並列処理の実験支援システム COOP/VM†

金井達徳††* 藤井啓明††
柴山潔†† 萩原宏††**

COOP/VM は，種々のネットワーク・トポロジを持つメモリ非共有・メッセージ交換型疎結合マルチプロセッサ上での，各種並列処理プログラムの実行特性の評価実験を支援するシステムである．このシステムは，C 言語をベースに並列処理のための機構を追加したオブジェクト指向プログラミング言語 COOP と，汎用計算機の仮想計算機システム上に構築した並列計算機において，COOP 言語で記述した並列処理プログラムの実行をシミュレートする「仮想並列計算機」と呼ぶ実行環境から構成される．COOP には，言語処理系内部のデータにアクセス可能で，並列実行の制御等，メタレベルの記述が可能な「論理プロセッサ」と呼ぶモジュール型を導入した．論理プロセッサのプログラミングによってスケジューリングや負荷分散の戦略等の実験が可能になる．COOP/VM では，論理プロセッサのプログラミングによって要素プロセッサ間の時間の同期や結合網のルーティング，通信遅延等を手続き的に記述することで，仮想的に並列計算機システムを実現している．

1. はじめに

多数の要素プロセッサを結合した並列計算機のアーキテクチャや，その上で実行する並列処理アルゴリズムの研究・開発が盛んになっている．このような研究・開発においては，その対象とする並列計算機上で並列処理プログラムを走らせた結果得られる動的な処理特性をあらかじめ把握し，それを反映させてより最適な設計を行うために，シミュレーションが重要な技術となっている．本論文で述べる COOP/VM は，共有メモリを持たないメッセージ交換型並列計算機を対象として，このような目的を持ったシミュレーション実験を支援するシステムである．

一般にシミュレーションには，そのシミュレータの開発に要するコストと，シミュレーションに要する時間，そしてそのシミュレーション結果の精度との間のトレードオフが存在する．すなわち，より高速で高精度のシミュレーションを可能にするためには，シミュレータの開発における生産性の向上が必要である．従来の並列処理のシミュレーションにおいては，問題に応じて専用のシミュレータを作成することが多かった．このようなシミュレータでは，シミュレーション対象の時間の進み方を正確に記述することが精密なシミュレーションにつながる．しかし，処理に要する時間を正確に見積もり，その記述を与えることは，シミュレーションの必要性が高い設計の初期段階であるほど難しい問題となっている．

また，並列に動作する複数の対象のシミュレータを逐次型のプログラミング言語で記述しようとすると，複数の処理の流れを時分割によって1つの処理の流れとして実行するために，特殊なディスパッチ処理のプログラミングを行う必要がある．並列プログラミング言語を用いてシミュレータを記述したとしても，時間に対する同期は，時間を管理する部分と各動作対象との協調処理が必要であり，やはり複雑なプログラミングが必要となる．

そこで我々は，並列処理のシミュレータに必要となる時間の管理や，複数の動作対象の実行制御を記述する能力を持つプログラミング言語 COOP を設計した．COOP はシミュレーション言語であると同時に，特定の相互結合網のトポロジに依存しないメッセージ交換型並列計算機のための汎用の並列プログラミング言語でもある．COOP には，時間の管理機能を始めとして，並列処理におけるさまざまな概念について実験するために「論理プロセッサ」と呼ぶメタレベルの記述能力を持たせた．また，COOP を用いたシミュレーション実験を支援する COOP/VM と呼ぶ実行環境を開発した．COOP/VM は，汎用計算機の仮想計算機システム上で任意の相互結合網のトポロジを持つメッセージ交換型並列計算機をシミュレートし，その上で COOP で記述した並列処理プログラムを仮想的に並列実行す

† COOP/VM: A Programming Environment for Experimental Parallel Processings by TATSUNORI KANAI, HIROAKI FUJII, KIYOSHI SHIBAYAMA and HIROSHI HAGIWARA (Department of Information Science, Faculty of Engineering, Kyoto University).
†† 京都大学工学部情報工学教室
* 現在 (株)東芝総合研究所
Research and Development Center, Toshiba Corp.
** 現在 龍谷大学理工学部
Faculty of Science and Technology, Ryukoku University

る．COOP/VM 自身も，COOP の論理プロセッサ・プログラミングによって実現している．

2. COOP の実験支援方式

COOP は，メモリ非共有メッセージ交換型の疎結合並列計算機を対象とした実験支援のためのプログラミング言語である．**図1**に COOP が対象とする並列計算機のモデルを示す．このような並列計算機や，その上で実行する並列処理プログラムの設計・開発時におけるシミュレーションは**図2**に示すような構造を持つ．まず，対象となる並列計算機の記述と，その上で処理したい問題が存在する．並列計算機の記述は，それを構成する要素プロセッサと，それらを結合する相互結合網に関する記述からなる．処理したい問題は，プログラマによって，直接，要素プロセッサのオブジェクトコードとしてコーディングするか，なんらかの高級言語のプログラムとして記述する．高級言語で記述したプログラムは，対象並列計算機用のコンパイラによってオブジェクトコードとなる．この並列計算機とオブジェクトコードの記述をもとにその実行を行い，動的な評価データを収集するのが，シミュレーションの目的である．

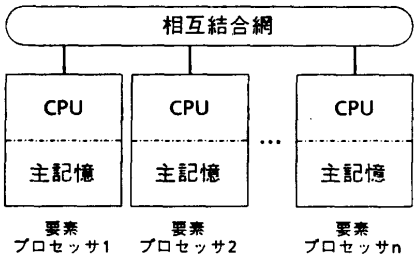


図 1 対象とする並列計算機のモデル
Fig. 1 Model of target parallel computers.

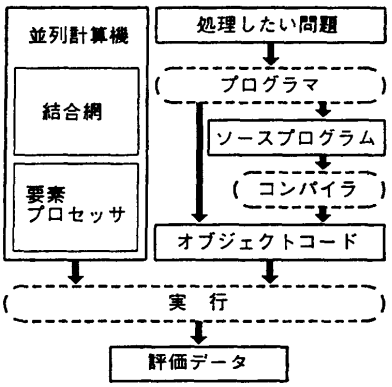


図 2 シミュレーションの構造
Fig. 2 Simulation structure of a parallel processing.

