計算機アーキテクチャ 77－1
（１９８９．７．１３）

並列推論マシン PIE64 のシステムソフトウェア

吉田 実, 田中 英彦
東京大学工学部

概要

従来からコミッティッドチョイス型言語とシステムプログラミングの親和性の高さは指摘されてきた。並列オブジェクト指向言語 Fleng++ は、この特徴をストリーム通信を行なうプロセスをオブジェクトとして捉えることにより、さらに発展させ、プログラムの表現能力、可読性を高めた言語である。この言語を中心に高性能な I/O ディバイスを持つ並列推論マシン PIE64 上のシステムソフトウェアについての検討を行なった。

System Programing on Parallel Inference Engine PIE64

YOSHIDA Minoru and TANAKA Hidehiko

Department of Electrical Engineering, Faculty of Engineering,
University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo 113, JAPAN

Abstract

It has been pointed out that committed-choice languages are suitable for system programing. Fleng++ is a parallel object-oriented language based on a committed-choice language Fleng. Fleng++ shows Fleng processes as objects and improves expressional power and readability. In this paper, system software on parallel inference machines is discussed focusing on Fleng++. The parallel inference machines are supposed to have high performance I/O devices.

IU間ネットワーク 4 × 4のクロスバスイッチ (SU chip として1つのLSIにまとめられている) が3段で回線交換の64 × 64のオメガ網をなす。そのネットワークが2系統ある。2系統は全く同じもので、システムが自由に使い分けることができる。各ネットワークは、回線接続に要する時間が1μsec、接続状態の転送量は40MByte/secである。 **I/Oインタフェース** SCSIが8系統である。各SCSIは、IU 4台、DM 1台に継っている。よって、最大I/O転送量は、4MByte/sec × 8 = 32Mbyte/secである。

表 1: PIE64のネットワーク

1 はじめに

我々の研究室では並列推論マシン PIE64[Koike88] の開発をすすめている。PIE64 は大規模記号処理を高速に実行することを目標にしている。PIE64 と複数のパスを通じて通信し合う高性能I/Oディバイスは一体化して一つのプログラムを実行する。このようなハードウェアを制御するOSの記述言語は、プログラム自身の実行の制御、コード、データの物理的な位置の管理などができなければならない。我々は、OSの大部分を記述する能力を持ち、なおかつユーザプログラムを充分容易に書けるプログラム言語として、並列オブジェクト指向言語 Fleng++[Nakamura88a] の開発を進めている。Fleng++ はコミッティッドチョイス型言語 Fleng[Nilsson86] のストリームを利用したプロセス間通信をオブジェクトへのメッセージパッシングという形で自然に記述できる。

2 PIE64のアーキテクチャ

PIE64は、64台の推論ユニット (IU) と回線交換の密結合ネットワークなどで構成される。IUは、推論プロセッサ (UNIRED)、ネットワークインタフェスプロセッサ (NIP)、SPARC、FPUなどから構成される。ネットワークは、4×4のクロスバスイッチを持つSU chip 3段で構成する。ネットワークは2系統あり、ハードウェア的には全く同じ能力を持っている。その他に全てのIUには、ホストマシンとの通信を行なうホストインタフェス、32個のIUには、I/Oディバイスとの通信を行なうI/Oインタフェスがある。

I/Oディバイスには、高性能な関係データベース処理能力を持った8台のディスクモジュール (DM) で構成されるデータベースマシンがある。DMは、プロセッサ、ソータ、2台のディスクなどで構成される。各DMは、IUボードへのインタフェスを持っている。

I/Oインタフェスは、8系統のSCSIからなる。各SCSIには、4台のIUおよび1台のDMが継る。

全体の構成は図1のようになっている。ネットワークについては、表1に示す。

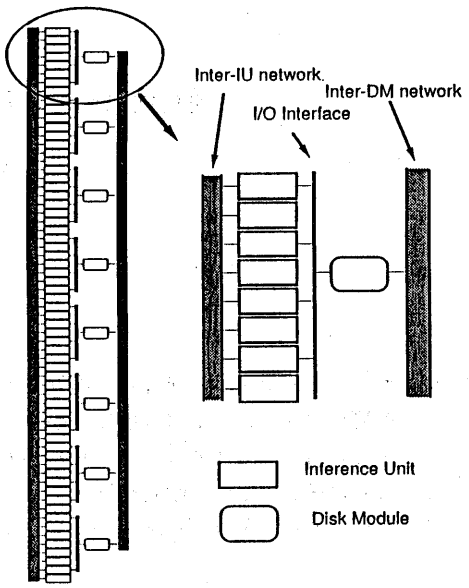

図 1: PIE64の全体構成

3 並列オブジェクト指向言語 Fleng++

Fleng++ は、システム記述もできる記述能力の高い言語として設計されている。ここでは、Fleng++ の仕様およびそのインプリメントの概略を述べる。

3.1 Fleng++ の概要

Fleng++ は、コミッティッドチョイス型言語 (CCL) であるFlengを基にオブジェクト指向パラダイムを導入したものである。現在のところ、Fleng++ は Fleng−− にコンパイルして実行される。Fleng−− は、Flengにアノテーション等の機能を付け、より低レベルの実行時処理の指定ができるようにしたものであり、Flengのスーパセットになっている。

