ソフトウェア基礎論　37－2
プログラミング言語　27－2
（1990. 12. 13）

# 並行オブジェクト指向言語のマルチプロセッサ上での実現

中村 宏明
日本アイ・ビー・エム (株) 東京基礎研究所

オブジェクト指向言語 COB をもとにした並行オブジェクト指向言語 ConcurrentCOB の設計を行ない、マルチプロセッサ・システム上に実装を行なった。ConcurrentCOB 言語は、COB からの書換えや COB との混在が容易に行なえるよう、COB との連続性を重視して設計されている。ConcurrentCOB の実行系は 2 階層からっており、変更や拡張、並列性の抽出、性能の向上が簡単に行なえる。1 層目はオブジェクトの並列性とオブジェクト間の通信の基本的な機能を、2 層目はメモリ管理などの機能を実現している。また、システムの性能の評価についても述べる。

# Implementation of an Object Oriented Concurrent Programming Language on a Multiprocessor System

Hiroaki NAKAMURA
IBM Research, Tokyo Research Laboratory
5-19, Sanbancho, Chiyoda-ku, Tokyo, 102 Japan

In this paper, the design of an object oriented concurrent programming language ConcurrentCOB and an implementation on a multiprocessor system are described. ConcurrentCOB is a concurrency extension to an object oriented programming language COB. The desing principle is that ConcurrentCOB should be a minimal extension to COB. The main feature of the ConcurrentCOB runtime system is that it has two-layered structure. The first layer provides concurrency of objects and communication between them. The second layer provides extra functions, such as a memory management, based on the first layer. Results of the evaluation of the runtime system are also mentioned.

# 1 はじめに

近年のVLSI技術等により、並列コンピュータを構成することが可能となってきている。これらを用いて、複数タスクのスループットの向上は行なわれているものの、1つのまとまった処理を並列に高速実行することは容易でない。その理由のひとつとして、適当なプログラム言語がないことがあげられる。

並行オブジェクト指向プログラミングは、並列性と同期がオブジェクトとメッセージ受渡しに埋め込まれているため理解しやすいことなどから、並列処理の記述に適した方法である。しかし、従来の並行オブジェクト指向言語の研究は、オブジェクトの表現力を調べることに重点が置かれ、並列環境で処理系を実現する方法や、実用的な性能を得る方法に関する研究は多くなかった。

そこで我々は、Cをベースにしたオブジェクト指向言語COB[11]をもとにして並行オブジェクト指向言語ConcurrentCOBを設計し、マルチプロセッサ・システムTOP-1[13]上での実行系の作成を行なうことによって、並列環境での並行オブジェクト指向言語の実現方式や性能など、並行オブジェクト指向言語の実用化へ向けてのさまざまな問題を考えていくことにした。

# 2 ConcurrentCOBの言語仕様

## 2.1 設計方針

ConcurrentCOBはCOBを拡張した並行オブジェクト指向言語である。ここでCOBの特徴をまとめると次のようになる。

- Cと互換性のあるコンパイル方式のオブジェクト指向言語である。
- インターフェースとインプリメンテーションを完全に分離することにより、大規模かつ複雑なプログラムの作成に対応する。またプログラムの修正による再コンパイルを極力小さくする。
- 実行時の暴走を避けるため、型システムを強化するとともに、配列のインデックス値の範囲のチェックなどをおこなう。
- プログラムが完成した時点で強力な最適化をおこない、実行効率を回復する。

特に並列化に有利な特徴としては、オブジェクトをヒープ上にだけ配置するため、セマンティクスが単純で統一されていることがあげられる。

ConcurrentCOBの設計に際して、COBとの連続性・差異の少なさを重視することにした。これは、COBの特徴を受け継ぐためとともに、次のような目的による。

- わずかな変更でCOBのプログラムをConcurrentCOBのプログラムに書き換えられること。これによって
  - 従来のCOBのプログラムを徐々に並列化して実行できるようになる
  - 並列プログラムのデバッグは困難なので逐次プログラムの段階である程度のデバッグを済ませておくことができる