理想的には，図2中の各構成要素の記述を正確に与えることができれば，正確な評価データを得ることができる．しかし，設計・開発の初期段階ほどシミュレーションに対するニーズは高いにもかかわらず，各構成要素は十分に詳細化されていないプロトタイプであり，完成時に得られるような正確な記述を与えることは困難であるのが普通である．また，対象並列計算機の仕様が確定していない段階でコンパイラを正確に与えることは非現実的な要求であり，このことも，図2の構造に即したシミュレーションを難しいものにしている．

我々は，設計・開発の初期段階において，できるだけ図2の構造に即したシミュレーションを手軽に行える手法として，次のようなモデルに基づいたシミュレーション方式を提案する．

(1) シミュレーション対象の並列計算機の要素プロセッサをシミュレーションを行うホストのプロセッサと同じものであると考えて，並列計算機のシミュレーションを行う．つまり，シミュレーション対象の要素プロセッサ上で命令の実行に要する時間は，ホスト計算機においてその実行に要する時間と同じものとする．これは，『並列計算機を構成する要素プロセッサにおいて，各種の処理に要する時間の割合は，現在の技術で構成されたフォン・ノイマン型計算機であれば，巨視的にみればどのようなプロセッサにおいても相似である』という仮定の上に立っている．

(2) シミュレーション対象の並列計算機上で実行したいプログラムを，並列計算機の構造に依存しない汎用の並列プログラミング言語で記述する．シミュレーション対象の並列計算機上でのそのプログラムの実行過程を(1)の仮定に基づいてシミュレートする．

このモデルにより，シミュレーション対象における処理時間をパラメタとして陽に与えなくても，(1)で述べたような仮定によって，適度な近似を行うことができる．また，シミュレーションを行うホスト計算機構造に依存するが，シミュレーション対象の並列計算機構造には依存しないプログラミング言語の処理系を用いることにより，対象並列計算機の仕様が決まらないとコンパイラが作れないという関係を解消することができる．

このようなモデルに基づいたシミュレーションを行うために設計したプログラミング言語が，本論文で述べるCOOPである．COOPは，結合網のトポロジに依存しない汎用プログラミング言語として設計した．COOPのシミュレーション対象の並列計算機上での実行を任意のトポロジを持つホスト計算機上でシミュレートするプログラムを記述するのもCOOPである．

COOPでは，シミュレータ記述言語としての能力を高めるために，プロセッサの持つ実行制御機構を，プログラミング言語レベルで抽象化した「論理プロセッサ」と呼ぶ概念を導入した．論理プロセッサは，COOPで記述したプログラムのレベルから見ればメタなレベルにあたる並列実行の制御を，COOP言語自身で記述可能にするものである．シミュレーション対象の並列計算機の要素プロセッサ間の時間の同期や，結合網のルーティング，通信遅延等は，論理プロセッサのプログラミングにより，手続き的に記述してシミュレートする．これにより，シミュレーション対象のプログラムの記述と，その対象並列計算機上での実行をシミュレートする部分の記述の分離が可能となっている．また，論理プロセッサのプログラミングによって，並列処理におけるスケジューリングや負荷分散の戦略等の実験も可能になる．

3. プログラミング言語 COOP

3.1 COOP 言語の概要

プログラミング言語COOPは，C言語をベースにした並列オブジェクト指向言語である．C言語をベースにすることで，C言語の持つ汎用性や，低レベルの記述が可能であるような柔軟性をそのまま活かすことができる．また，COOPの記述対象としているのは，メッセージ交換型の並列計算機上で実行する問題解決プログラムのプロトタイプである．そのため，COOPを並列オブジェクト指向言語とすることで，プログラムのモデル化とその並列計算機上へのマッピングが容易になる．COOPでは，Smalltalk[1]等のオブジェクト指向言語におけるオブジェクトに対応するものを「モジュール」と呼ぶ．モジュールは，C++[2]などと同様，型を持っている．

COOPは，並列処理における様々な実験を支援することを目的としている．そのため，それ自身が実験・研究の対象となる自動的なガーベジコレクションを伴うようなメモリ管理や，モジュールやそれに関する制御の流れを要素プロセッサ間で移動させるような自動的な負荷分散機能は，COOP言語の機能としては含めない．すべてのメモリ領域は，プログラマの責任で割り付け・解放し，モジュールの要素プロセッサへの割り付けはプログラムの記述によって静的に決定される．

3.2 COOP の並列処理機構

COOPで記述したプログラムの実行は，**図3**に示すように，要素プロセッサ上に分散して配置されたモジュール間のメッセージ通信によって進められる．モジュールはメッセージを受け取ると，そのメッセージに対応する手続きの実行を開始する．この手続きのことを「メソッド」と呼ぶ．メソッドの実行の流れを「スレッド」と呼ぶ．COOPでは，ひとつのモジュールに対して複数のスレッドが同時並列に存在可能である．メッセージの処理の直列化 (serialization) は行わない．同一モジュールに属するスレッド間で同期をとる必要がある場合は，COOPの提供する同期プリミティブであるTest&Set命令を用いて，プログラム上で明示的に記述する．

モジュールへのメッセージ送信には，**図4**に示すような分岐型と待機型の2種類を用意した．分岐型は，他のモジュールにメッセージを送った後，その結果の返送を待たずに次の処理を続けるもので，並列した処

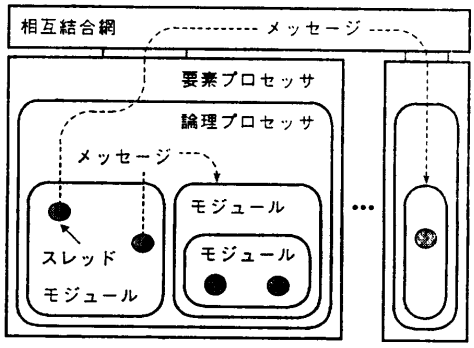


図3 COOPプログラムの実行の様子
Fig. 3 Snapshot of executing COOP programs.

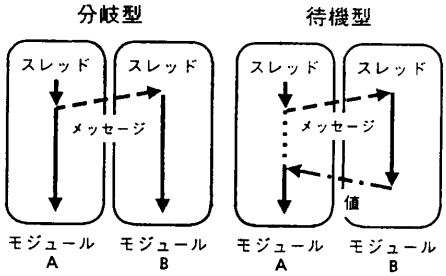


図4 COOPのメッセージ送信パターン
Fig. 4 Patterns for message passing in COOP.