3.2 Fleng−−

Fleng−− は、Flengにアノテーションの機能を付け加えたものであり、並列記号処理のアセンブラを目指したものである。Fleng自体は、大まかに言えば、GHCからガードをなくし、さらに、ゴール間の論理的なAND関係を取り除いたものである。GHCに較べ、必要と思われる機能を絞り込み、その分処理系を軽くて速いものにすることが目的である。

アノテーションは、active unify("!")、single reference (" ' ")、bind to non-variable("#") などがある。このうち、single reference annotation は、そのセルが他のゴールから参照されていないことを示すもので、そのセルには破壊的代入を行なえ、コンパイル時の最適化に役立つ [Koike89]。bind to non-variable annotation は、その引数が具体化されるまでゴー

ルをサスペンドさせる。逆に、active unify annotation は、引数とそこに書かれているものをコミットされた後にユニファイするものである。bind to non-variable annotation と active unify annotation は、それぞれ、モードが入力、出力であると言うふうに読むことができる。

3.3 Fleng++ の言語仕様

Fleng++ は、クラスという単位でプログラムを記述する。述語にはクラス内でしか参照できないものと外からも参照できるものがある。前者をローカル述語と呼び、後者をメソッドと呼ぶ。クラスは継承機能を持ち、他のクラスの定義を取り込むことができる。また、何度でも代入することのできる変数を持っていて、これをインスタンス変数と呼ぶ。クラスは、システムの上限値まで任意の数のインスタンスオブジェクトを作ることができる。このインスタンスオブジェクトがそれぞれ独立したインスタンス変数を持つ。このクラス単位のプログラムのモジュール化、階層化の機能によって Fleng++ はプログラム開発の効率化をはかっている。

クラスの定義の例を以下にあげる。

```
class ex
    inherit stdio
    var $v := 0.
  :p(Self,V) :-
      V > $v -> $v := V.
  1(X) :- X = a.
end.
```

はじめにクラス名を宣言し、継承クラス、インスタンス変数を、必要なら宣言する。インスタンス変数は、"$" で始まる。":" で始まるのはメソッド定義である。これに対し、ローカル述語は ":" で始まらず、普通の Horn 節で定義される。ここで出てくる "->" という記号はマクロで矢印の左側が true の時に右側の部分が実行される。マクロについての詳細は後述する。

3.3.1 オブジェクトの生成

オブジェクトの生成は、

```
Obj = #ex
```

とする。クラス名の前に # を付けたものを、コンストラクタと呼ぶ。クラス ex のインスタンスを作り、その:p というメソッドを呼ぶのなら、

```
Obj = #ex, :p(Obj,A)
```

とする。この場合、メソッド呼び出しが 1 つだけなら、

```
:p(#ex,A)
```

と書くこともできる。

3.3.2 マクロ機能

マクロ機能はよりプログラムを書きやすく、読みやすいものにするために加えられた機能である。Fleng++ だけでなく Fleng(--) でも使用できる。GHC ライクなガードと Prolog ライクな if-then-else がある。ガードは、

```
Cond_1 | Goals_1;
Cond_2 | Goals_2;
.
.
Cond_n | Goals_n;
```

のように記述する。Cond_i($1 \leq i \leq n$) のうちで true になったものの 1 つが非決定的に選ばれそのゴールが実行される。if-then-else は、

```
Cond -> TrueGoals; FalseGoals
```

または、

```
Cond -> TrueGoals
```

のように書く。Cond が true なら TrueGoals が、false ならば FalseGoals が実行される。これらのマクロは簡単に Fleng のプログラムに展開できる。

3.3.3 継承

オブジェクト指向言語と呼ばれる言語はほとんど継承の機能を持っている。Fleng++ も多重継承の機能を持ち、複数のクラスを継承することができる。継承順は、depth first up to join と呼ばれる方式である。これは右から左へ深さ優先でクラスの階層をたどるが、サブクラスはスーパクラスより必ず先にたどられるというものである。コンパイル時のオプションにより depth first right to left でたどることもできる。

