マルチメディア通信と分散処理 58－17
（1992. 11. 20）

モジュールのパイプライン結合による分散処理の一方式

仙田 修司　美濃 導彦　池田 克夫
京都大学工学部

近年、コストパフォーマンス、耐故障性、拡張性といった点から分散システムが注目されている。しかし、分散システムにおける並列処理の実現には、負荷の分散、協調アルゴリズムの記述、障害からの回復といった問題に対処しなければならない。本稿では、モジュールの非同期呼び出しに基づく分散並列処理モデルを提案する。本モデルは、動的な負荷分散、メタモジュールの構成、障害発生時の自律的回復などの特徴を持つ。また、分散システムにおける共有メモリを実現する仮想ポインタを提案する。モジュール間の通信手段として仮想ポインタを用いることで、効率的なデータ転送とネットワークI/Oを意識しない分散処理プログラミングが実現できる。

A DISTRIBUTED PROCESSING METHOD BY PIPELINE CONNECTED MODULES

Shuji Senda　Michihiko Minoh　Katsuo Ikeda
Kyoto University

Recently, distributed systems are getting attention because of their cost performance, fault-tolerant feature and extendibility. But, in order to realize a parallel processing on a distributed system, we have to solve the problems of load balancing, programming for cooperative algorithms, and fault recovery. In this paper, we describe a Distributed Parallel Processing model which is based on asynchronous remote module calls. The advantages of the model are dynamic load balancing, construction of meta-module and autonomous fault recovery. We also describe a implementation of virtual shared memories in distributed systems in which Virtual Pointer is defined to point the address of the virtual shared memory. Using VP as a communication method among modules enables us to transfer data efficiently. At the same time, we can easily write distributed applications.

1 はじめに

単体ハードウエア性能の限界、高機能マイクロプロセッサの出現、ネットワーク技術の発展といった要因から、分散システムによる分散処理が注目されている。分散システムの特徴は、ネットワークで接続された様々な資源を共有することによって、小規模の機器群を組み合わせた大規模な仮想的システムを構築できることである。こうした分散型大規模システムは、コストパフォーマンス、耐故障性、拡張性といった点で集中型の大規模システムよりもすぐれている。

プリンタサーバやファイルサーバといった特定の機能を共有する機能分散に関してはすでに実用段階にある。しかし、複数のプロセッサを用いて処理速度の向上をめざす負荷分散に関しては未だに実用には至っていない。この理由として、協調しながら並列動作を行うプログラムを作成することが困難であること、分散システムは疎結合のシステムであるためネットワーク通信のオーバーヘッドが密結合のシステムに比べて非常に大きいことなどが考えられる。

本稿では、モジュールの非同期呼び出しに基づく分散並列処理モデルを提案する。このモデルは、入力のみから出力が決定する関数的なモジュールを分散処理の単位し、動的な負荷分散、メタモジュールの構成、障害発生時の自律的回復といった特徴を持つ。また、モジュール間での通信手段として、分散システムにおける仮想的な共有メモリを実現する方式を提案する。具体的には、仮想ポインタを用いることにより、アプリケーションプログラマはネットワークI/Oの詳細を意識することなく、分散処理プログラムを作成することが出来る。

2 分散並列処理モデル

2.1 従来の分散処理

従来の分散システムでは、遠隔手続き呼び出し(RPC)を用いて処理の分散を行っていた。RPCを用いた分散処理の利点として、明解で簡潔なセマンティクスを持つこと、効率がよいこと、一般性があることが挙げられている[1]。

現在最も普及しているRPC/ONC[2]などの同期型のRPCでは、呼び出し元は呼び出し先の処理が終わるまでブロックされる。これに対して、非同期RPC[3]では、呼び出し元と呼び出し先が並列に動作することができ、処理速度の向上が期待できる。

並列処理による処理速度の向上のためには、分散システム内での負荷のバランスを考慮する必要がある[4]。システム構成の変化などにも対応した柔軟な負荷分散を行うためには、処理の割り付けを実行時に決定する動的負荷分散が必要となる[5]。