  などの利点を得る。
- COBのプログラムとConcurrentCOBのプログラムを混在できること。これは
  - ライブラリ等のCOBのプログラムの再利用を行なう
  - 並列性や同期の必要がない場合には軽いCOBのメソッド呼び出しを用いることによってオーバーヘッドを削減する
  - 再帰呼びだしなど並行オブジェクトで表現しにくいアルゴリズムをCOBで記述する

  などのために必要である。

## 2.2 オブジェクト間通信

COBの各オブジェクトに対して、1つのアクティビティを割り当て一体化したものがConcurrentCOBのオブジェクトである。これらのオブジェクト間での通信の方法として、同期型のみを採用した。理由は次の通りである。

- CやCOBにおける呼び出しと同じように利用することができ、従来のプログラミングと連続性がある。
- オブジェクト間通信は、処理の要求と結果の返答の組で行なわれることが多い。
- 手続きベースの言語に非同期通信を導入すると、変数の意味などの言語仕様に大きな拡張が必要になり[4]、ConcurrentCOBの目的に適さない。
- 並列処理では、エラー・例外の扱いを局所化する必要があるが[1]、範囲の限定や復帰が容易である。
- 必要な場合には、バッファリングを行なうオブジェクトを用いることによって、非同期通信を模倣できる[14]。

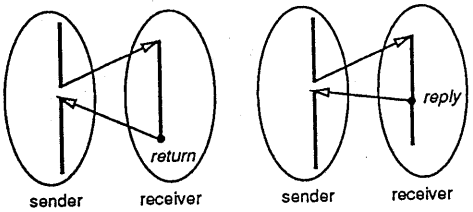


図 1: 通信の種類

このような通信はリモート・プロシージャ・コール (RPC) と同様なものと考えることができるが、RPC を並行オブジェクト間の通信に用いるときの問題点は、処理の軌跡が 1 本になり、並列性が阻害されることである。そこで送信側オブジェクトへの値の返し方を 2 種類用意し、これに対処した (図 1)。1 つは値を返すと同時に実行を終了する通常の方法で、return で示される。もう 1 つは、値を送信側に返すが、処理はそのまま続ける方法で、reply で示される。なお return は reply の特殊形として実装が可能である。

## 2.3 同期のモジュール化

インヘリタンスは、特に、強い型付けを行なう言語に柔軟性を導入する重要なはたらきをする。しかし、並行オブジェクト指向言語におけるインヘリタンスは種々の問題をもたらしてきた [6]。ある手続きの途中でメッセージの受付けを行なう方法 (Ada のランデブや、Occam の?と!の組など) では、新たな手続きが増える度に同期部分のプログラムの書換えが必要で、インヘリタンスを含む差分プログラミングに馴染まない。このため多くの並行オブジェクト指向言語では、メソッドの実行を開始するかどうかのレベルで同期がとられる。しかし、ある時点で受付け可能なメッセージの種類を、間接的にでも名前で指定する限り [15]、サブクラスの記述にはスーパークラスのインプリメンテーションに関する知識が必要になり、同期に関してモジュール化されていると言えない。

そこで、メッセージの引数、インスタンス変数、private 関数で構成される式をメソッドの前に付加し、この式を評価した結果が真になるメッセージだけが受付けられるようにした。この式をガードと呼び、when(<式>) で表現する。ガードにより同期のモジュール化が達成される。

[8] ではガードで表現できない、つまり 1 階の論理式で記述できない同期制約の存在が示されているが、実用的にはガードは十分な表現力を持つものと思われる。またガードは新たな言語要素の付加が少なく、ConcurrentCOB の目的に適しているため採用した。