3.3.4 メッセージの直列化

Fleng では各ゴールは独立に実行される。実行順序を決めるものは、中断・再開の機構とリダクションによって新たなゴールが生成される過程のみである。Fleng++ では、それ以外にメッセージの受信とメソッド起動に関する順序付けがある。オブジェクトが副作用を持つため、メッセージの到着順序によって実行結果が異なることがあり、メッセージの順序を考慮しなければならない。Fleng++ では、このメッセージの順序付けを arrival ordering と呼び、「逐次解釈、並列実行」という原則を設定した。プログラムは左から右へ深さ優先で読むことができる。しかし、直列化が必要なところを除き、実行は並列に行なわれる。

3.4 Fleng++ のインプリメント

現在、Fleng++ は、Fleng-- のプログラムを出力するコンパイラがインプリメントされている。コンパイラの出力した Fleng-- のプログラムを Fleng-- 処理系に実行させている。Fleng++ コンパイラは SICStus Prolog で書かれていて、また、Fleng-- 処理系は C 言語で書かれていて、それぞれ、Sun Workstation などで実行することができる。ここでは、Fleng++ コンパイラのインプリメントの概略を示す。なお、ここで紹介されている Fleng-- のコードは、説明のために単純化してあり、実際のコードとは異なっている。

3.4.1 クラス定義の枠組

ここでは、Fleng++ のコードがどのようにコンパイルされるかを簡単に述べる。

例として、以下のようなクラス定義があったとする。

```
class a.
  :msg1(S,A) :- ...
```

```
  :msg2(S) :- ...
    .
    .
end.
```

これをコンパイラは次のようなコードに変換する。

```
a([#M|Self],Super) :-
    my_method(M,[msg1(_,_),msg2(_),...],Result),
    a_aux(M,Result,Self,Super).
a_aux(msg1(Arg1,Arg2),_,Self,Super) :-
    ... , a(Self,Super).
a_aux(msg2(Arg1),_,Self,Super) :-
    ... , a(Self,Super).
  .
  .
a_aux(M,false,Self,Super) :-
    Super = [M|Super1], a(Self,Super1).
```

my_method/3 は受け取ったメソッドが自分のクラスで定義されているかどうかを調べる述語である。自分のクラスで定義されていないメソッドの時は、第3引数が false になり、スーパクラスにメッセージを渡す。新しいインスタンスオブジェクトの生成は、

```
a(A,O),object(O)
```

とする。object/1 は、全てのクラスがインプリシットに継承するクラス object を実現する述語である。a→b→c の順に継承していれば、上の例の代わりに

```
a(A,B),b(B,C),c(C,O),object(O)
```

とすればよい。

3.5 メッセージの送信

メッセージの送信は、送信するオブジェクトのストリーム S を

```
S = [msg|S1]
```

のように具体化すればよい。メッセージ送信用の新しいストリームは S1 になる。Fleng++ のメッセージ送信では、第1引数に送信先のオブジェクト自身が渡される。そのような場合のメッセージ送信は、まず、ストリームを分岐させ、一方のストリームを送信先のオブジェクトに、他方を自分で使うオブジェクトに使用する。つまり、

```
S = [msg(S1)|S2],merge(S1,S3,S2)
```

のようにする。

送信先が自分自身の時は直列化のために、

```
S = [msg(S1)|S2],append(S1,S3,S2)
```

のように append/3 を使用する。そのメソッド内で使われなくなったオブジェクトはストリームを閉じることによって捨てられる。

3.5.1 部分計算による最適化

Fleng++ の出力コードの最適化前の状態では、無駄な append、merge が現れる。例えば、append の第1引数がわかっていたら、

```
..., append([A1,A2,...],B,C), ...
```

のかわりに、

```
C = [A1,A2,...|B]
```

とすることができる。このような部分計算は比較的少ない手間で計算の効率を大きく上げることができる。

4 PIE64 の OS

4.1 概要

プログラムの独立実行の単位は、実行モジュール (Execution Module, EM) と呼ばれるもので、この単位で、EM 間のスケジューリング、EM 間の通信などの複数の EM 間に渡る処理を除く部分は実行される。EM は、自分の名前、コード / コードテーブル、共有ストリーム、シンボルテーブル、リフレクティブオブジェクトテーブル、EM ディレクトリなどからなる。また、本 OS には、リフレクティブなオブジェクトを導入した。これは、Fleng++ からは優先通信を受け付けるオブジェクトに見える。この2つが、実行時の動的な実行制御の単位となる。

