ソフトウェア工学 91－5
（1993. 3. 2）

UNIXネットワーク上の並列プログラム処理系の作成

高宮 広佳* 石黒 淳* 山田 剛** 小原 啓義*

早稲田大学理工学部* サリオンシステムズリサーチ **

並列プログラムのパラダイムとしてCSPを採用し、TCP/IPを用いたUNIXネットワーク上に並列プログラム処理系を作成した。具体的にはUNIXネットワーク上でCSPの同期・通信機構を実現するためにRPCを用いて、複数のプロセスを1つのプログラムとして管理するマネージャ、各プロセスを管理するマネージャ、プロセス間通信を管理するマネージャ、という3種類のマネージャを作成し、さらにCSPに基づいたOccamをソース言語としてOccamからC言語へのトランスレータを作成した。本稿ではこのような同期・通信機構を現状のUNIXネットワーク上に実現する際の問題点を考察し、3種類のマネージャの実現について述べる。

An implementation of parallel processing system on UNIX network

Hiroyoshi Takamiya* Atsushi Ishiguro* TsuyoshiYamada** Hiroyoshi Ohara*

School of Sci. and Eng., Waseda Univ.* Sarion Systems Research**

A parallel processing system which uses CSP as a paradigm, has been developed on the UNIX network employing TCP/IP. RPC was used to attain synchronization and communication of CSP on the UNIX network, and three types of managers were implemented; the first manager which manages several processes as a single program, the second which manages each process, and the third which manages interprocess communication. By a translator internal, the source language scripts, Occam based on CSP, are converted to C source codes. This paper focuses on the problems raised on the implementation of such synchronization and communication facilities on the UNIX network, and discusses the three types of managers.

1 はじめに

近年、ワークステーション、パーソナルコンピュータを相互に接続したLAN(Local Area Network)の普及はめざましく、このようなLAN環境において計算機資源を効率良く利用するために、ネットワークを対象としたオペレーティングシステムや周辺システムの研究が進められている。これらの研究により、多くのネットワークシステムが開発され実用に供されている。

しかしながら、従来の提案は主として資源の共有や負荷の分散に重点がおかれ[1]、並列プログラミングの環境としては不十分であった。将来、プログラミング環境としてネットワーク全体を利用した並列処理のサポートは必要不可欠なものとなることは間違いないと考えられる。

このような点から、我々はコンピュータネットワークを利用した並列処理環境の実現を目指すこととした。その際対象とするオペレーティングシステムとして、次のような理由からUNIXを採用することとした。

- システムについての詳細が公開されている
- ネットワークを利用したプログラミングにおいて強力な機能を持っている
- ワークステーションのオペレーティングシステムとしてほぼ標準として認知されており、さまざまな企業、教育機関において多くの人々に利用されている

もちろん、UNIXにはプロセスを並列に動作させる機構があるが、複数プロセスが協調して動作するということに関しては必ずしも十分であるとはいえない。

また、並列処理の重要な項目の一つであるプロセス間の通信には、その記述が容易であるという点から、送信と受信が対称的な形となる対等通信を行ないたいと考えた。しかし、UNIXに用意されているマシン間にまたがるプロセス間通信機構は、TCP/IPに基づいたクライアント-サーバモデルが主流であるため、送信と受信を対称的な形で記述することができず、複雑な関係を持つ通信処理の記述が困難であった。

本稿では、既存のUNIXシステムを拡張することにより、対等通信の機構を持つ並列プログラム実行システムをUNIXネットワーク上に実現する際の問題点を考察し、作成したシステムの構成について述べる。同時に、システムの検証のために作成した言語処理系についても述べる。

2 通信、同期機構

我々は、並列プログラムのモデルとしてHoareが提唱したCSP (Communicating Sequential Processes)[2] [3]を採用した。CSPとは、逐次的に動いている複数のプロセスが互いに通信し合いながら同時に動くことによって並列処理を行なう、というものである。CSPにおいては、同期と通信は密接な関係にありそれらは同じ概念で扱われる。すなわち、通信の際には常に、先に通信の命令に到着したプロセスが相手を待つことによって同期がとられる。

