上記と同様にしてWhereAreYouがAに対して送信される。最後に、オブジェクトとストリームAの接続が行なわれると、A中のメッセージがオブジェクトに対して送信される。その結果、BからAに送られていたWhereAreYouがオブジェクトに到着し、その返答であるIamHere(引数はオブジェクトの位置)がBに対して返信される。IamHereを受信することによって、Bの宛先は、直接オブジェクトとなりB中のメッセージがオブジェクトに対して送信される。以下、同様にしてCの宛先もオブジェクトとなり、C中のメッセージがオブジェクトに対して送信される。

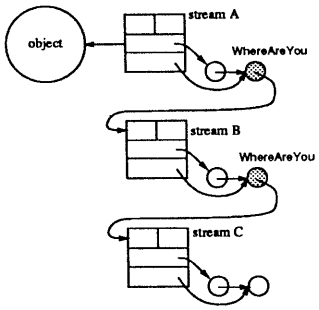

図7: ストリームの接続

上記の手順は基本的なものであり、それ以降そのストリームに対してメッセージが送信されないことが明確である場合、及びストリームが直接オブジェクトに接続された場合には、メッセージの直接転送等によるメッセージ送信回数の低減を行なうことが可能である。また、メッセージの複合化によるメッセージ送信回数の低減も行なっている。

4 評価

一般に、並列化による速度向上は、以下の2点のトレードオフにより決まる。

- 並列化したことによるオーバーヘッドによるスピードダウン
- 並列化したことによるスピードアップ

以下、並列処理に伴うオーバーヘッドと、並列化による速度向上について、前章で述べたメッシュ構造を生成する例題および素数生成プログラムに対する評価結果を示す。

4.1 並列処理のオーバーヘッド

A'Um並列版処理系(PAS)では、並列化によるオーバーヘッドとしては、

1. プロセッサ間でのメッセージ送信
2. 制御メッセージの送受信(ガーベジコレクションを含む)

の2点がある。

メッセージ送信の内外比

メッセージをプロセッサ内で送受信する場合と、プロセッサ間で送受信する場合の処理時間の比を表1に示

表 1: メッセージ送受信の比

引数の型	アトム	ストリーム
ローカル通信	1.00	1.26
グローバル通信	1.63	2.20

表 2: 並列処理のオーバーヘッド

	Primes	Mesh
PE内として処理	1.00	1.00
PE間として処理	1.37	3.45

す。表1に示した値は、

1. メッセージ生成
2. メッセージ送信
3. メッセージ受信
4. オブジェクトのスケジューリング
5. メソッド検索

の時間の和であり、それぞれ1引数の場合のものである。引数がアトム (整数) である場合の処理時間を1としている。プロセッサ間でのメッセージの送受信において、メッセージ中の引数のチェックを行ない、必要に応じて輸出入表に登録するため (このときジョイントが生成される)、ローカルな通信に較べると、約60~70% 増の処理時間となっている。ローカルな通信の処理時間とプロセッサ間での通信の処理時間の比が比較的小さいのは、プロセッサ間の通信に Symmetry の共有メモリを用いているためである。

システムメッセージ

処理の並列化に伴いシステムメッセージが必要となる場合としては、以下の場合が考えられる。

- オブジェクトを他 PE 内に生成
  オブジェクト生成要求メッセージの送信と
  IamHere メッセージの受信
- プロセッサ間でのストリームの接続
  WhereAreYou メッセージの送信と
  IamHere メッセージの受信
- 参照カウント操作
  Close メッセージの送信と受信

例題 Primes と Mesh において、プロセッサ1台を用いて以下の2種類の場合について処理速度を比較した結果を表2に示す。

1. 全ての処理を PE 内でローカルに行なう。
2. 全ての処理を PE 間での処理と同様にメッセージを用いて行なう。

Primes の場合には、並列化による処理速度の低下は、約30% と非常に小さい。これは、表3に示したように、Primes においては並列化によってほとんどシステムメッセージが必要とはならないからである。メッセージを PE 間で通信することのオーバーヘッド