## 2.4 プログラム例

```
class MemoryManager {      /* Interface */
    void *malloc(size_t);
    void free(void *);
     ...
};

class impl MemoryManager { /* Implementation */
    size_t left; /* private instance variable */
     ...
definition:
    void *malloc(size_t req) when (left >= req) {
        <割り当てる領域のアドレス addr の計算>
        reply addr;
        <left の更新などの後処理>
    }
    void free(void *p) {
         ...
    }
     ...
};
```

図 2: ConcurrentCOB のプログラム例

図 2 は ConcurrentCOB による簡単なメモリ管理オブジェクトのクラス記述例である。クラスの定義は型構成子であるインターフェース部と、実際にクラスをどう実現するかを記述するインプリメンテーション部に厳格に分離されている。またこのため、クラスのインプリメンターだけが知っていればよいインスタンス変数などは、すべてインプリメンテーション部におかれている。このような特徴は、そのまま COB から受け継いだものである。

メモリ管理オブジェクトはメモリの要求を行なうメッセージ malloc が到着すると、ガードの評価を行なう。割り当て可能なメモリがあるときガードの評価値は真になり、そのメッセージは受付けられる。malloc メソッド内では、割り当てる領域の先頭アドレスを計算して送信側オブジェクトに返した後、残りの処理をおこなう。メモリが足りないときはガードの評価値は偽になり、malloc の受付けは中断される。中断されたメッセージ malloc は、メモリを解放するメッセージ free によって十分な領域が確保されてガードの評価値が真になった時に受付けられる。

## 3 実装

### 3.1 TOP-1 上での実装の概要

TOP-1 は Intel80386(14MHz) を 10 台、スヌープキャッシュによってバス結合した共有メモリ型マルチプロセッサである。また TOP-1 上では、マルチプロセッサ対応のオペレーティング・システム TOP-1 OS[7] が稼働している。我々は ConcurrentCOB の実行系を、TOP-1 OS 上に実装した。

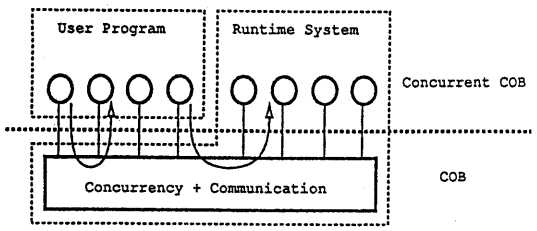


図 3: 実行系の構造

ConcurrentCOB の実行系の大きな特徴は、2 つの階層からなっていることである (図 3)。1 つ目の階層は、オブジェクトの並列性とオブジェクト間の通信を実現し、COB で記述されている。2 つ目の階層は、並列性と通信以外の並行オブジェクトを動作させるのに必要な機能を提供している。この階層で実現している機能としては

- メモリ管理 (図 2 を拡張したもの)
- I/O 管理
- 他のドメインとの通信管理

などがあげられる。1 つ目の階層でオブジェクトの並列性とオブジェクト間の通信が実現されているため、1 つ目の階層だけで並行オブジェクトの最小限の動作が可能である。このため、2 つ目の階層は ConcurrentCOB のユーザ・プログラムと同じ枠組で、ConcurrentCOB 自身で記述することができる。

このような階層構造の利点としては次のものがあげられる。

1. 実行系の再構成や拡張が容易にできる
2. 実行系の内部の並列性を引き出すことが容易にできる
3. 実行系の調整が行ないやすく、性能を向上させることが容易にできる

1. は、ConcurrentCOB に限らず、他のほとんどの階層構成のシステムに共通の利点である。2. に関しては、並列コンピュータ上でのプログラム言語の実行系では、排他制御を行ないつつ、実行系の内部ができる限り並列に動作することが望ましい。ところが、シングル・プロセッサ・システムでは排他制御は割り込みを禁止するだけで達成できるのに対して、マルチ・プロセッサ・システムでは各資源をきめ細かく排他制御することは簡単ではなく、実行系内部の設計は非常に難しいものとなる。ConcurrentCOB の実行系では、このような並列性の制御を行なう部分を並行オブジェクトの枠組で記述することができるので、並列性を自然に引き出すことができる。また 3. に関して、階層構成のシステムは性能の点で不利になると考えられがちである。しかし、階層化された構造がボトルネックを局所化するのに役立っているため、性能の調整が行ないやすく、ある程度より複雑なシステムでは性能の高いものを得ることが簡単になる。性能の調整の経験について 4で述べる。