CSPではプロセスの通信、同期が簡潔にモデル化されており、また容易に他の同期・通信機構をシミュレートできる。また、通信の記述は送信と受信が対称的な形となり、我々の要求と合致することなどから本システムの通信、同期機構として採用した。

3 システム実現の問題点

UNIXネットワーク上にCSPに基づく並列処理システムを実現するための問題点を考える。

3.1 対等通信の問題点

今回、先に述べた同期、通信機構を実現するために、取り扱いが容易であり、異なるアーキテクチャ間でも正しくデータを送ることができる、という理由からRPC(Remote Procedure Call)を採用している。しかし、RPCはクライアント-サーバモデルに基づいているため、要求を出すクライアントと処理要求を待つサーバ、といった形をとり、対称的な形の相互通信ということには適していない。

そこで、対等通信を実現するためには何らかの機構が必要となる。

3.2 並列プロセスの問題点

並列プロセスの生成、実行には二つの問題が考えられる。一つはどのようにして複数のマシンで複数のプロセスの起動を行なうかということであり、もう一つは起動された複数のプロセスを一つの関連しあった集合として扱うにはどうするかということである。

- プロセスの配送と起動

  並列プロセスを他のマシンで起動させるためには、プロセスを起動させたいマシンがそのプロセスの実行ファイルに対してアクセス可能な状態にする必要がある。

  UNIX にはプロセスをリモート実行するための機能が存在するが、実行要求を出したプロセスは、リモートマシンで実行されるプロセスが終了するまでブロックされるということ、またリモートマシンでのプロセス ID の収得が困難であるという理由から、我々の要求には適していない。

- プロセスの識別

  システムはどのプロセスがどのマシンに存在するかを知っている必要がある。さらに、複数のマシンで動いている複数のプロセス一つのアプリケーション (我々はこれをジョブと呼ぶこととする) として認識されなければならない。よってネットワーク全体のプロセスの状態を把握するための機構が必要となる。

4 システムの実現

3で述べた問題点の解決法とシステムの実現について述べる。

4.1 対等通信の実現

クライアント-サーバモデルにおいて行なわれる通信は、クライアントがサーバに対して要求を出し、サーバは複数のクライアントからの要求を受け付けて返事を返す形となるため、サーバとクライアントは対等に送受信を行なう形にはならない (図 1)。

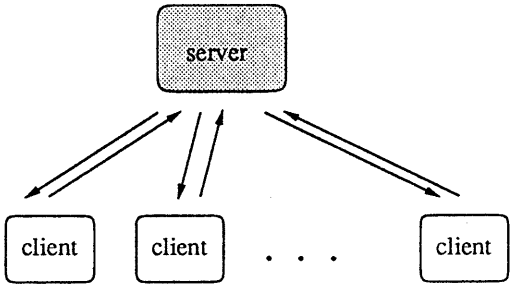

図 1: クライアント-サーバモデル

しかし、サーバをはさんだ二つのクライアントプロセスに注目すると、その二つのプロセスは互いに対称的な形である。つまり、サーバとクライアントの通信は非対称でも、間に入ったサーバに通信の仲介をさせ、クライアント同士がサーバに対して通信の要求、および結果の受け取りを行なうことによって対称的な通信を行なうことができる。

そこで、以下のようなステップで CSP の対等通信を実現することとする。

1. 通信を行なうプロセスは通信サーバに対し送信、または受信のリクエストを送る
2. もし通信が成立しなければ通信サーバは応答を保留し、リクエストを出したプロセスはブロックし続ける
3. 通信が成立したら受信側には送られるデータとともに ack が、送信側には ack だけが送られる