理の流れを生み出すために用いる．プログラム上は，

モジュール！メソッド名 (引数 1, 引数 2, …, 引数 *n*)

と記述する．待機型は，メッセージに対する結果の返信を待ち合わせて，それを得てから次の処理を進めるものである．プログラム上は，

モジュール．メソッド名 (引数 1, 引数 2, …, 引数 *n*)

と記述する．モジュールは「モジュール・ポインタ」と呼ぶデータによって参照する．COOP のモジュール・ポインタは，そのモジュールの属する要素プロセッサ番号を含んでいるため，他の要素プロセッサ上にあるモジュールへのメッセージ送信も，同一要素プロセッサ上の場合と同様に記述することができる．

3.3 モジュール

COOP のモジュールは型を持ち，**図 5** のように宣言する．モジュール型の宣言は，そのモジュールの局所変数，局所関数，メソッド，そのモジュールの初期化を行うためのメソッドの記述を持つ．初期化メソッドはモジュール型と同じ名前を持つ．宣言は他のモジュール型の定義を継承 (inheritance) することもできる．COOP の継承機構は多重継承も可能であるが，同じ名前を持つ変数やメソッドなどプログラム上の言語要素を深さ優先で上書きするという単純な実現法を採っている．新しいモジュール型を定義すると，その型のモジュールのポインタを保持することができる変数の型が，同じ名前で定義される．プログラム中では，これを用いてモジュールの参照を行う．

COOP では，モジュール型の定義の中で局所的なモジュール型を定義することができる．また，COOP で記述したプログラムのテキスト上のトップレベルは root と呼ぶモジュール型の宣言であると解釈される．その結果，COOP のプログラム中のすべてのモジュール型間には，木構造をなす親子関係が存在する．モジュール型を定義すると，そのモジュールの実体を生成するためのメソッドが，そのモジュール型の構造上で親となるモジュール型のメソッドとして作られる．たとえば図 5 の例のように，モジュール型 M2 は M1 の中で定義されているとする．このときモジュール M1 には，新しい M2 型のモジュールを生成するためのメソッド M2 が定義される．M1 のメソッド M2 は，モジュール M2 の初期化メソッドと同じ引数をとり，以下の処理を順に行う．

(1) モジュール M2 の実体に必要な領域を割り付けて初期化する．

(2) 生成したモジュールに対して初期化のためのメッセージ M2 を送る．

(3) M2 が後述の論理プロセッサ・モジュールの場合は，分岐型で dispatch メッセージを送る．

(4) 生成したモジュールのポインタを値として返す．

このように，すべてのモジュールの実体の生成は，親モジュールに対するメッセージ送信によって行われる．この結果，モジュールの実体間には，モジュール型の木構造を拡大した構造を持つ親子関係が生まれる．新しいモジュールを生成するためのメッセージ呼び出しは，そのメッセージを受信すべきモジュールの指定をプログラム上では省略することができる．この場合，そのメッセージを受信可能なモジュールを，コンパイル時に自モジュール，親モジュール，その親モジュールへと探索して決定する．

モジュール内で定義された関数やメソッドからは，その構造上で親となるモジュール内の変数を参照できる．この機構によって，親モジュールの持つ変数を子モジュール間で共有するような大域変数として扱うことが可能になる．各モジュールに対して暗黙に定義される疑似変数としては，モジュール自身を指す self，親モジュールを指す parent，後述の論理プロセッサ・モジュールを指す processor がある．

```
module M1{
  /*インヘリタンスの宣言*/
  inherit    M3,M4;
  /*局所変数の宣言*/
  integer    i;
  char       c='a';
  M2         m2ptr; /*モジュール・ポインタ*/
  /*局所モジュール型の宣言*/
  module     M2{…};
  /*局所関数の定義*/
  local      func1(…){…}
  /*初期化メソッドの定義*/
  M1(…){…}
  /*メソッドの定義*/
  method1(…){…}
  ……
};
```

図 5 モジュール型の宣言
Fig. 5 Declaration of module types in COOP.

4. 論理プロセッサ

4.1 論理プロセッサの概要

「論理プロセッサ」は COOP を用いた並列処理の実験を柔軟に支援するために導入した特殊なモジュール型である．一般にプロセッサは，プログラムの実行を進めるエンジンとしての働きを持つ．このプロセッサの機能をプログラミング言語のレベルで抽象化したのが，論理プロセッサである．論理プロセッサはその

管理下にあるモジュールの関与するメッセージの送受信や，それらのメソッドを実行するスレッドのスケジューリングを行う一種のインタプリタとして働く．このインタプリタ部分のプログラムをユーザが COOP 言語自身で記述することによって，並列処理に関するさまざまな実験を可能にしている．

COOP プログラムのトップレベルである root モジュールも論理プロセッサ・モジュールのひとつである．この結果，すべてのモジュールの実体からそのモジュールから構造上で親となるモジュールの方向へたどると，必ず論理プロセッサ・モジュールにぶつかる．こうして最初に見つかる論理プロセッサ・モジュールを「そのモジュールを制御する論理プロセッサ・モジュール」であるという．モジュールとそれを制御する論理プロセッサ・モジュールとは必ず同じ要素プロセッサ上に存在する．モジュールを参照するために用いるモジュール・ポインタは，

- モジュールの存在する要素プロセッサ番号；
- モジュールの存在するアドレス；
- モジュールを制御する論理プロセッサ・モジュールの存在するアドレス；

の3つのデータの組からなる．この結果，モジュール・ポインタが与えられると，そのモジュールを制御する論理プロセッサ・モジュールを見つけるのは容易である．

論理プロセッサ・モジュールは，

- processor と呼ぶモジュール型を必ず継承している；
- 次項で述べる論理プロセッサ・インタフェースと呼ぶ3つの特殊な働きを持つメソッドを持つ；
- 論理プロセッサ・モジュール内で processor 疑似変数は，その上位の論理プロセッサ・モジュールを指す；