4.2 目標

PIE64 の OS 開発が一つの目的であるが、なるべく広い範囲の並列マシンで応用できる技術を開発することが目標である。

並列推論マシンの OS には、以下の管理をする必要がある。

ゴール管理

ゴールは粒度のかなり小さい並列実行の単位である。推論マシンでは、CPU 資源の評価単位としてゴールのリダクションの回数を使うのが実用的である。実行時に、並列性を十分生かし通信コストのなるべく少なくなるようにするのは、大部分 OS の責任である。その際、ゴール管理と次の項で述べるメモリ管理が重要になる。

また、PIE64 の IU のように全ての IU が同じ機能を持っている訳ではないような状況では、ゴールをどこに送るかは、単に負荷を均等にばらまけば良いというだけではない。その機能を実行できる IU にゴールを送らなければならない。

メモリ管理

メモリ領域はデータ領域と実行コード領域に分かれる。データ領域はほとんどヒープ領域として使われ、GC によって不要になったセルは回収される。実行コードは、OS の常駐部分以外は主メモリ上から追い出される可能性がある。また、並列マシンでは、シングルプロセッサマシンに見られるメモリの階層以外に、メモリのある場所によって同じディバイスを使っていてもアクセススピードに差が出るのが普通である。メモリ領域が必要になった時、それをどこに確保するかは大部分 OS の責任である。

シンボルテーブルの管理

記号処理に使うシンボルと文字列の対応をシンボルテーブルとして管理する。効率的に処理できるシンボルのための識別子の数は有限であるから、これを管理してやる必要がある。あるユーザプログラムが、シンボルの識別子を喰い潰すようなプログラムを実行したり、シンボルの識別子が単調増加するようになっていると識別子が足りなくなる場合が考えられる。

その他の資源管理

上記以外の資源として、I/O などを管理する。

目標は、以下の通りである。

並列性の自然な抽出・IU 間通信の抑制

並列マシンにおいては並列性の自然な抽出は重要である。PIE64 で採用した Fleng++ は、その下のレベルでは Fleng のゴールに相当するものが考えられる。ゴールという単位はかなり粒度が小さい。これを無条件にばらまいていたら、PIE64 ぐらいの中程度の並列マシンでは、プロセッサ間の通信を増やすことになり、高並列のマシンでは遠距離通信が増え、結局のところ、アーキテクチャによらず通信コストを増大させてしまう。通信の多いゴールは近くに、少ないゴールは遠くにという戦略をとりやすいようにする工夫が必要である。

強力なメモリ管理

推論マシンのメモリ管理の中心は GC である。高速でリアルタイム性の高い GC やオーバヘッドの少ない実行時回収を行なう GC が必要である。しかし、ゴールの制御機構を使えば、GC をしてもメモリが回収できなくて破綻するようなプログラムを書くことができるそのようなプログラムによっても OS がダウンしないような工夫が必要である。

マルチユーザ、ネットワーク環境

MIMD 型の高並列マシンは、巨大な CPU パワーを提供するものである。一人のユーザが独占する様な性質のものではない。また、大規模な記号処理プログラムでは、複数の人間が共同で開発し利用するのが一般的である。マルチユーザシステムは、実験機としては複雑すぎる面があるかも知れないが、少なくとも目標としては掲げねばならないし、そのための技術的問題点を明らかにし、解決する必要がある。

一方、ネットワーク環境で使えることも重要である。自分の手元のワークステーションから気軽にログインできて、また、ウィンドウアプリケーションも十分そろっていてこそ、知的な活動もできるというものだろう。

持続性

OS の運用に当たって、単調増加するようなものがあり、無限まで増加可能なインプリメントをしていないとやがて破綻をきたす。特に問題になるのは、シンボルテーブルのサイズおよび識別子の数が単調増加する時だろう。また、ユーザのミスなどで、シンボルテーブルを大量に使うプログラムが走ると困る。

4.3 リフレクティブオブジェクト

リフレクティブオブジェクトと次の節で述べる実行モジュールは、PIE64 の OS の基本的実行制御の単位である。リフレクティブオブジェクトは比較的粒度の小さい実行単位であり、実行モジュールはプログラムの独立実行を行なうための単位である。

4.3.1 リフレクティブオブジェクトとは

リフレクティブオブジェクトとは、自分の実行を監視しながら計算を進めるオブジェクトのことである。リフレクティブオブジェクトは、そのオブジェクトの実行を行なっているゴールの集合からなる。本 OS では、このオブジェクト単位で、実行を停止、強制終了、優先度の変更を行なうことができる。このオブジェクトは、プログラムの細かな実行制御を司る。