プロセスと通信サーバの関係を図 2に示す。

4.2 並列プロセスの実現

- プロセスの配送と起動

  プロセス配送の問題に関しては、現在 NFS(Network File System) などのファイル共有システムが実現されている。今回はこの機構を用いており、ファイル配送のための機構は特に考慮していない。

  起動に関しては、クライアント-サーバモデルに基づき、プロセス生成の要求を受けて各マシンにプロセスを起動するサーバを作成する。

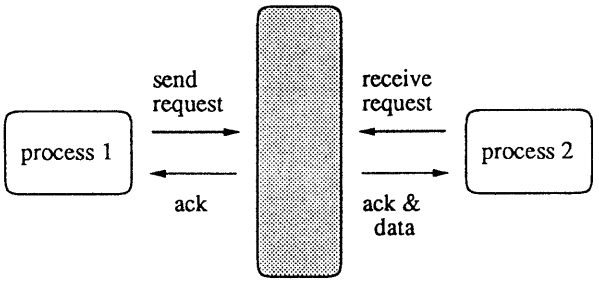

図 2: 通信サーバの概念

- プロセスの識別の実現

  プロセスの実行を管理する各マシン上のサーバの情報や、通信を管理するサーバの情報をもとに、ネットワーク全体のプロセスの状態を管理するサーバを作成する。

5 システムの構成

前述のように、通信の仲介をするサーバを Channel マネージャ、各並列プロセスの管理をするサーバを Process マネージャ、ネットワーク全体の状態を管理するサーバを Job マネージャと呼び、3 種類のマネージャによってシステムを構成した。以後、この 3 種類のサーバをまとめて CSP サーバと呼ぶ。

5.1 CSP サーバ

Job マネージャ、Process マネージャ、Channel マネージャはそれぞれが管理する情報をテーブルとして保持している。ここでは、各マネージャが管理するテーブルについて述べる。

5.1.1 Job マネージャ

Job マネージャは並列プロセスの管理とリクエストに応えるために二つのテーブルを内部に持つ。一つは Job テーブルで実行中の並列プログラムの状態を記録しているものであり、表 1 に示すエントリを持つ。新しい Job が実行される都度、その情報が新しいテーブルのエントリに登録される。

表 1: Job テーブルのエントリ

項目	内容
job_id	ジョブ id
map_file	マップファイルのファイル名
root_name	ジョブの中で最初に実行したプロセスのプロセス名
root_id	ジョブの中で最初に実行したプロセスのプロセス id
path	ジョブのワーキングディレクトリのパス
p_list	並列実行するプロセスのリスト
c_list	ジョブの中で使用するチャネルのリスト
m_list	ジョブが使用するマシンの名前のリスト
start_time	ジョブの実行が始まった時刻

表 1で、 p_list、c_list、m_list とは表 2〜4の項目を要素とするリストである。

Job マネージャが持つもう一つのテーブルはプロセスの終了待ちを管理する Wait テーブルである。この Wait テーブルは表 5 に示すエントリを持つ。このテーブルは他の並列プロセスの終了待ちを管理するために使われる。waiting_process には他の並列プロセスの終了を待つプロセスが登録される。waiting_process に登録されたプロセスは par_process_list に登録したプロセスが全て終了するまで実行を停止する。

5.1.2 Process マネージャ

Process マネージャはプロセスの管理のため内部に Process テーブルを持つ。Process テーブルのエントリは表 6 のようになっている。

表 2: p_list の要素

process_name	並列プロセスの名前
file_name	プロセスの実行ファイルのファイル名
machine_name	プロセスを実行するマシン名
stat	プロセスの状態

表 3: c_list の要素

channel_name	ジョブが使用するチャネルのチャネル名
machine_name	チャネルを管理するチャネルマネージャのあるマシンのマシン名

表 4: m_list の要素

machine_name	ジョブが使用するマシンのマシン名

表 5: Wait テーブルのエントリ

job_id	ジョブ id
waiting_process	終了を待つプロセスのプロセス名
waiting_process_id	終了を待つプロセスのプロセス id
par_process_list	並列プロセスのリスト