ことを除いて，他のモジュール型と同様である．論理プロセッサ・インタフェース以外の論理プロセッサ・モジュールのメソッドの実行は自論理プロセッサの制御に従う．

4.2 論理プロセッサ・インタフェース

すべての論理プロセッサ・モジュールの定義は，**図6** に示すような仕様を持つ3つのメソッドを備えなければならない．COOP のコンパイラは，各論理プロセッサがこれらのメソッドを持つことを前提として，オブジェクト・コードの生成を行う．

sendMessage と sendValue はそれぞれ，その論理プロセッサの制御下にあるモジュールの関与するメッセージの送信，および，メソッドの実行結果の返信に伴って起動されるメソッドである．

```
void sendMessage(_modulePointer callee,
                 _method        method,
                 _valueBlock    *args,
                 _threadPointer caller);

void sendValue(_threadPointer caller,
               _valueBlock    *result);

void dispatch();
```

図6 論理プロセッサ・インタフェースの仕様
Fig. 6 Specification of logical processor interfaces.

sendMessage は以下の引数を伴って呼び出される．

- callee：メッセージを送る相手モジュールを指すモジュール・ポインタ；
- method：メッセージに対応して起動されるメソッドのアドレス；
- args：引数をパケット化した構造体へのポインタ；
- caller：メッセージを送る側のスレッドを指すスレッド・ポインタ；

スレッド・ポインタは以下のデータの組からなる．

- スレッドの存在する要素プロセッサ番号；
- スレッド・ディスクリプタの存在するアドレス；

sendValue は以下の引数を伴って呼び出される．

- caller：値を送り返す先のスレッドを指すスレッド・ポインタ；
- result：値をパケット化した構造体へのポインタ；

dispatch はその論理プロセッサの管理するスレッドの実行のスケジューリングを行うためのメソッドである．dispatch は，スケジューラの実行を開始するために論理プロセッサ・モジュールの生成時に呼び出され，上位の論理プロセッサの制御下で実行される．

論理プロセッサ・インタフェースのメソッドにおいては，前節で述べたモジュール・ポインタやスレッド・ポインタといったシステムの内部のデータにアクセス可能である．これによって，言語システムの内部データを参照し，実行時のセマンティクスに影響を及ぼすようなプログラミングが可能となる．

4.3 COOP のスレッド・ライブラリ

COOP は，論理プロセッサ・インタフェースで規定されたメソッドにおいて，スレッドの操作を行う．そのため，**図7** に示すようなスレッド操作用のライブラリを提供している．

新しいスレッドは，スレッドの属するモジュール (callee)，実行するメソッドのアドレス (method)，メッ

```
_threadPointer
_create_thread(_modulePointer   callee,
               _method          method,
               _threadPointer   caller);

_real_time
_resume_thread(_threadPointer   thread,
               _valueBlock      *value);

_valueBlock  *_suspend_thread();

void _exit_thread();
```

図 7 COOP のスレッド・ライブラリ
Fig. 7 Specification of a thread library in COOP.

セージを送った側のスレッド (caller) の3つの引数とした _create_thread によって生成する．このとき，スレッドに固有のスタック等のデータ領域が割り付けられ，スレッド・ディスクリプタが作成される．スレッドは，スレッド (thread) とそれに引き渡す値 (value) を指定した _resume_thread によって実行を開始させる．新しいスレッドの最初の起動時にはそのスレッドに対応するメソッド呼び出しの引数を value として与える．それ以降に起動をかけるのはスレッドが他のモジュールへ送ったメッセージに対する返答があったときであり，返ってきた値を保持する構造体のアドレスを value として与える．_resume_thread はその実行に要した CPU 時間を値として返す．_resume_thread が起動をかけたスレッドは _suspend_thread あるいは _exit_thread によって実行を停止し，_resume_thread を発行したスレッド側に制御を移す．また，_suspend_thread は次に _resume_thread で再起動をかけられたときに，その value 引数を値として返す．一方 _exit_thread は，スレッドの持つ領域を解放してその実行を終える．

4.4 論理プロセッサ・モジュールの動作

最も単純な論理プロセッサ・モジュールは，図8に示すような動作を行う論理プロセッサ・インタフェース・メソッドをデフォルトとして持つ．COOP のコンパイラは，すべてのモジュールのメッセージ呼び出しをそのモジュールを制御する論理プロセッサの sendMessage メッセージに変換する．このとき，分岐型のメッセージ呼び出しの場合には第4引数の caller に NullThread と呼ぶダミーのスレッド・ポインタが与えられる．待機型のメッセージ送信の場合には送信側のスレッドが _suspend_thread で実行を中断し，値の返送を待つ．論理プロセッサは，sendMessage の第1引数で与えられるメッセージ送信先のモジュールが自分と同じ要素プロセッサ上にあれば，m2lp でそのモジュールを制御する論理プロセッサ・モジュールを見つけ，それに sendMessage メッセージを転送する．異なる要素プロセッサ上にある場合にはさらに上位の論理プロセッサに転送する．こうして，メッセージ送信相手モジュールを制御する論理プロセッサが sendMessage を受け取るとその実行を開始するために新しいスレッドを生成し，スケジューリングのためにキューに入れる．

```
sendMessage(callee, method, args, caller) {
  if (calleeは他の要素プロセッサ上にある)
    processor.sendMessage(callee, method, args, caller);
  else if (calleeは自分の制御下にある) {
    newThread=_create_thread(callee, method, caller);
    <thread, args>対をスレッド・キューに入れる;}
  else
    m2lp(callee).sendMessage(callee, method,
                             args, caller); }

sendValue(caller, result) {
  if (callerがNullThreadでない) {
    if (callerは他の要素プロセッサ上にある)
      processor.sendValue(caller, result);
    else if (callerは自分の制御下にある)
      <caller, result>対をスレッド・キューに入れる;
    else t2lp(caller).sendValue(caller, result); }}

dispatch() {
  if (スレッド・キューが空でない) {
    スレッド・キューから
      <thread, value>対を取り出す;
    _resume_thread(thread, value); }
  self.dispatch(); }
```

図 8 単純な論理プロセッサ・インタフェースの例
Fig. 8 An example of a simple logical processor interface.

