プログラミング 8－15
（1996. 8. 28）

Parallel STL による並列プログラミング

中田秀基[†] 佐藤三久[†††] 松岡聡[††]
石川裕[†††] 松田元彦[†††]

C++ においてテンプレートを用いた並列プログラミング向けライブラリ構築の試みがなされている。本稿では、並列プログラミングのインターフェイスとしてのテンプレート技術の応用について考察する。並列テンプレートライブラリを用いると、プロセッサの構成を意識することなく並列プログラミングを行うことができる。データ並列のテンプレートライブラリは、データの局所性を利用した計算を可能にする。タスク並列のテンプレートライブラリは、負荷分散と同期をユーザから隠蔽し、困難なタスク並列計算を支援する。

Parallel Programming using Parallel STL

Hidemoto NAKADA,[†] Mitsuhisa SATOH,[†††]
Satoshi MATSUOKA,[††] Yutaka ISHIKAWA[†††]
and Motohiko MATSUDA [†††]

There are several works on parallel processing with C++ using template library. In this paper, we discuss template technology as an interface for parallel programming. Parallel template library allows us to program parallel programs taking no thought of target machine configuration. Data-parallel template library enables optimization using data locality. Task-parallel template library provides methods to distribute work-loads.

1. はじめに

C++[1] を並列に拡張する研究が数多くなされている。HPC++[2] は、これらの並列の C++ を統合する試みとして行なわれている。この HPC++ の並列化の中核をになうものが並列化版 STL である PSTL である。これは言語レベルでの機能拡張を最小限にし、テンプレートでインターフェイスを提供するものである。

テンプレートは、一種のパラメータ型を実現する機構であり、型整合性を保持したまま柔軟なプログラミングを実現することができる。C++ にはこれに加えて演算子のオーバーロード機能があり、これと複合することで、テンプレートはユーザーインターフェイスを提供するための強力な機能となりうる。

テンプレートを用いると、低オーバヘッドで、データ集合に対して抽象度の高い、高階な演算を定義することが可能である。このため、注意深くライブラリを設計すれば、ライブラリがプロセッサのコンフィギュレーションを吸収することで、逐次でも並列でも、プロセッサのコンフィギュレーションに依存しないプログラム環境を提供することが可能なはずである。

これを実現するためには二つのアプローチが存在する。一つは逐次むけに存在するデータ集合に対する演算を定義しているライブラリを拡張する方法。もう一つは、並列計算に特有の演算の構造を抽象化して構築する方法である。本稿では、これら双方のアプローチに関して検討し、テンプレートを用いて並列演算用のライブラリを構築法について述べる。

2 章ではテンプレートとその周辺技術について概観する。3 章ではデータ並列実行に必要とされるインターフェイスについて考察し、データ並列におけるテンプレートライブラリについて述べる。4 章でタスク並列におけるテンプレートライブラリについて考察する。

2. テンプレートとその周辺技術

2.1 C++ のテンプレート機能

C++ のテンプレートは、一種のパラメータ型を実現する機構で、あるクラスをパラメータとして他のクラスや関数を定義する機能である。テンプレートを用いることで、型整合性とコンパイル時の型チェックを保ったまま、型のない言語と同様の柔軟な記述が可能になる。

† 電子技術総合研究所
Electrotechnical Laboratory
†† 東京大学工学部
Faculty of Engineering, The University of Tokyo
††† 新情報処理開発機構
Real World Computing Partnership

例えば、List というようなクラスをテンプレートで定義すれば、すべての型に対して List を用いることができる。

```
template <class T>
class List{
  T & item;
  List * next;
public:
  List(T & i, List * n):item(i),next(n){}
};
```

継承を用いて同じことを実現することは不可能ではない。ListItem というようなクラスをつくって、List に入れるクラスすべてがこれを継承するようにするのである。これには多重継承を用いなければならないので、int などの基本型には用いることはできないし、すでに定義された型に対しては変更が必要であり、ライブラリとしての性質は良くない。

関数に対してもパラメータ型を用いた定義が可能である。これは関数テンプレートと呼ばれる。

2.2 STL の概要

STL (Standard Template Library)[3] とは、テンプレート をもちいたライブラリで主にデータを格納するさまざまな構造とその上での演算を提供するものである。STL は大まかに別けて、データを格納する container、container の一つ一つの要素を指す iterator、iterator を用いてデータ構造に対する一般的な計算を提供する generic algorithm の 3 つからなる。