4.3.2 リフレクティブオブジェクトのインプリメント

リフレクティブオブジェクトは、そのオブジェクトが生成される時に Fleng−− に新たに付け加えられたメタな機能であるゴールグループ生成述語 gg を用いる。つまり、Fleng++ のリフレクティブオブジェクトは、Fleng−− からは、ゴールグループとして捉えることができる。例えば、max というクラスのインスタンスを作る時には、

```
max(O,P),object(P)
```

とするかわりに、

```
gg([max(O,P),object(P)],S)
```

とする。gg の第2引数 S は、ゴールグループの制御用のストリームである。

ゴールグループ (GG) は、後述する実行モジュールに比べライトウェイトな制御単位である。従来、Fleng(−−) では各プロセッサに一つのアクティブゴールキューを持ちそれを頭から実行していた。GG の導入にあたって、ゴールキューの数を複数にした。GG の生成時に GG の識別子 (GGid) を受け取り、以後、IU の外にゴールを投げる時にはその GGid を一緒に付ける。受け取り側は、その GGid のキューに受け取ったゴールを加える。GGid は、そのゴールが複数の IU に渡るときは、その IU 間で一意になるように管理しなければならない。その管理のオーバヘッドは、特に、高並列マシンでは問題になる。

GGid の管理に当たっては、GG の実行される IU をソフトウェア的に階層化して、全体の負荷やプログラムの並列度によってどの階層を使うか決めるのが良い。一度決めた階層をかえるのは、それほど大変ではないので、動的に全体の負荷を見て変化させることも可能である。

GG のメタコントロールとしては、停止、実行再開、強制終了、優先度変更などが考えられる。停止は、単にその GG にリダクションの機会を与えなければ良い。強制終了はその GG をテーブルから除く。次の GC の時にメモリ上から除かれる。また、GG の強制終了時に GG の使っていた領域を即時回収することもできる。

4.3.3 オブジェクトの保存

オブジェクトは、内部状態を持つのでそのスナップショットをコピーしたり、ディスクに保存することができる。その場合、内部状態、つまり、インスタンス変数の内容のみをコピーしたり保存する。Fleng++ におけるオブジェクトの実行は、インスタンス変数を引数として保持し、逐次的に解釈してゆく解釈部とその解釈に基づいた実行を行なう実行部がある。スナップショットをコピーする時には、実行部は実行が終了するまでそのまま実行を続け、インスタンス変数の保持する解釈部はその時点のインスタンス変数の内容を保存する。

4.4 実行モジュール

実行モジュール(EM)とは、プログラムの独立実行の単位である。UNIXのプロセスに近いイメージである。リフレクティブオブジェクトが、立ちあげの軽さを狙っているのに対し、EMは立ちあげは重くなるが、コースグレインな実行を受け持ち、EM間の通信は特別な場合を除き比較的少ないように設計される。EMは、以下の属性を持つ。

自分のEM名 EMは名前で参照されるので名前を持っている。

シンボルテーブル そのEMにローカルなシンボルは、EM内で持っている。EM実行の終了と共にこの領域は解放されるので、シンボル識別子を喰い潰すようなプログラムを書いたとしても、その影響はそのプログラムを実行するEM内に抑えられる。一方、システム述語の述語名、引数などに現れるシステムで良く使うシンボルは、予めグローバルシンボルとして定義されている。これにより、システム述語の実行時やEM間通信でシンボル識別子の変換のオーバヘッドを減少させている。

GGテーブル そのEMに属するゴールグループのテーブルである。

EMディレクトリ EMの親子関係を保持している。

4.5 EMのメモリ管理

データ領域、コード領域を大量に消費するEMが走る可能性がある。誤ったスケジューリング指定、停止など何らかの理由でストリーム通信をするプロセスの受信側が動かなくなり、送信側がどんどん送ってしまった場合などが考えられる。また、バグではなくてもメモリを大量に使用するプログラムを実行するような場合も考えられる。そのようなユーザEMを放置すると、GCが頻繁に起こり、かつ、GCによるメモリ回収率も悪くなり、やがて、システムのダウンを招くことが考えられる。

EMはメモリ管理の単位でもある。EMと他のEMとの通信はEM間通信のストリームを通じてしかできないようになっている。そのストリームを管理することによって、GCのマーキングフェーズの時、比較的小さなオーバヘッドで使用メモリ量を調べられる。我々は高並列並列推論マシンにおけるメモリ使用量による管理がどの程度重要なものであるかの知見はほとんど得ていない。しかし、少なくともマルチユーザ、マルチタスクOSを実用的に稼働させるためにはEMのメモリ使用量を知り、CPU使用量と共に管理する必要があると考えている。