メソッドの実行結果の返信はそのスレッドを制御する論理プロセッサに対する sendValue メッセージに変換される．sendValue も sendMessage と同様，論理プロセッサ間でメッセージの転送を行う．このとき，sendValue の第1引数が NullThread であれば転送しない．返答先のスレッドを制御する論理プロセッサはスレッドと返された値の対をスケジューリングのためのキューに入れ，実行を再開させる．

dispatch はキューの中から実行すべきスレッドを選択し，そのスレッドに起動をかける．その後，自己再帰的に dispatch メッセージを送ることによってスケジューリングのループを構成する．

COOP のコンパイラは，dispatch に見られるような末尾再帰的メッセージ送信に対して，無駄なスレッドを生成しないように最適化を行う．また，論理プロセッサの sendMessage 等に見られるようなメッセージの転送による多段のメソッド呼び出しは，コンパイル時に畳み込んで最適化する．

このような論理プロセッサ・インタフェースのプログラミングをユーザに開放することによって，単にプログラムのロジックだけでなく，実行の手順をも記述することを可能にしている．すなわち，

（1） 論理プロセッサ・モジュールの管理するスレッド・キューの構成；
（2） sendMessage や sendValue においてスレッド・キューへ実行可能なスレッドを入れる手順；
（3） dispatch においてスレッド・キューから実行可能なスレッドを取り出す手順；

を変化させることによって，デフォルトで提供される単純なラウンド・ロビン方式ではなく，優先度を考慮したスケジューリングなどを実現することができる．また，論理プロセッサは階層的に持つことができるため，特定の論理プロセッサ・モジュール以下のモジュールに対して特殊なスケジューリングを行うことも可能になる．この機能を利用してモニタやセマフォのような同期機構も論理プロセッサ・モジュールとして実現することが可能になる．

5. COOP のモジュール構成とシミュレーション方式

COOP 言語で記述したプログラムは，**図 9** に示すようなモジュール構成を持つ．並列計算機を構成する各要素プロセッサには root と呼ぶ論理プロセッサ・モジュールが 1 つ存在する．root モジュールは COOP のプログラムのトップレベルであり，プログラムを構成する各モジュールは root モジュールを根とする木構造をなす．root モジュール自身の実行制御はその上に存在する「PE モジュール」と呼ぶ論理プロセッサ・モジュールが行う．たとえば root モジュールの論理プロセッサ・インタフェースの記述において，他の要素プロセッサ上のモジュールに対する sendMessage は root を制御する PE 論理プロセッサ・モジュールの sendMessage の呼び出しに変換する．実際の要素プロセッサ間通信は PE モジュールが行うことになる．PE モジュールはその計算機上で COOP で記述したプログラムを実行する時の核となる論理プロセッサ・モジュールであり，その計算機に固有のものである．

COOP 言語を用いたシミュレーションを行う場合，シミュレーションのターゲット並列計算機上で実行するプログラムも，ホスト計算機上でターゲット並列計算機をシミュレートするプログラムも，どちらも COOP のプログラムである．この関係は，**図 10** に示すようなモジュール構成を採ることによって実現する．ホスト計算機の各要素プロセッサ上にはシミュレータのトップレベルとなる root 論理プロセッサ・モジュールが存在する．シミュレータはターゲット並列計算機の要素プロセッサをシミュレートする PE′ と呼ぶ論理プロセッサ・モジュールを提供する．ターゲット並列計算機で実行するアプリケーション・プログラムのトップレベルはホスト計算機上の PE′ モジュールをターゲット並列計算機における PE モジュールとみなし，図 10 の root′ モジュールの位置に置く．

PE′ モジュールは，root′ 以下のモジュールにおいてプログラムの実行に要した時間を累積することによってターゲット並列計算機の要素プロセッサにおける処理時間を管理する．また root′ モジュールは，ターゲット並列計算機の他の要素プロセッサに対する sendMessage や sendValue を PE′ モジュールに依頼する．PE′ モジュールは依頼されたメッセージに送信時刻と受信されるべき時刻をタイムスタンプとして付け，必要ならば PE モジュール間で通信を行って，送信先の PE′ モジュールへ伝える．PE′ モジュール

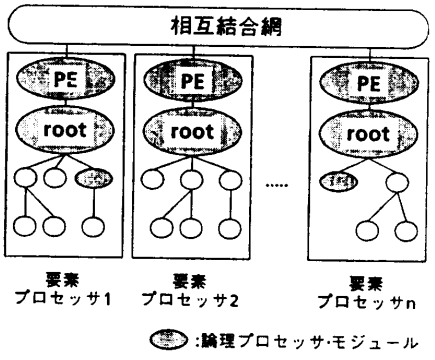


図 9 COOP のモジュール構成
Fig. 9 Configuration of modules in COOP.

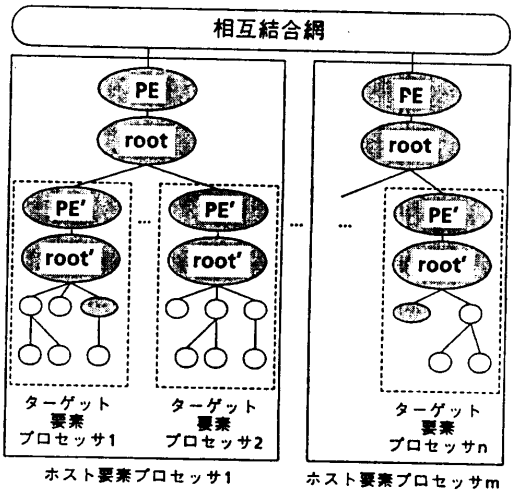


図 10 シミュレーションにおけるモジュール構成
Fig. 10 Configuration of modules for simulation.