表 3: メッセージ送信回数

	ユーザ	Close	W.A.Y.	I.H.	Create
PRIMES	47572	303	0	303	303
MESH	680	430	509	850	341

が約60~70% であることを考えると、全処理の約半分がメッセージ送信であることがわかる。

Mesh の場合には、並列化によるオーバーヘッドが非常に大きい。これは、表3に示すように、全ての処理をメッセージで行うことによって、PE 内でローカルに行なう場合には単純なメモリへのアクセスであった処理が、PE 間でのメッセージ送信となったためである。

Mesh において、並列化によるオーバーヘッドが大きく見えるもう1つの理由は、ストリームの機能を用いることによって PE 内でのローカルな処理が非常に高速化されているためである。ストリームの機能を用いずにこのようなプログラムを記述するためには、前章で述べたように、オブジェクトの生成の待ち合わせをしないといけない。A'Um を用いてこのようなプログラムを書くと、表3に示したものの約2倍のユーザ定義メッセージの送信が必要となり、約1.8倍の処理時間が必要となる (処理時間の増加はユーザメッセージ送信回数の増加および待ち合わせのために生じるものと考えられる)。このような場合と比較すると並列化したことによるオーバーヘッドは約87% となるり、それほど大きな値ではない。

4.2 台数効果

例題 Primes と Mesh における台数効果を図8と図9に示す。図8と図9は、1PE で全ての処理をローカルに行なった場合の性能を1としたものである。図8と図9において、実線は同一 PE 内の処理をローカルに行なった場合のものであり (こちらが通常の処理である)、点線は全ての処理をプロセッサ間の場合と同様にメッセージを用いて行なった場合のものである。

Primes の場合、並列処理の効果は最大15プロセッサの場合で約8倍である。この比は、プロセッサ1台がローカルな処理を行なった場合と比較したものであり、並列化によるオーバーヘッド約27% による性能低下が含まれている。全ての処理をプロセッサ間の処理と同様にした場合の性能を基準として考えればプロセッサ15台の場合の性能向上は約11倍であり、さらにプロセッサ数を増やすに従って性能が向上することが期待される。

また現在用いている負荷分散方式では、各プロセッサは原則的に他のプロセッサ内にオブジェクトを生成する。このため、Primes では、次段のフィルターが必ず他の PE に生成され、プロセッサ数が2以上の場合には、全ての処理が他のプロセッサとの間で行なわれている。このため、より適当な負荷分散を行なうことによって、よりよい台数効果を期待できると考えられる。

Mesh においては、15プロセッサの場合で、約2.6倍であるがこれは並列化のオーバーヘッドを含んだ場合のものである。全ての処理をメッセージで行なった場合を基準として考えれば、15プロセッサの場合で、

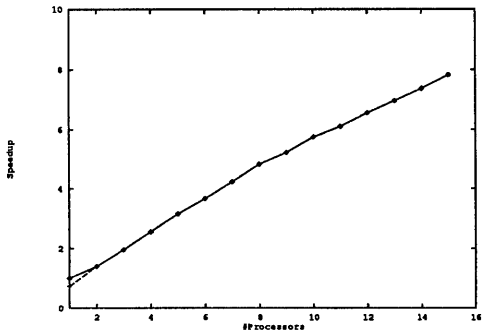

図 8: 台数効果 Primes

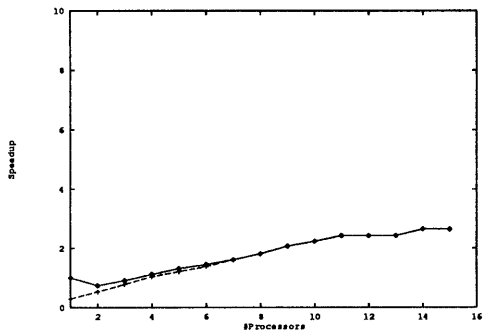

図 9: 台数効果 Mesh

約 9.1 倍の性能向上となっている。

4.3 システムメッセージの効率化

PAS では連続して送信されるシステムメッセージを最大 2 個まで、複合化して一つのメッセージとして送信している。これによるメッセージ転送の減少率を図 10に示す。図 10より約 20% のメッセージ送信が削減されていることがわかる。現在の実装では、ユーザ定義のメッセージと close 命令との複合化が行なわれていないため、そのような複合化を行なうことによってより、更にメッセージ転送回数の削減を行なうことができると考えられる。