テンプレート ベースのライブラリの継承ベースのライブラリに対する利点は、その性能と柔軟さである。テンプレート はコンパイル時に展開されてしまうので、参照が静的になるため、動的ディスパッチのコストがかからない。このため性能面で有利である。また、テンプレートのライブラリではそれぞれの要素クラスに対して C++ における型に合致していることを要求しない。このため、signature さえ一致していればどのようなクラスでも互換性があることになり、一部のクラスをオリジナルのものと置き換えるなどの柔軟な使用が可能となる。

もう一つの STL の特徴として、function object を用いた高階関数がある。function object とは、一見関数のように用いることのできるオブジェクトのことである。function object は、operator () をオーバーロードする。すると、*object*(*args*) というような記述で、この演算子定義が呼び出されるため、関数のように用いることができるのである。手続きを渡すための方法としては、関数へのポインタも考えられるが、function object は inline での展開が可能でありコンパイラによる最適化が期待できる点が優れている。このように STL は高階関数が基本となっているため、集合データに対する手続きの記述が容易になっている。

STL の container には、Vector, List, Map などの構造がある。iterator はそれぞれの container に対して定義され container 上での位置を表現し、container 上をトラバースする手法を提供する。generic algorithm は iterator を用いて定義されている関数群である。これは iterator のみを用いているため、iterator が定義できさえすれば、任意のデータ構造に対して同じ演算を提供することができる。STL に含まれていないデータ構造に対しても同様で、プログラマが記述した任意の構造であっても、iterator を用意してやりさえすれば、演算を利用できる。演算には、Sort, Merge, Accumulate, InnerProduct などがある。

iterator は STL で非常に大きな役割を果たす。iterator はポインタを一般化したもので、従来のポインタに対して可能であった操作を実装したクラスの総称である。この操作とは、* によるリファレンスの取り出し、++、-- による参照点の移動、==、!= などによる比較などである。container の実装は、通常の C++ の配列とは異なり、必ずしもメモリにフラットにマップしたものではない。したがって、container の上を通常の pointer を用いてトラバースすることはできない。しかし、container の構造にあわせた iterator を用いれば、pointer と同じインターフェイスを提供してくれるので、pointer を用いるのとほぼ同じような感覚で、プログラミングすることが可能である。

下に示すのは、vector を用いて 0 から 9 までの加算をしているプログラムである。

```
main(){
  vector<int> v(10);
  vector<int>::iterator vi = v.begin();
  int sum = 0;
  for (int i = 0; i < 10; i++) *vi++ = i;
  for (vi = v.begin(); vi != v.end(); vi++)
    sum += *vi;
}
```

vi が iterator である。定義を除けばほとんど pointer と同じ使いかたができることが分る。

下に示すのは 同じプログラムの加算を行っている部分を generic algorithm のひとつである accumulate を用いて記述した例である。

```
  sum = accumulate(v.begin(),
                   v.end(), 0, plus<int>());
```

accumulate の第 4 引数 のplus<int>() は function object で次のように定義されている。

```
template <class T>
struct plus : binary_function<T, T, T> {
    T operator()(const T& x, const T& y)
                const { return x + y; }
};
```

accumulate 自身は以下のように定義されている。

```
template <class InputIterator, class T,
          class BinaryOperation>
```

```
T accumulate(InputIterator first,
             InputIterator last, T init,
             BinaryOperation binary_op) {
    while (first != last)
        init = binary_op(init, *first++);
    return init;
}
```

binary_op には、オブジェクト plus<int> がバインドされているので、binary_op(init, *first++) とすると、plus<int> で定義した operator () が呼び出される。この結果 int に対して operator + が実行され、加算が行われるのである。

2.3 Expresssion Template

Expresssion Template[4] は、C++ のテンプレートと演算子のオーバロードを用いて、Lisp などの Closure に似た機能を実現するものである。これを用いると次のような記述が可能になる。

```
main(){
  DExpr<DExprIdentity> x;
  sum(x/(1.0+x), 0.0, 10.0);
}
```

これは、x を 0 から 9 まで動かしながら、$x/(1.0+x)$ を計算するものである。sum の第一引数の一見通常の式に見えるものが、closure のように働くのである。