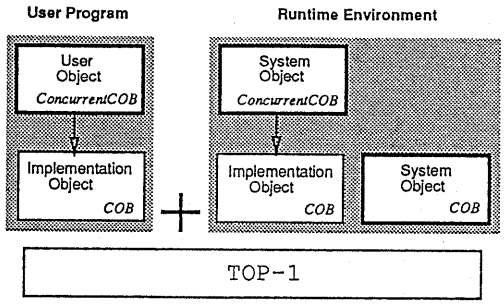


図 4: ユーザ・プログラムの実行

ユーザ・プログラムの ConcurrentCOB オブジェクトは COB のインプリメンテーション・オブジェクトにトランスレータで変換され、それが実行系と結合されることによって TOP-1 上で実行される (図 4)。インプリメンテーション・オブジェクトは、本来のオブジェクトにアクティビティを割り当てるためのオブジェクトを付加した複合オブジェクトである。すでに述べたように、実行系は COB で記述された部分と ConcurrentCOB で記述された部分からなる。ConcurrentCOB で記述された部分は、ユーザ・プログラムと同様に、COB のインプリメンテーション・オブジェクトに変換されている。

以降の節では、1 つ目の階層で実現されている機能に関して述べる。オブジェクトの並列性を実現する部分を 3.2で、オブジェクト間の通信を実現する部分を 3.3で説明する、

### 3.2 オブジェクトの並列性の実現

我々は TOP-1 OS 上に ConcurrentCOB の実行系の構築を行なった。TOP-1 OS は従来の OS と同じプロセス・モデ

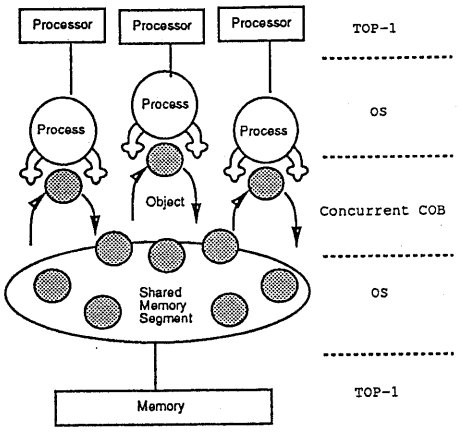


図 5: オブジェクトの並列性の実現

ルを採用しているため、TOP-1 ハードウエアの並列性はプロセスの並列性として利用される。このため、プロセスの数以上の並列性をユーザが利用することはできない。TOP-1 OS では、各プロセスはプロセッサ台数を越えないかぎり異なるプロセッサ上に配置されるので、システムの起動時に特定の数のプロセスを立ちあげ、以降これらのプロセスを仮想的なプロセッサとして用いる。

また、並行オブジェクトの特徴はアクティビティがオブジェクトに割り当てられていることであったが、処理の過程でオブジェクトの生成・消滅が数多く繰り返されることや、受信準備ができていないオブジェクトへのメッセージ送信などで動作を中断することが頻繁にあることなどから、オブジェクトの生成や切替えはかなり軽くなくてはならない。このため OS の提供するプロセスをそのまま並行オブジェクトとして用いることは行なわず、プロセスとオブジェクトの対応付けを動的に行ない、オブジェクトの生成・切替えはユーザ・レベルで行なう。

[3] では、OS を介して行なう並列システム上の並列プログラムの実行方式を 4 種類に分類しているが、我々の方式は User Concurrency と呼ばれるものである。実プロセッサと OS の提供する並列性が同じ数で、ユーザ・プログラム上の並列性は OS のプロセス数よりも一般に多いため、並列性の制御は、OS でなく、ユーザ・レベルで行なうことになる。