4.5.1 EMのスワップアウト

前節ではメモリ使用量を知る方法を述べたが、その使用量を基にEMを管理する方法がなければ、ディバックや性能測定などにしか役立たない。メモリを大量に使っているEMは、停止させるとそれ以上のメモリ消費はないが、既にメモリを多く使っているので、他のEM実行の障害になる。強制終了させるのは、スナップショットを2次記憶に退避しておき、後でその時点からの実行再開が可能ならば良いが、そうでない場合には大きな無駄を生じることになる。そのような場合に、EMをスワップアウトできるとシステムの効率は非常に良くなる。EMはスワップアウトしやすいようにEM間の通信は経路を限定している。

EMのスワップアウトは、コンパクションGCの時に行なうと効率が良い。マークフェーズの時に各EMの使用メモリを調べ、その結果などからスワップアウトするEMを決める。そのEMをスワップアウトとした後、コンパクションを行なう。

4.6 全体の管理

全体で管理しなければならないものは、コード、EM間通信、CPUなどの資源といったEM間で共有するものである。

4.6.1 コード / コードテーブル

コードは、クラス単位で管理される。全てのIUが同じコードを同じアドレスに持つわけではない。静的に実行状態を決定できないような複雑な計算を行なう高並列計算機においては、全てのIUに同じコードを置くと、制御上は楽かも知れない。しかし、大きなコードを保持するためには、そのためにコード領域が増え、その分、データ領域を削ってしまう。特にファイングレインの並列計算機においては、プロセッサ1台当たりのメモリ量がそれほど多くないのでこの問題は重要である。

本OSでは、コードは全てのIUで同じものを持つわけではない。そのため、コードの状態をより細かく管理するためにコードテーブルをもつ。コードテーブルは、コードがどのような状態にあるかを保持している。コード状態を表す属性は、メモリ上の状態、そのコードを参照しているEMの数などである。メモリ上の状態とは、そのコードを、1) 全IUが保持している、2) 保持しているIUはなく完全にスワップアウト状態である、3) 保持しているIUもあるなどである。PIE64の場合は、この情報はコード当たり64bitで保持できる。あるIUが実行のためのコードが存在しないことに気づくと、他のIUがそのコードを持っているかどうかを調べる。もし、なければI/Oディバイスから読み込まなければならない。あれば、IUの忙しさ、ネットワークトラフィックなどから、他のIUから読み込むか、I/Oディバイスから読み込むか決定する。また、コードを完全にスワップアウトするとI/Oディバイスからそのコードを呼び込むのに時間がかかるので、なるべく1つのIUにはコードが保持されているようにする。

PIE64では、全てのIUがハードウェア的に全く同等というわけではない。つまり、直接、I/Oディバイスと接続されていないIUが存在している。そのIUで実行できないI/O操作をしなければならなくなった時には、実行可能なIUを捜し処理を依頼する必要がある。

4.6.2 EM間の共有ストリーム

EM間はストリームを通じて通信をする。ところが、未定義変数の取り扱いを工夫しないと、スワップアウト時に問題が生じる。まず、EMからの非変数の参照はそのままディスクへコピーできる。しかし、未定義変数があり、その変数を他のEMと共有している可能性があれば、コピーという戦略にでることは簡単にはいかない。つまり、他のEMと共有している

未定義変数へのポインタの扱いが問題になる。未定義変数のあるセルは、スワップアウトされている間にGCが行なわれると、アドレスが変更される。また、その変数を共有しているEMがその変数を参照しなくなったら、そのセルはなくなってしまう。しかし、スワップアウト時のGCをディスクにまでおよばせるのは現実的ではない。それを避けるためには、他のEMを参照しているポインタは主メモリ上に残しておかねばならないが、制御も難しく、そのポインタの分のメモリも消費される。そこで何らかの工夫が必要になる。まだ、決定版の方法はないが、ここでは考えられる戦略を示す。

簡単な方法は、ストリーム通信の間に監視用のオブジェクトを一ついれ、お互いに厳しい入出力ルールを決めて、相手に渡す時はグランドに落ちるまで待つという方法がある。例えば、以下のようなルールを設定する。

1. ストリームを開いた方がイニシアティブをとり、ストリームの構造の具体化を行なう。
2. ストリームの構造は、
   `[(A1,B1),(A2,B2), ... |ABx]`
   の形に限定する。この場合、ストリームの構造を決定するとは、
   `S = [(A,B)|AB]`
   のようなユニファイで実現できる。
3. 監視用のオブジェクトは、ストリームの構造ができたら、そのコピーを作り、スレーブ側のストリームをそのコピーで具体化する。
4. マスタ側のストリームの変数Aがグランドにおちるのを待ち、スレーブ側のストリームにユニファイする。
5. スレーブ側の送信のための変数がグランドにおちるのを待ち、マスタ側のストリームの変数Bとユニファイする。