Expression Template のポイントは、演算子のオーバーロードを巧みに用いていることである。"+"などの演算子が、実際の演算を行うかわりに、演算を表現するシンタックスツリーを構成する。このシンタックスツリーをコンパイル時に静的に評価することで、この機能が実現されるのである。

expression template の機構は非常におもしろいが、式の内部で使用するすべての関数をオーバロードしておかなければならないという制約があり、実用性には疑問が残る。

Expression Template が目指していると思われるものを実現する一つの方法として、コンパイル時メタを使用することが考えられる。コンパイル時メタとは、リフレクションの一種で、コンパイル時にシンタックス拡張やセマンティクスの変更を実現するものである。C++ にコンパイル時メタを導入する研究には、MPC++[5] や Open C++[6] などがある。これらの言語を用いれば無名の関数を言語に導入できる。これを Lisp の Lambda 式のように用いれば、Expression Template 以上に有用な枠組になりうるだろう。また無名関数は並列インターフェイスとしてのみでなく、例えば GUI のコールバックの記述など、さまざまな応用が考えられる。

3. データ並列とテンプレート

データ並列を分散メモリ上で実現する際に最も重要になるのは、データの配置である。データ並列では一般にデータがおかれているプロセッサがそのデータを処理するため、データの配置がそのまま計算の配置に直結するからである。

3.1 データ並列に必要な機能

本節では、データ並列の言語 / ライブラリの提供すべき機能について、特にデータの配置に着目して説明する。

3.1.1 HPF

HPF[7] は、並列 Fortran の標準化の試みとして研究されている言語である。HPF の特徴は、配列を分散配置する手法が整っていることである。HPF の配列の配置は配列同士のアラインメントの設定、配列のディストリビューション、実プロセッサへのマップ、の三段階で行なわれる。

3.1.2 A++/P++

A++/P++[8] は、配列演算を支援するために、C++ のデータ型にアレイ型を追加するライブラリである。P++ は、A++ をインターフェイスはそのままに、並列実行できるようにしたもので、SPMD 型の実行を前提としている。

A++/P++ の特徴は、配列に対して view という概念を定義していることである。view は配列の一部を別の配列としてみるもので、これ自身を配列と同様に扱うことができる。view を定義するための方法として、配列の各次元ごとの範囲をしめす Range と Index と呼ばれるオブジェクトがある。Range は、index の上限と下限を設定するものである。Index はよりフレキシブルで、 position, size, stride の 3 つ組で指定する。

P++ では、配列をプロセッサ間に跨って定義できる。この場合のパーティショニングは Partition_Type と呼ばれるオブジェクトを用いて行う。パーティショニングの指定法は、プロセッサ数の指定、使用するプロセッサ番号の指定、負荷分散を書いた double の配列による指定、の 3 つがある。同じサイズの配列を同じパーティションタイプを用いて配置するとアラインメントは完全に一致する。

3.1.3 Template Closure

A++/P++ と同様のインターフェイスをテンプレートで提供するものとして、Template Closure[9] がある。これは、Expression Template と同様の発想に基づくもので、演算子のオーバロードを用いて、行列の計算式そのものを一度パーズツリーとして構成し、これをコンパイル時に静的に評価することで、ループフュージョンと一時変数の除去を行なうものである。

3.1.4 考　　察

HPF と A++/P++ の機能から分ることは、データ並列においては、データの相対的配置が重要であるということである。特に、HPF においてはデータ同士の相対的アラインメントの指定が重視されている。HPF では、アラインメント情報を用いて、コンパイラが

FORALLループにでてくるデータがどのプロセッサにあるか、を解析し、最適化を行なう。

双方の問題点は、データ集合に対する手続きがが貧弱であることである。HPFではFORALLを使って、逐次のループと同様にインデックスを用いて各エレメントのにアクセスし、操作を行なうしかない。P++では並列化されるのは幾つかの用意された関数だけである。この問題点は、function objectによる高階関数をもちいることで解決できると思われる。

3.2 HPC++ 版PSTL

HPC++[2]はいくつかのC++の並列拡張を統合しようという試みとして設計されている言語である。HPC++の設計はLevel1とLevel2の2段階に別れている。Level1では、compiler directiveとライブラリで並列を支援し、Level2では、言語自体の拡張を行う。現在はLevel1の設計が行われている。このLevel1の並列支援を行なうものが、並列版STLであるPSTLである。Level1はSPMDやshared memoryを対象としている。