は自分の管理する時間とメッセージの受信されるべき時刻の同期をとりながらターゲット並列計算機上でのメッセージ送信をシミュレートする．メッセージの受信されるべき時刻をタイムスタンプとして付加するときに，ターゲット並列計算機のネットワーク・トポロジを考慮して通信遅延時間を計算することによってネットワークのシミュレーションを行う．

このように，root および PE′ 論理プロセッサ・モジュールの記述によってターゲット計算機をシミュレートし，その上でターゲット並列計算機上のアプリケーション・プログラムを実行するのが COOP を用いたシミュレーション方式である．このような構成にすることによって，単一プロセッサからマルチプロセッサまで任意の構成の計算機上で任意のトポロジを持つ並列計算機のシミュレーションが可能となる．また，各 PE′ モジュールが管理する時刻をメッセージに付加したタイムスタンプのみを用いて合わせるようなシミュレータを記述することによって，シミュレータをホスト計算機の構成に依存しないように作成することができる．逆に，ホスト計算機に依存する機能を用いてシミュレータを記述することによって，効率の良いシミュレーションを実現することも可能になる．

6. 並列処理実験環境 COOP/VM

6.1 仮想並列計算機の構成方式

COOP/VM は，COOP 言語を用いた並列処理の実験を支援する環境として汎用計算機の仮想計算機システム (Virtual Machine; VM)[3] 上に開発したシステムである．本システムでは，仮想計算機システム上に数十から数百の要素プロセッサが結合した仮想的な並列計算機を構成する．この仮想並列計算機上で COOP で記述したプログラムの実行を行う．

仮想計算機システムは，1台の実計算機を時分割で使用することによって同じアーキテクチャを持つ複数台の仮想計算機があるように見せる技術である．COOP/VM では，シミュレーション対象の並列計算機を構成する各要素プロセッサに対してそれぞれ1台の仮想計算機を割り当てることによって，仮想並列計算機を構成する．この方式は，次のような特徴を持つ．

(1) 仮想計算機が持つメモリやタイマなどの資源は論理的には実計算機と同じものである．そのため，シミュレーション対象の要素プロセッサはホスト計算機のプロセッサと同じものと見なす COOP によるシミュレーションの考え方を実現しやすい．1要素プロセッサを1仮想計算機としない場合，各要素プロセッサにホスト計算機と同じアドレス空間を与えるためには，1プロセスに対して1仮想空間を与えることのできる多重仮想記憶を用いて1要素プロセッサを1プロセスで実現する必要がある．

(2) 仮想計算機システムの効率を向上させるために，仮想計算機補助機構 (Virtual Machine Assist feature; VMA)[3] に代表されるようなハードウェア／ファームウェア／ソフトウェアによる支援機構が提供されている．そのため，1台の仮想計算機上のなんらかのゲスト・オペレーティング・システム上でシミュレーション対象の各要素プロセッサをプロセスとして実現する場合に比べてゲスト・オペレーティング・システムによるオーバヘッドがなく，直接にシステムの支援を受けられる．

シミュレーション対象の要素プロセッサ間の相互結合網を介した通信は仮想計算機間通信機構 (Virtual Machine Communication Facility; VMCF)[3] を用いてシミュレートする．VMCF は，転送したいデータと転送先の仮想計算機を指定すると転送先の仮想計算機には割込みによって転送要求が伝えられ，両者のアドレス空間の間でブロック転送を行うことにより通信を行う機構である．この機構を用いて，他の要素プロセッサへのデータ転送はその要素プロセッサを担当している仮想計算機を指定した VMCF による通信によって実現している．

各要素プロセッサは，対象並列計算機上で経過する仮想的な時間 (COOP/VM では「仮想時間」と呼んでいる) 軸に同期するよう，時間の管理を行う必要がある．この要素プロセッサ間の時刻合わせという同期は，要素プロセッサ間の通信メッセージに付加するタイムスタンプのみでとることも可能ではあるが，デッドロックを発生させないような特別の配慮が必要となり，効率を低下させる要因となる．そこでメッセージのタイムスタンプに加えて仮想計算機間の共有メモリを用い，各要素プロセッサの状態を知らせ合うことによって同期をとる方式を採用した．仮想計算機システムは仮想計算機間のメモリ共有の手段として非連続保管セグメント (DisContiguous Saved Segments; DCSS)[3] を提供している．DCSS は，仮想計算機間で

共有するデータの入ったメモリ・セグメントを用意しておき，各仮想計算機が自分のアドレス空間とそのセグメントをマップすることで，共有セグメントを実現する機構である．COOP/VM では DCSS を非プロテクト状態で使うことにより共有メモリを実現している．共有メモリの参照は臨界領域となるので，共有メモリに対する Test&Set 命令によって排他制御を行う．

COOP/VM の仮想並列計算機はシミュレーション対象の並列計算機と同じ数の要素プロセッサを持っている．そのため，シミュレータの記述に際しては，図 10 で示した root モジュールと PE′ モジュールの役割分担は不要である．そこで PE′ モジュールの役割は root モジュールが兼ねるようにシミュレータの記述を行う．また，COOP/VM の仮想並列計算機の PE モジュールは要素プロセッサ間の通信をサポートしていない．そのため root モジュール（PE′ モジュールも兼ねている）の記述において，他の要素プロセッサとの通信は VMCF を用いた通信を行うように，sendMessage や sendValue の記述を行う．この記述をサポートするために，VMCF を利用した要素プロセッサ間の通信を行う**図 11** に示すような低レベルの関数を，「物理プロセッサ・インタフェース」として提供している．

6.2 COOP/VM による実験方式

COOP/VM を用いた実験の例として，格子状結合の並列計算機を実現する root モジュールの概略を**図 12** に示す．この root モジュールがシミュレーション対象の要素プロセッサの PE モジュールとなる．まず初期化メソッドにおいて共有メモリを初期化する．共有メモリには各要素プロセッサに対して，

- root モジュールのモジュール・ポインタ；
- 現在，処理を実行中か実行待ちか，およびスレッド・キューが空か否かを示すフラグ；
- 現在実行が進んでいる仮想時間；
- 次に受信すべきメッセージの受信仮想時間；

のエントリを持っている．次に，自分の担当する要素プロセッサ上のシミュレーション対象プログラムの root モジュールを生成する．初期化メソッド内では，対象並列計算機に依存したパラメタをユーザ・プログラムに対して参照可能にするために，大域変数の設定を行う．図 12 の例では，格子状結合した隣接する 4 つの要素プロセッサ上のシミュレーション対象プログラムの root モジュール・ポインタを変数 N, S, W, E に代入し，ユーザ・プログラムが他の要素プロセッサ上にモジュールを生成することを可能にしている．

sendMessage メソッドでは，他の要素プロセッサへのメッセージ送信を，VMCF を用いて仮想計算機間で

```
void _send_message(_modulePointer   callee,
                   _method          method,
                   _valueBlock      *args,
                   _threadPointer   caller,
                   _real_time       time);

void _receive_message(_modulePointer   *callee,
                      _method          *method,
                      _valueBlock      **args,
                      _threadPointer   *caller,
                      _real_time       *time);

void _send_value(_threadPointer   caller,
                 _valueBlock      *result,
                 _real_time       time);

void _receive_value(_threadPointer *caller,
                    _valueBlock    **result,
                    _real_time     *time);

_event_type  _event_check();

void _wait_event();

void _wakeup(_penum  penum);
```