表 6: Process テーブルのエントリ

job_id	ジョブ id
process_name	プロセスの名前
p_id	プロセスの id
process_file	プロセスの実行ファイル
parent_process_name	親のプロセスの名前
parent_process_id	親のプロセス id
start_time	プロセスが実行を開始した時刻

5.1.3 Channel マネージャ

Channel マネージャは各並列プロセス間の Channel を用いた通信を実現し、転送されるデータを保持するために Channel テーブルを持つ。Channel テーブルのエントリは表 7のようになっている。

5.2 マネージャの配置

Process マネージャは各マシン固有の情報を管理するため、プロセスを生成する可能性のあるマシンすべての上になければならない。

Channel マネージャは通信の仲介をするが、必ずしもすべてのマシンにある必要はない。しかし、マネージャが少ないとプロセス間通信が集中して発生した場合にボトルネックになる危険性があり、Channel マネージャはある程度分散させるべきであろう。今回は、実験的に Channel マネージャを Process マネージャと同様にすべてのマシンに配置することとした。

Job マネージャは、ネットワーク上のプロセスの配置状況、およびジョブの構成の情報を管理しているため、プロセスからのリクエストが集中することが考えられる。しかし、Job マネージャはジョブ全体の情報を管理するという機能から分散させることは困難である。現段階では、Job マネージャはネットワークに一つとし、すべてのジョブの管理を行なわせている。

ネットワーク上のマネージャ配置の様子を図 3に示す。

表 7: Channel テーブルのエントリ

job_id	ジョブ id
channel_name	チャネルの名前
channel_id	チャネルの id
sender_name	送信側のプロセス名
sender_id	送信側のプロセス id
sender_machine	送信側のプロセスのあるマシン
receiver_name	受信側のプロセス名
receiver_id	受信側のプロセス id
receiver_machine	受信側のプロセスのあるマシン
trans_data	転送されるデータ
connect	接続状態を示すフラグ
standby	通信待ち状態を示すフラグ
caller	通信の呼出側を示すフラグ

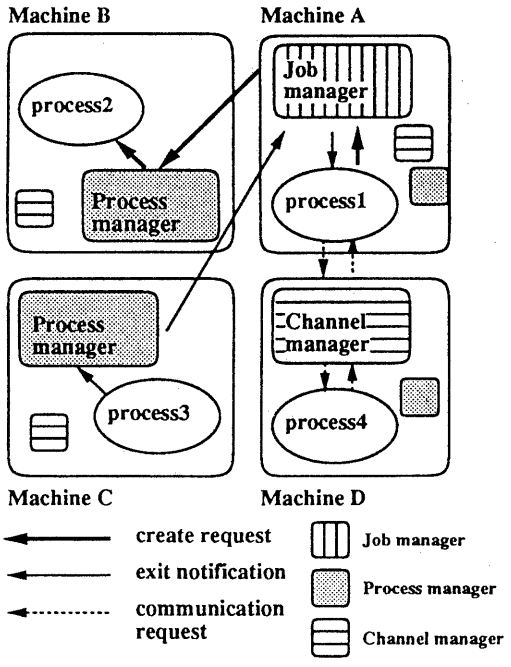

図 3: マネージャの配置

5.3 Occam トランスレータ

今回、並列処理システムのプリミティブとして CSP を採用し作成を行なったが、作成したシステムの検証、および利用という目的から、CSP に基づいた並列処理言語 Occam [4] の言語処理系を作成した。今回作成したのは Occam のプログラムを複数の C 言語のプログラムへ変換するものである。