次に記憶の管理であるが、OS を介して共有メモリ・セグメントを各プロセスに張り付け、並行オブジェクトをすべてこの共有メモリ・セグメントに配置している。このため、並行オブジェクトは全てのプロセスからアクセスすることができる。またメソッドの定義などは、プロセス固有の領域に配置されているが、OS によって共有がなされている。

ここで、プログラムが終了するまで、各プロセスは次の動作を繰り返し、並行オブジェクトの処理が進められる (図 5) 。

1. オブジェクトを一つ選び出す
2. そのオブジェクトのメソッドを実行する
3. そのオブジェクトを解放する

## 3.3 オブジェクト間の通信の実現

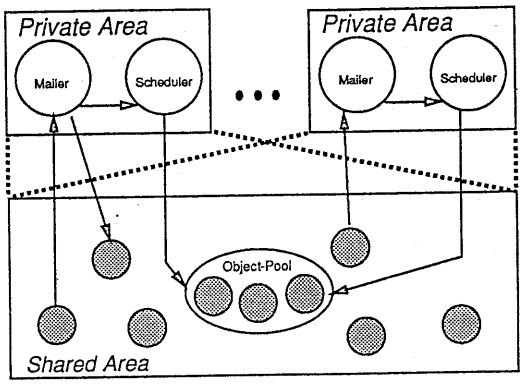


図 6: オブジェクト間通信の実現

複数のプロセスを用いて並列実行系を実現するとき、プロセス共有領域とプロセス固有領域にそれぞれ何を配置するかが重要である。これは、シングルプロセッサ環境をマルチプロセッサに拡張するときにオブジェクトの serialization と replication を考えること [12] に対応する。図 6に、主要なオブジェクトの配置を示す。並行オブジェクトが通信を行なおうとするとき、Mailer に通信の要求を出す。Mailer はオブジェクト間の通信の要求を受け付け、オブジェクトの切替の必要性を調べる。Scheduler はプロセスが実行するオブジェクトの切替えを行う。ObjectPool は実行可能なオブジェクトの管理を行う。これらのオブジェクトは COB のオブジェクトであるが、Mailer や Scheduler など、呼び出されるオブジェクトがプロセス固有領域にある場合には通常の COB のメソッド呼びだしで実現されるのに対して、ObjectPool のようにプロセス共有領域にあるオブジェクトへのアクセスは, ハードウエアレベルの同期機構による相互排除をともなったメソッド呼び出しが必要である。このようなハードウエアレベルの同期をおこなうときに不必要な

キャッシュの無効化・更新をなるべく起こさないために、不可分命令の実行回数が少ないスピン・ロックを用いる [17]。

ConcurrentCOB オブジェクトのメソッド呼び出しは、COB インプリメンテーション・オブジェクトが Mailer を介して通信するように変換される。Mailer は receive 、send 、reply の 3 つのメソッドを持つ COB オブジェクトある。

**receive** メソッドの実行を完了したときに呼び出される。受信側オブジェクトは、自分あてのメッセージが来ているとき、新たなメソッドの実行をはじめる。メッセージが来ていないときは、受信側オブジェクトの実行を中断する。

**send** 並行オブジェクト間のメソッド呼び出しを要求するときに呼び出される。受信側オブジェクトが receive を呼び出した結果として中断しているとき、受信側オブジェクトの中断を解除してメソッドの実行をはじめる。受信側オブジェクトがメソッドの実行中か send を呼び出した結果として中断しているとき、メッセージが受付けられるのを待つオブジェクトとして送信側オブジェクトを受信側オブジェクトに登録する。いずれの場合も、送信側オブジェクトの実行は中断される。