図 11 物理プロセッサ・インタフェースの仕様
Fig. 11 Specification of physical processor interfaces.

```
root(){
  共有メモリの自分のエントリを初期化する;
  自分の管理するシミュレーション対象プログラムの
    rootモジュールを生成する;
  変数N,S,W,Eに自分の上下左右に結合している要素
    プロセッサ上のシミュレーション対象プログラムの
    rootモジュール・ポインタを代入する;
  スレッド・キューとメッセージ・キューを初期化する;}
sendMessage(callee, method, args, caller){
  if(calleeは他の要素プロセッサ上にある)
    _send_message(callee, method, args, caller,
                  _current_time + DelayTime);
  else if(calleeは自分の制御下にある){
    newThread=_create_thread(callee, method, caller);
    <thread, args>対をスレッド・キューに入れる;}
  else
    m2lp(callee).sendMessage(callee, method, args, caller);}
sendValue(caller, result){...}
dispatch(){
  if(_check_event())
    _receive_messageあるいは_receive_valueで
      メッセージを受信し,メッセージ・キューに入れる;
  while(_current_time以前に受信すべきメッセージがある)
    そのメッセージを取り出してsendMessageを起動;
  if(_current_time以前で
       実行待ち中の要素プロセッサpがある){
    _wakeup(p); _wait_event();}
  else if(スレッド・キューが空でない){
    スレッド・キューから実行するスレッドとその
      引数を選択して<thread, args>とする;
    _current_time += _resume_thread(thread, args);}
  else if(メッセージ・キューが空)_wait_event();
  else if(次のメッセージ受信時刻以前で
            実行待ち中の要素プロセッサpがある){
    _wakeup(p); _wait_event();}
  else
    メッセージ・キューから次に受信すべきメッセージ
      を取り出し,スレッドを生成してキューに入れ,
      _current_timeをその受信時刻に合わせる;
  self.dispatch();}
```

図 12 格子状結合を持つ並列計算機を実現する PE モジュールの例
Fig. 12 An example of PE module realizing mesh-connected multi-processors.

メッセージ転送を行う _send_message 関数の呼び出しに変える．このとき，相手側の要素プロセッサがそのメッセージを受信すべき仮想時間を _send_message の第5パラメタとして与える．図 12 の例では，単純に DelayTime 時間の通信遅延のみを与えているが，この部分の記述を変えることにより，より複雑な遅延モデルを実現することができる．sendValue についても sendMessage と同様の処理を行う．

dispatch メソッドは，共有メモリを参照して仮想時間に対する同期をとりながら，自要素プロセッサ内のスレッドの実行のスケジューリングと他の要素プロセッサから送られてくるメッセージや値の受信を行う．各要素プロセッサの状態を共有メモリを用いて知らせ合うことで，メッセージに付加したタイムスタンプのみによる分散的な時間合わせに付随するデッドロックの発生や空メッセージの送信による非効率性を改善することができる．

このように COOP/VM は，シミュレートしたい対象並列計算機の動作を手続き的に記述することによって，その上で COOP で記述したアプリケーション・プログラムを仮想的に並列実行する環境を実現することができる．

COOP/VM は，汎用計算機の仮想計算機システム上に構築したが，マルチ・プロセスのオペレーティング・システム上で1つのプロセスを1つの要素プロセッサとし，プロセス間通信を用いて要素プロセッサ間の通信を実現するような構築法も可能である．

7. 考　　察

7.1 COOP による実験支援の特徴

COOP 言語を用いた並列処理の実験支援方式は，以下に示すような特徴を持つ．

（1） シミュレーション対象の並列計算機の動作の記述と，その上で実行するプログラムの記述を分離することができる．

従来のシミュレーション用言語を用いた場合，この両者をまとめてモデル化することが多かった．COOP ではプログラム・モジュールとそれを制御する論理プロセッサ・モジュールの記述を分けることにより，両者を分離したモデル化が可能になっている．この両者を分離することによって，ある並列計算機上で異なるプログラムを実行したり，あるプログラムを異なる並列計算機上で実行するようなシミュレーション実験が容易になる．もちろん，両者を明確に分離しない記述も COOP では可能である．

（2） スケジューリングや通信のルーティング等の戦略を記述することができる．

シミュレーションの対象プログラムから見て，PE モジュールの論理プロセッサ・インタフェースの記述では，対象並列計算機の相互結合網のシミュレーションや時間の管理を行う．一方，対象プログラムの root モジュールの論理プロセッサ・インタフェースにおいては，そのプログラム独自のスケジューリングやアーキテクチャ上直接通信することのできない要素プロセッサ間の通信のルーティングなど，PE モジュールのシミュレートするアーキテクチャを補うオペレーティング・システムの機能を記述して実験することができる．

（3） 時間の管理が容易である．

COOP では，シミュレーション対象の計算機における処理時間をシミュレーションのホスト計算機の処理時間と同一であるという仮定の下で，シミュレーション対象の時間を進める．このアプローチは，処理時間をパラメタとして陽に与える必要がなく，仮定の成立する範囲で正確な時間が得られるという特徴を持つ．実際には，スレッド・ライブラリによってスレッドの実行に要した実時間を得ることができるので，論理プロセッサ・モジュールの記述においてその時間を累積することで時間の管理を行う．もちろん COOP では従来のように処理時間をパラメタとして陽に与えるシミュレーションも可能である．

（4） 同期機構の拡張が容易である．

論理プロセッサ・モジュールのプログラミングによってモニタ等の特殊な同期機構を実現することが可能である．COOP はこのような，並列処理に関する種々の概念の実験を支援することができる．