Level1のcompiler directiveには、ループが独立であることを示す`#pragma HPC_INDEPENDENT`、独立なループ内での変数へのアクセスのアトミック性を保証する`#pragma HPC_REDUCE`などがある。これらの構文を用いてPSTLを作成する。

PSTLの基本は、distributed containerとparallel iteratorである。distributed containerは、複数のプロセッサに跨るContainerである。Distributionについては、P++同様に、インスタンシエイト時に指定する。parallel iteratorは、distributed containerに対して、プロセッサを跨いでのトラバースを与える。これを用いて並列実行するためのアルゴリズムとして次の3つが提供されている。

- `par_apply(f(), begin1, end1, begin2, begin3,....)`
  `f()`を各要素にアプライする。
- `par_reduction(op(), f(), begin1, end1, begin2, begin3,....)`
  それぞれのコンテナの要素に`f()`したあと、`op()`でリダクションする。
- `par_scan(op(), f(), begin1, end1, begin2, begin3,....)`
  begin2以降のコンテナの要素同士に`f()`したあと、`op()`の結果をbegin1で始まるコンテナに格納する。

これらのアルゴリズムは、引数として与えられるiteratorがparallel iteratorでないときには通常の逐次版のアルゴリズムとして動作するが、parallel iteratorのときには、プロセッサごとに行なうことで並列化をおこなう。

3.2.1 ディストリビューション

Distributed Containerの一種として、Distributed Arrayがある。Arrayのディストリビューションを指定するためのクラスとして、ArrayDistributionというクラスがある。この抽象クラスから具体クラスを導出し、これをDistributedArrayのインスタンス作成時に指定することで、Arrayのディストリビューションを指定する。

このArrayDistributionクラスを具体化するには、次の3つのメソッドを実装しなければならない。

- アレイの参照空間から1次元のリニア空間へのマップ
- 1次元リニア空間からアレイの参照空間へのマップ
- 1次元リニアのインデックスからプロセッサ番号とオフセットへのマップ

この手法は非常に汎用性があり、blockでもcyclicでも任意のディストリビューションを記述することが可能である。

また、DistributedArrayのアクセス法としては、A++と同様なRangeやviewに相当するものが用意されている。

3.3 HPC++ の問題点

HPC++の並列機能の問題点は

(1) データ間のアラインメントを記述する方法がない
(2) アラインメントを考慮して実行する枠組みがない
(3) function objectを用いているため、記述が煩雑

の3点にあると思われる。3に関してはすでに述べたので、1と2について述べる。

HPC++のアレイ機能を にはデータ構造同士のアラインメントを指定する方法がない。ディストリビューションを指定することで、相対的な関係を間接的に設定することは可能ではあるが、これで、任意のアラインメントを設定することは難しいだろう。

これは他のparallel containerにおいても同様である。例えば、2つのparallel containerの内積をとろうとする場合を考えてみよう。個々のプロセッサでローカルに内積をとり、それらをすべて合わせて解を出すわけだが、2つのcontainerの対応する要素が、同じプロセッサにあるとは限らない。したがって、1つめのcontainerの要素があるプロセッサでローカルな処理をするならば、2つめのcontainerの要素に対するアクセスは、単なるリモートアクセスになってしまう。すべてのアクセスでリモートかローカルかのチェックが入るだけでなく、リモートアクセスは要素ごとに行われることになり非常に効率が低下する。

HPFでは、二つのArrayの間のアラインメントを用いて、コンパイラが静的に解析を行なってリモートアクセスの最適化を行なう。しかし、テンプレートを用いたライブラリのみで、これを行なうことは非常に難しい。HPC++でアラインメントを指定する方法を用意していないのは、コンパイル時の解析が難しいためアライン

メントがあっても意味がないためであろう。

しかし、並列計算においてデータアクセスの局所性は本質的である。何らかの方法で通信時間を削減しなければならない。これに対処するにはいくつか方法が考えられる。

一つは、ディレクティブを言語に導入することである。また、コンパイラ自身の能力として、解析機能を作り込んでしまうことも考えられる。しかし、これらの手法では、ライブラリが言語システムに依存することになり、新たなデータ構造のライブラリを追加しようとした際に、コンパイラにまで手を入れなければならないことになりかねない。