**reply** 送信側オブジェクトを中断から解除する。返値を送信側オブジェクトにセットする。その後、もとの処理に戻る。

これらのメソッドは、この他に、コンテクストの切替え、Scheduler による別のオブジェクトの起動、ガードの評価とその値のチェックなどを行なう。

ここで、図 2のメソッド malloc を取り上げ、Mailer を用いた通信へ変換されたものを図 7に示す。なお、メソッドの呼び出し方は変換後も変化しない。

```
void *malloc(size_t req) {
  /* コンテクストは送信側 */
  <ガード評価用の環境を保存>
  switch (Mailer->send(<受信側オブジェクト>)) {
  case RECEIVER:
    /* send と receive のハンドシェーク成功 */
    /* コンテクストは受信側                 */
    malloc_body(req);
    Mailer->receive();
    /* ここへは戻ってこない */
  case SENDER:
    /* 送信側の中断を解除 */
    /* コンテクストは送信側 */
    return <reply によってセットされた値>
  }
}
void *malloc_body(size_t req) {
     ...
    Mailer->reply(addr);
     ...
}
```

図 7: Mailer を用いた通信への変換

# 4 性能評価

## 4.1 メッセージ送信のオーバーヘッド

我々は、TOP-1 上の処理系に先だってシングル・プロセッサ・ワークステーション上で ConcurrentCOB の処理系の試作を行なった。このとき ConcurrentCOB のプログラムの実行時間の大部分がメッセージ受渡しに費やされていることがわかった。このため、メッセージ受渡しの機構の改良を行ない、あるプログラムについては、はじめの版に対して 4.4 倍の性能向上が達成できた。この改良で得られたメッセージ受渡しの性能向上のためのポイントは、

- メッセージの受渡しの実現に用いる一時的なオブジェクトは、メッセージ受渡しごとに生成するのでなく、使い終った後で再利用するようシステムが管理すること
- ある時点で受け付け可能なメッセージの集合（静的に定義されたメッセージ集合の部分集合）の走査を高速化すること
- 性能の調整が容易なようにシステムが構成されていること

等であった。TOP-1 への実装に当たっては、特に性能の調整が容易になるように、実行系を階層構成にしたことをすでに述べた。このため、ユーザ・プログラムと実行系内部の同期の機構が統一されていて、性能の向上を行なうために、COB で記述された 1 つ目の階層に集中することができる。この部分に対して、更に性能のボトルネックとなっている部分について次のような改良をおこなった。

- 重たい処理の機械語への書き換え
- アルゴリズムの変更（命令順序の入れ変え等）

この結果、現在の実行系は最初の TOP-1 上の実行系と比して 2.4 倍の性能の向上を得ている。

各種操作に必要な処理時間を測定した結果を表 1に示す。COB のメソッド呼び出し (1) は 1 段の間接関数呼び出しで実現されているので極めて効率が良いのに対して、ConcurrentCOB のメソッド呼び出し (3) は、オーバーラップ

| 操作 | 処理時間 |
|---|---|
| (1) method-call(COB) | $5.2\mu s$ |
| (2) context-switch + schedule | $28.8\mu s$ |
| (3) method-call(ConcurrentCOB) | $104\mu s$ |

表 1: 各種操作の処理時間

実行によって実質的に実行時間が短縮されることを考慮しても、かなり遅いように思われる。一方、同期通信と同様の操作を実現するには、少なくとも 2 回の context-switch と re-scheduling(2) を含んでいる必要があるため $60\mu s$ 程度はどうしても必要であり、また、ConcurrentCOB のメソッド呼び出しは条件同期も実現していることを考えると、同期の機構としての ConcurrentCOB のメソッド呼び出しは許容できる速度を達成していると言えるだろう。この結果は、ConcurrentCOB は並列システムを記述するのに有効な手段となるが、より現実的なシステムの記述には ConcurrentCOB だけでなく、COB のオブジェクトも用いるべきであることを示唆している。