6 システムの動作

ここでは、並列プロセスを動作させる際のマネージャ間のリクエストの流れ、および Occam トランスレータの処理について述べる。

6.1 マネージャ間のリクエストの流れ

マネージャ間のリクエストの流れを図 4に示す。

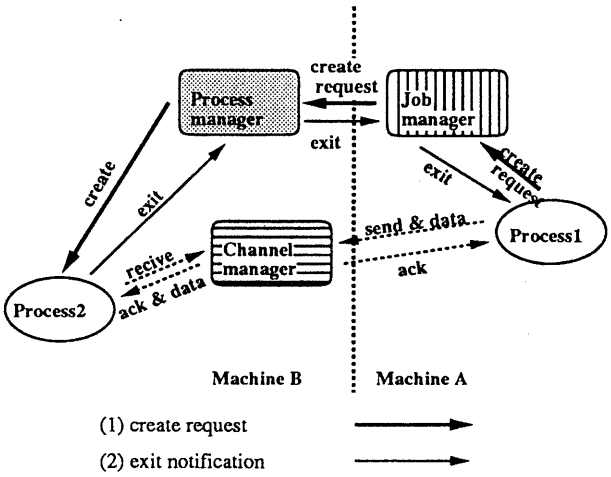

図 4: マネージャ間のリクエストの流れ

(1) process1 が process2 を起動しようとする時、その実行の流れは create request の矢印のようになる。まず、process1 が Job マネージャに対して "MachineB に process2 を生成せよ" という要求を出す。すると、Job マネージャは MachineB の Process マネージャに process2 を作るよう要請する。そして、MachineB の Process マネージャによって process2 が起動される。

(2) 一方プロセスの終了は、exit notification の矢印のようになる。プロセス自身が自分のいるマシンの Process マネージャに対して終了を知らせる。プロセス終了のシグナルを受けた Process マネージャは Job マネージャに対してそのプロセスの終了を知らせる。Job マネージャは、終了しようとしているプロセスの起動を依頼したプロセスに、そのプロセスの終了を知らせる。

プロセス間通信に関して、より具体的なリクエストおよびデータの流れを図 5に示す。

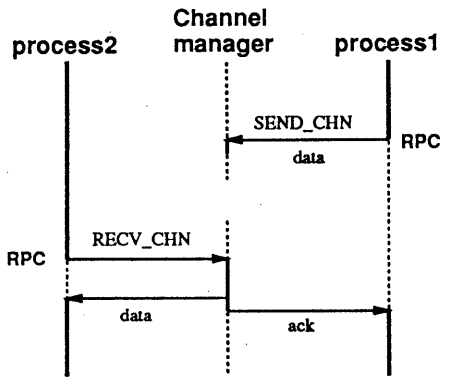

図 5: Channel マネージャを介したプロセス間通信

図 5では、process1 が process2 にデータを送信しようとしている。このとき process1 は Channel マネージャにリクエスト SEND_CHN を送る。process2 はデータを受信するので、同様にリクエスト RECV_CHN を送る。これらのリクエストを受けた Channel マネージャはそれぞれ対応するリクエストが来るまで応答をしない。そのため先にリクエストを送ったプロセスは対応するリクエストが来るまでブロックされる。そして、対応したリクエストが来ると Channel マネージャは双方のプロセスに応答を返す。これにより、通信が成立し送信側、受信側のプロセスが実行を再開する。こうして、CSP の対等通信が実現できる。