もう一つは、コンパイル時メタを用いる方法である。メタを用いて、Distributed Container 間の関係を解析し、適切なコードを出力させることができるだろう。

もう一つの方法は、ローカルなイタレーションの前後に、動的にアラインメント解析を行ないそれに従ってバルクでのデータ転送を行なうルーチンを挿入する方法である。スタティックな解析を行なうのに対しては効率は低下するが、ナイーブに実装するよりははるかに高い効率が期待できる。この手法を実装するには、現在の HPC++ のディストリビューションはあまりに一般的である。あらゆるディストリビューションが記述可能なため、データの配置を見究めにくく、最適な転送パターンを導出することは難しい。これは、HPF のアラインメントにもいえる。実際のアプリケーションでの使われ方をサーベイして必要な機能を絞り込み、動的なアラインメント解析が容易で、なおかつ必要な機能はあるというようなアラインメントの指定方法を考察する必要がある。

4. タスク並列とテンプレート

タスク並列を用いたテンプレートライブラリを考えてみよう。タスク並列プログラムの記述が困難なのは、タスク同士の同期や負荷分散が困難だからであると考えられる。これらを考慮しながらプログラミングすることは難しいし、これらがアーキテクチャに依存するために、プログラムのポータビリティが低下する。これらの困難なポイントを、テンプレートライブラリで隠蔽することで、タスク並列を一般的に用いることを可能にする。

タスク並列テンプレートは、タスク並列にありがちな問題の解法の骨組みを抽出して、これをテンプレートライブラリとして提供する。ユーザが自分の問題に適した骨組みを選び、自分の問題を解くための部分演算を与えると、計算が実行されるのである。

テンプレートで計算の骨組みを提供する利点は、実際にどのように計算が行われるかが、完全に隠蔽されている点である。テンプレートライブラリは、共有メモリマルチプロセッサ向けに実装されているかも知れないし、ワークステーションクラスタ向けに実装されているかも知れない。プログラマは、実際の実装を意識せずに、部分関数を記述するだけで、プロセッサの構成にマッチした実行方式で、自動的に実行が行なわれるのである。

4.1 タスク並列テンプレートライブラリ

タスク並列テンプレートライブラリには、以下のようなものが考えられる。

- たくさんのオブジェクトが互いにメッセージパッシングを行いながら同じタイムクロックで稼働する離散シミュレーション
- 一つの黒板オブジェクトを複数のオブジェクトが共有し、黒板オブジェクトを書き換えることで計算を進める黒板法
- 探索木をスタックとして表現、共有し、部分問題に対する操作で探索を行なう方法

これらに共通して用いられる負荷分散の手法や、サブ問題のキューイング手法などを低レベルのテンプレートライブラリとして設計する。高位のテンプレートライブラリは、これらの低レベルのライブラリを用いて組み立てるように設計する。アーキテクチャに依存するのは、低位のライブラリのみになると考えられるので、低位のライブラリのみを書き換えるだけで、ライブラリを移植することができる。

4.2 スタック分割動的負荷分散方式

このように、タスク並列の枠組と実際の演算を分離した例として、並列論理型言語 KL1 むけに ICOT で開発された、スタック分割動的負荷分散方式 (STB) という手法がある[10]。

STB は、深さ優先探索アルゴリズムなどのスタックを用いた問題に対する負荷分散手法として考えられたものである。この手法では探索木をスタックで表現し、スタックを各プロセッサに分配する。ユーザは、問題に対して処理を行ない、新たな副問題と部分解をかえすプログラム expand と、部分解から全体の解を作り出すプログラム combine を作成する。システムは、あらかじめ対象を処理するプロセスをすべてのプロセッサに分散させておく。それぞれが、スタックから取り出した問題に expand を適用し、副問題をスタックに戻し、部分解を combine でマージする。仕事がなくなると乱数で選択した他のプロセッサに対して仕事を要求する。そのクラスタに仕事がないと、新たに乱数で他のプロセッサを選択してそちらに要求を行なう。図 1に 4 プロセッサでの STB の様子を示す。

これをテンプレート で C++ 上に実装してみよう。この場合は、 expand と combine をそれぞれユーザーに、 function object として記述させることになる。