## 4.2 マルチ・プロセッサによる速度向上比

次に、分散型クイックソート・プログラムを用いて実行系の速度向上比を測定した結果について述べる。このプログラムは、値を保持する各オブジェクトが近接するオブジェクトとの値を比較することによって自分の位置を決定して、binary tree 上に全順序を構成するものである。すべてが ConcurrentCOB のオブジェクトであり、並行度が高いので並列処理に有利な一方、オブジェクト間の通信が多いのでオーバーヘッドが大きい、という性質をもつ。また、多くのオブジェクトを動的に生成するので、メモリ管理オブジェクトへのアクセスが多い。

プロセス数を変化させたときの処理速度向上比を調べた結果を図 8 に示す。ここで実行過程を単純にモデル化して、プログラムが完全に並列に実行できる部分と逐次的にしか実行できない部分からなると考え、プログラム内の並列化率を $p$、プロセス数を $N$ とすると、速度向上比 $S_p$ は次式で表される。

$$S_p = \frac{1}{(1-p) + \frac{p}{N}}$$

$p$ を算出すると、プロセス数に関わらず約 85%になる。逐次化の要因は次の 2 つに分類できる。

1. 並行度の低化
   (オブジェクトの入口でおこる直列化など)
2. ハードウエア同期によるアイドリング
   (ObjectPool、インプリメンテーション・オブジェクトへのアクセスの衝突など)

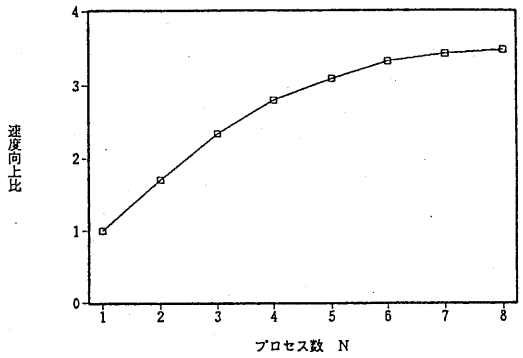


図 8: 速度向上比

TOP-1 上の Parallel FORTRAN での応用プログラムの実行 [10] ではプロセス数の増加とともに並列化率 $p$ が低下しているが、ConcurrentCOB のクイック・ソートでは $p$ がほとんど変化しないことから、ハードウエア同期の影響が少ないことがわかるが、詳細な分析のためには多くの種類のプログラムについての測定が必要である。

# 5 関連研究

ConcurrentCOB の実行系が提供する機能は、Thread または Light Weight Process と呼ばれる並行プログラムのための機構と同種のものである。Thread には、OS のカーネル内で実現するものと、ユーザ・レベルで実現するものの 2 種類があるが、マルチ・プロセッサで動作させるとき、さまざまな問題点が両者に存在する [9]。OS のカーネル内で実現する方式は、スレッドの生成や切替が非常に重く、このオーバーヘッドがあたえるプログラミング・スタイルへの影響が大きい。またユーザ・レベルで実現する方式では、一つのプロセス内に実現される限り、複数の Thread が同時に走ることがない。

PCR[16] は複数のプロセスを用いてユーザ・レベルの Thread を実装することによってこの問題を解決している。本稿で説明したオブジェクトの並列性の実現方法は、PCR と同じものである。PCR は C++のライブラリとして用いられることも意図されている。しかし、オブジェクト指向のインターフェースは PCR の持つ同期機構を組み合わせて実現する必要があるため、はじめからオブジェクト指向のインターフェースを持つものとして設計したのに比べてオーバーヘッド大きくなる可能性がある。

オブジェクト指向言語とのインターフェースを持った Thread ライブラリとして、C++の Task System[14] がある。また PRESTO[2] はマルチ・プロセッサ上で動作する

C++とのインターフェースを持った並行プログラミング実行環境である。C++TaskSystem や PRESTO がライブラリ方式で並列性や同期を与えているのに対して、ConcurrentCOB は並列性が埋め込まれた言語を提供している。