つまり、この戦略ではグランドにおちたものしか通信されないから、スワップアウトの時に全てコピーできるようにするものである。このような方法は、EMという大きな実行単位での同期を実現するには、比較的適していると思われる。また、内部ネットワークに較べ圧倒的に低速なLANなどの外部ネットワークを通じて他のOSとの通信も簡単にできる。

相互の通信が非常に頻繁で上記の戦略をとっていたら遅くなってしまうような場合は、リフレクティブオブジェクトを使う。

巨大なデータ構造をEM間で共有して使用する場合は、あるEMのオブジェクトとして実現し、各EMはそのオブジェクトへメッセージを送ることにより、データをアクセスする。この場合、そのオブジェクトのストリーム自体は共有しなければならない。

4.7 EMに対するリフレクティブな操作

EMのスワップアウトは、メモリの有効利用をはかったり、高負荷時のダウンを防ぐための要求駆動的な方法であった。しかし、EMのディスクへの退避を明示的に指示してに使うことによって、入出力を行なうことができる。また、ディスク上にしか存在しない巨大なEMを作ることもできる。このEMの実行は、PIEの主メモリ上に持ち込まれることはなく、I/Oディバイス側で行なわれる。巨大な構造を持つが、アクセスされるのはそのうちのごく一部であるような時に適している。

この機能により、従来のOSのファイルとプロセスを統一的に扱える。

4.8 シンボルテーブルの管理

シンボルテーブルはシンボルの識別子と文字列を対応付けるものである。このテーブルは、従来の処理系では、一元的に管理し、かつ、単調増加するものが多かった。しかし、そのような管理方法は以下の点で不都合である。

ユーザプログラムからのプロテクション
　あるユーザプログラムがストリングテーブルを消費するようなことをした時の影響がすべてのプログラムに及ぶ。

OSの連続稼働
　OSが連続稼働している内にやがてシンボルテーブルが足りなくなってしまう。

2次記憶への退避
　2次記憶に退避されたEMがあると、OSはGC、その他、シンボルテーブルの項を減少させることが困難になる。

上記の問題点の内、OSの連続稼働についてはシンボルテーブルのGCを実行することによって回避することができる。つまり、GCのときに少し手間をかけて、シンボルが使われていたらそのシンボルテーブルのエントリにマークを付け、マークの付いていないシンボルは回収して再利用することができる。ただし、シンボルテーブル自体のコンパクションはかなり重い仕事になるので普通は行なわない。しかし、通常、コンパクションは必要ない。

ところで、EMのスワップアウトについて既に述べたが、スワップアウトしたEMの分までシンボルテーブルに含めることは現実的でない。EMの積極的なスワップアウトまで含めるとなおさらのことである。全体で1つのストリングテーブルしか持っていなければ、スワップアウトしたEMがある限り不用意にシンボルテーブルのGCを行なうことができない。そのようなことを考えると、EM内で使うシンボルのシンボルテーブルは、そのEM内で管理するのが妥当である。そして、EMのスワップアウトと一緒にそのシンボルテーブルもスワップアウトされると問題はない。

5 負荷分散、データ分散に関して

負荷分散は並列計算機の性能を引き出す上で非常に重要なことは言うまでもないが、その際、ただ負荷が均等であれば良いわけではない。実行に必要な通信コストが少なくなることも重要である。それゆえ、実行すべきゴールを負荷の少ないところに無差別に投げるのは、特殊な場合を除けばあまり良い方法ではない。実行前に静的に最適化することができるような問題を扱うことは比較的少ないであろうし、バッチジョブのような使い方は考えていない。よって、本OSでは静的には、ある程度の最適化を行ない、実行時の最適化のための情報を作り、動的にはその情報を基に動的負荷分散を行なう。

動的な負荷分散の際の情報として考えられるのは、EM、オブジェクト、オブジェクトの送信者、受信者関係の有向グラフ、コンパイラの出す情報などである。EM間は、ソフトウェア的には、特定の通信経路しか持っていないので、通常、大きな

通信が生じることはないので、EM の使う IU を負荷に応じて限定するとよい。例えば、同じぐらいの計算量や優先度を持つ EM が同時に実行されていれば、どちらも 64 個の IU を使って実行するより、32 個ずつに分けた方が通信コストは少なくなるだろう。オブジェクトも EM に近いが特性を持つ。つまり、内部の通信の方が外部の通信より頻繁であることが多い。プログラミングスタイルにもよるが、PIE64 では、このオブジェクトを並列実行の単位にしても十分な小さな粒度を得られると考えている。オブジェクト実行の並列化の効果は、もっと、高並列の計算機でないと効いてこない可能性が高い。