例として N-queen を記述してみる。この場合、後処理 combine は必要ないので空の関数になる。expand は以下のようになる。

```
template <class Que>
class quExpand{
public:
```

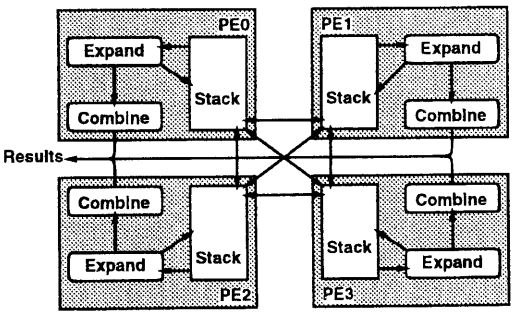

図1 スタック分割動的負荷分散方式

```
  void operator()(qu & brd,
                  Que &que, Que &ans){
    if (brd.current >= brd.max)
      ans.push_back(brd);
    else
      for (int i = 0; i < brd.max; i++){
        qu & brd2 = *(brd.copy());
        brd2.addlast(i);
        if (brd2.check())
          que.push_back(brd2);
      }
  }
};
```

qu は、チェスの盤面を表現するクラスである。部分解は、途中まで整合性がある状態でクイーンを置いたものに相当する。この expand ルーチンは、queen の部分解 brd を受け取り、それが解の一つであれば、解を蓄積するキュー ans にプッシュする。brd が、まだ解でなければ次の部分解を生成し、部分解キュー que にプッシュしている。すなわち、もう一つ駒を置いてみてそれが条件を満たさなければ捨て、条件を満たせば、つぎの部分解としてキューに入れるのである。

stb を呼び出す部分は以下のようになる。

```
deque< qu > ans =
     stb(quExpand < deque< qu > >(),
         quCombine< deque< qu > >(),
                    *(new qu(N)));
```

関数stb は、expand 関数オブジェクトと combine 関数オブジェクトと初期値を引数として受けとり、これらの関数オブジェクトを呼び出して実行を行う。

4.3 考　　察

タスク並列の枠組をテンプレートで提供することは、データ並列では記述しにくいアプリケーションにまで C++ の適用可能範囲を広げることになる。

タスク並列のテンプレートライブラリを構築するためには、タスク並列の計算法を収集解析し、基本的なパターンを抽出して、ライブラリに必要とされる機能を整理しなければならない。

5. おわりに

並列演算のインターフェイスとしてのテンプレートの可能性について議論した。テンプレートによる並列ライブラリには以下の利点がある。

- 柔軟で拡張性のあるライブラリが構築可能。
- プロセッサアーキテクチャなどの相違をライブラリが吸収するので、ポータブルな並列プログラムの記述が可能
- タスク並列に関しても対処可能

しかし、一方、

- コンパイラレベルではないため、静的解析による最適化が難しい
- 高階関数の実現を function object に頼っているため、記述が煩雑

という問題点もある。今後は、コンパイル時メタの利用も考慮しつつライブラリの実装を行なっていく。

参 考 文 献

1) A.Ellis, M. and Stroustrup, B.: *The Annotated C++ Reference Manual*, Addison Wesley (1990).
2) : High Performance C++, http://www.extreme.indiana.edu/hpc++/index.html.
3) R.Musser, D. and Saini, A.: *STL Tutorial and Reference Guide*, Addison Wesley (1996).
4) Velhuizen, T.: Expression Templates, http://www.roguewave.com/forum/exchange/extempl.htm.
5) 石川裕, 堀敦史, 小中裕喜, 前田宗則, 友清孝志: 並列プログラミング言語 MPC++ の実現, *JSPP'94* (1994).
6) Chiba, S.: A Metaobject Protocol for C++, *OOPSLA '95* (1995).
7) : High Performance Fortran Forum Home Page, http://www.crpc.rice.edu/HPFF/home.html.
8) Quinlan, D. J.: A++/P++, http://www.c3.lanl.gov/dquinlan/A++P++.html.
9) 松田元彦, 石川裕, 佐藤三久: C++ テンプレートを使ったデータ並列処理ライブラリの 効率化手法, *JSPP'96* (1996).
10) 古市昌一: PIMOS 負荷バランスユーティリティマニュアル, TR 796, ICOT (1992).