7.2 他のシステムとの比較

VM/EPEX[4] や ESMM[5] は，COOP/VM と同じく仮想計算機システム上に構築した並列処理の実験システムである．これらも仮想計算機上に任意台数の並列計算機を構成する点は COOP/VM と同様だが，ESMM と COOP/VM がメッセージ交換型の並列計算機を対象としているのに対し，VM/EPEX は，共有メモリを持った並列計算機上での SPMD (Single Program Multiple Data) と呼ぶ並列処理方式を対象としている点が異なる．特に ESMM は，仮想計算機の機能を拡張することにより効率のよいメッセージ通信機構を実現している．また，VM/EPEX や ESMM

では，COOP/VM のような仮想時間に対する同期という概念は陽には扱わない．

PRESTO[6] は汎用計算機の上に構築した並列処理の実験システムである．オブジェクト指向言語 C++ をベースに，様々な形態の並列処理を実現するための環境である点は COOP/VM と共通である．また，スレッドのスケジューリングを行うオブジェクトや同期をとるオブジェクトの記述をユーザに開放するという考え方は，COOP/VM の論理プロセッサと同じ目的を持つものである．しかし，PRESTO はメモリ共有型の並列計算機を対象として各種の並列処理方式の研究を行うためのソフトウェア指向のシステムであり，COOP/VM のように，対象並列計算機のアーキテクチャや，その上で経過する仮想時間の流れというものを明確に意識したものではない．

7.3 論理プロセッサのプログラミング機構

論理プロセッサのプログラミングは，COOP で記述したアプリケーション・プログラムのレベルから見ればメタレベルのインタプリタと考えることができる．このような観点から，システムが自己参照とその操作を行う「リフレクション」と呼ぶ考え方が研究され[7]，その強力さが示されている．並列に動作するシステムにおけるリフレクションはまだまだ未解明な部分が多く，現在さまざまな試みがなされている．COOP の論理プロセッサ・インタフェースのプログラミングは，記述可能な範囲がメッセージ送信の管理とスレッドのスケジューリングという限られた部分でしかないが，非常に制限されたリフレクションの一種であると考えることができる．しかし，COOP をリフレクションのための機構を持つ言語として見た場合，まだ不完全な言語機能しか持たず，そのセマンティクスも曖昧なものが多い．今後，リフレクションの観点から COOP 言語の機能を見直し，より強力な言語へと発展させていくことも必要である．

8. おわりに

COOP は，並列計算機やその上で実行する並列処理プログラムの設計・開発の初期段階でのプロトタイピングにおいて，より高精度のシミュレーションを行うためのシステムである．COOP では，論理プロセッサというエンジンとしての計算機構の抽象化概念を言語機能として取り込んだ．論理プロセッサのプログラミングにより，並列実行のスケジューリングという並列処理の本質的な部分をユーザ・レベルで記述可能にすることで，柔軟で汎用性の高い実験支援環境を提供することができる．

本論文で示した COOP/VM は，汎用計算機の仮想計算機システム上に開発した COOP 言語処理系の一実現例である．現在，各種の並列計算機が開発され，商用化されている．このような並列型の計算機上で COOP の処理系を実現し，シミュレーション自体の並列処理を実現していくのは今後の課題である．

COOP 言語の処理系は C 言語へ変換するプリプロセッサとして実現している．そのため，COOP 上で開発したプログラムは，設計・開発対象の並列計算機が完成すれば，その要素プロセッサのための C コンパイラさえあれば，その PE モジュールを実機上に用意することによってそのまま実行することができる．これにより，対象並列計算機のためのソフトウェア開発支援ツールとしても有効であり，並列処理の実験から開発に至るまでのトータル・システムとしての発展が期待できる．

参考文献

1) Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and Its Implementation*, p. 714, Addison-Wesley (1983).
2) Stroustrup, B.: *The C++ Programming Language*, p. 328, Addison-Wesley (1986).
3) IBM: *Virtual Machine/System Product High Performance Option System Programmer's Guide*, Release 4.2, SC 19-6224-6, p. 878, IBM (1985).
4) Darema-Rogers, F., George, D. A., Norton, V. A. and Pfister, G. F.: Environment and System Interface for VM/EPEX, *Research Report RC 11381*, p. 19, IBM (1985).
5) Hsieh, S. C.: An Extendable Simulator for Multi-processor Machines, *IEEE Second Int. Conf. on Comput. and Appl.*, pp. 359-365 (1987).
6) Bershad, B. N., Lazowska, E. D., Levy, H. M. and Wagner, D. B.: An Open Environment for Building Parallel Programming Systems, *Proc. ACM Symp. on Parallel Programming*, pp. 1-9 (1988).
7) 菅野博靖，田中二郎：メタ推論とリフレクション，情報処理，Vol. 30, No. 6, pp. 694-705 (1989).

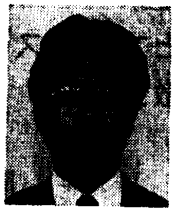
(平成元年8月28日受付)
(平成2年9月11日採録)

金井 達徳（正会員）

1961 年生．1984 年京都大学工学部情報工学科卒業．1989 年同大学院博士課程修了．1989 年(株)東芝入社．現在，総合研究所情報システム研究所に所属．京都大学在学中に本研究に従事．

藤井 啓明（学生会員）

1966 年生．1989 年京都大学工学部情報工学科卒業．現在，同大学院修士課程在学中．並列計算機システムの研究に従事．



柴山 潔（正会員）

昭和 26 年生．昭和 49 年京都大学工学部情報工学科卒業．昭和 54 年同大学院博士課程単位修得退学．同年同大学工学部情報工学教室助手．昭和 61 年同助教授，現在に至る．工学博士．計算機システム，計算機アーキテクチャなどの教育・研究に従事．電子情報通信学会，人工知能学会，IEEE, ACM 各会員．ICOT・WG 委員．昭和 61 年度本学会論文賞受賞．

萩原 宏（正会員）

大正 15 年生．昭和 25 年京都大学工学部電気工学科卒業．NHK を経て，昭和 32 年京都大学工学部助教授，昭和 36 年同教授，平成 2 年 3 月京都大学を停年退官．平成 2 年 4 月より龍谷大学理工学部教授．工学博士．情報理論，パルス通信，電子計算機などの研究に従事．昭和 31 年度稲田賞受賞．昭和 50，62 年本学会論文賞受賞．昭和 56～58 年度本学会副会長．著書「電子計算機通信 1～3」「マイクロプログラミング」など．電子情報通信学会，ACM，IEEE 各会員．