言語として並列性を提供する利点としては、並列性とデータ、手続きを一体化することによるプログラムの複雑度の低減があげられる。また ConcurrentCOB では同期の機構がカプセル化されているので、プログラムの部品化が容易に行なえる。これに対しライブラリ方式では、同期のとり方がオブジェクトの使われ方に依存する場合があり、ライブラリを利用する部分の部品化がむずかしい。さらに、並列システムのデバッグは非常に難しいが、言語で並列性をサポートすることによって、コンパイル時にタイプチェックなどによって誤りの多くを発見できる。また不必要な同期を取り除く作業は、ライブラリ方式では全てユーザに任されているが、言語でサポートする方式ではコンパイル時の最適化が可能である。

## 6 おわりに

並行オブジェク指向言語 ConcurrentCOB のマルチプロセッサ上の処理系を実現した。幾度かの改良によって、実行系がある程度の性能で動作するようになったことを確認した。今後は、このようなシステムを効果的に利用する方法を開発することが重要になる。また、このシステムの拡張として分散環境用の処理系の検討を進めている。

## 参考文献


[1] H. E. Bal, J. G. Steiner and A. S. Tanenbaum, Programming Languages for Distributed Computing Systems, *ACM Computing Survey*, Vol.21, No.3, 1989.

[2] B. N. Bershad, E. D. Lazowska H. M. Levy and D. B. Wagner, An Open Environment for Building Parallel Programming Systems, *In Proc. of Symposium on Parallel Programming*, ACM, 1988.

[3] D. L. Black, Scheduling Support for Concurrency and Parallelism in the Mach Operating System, *IEEE Computer*, Vol.23, No.5, 1990.

[4] R. Chandra, A. Gupta and J. L. Hennesy, COOL: A Language for Parallel Programming, Technical Report CSL-TR-89-396, Stanford Univ. Computer Systems Laboratory, 1989.

[5] K. Hosokawa and T. Kamimura, Concurrent Programming in COB, *2nd UK/Japan Workshop on Computer Science*, 1989.

[6] D. G. Kafura and K. H. Lee, Inheritance in Actor Based Concurrent Object-Oriented Languages, *In Proc. of ECOOP'89*, Cambridge Univ. Press, 1989.

[7] 河内谷, 森山, 山崎, 白鳥, TOP-1 オペレーティング・システムの構造, 情報処理学会オペレーティング・システム研究会, 48-4, 1990.

[8] S. Matsuoka, K. Wakita and A. Yonezawa, 並行オブジェクト指向言語における継承について, 日本ソフトウエア科学会第 7 回大会論文集、A2-3, 1990.

[9] 根岸, 密結合マルチプロセッサのためのスレッドの実現法, 情報処理学会第 41 会全国大会, 7D-5, 1990.

[10] 大澤暁, TOP-1 における流体問題解析, 情報処理学会計算機アーキテクチャ研究会, 83-6, 1990.

[11] 小野寺, 上村, COB によるオブジェクト指向プログラミング, *ComputerToday*,Vol.7, No.6, 1990.

[12] J. Pallas and D. Ungar, Multiprocessor Smalltalk: A Case Study of a Multiprocessor Based Programming Environment, *In Proc. of Conf. on Programming Language Design and Implementation*, ACM, 1988.

[13] 清水, 山内, マルチプロセッサワークステーション TOP-1, *bit*, Vol.22, No.7, 共立出版, 1990.

[14] B. Stroustrup and J. E. Shopiro, A Set of C++ Classes for Co-routine Style Programming, *In Proc. of C++ Workshop*, USENIX, 1987.

[15] C. Tomlinson and V. Singh, Inheritance and Syncronization with Enabled-Sets, *In Proc. of OOPSLA'89*, ACM, 1989.

[16] M. Weiser, A.Demers and C. Hauser, The Portable Common Runtime Approach to Interoperability, *In Proc. of the Twelfth ACM Symposium on Operating System Principles*, 1989.

[17] N. Yamanouchi, Performance Effects of Program Structure on a Snoop-cached multiprocessor system, *In Proc. of InfoJapan'90 Computer Conf.*, IPSJ, 1990.