6.2 Occam トランスレータ

Occam トランスレータの処理の流れを図 6に、Occam トランスレータが出力するファイルを図 7に示す。

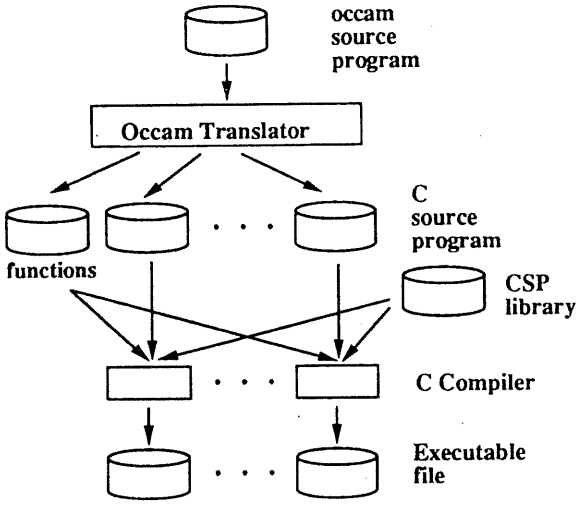

図 6: Occam トランスレータの処理の流れ

図 7において、main program は Job マネージャが存在するマシンで実行されるプロセスであり、parallel processes は並列に実行されるプロセスである。function definitions は関数定義をまとめたものである。chan_map はプロセスのチャネル利用状況について記述されたものである。

また、今回作成したシステムとのインターフェースとなる関数を作成し、これを CSP ライブラリと名付けた。C 言語のソースプログラムをコンパイルし CSP ライブラリとリンクすることにより、システム上で実行可能となる。

7 評価

Occam トランスレータの作成により、プログラム中で CSP の対等通信が容易に記述できるようになり、並列プログラムの記述性が高まると考えられる。

また、3 種類のサーバの作成により、UNIX 上で対等通信を用いた並列プログラムの実行が可能となっ

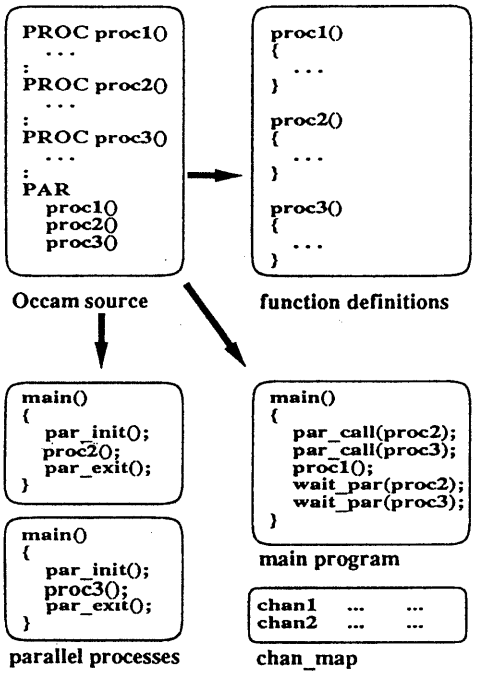

図 7: Occam トランスレータの出力

た。しかし、本システムは RPC の上層に構築するという形をとっているため、通信のオーバーヘッドはかなり大きい。また、ネットワーク上に Job マネージャが一つしか存在していないため、プロセスの数が増えると全体の処理速度が低下する。しかし、今回は試作段階であるため、処理速度の向上は今後の課題とする。

8 おわりに

以上、UNIX 上に CSP に基づく並列プログラムの実行環境の実現について述べた。

今後は非決定選択待ちなどのインプリメントを行なう。また、Job マネージャの複数配置による Job マネージャの負荷の軽減 [5]、およびプロセス再配置によるマシンの負荷の軽減 [6] を考慮し、より使いやすい並列処理環境にしていく予定である。

参考文献

[1] 前川・所・清水 編, "分散オペレーティングシステム-UNIX の次に来るもの-", 共立出版, 1991.

[2] C. A. R. Hoare, "Communicating Sequential Processes", ACM Comm., Vol.21, No.8, Aug., pp666-677, 1978.

[3] C. A. R. Hoare, "Communicating Sequential Processes", PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE, 1985.

[4] INMOS Limited, "occam2 Reference Manual", PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE, 1988.

[5] Philip A. Bernstein, Nathan Goodman, "Concurrency Control in Distributed Database Systems", ACM Comput. Surv., Vol.13, No.2, Jun., pp185-222, 1981.

[6] Roger S.Chin, Samuel T.Chanson, "Distributed Object-Based Programming Systems", ACM Comput. Surv., Vol.23, No.1, Mar., pp91-124, 1991.