6 Fleng++ コンパイラ第 2 版

現在の Fleng++ 処理系は、Fleng-- のプログラムを出力するものである。これは実験的なインプリメントであり、Fleng++ のプログラムがきちんと Fleng-- に変換できることを示したことに意義がある。また、ソフトウェア開発用としても活用されている。しかし、実行効率、ディバッグなどに問題点がいくつか存在する。

実行速度が遅いこと
これは、現在 Fleng-- 処理系の高速なものがないせいもある。そのようなものが得られるとかなり速くなるだろう。しかし、Fleng++ 特有の実行順序制御などが絡む部分は、直接、Fleng-- より低いレベルに落すコンパイラの方が高速に実行できるだろう。

メソッドサーチがメソッド数のオーダである
メソッドサーチは、オブジェクト指向言語ではかなりの時間を占めることが知られている。Fleng++ でも、メソッド呼び出しは頻繁に生じる。現在のインプリメントはリニアにメソッドを探しているが、ハッシュなどを使って定数時間で探すことができるようにしたい。

ディバッガが作りにくい
完全に Fleng-- におとしてしまった後で、オブジェクトレベルのディバッガを作るのは難しい。Fleng++ でのディバッグの中心は、ゴールのリダクションよりストリームを通じてのメッセージパッシングとその結果のオブジェクトの内部状態の変化などである。このような情報はコンパイラが持っているので、それを利用できるようにしたい。

これらのことを踏まえて現在、新しい Fleng++ の処理系を開発中である。この処理系は、SPARC へのネイティブコンパイラであり、以下の機能を持つ。

メソッドサーチの高速化
メソッドサーチは、ハッシュを用いてコンスタントオーダで高速に実行する。

インクリメンタルな開発環境
クラスをプログラムのコンパイルの単位にとる。クラスを変更変更した場合は、そのクラスを再コンパイルするだけですぐにプログラム全体が実行可能になる。ソフトウェア開発、特にプロトタイピングの場合は、実行・ディバッグ・再コンパイルのサイクルの短さが重要である。

実用的な処理速度
この処理系では究極の処理速度は求めない。しかし、簡単に速くなる部分と処理系の性能に大きな影響を与えるところは十分最適化する。

ディバッガの支援
少なくとも、あるオブジェクトへのメッセージとそれによって生成されたオブジェクト、外に投げられたメッセージ、内部状態の変化を監視する機能が必要だろう。他の大部分は、Fleng のディバッガと同じようにすればよいと考える。

7 終りに

現在、Fleng++ コンパイラ第 2 版と OS の実験バージョンを開発中である。OS に関しては、現在、我々は本格的なインプリメントを行なえるマシンを持っていないので、いくつかの機能を確かめるためとユーザインタフェース関係のインプリメントが中心になるだろう。Fleng++ は、SPARC 上で動くネイティブコンパイラを検討している。Fleng++ は UNIRED の仕様が固まりつつあるので、それを反映したより PIE64 上での動作を意識した処理系にして行きたい。

参考文献

[Koike88] Koike,H. and Tanaka,H.: *Multi-Context Processing and Data Balancing Mechanism of the Parallel Inference Machine PIE64*, In Proc. of The Int. Conf. on Fifth Generation Computer Systems, ICOT, 1988.

[Nakamura88a] 中村宏明 : "*Committed-Choice* 型言語に基づいたオブジェクト指向プログラミング・システムの研究", 東京大学工学系研究科情報工学専攻修士論文, 1989.

[Nilsson86] Nilsson,M. and Tanaka,H.: "*Fleng Prolog - The Language which turns Supercomputers into Prolog Machines*", In Wada,E. (Ed.): Logic Programming '86, LNCS 264, Springer-Verlag, 1986.

[Koike89] 小池, 田中: "単一参照アノテーションを用いた論理型言語の最適化コンパイル", 情報処理学会第 38 回全国大会, 6Q-1, 1989.

[Chikayama88] T. Chikayama, H. Sato and T. Miyazaki : "*Overview of the Parallel Inference Machine Operating System(PIMOS)*", In Proc. of The Int. Conf. on Fifth Generation Computer Systems, ICOT, 1988.

[越村] 越村、近山、佐藤、藤瀬、松尾、和田 : "並列論理型言語による OS の記述" 情報処理学会ソフトウェア基礎論研究会報告 No.29, 1989 年 5 月.