5 おわりに

以上、A'Um の並列版処理系の概要とその評価について述べた。ストリームを用いることによって、並列処理の記述が容易になること、またストリームを用いたプログラミングにおいて十分な台数効果が得られることが確認された。

処理系は、コンパイラを含めて全て C++ で記述されている。共有メモリの存在を仮定しない実装方式をとっており、マシン依存の部分は最小限に抑えられている。現在、ソケット通信を用いた並列版処理系についてもデバッグ中である。今後、より詳細な評価、処理系の効率化、より高度な負荷分散機能の処理系への組み込み等を行なう予定である。

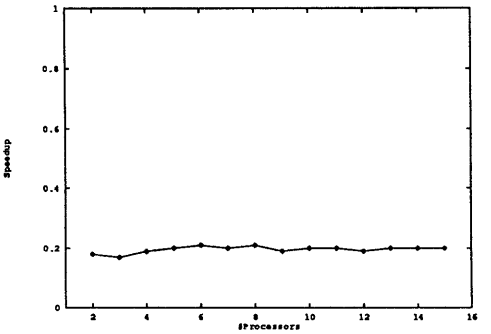

図 10: メッセージ数の削減

謝辞

本処理系の作成に協力して頂いた日本電気技術情報システム開発 (株) 小柳氏、丹下氏に深謝致します。

参考文献

[1] K. Yoshida and T. Chikayama, "A'UM – A Stream-Based Object-Oriented Language –," Proc. Int'l Conf. on Fifth Generation Computer Systems (FGCS '88), pp. 638-649, ICOT, November 1988.

[2] N. Carriero and D. Gelernter, "Linda in Context," pp444-458, CACM, Vol.32, No.4, 1989.

[3] 小西、丸山、小長谷、吉田、近山、「並列オブジェクト指向言語 A'UM-90 」, JSPP'90 論文集, 1990 年 5 月

[4] 丸山、小西、小長谷、吉田、近山、「ストリームに基づく並列オブジェクト指向言語 A'UM-90 - ストリーム分散実装方式」、情処学会第 4 1回全国大会講演論文集、2E-2, 1990 年 9 月

[5] 小西、丸山、小長谷、吉田、近山、「A'UM-90 のストリームの分散実装方式 」, JSPP'91 論文集, 1990 年 5 月

[6] S. Watanabe, Y. Harada, K. Mitani, E. Miyamoto, "Kamui88: A Parallel Computation Model with Fields and Events," pp153-175, Advances in Software Science and Technology, Vol.2, 1990, JSSST.

付録: 素数生成プログラム Primes

```
class aum.
:go(^Argv,^Env) ->
  #prime:primes(2000, #tio).
end.
```

```
class prime.
:primes(^Max, ^Tio) ->
    Tio:print(2):print(3)=^NewTio,
    :generate(3+2,
              Max,
              #filter:init(3,NewTio)).
:generate(^X,^Max,^Ns) ->
    (X < Max) ? (
      :'true  ->
         :generate(X+2,Max,Ns:n(X));
      :'false ->
    ).
end.

class filter.
  out me, next, to_next, tio.
:init(^V, ^Tio) ->
  V = !me, 0 = !next, Tio = !tio.
:n(^X) ->
  ((X mod !me) == 0)? (
    :'false ->
      (!next == 0)? (
        :'true ->
          !tio:print(X = !next),
          #filter:init(X,!tio)
                        = !to_next;
        :'false ->
          !to_next:n(X)
      );
    :'true  ->
  ).
end.
```