動的に負荷の分散を行うと、ある処理がどのプロセッサで実行されるかは非決定的である。このような理由から、処理を請け負うモジュールは、その内部に静的なデータ保持することは出来ない。つまり、与えられた入力だけから出力を得る必要がある。

ネットワークを介したデータ転送速度は、計算機の処理速度と比較すると非常に遅い。そのため、分散システムではデータの転送をどのように行うかが問題となる。特に、多量のデータを転送する必要がある場合には、データの到着順に処理を行うパイプ処理[6]などの工夫が必要となる。

本稿で提案する分散並列処理モデル(Distributed Parallel Processing model, 以下DPPモデル)は、サーバクライアントモデルによる関数型のモジュールの非同期呼び出しをその基礎としており、次の様な特徴を持つ。

- モジュールは内部に静的なデータを持たず、それへの入力のみから出力が決定する。
- モジュールを非同期に呼び出すので、モジュールの並列実行が可能である。
- 処理を実行するプロセッサを動的に決定することにより、負荷の分散と耐故障性の向上が期待できる。
- データ転送プロトコルとして仮想ポインタを用いることで、多量のデータを必要に応じて転送することができる。
- モジュールの入出力を結合していくことで、複数のモジュールを組み合わせたメタモジュールを構成することができる。
- モジュールの関数的な性質と仮想ポインタの使用により、処理を実行中のプロセッサがダウンした場合でも自律的に処理の回復を行うことが出来る。
- システム内のメタデータを管理する集中型のマネジャが存在しないので、マネジャのダウンによるシステム全体の停止といったボトルネックがない。

2.2 モジュールとモジュールインスタンス

DPPモデルにおけるモジュールは、内部に静的なデータを持たず、入力データのみから出力データを

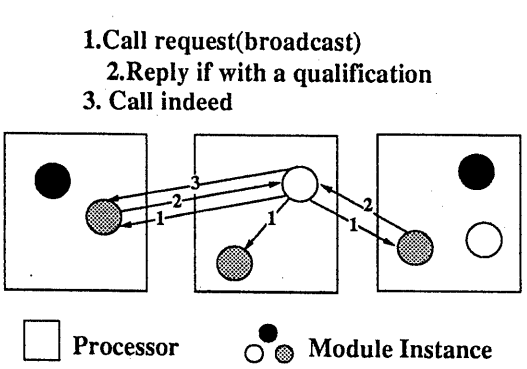

図 1　モジュールの呼び出し手順

得る関数的な動作をする。よって、モジュールとは、処理の機能を表すテンプレートであると定義できる。これに対して、実際にプロセッサ上で処理を行う実体をモジュールインスタンスと呼ぶ。

あるモジュールに対応するモジュールインスタンスは同時に複数個存在し得る。ユーザがモジュールの呼び出しを要求すると、システムはそのモジュールに対応するモジュールインスタンスの中から一つを選択して、実際には、そのモジュールインスタンスを呼び出す。このモジュールインスタンスの選択を適切に行うことによって、動的な負荷分散を達成できる。

具体的には、以下の手順によりモジュールインスタンスの選択を行う (図 1)。

1. あるモジュールを呼び出すモジュールインスタンス (クライアント) は、モジュールの呼び出し依頼をネットワーク内にブロードキャストする。この呼び出し依頼は、呼び出されるモジュール (サーバ) に引き渡すパラメータと、サーバの資源に関する条件 (例えば、負荷の上限や空きメモリの下限など) から構成される。
2. サーバは、クライアントの依頼内容を見て、依頼の条件を満たしている場合にかぎりクライアントに返事を返す。
3. クライアントは最初に返事を返してきたサーバを呼び出す。

この手順は、クライアントとサーバの間における双方向の局所的な情報交換に基づく契約手続きである [7]。条件を満たすサーバが複数あった場合、最も単純な方法として最初に返事を返してきたサーバを選択する。この方法の代わりに、ある一定期間内に返事を返してきた全てのサーバの中から最適なものを選ぶことも出来るが、現実問題として、何をもって最適とするかの判断が難しく、一定期間待つという時間の無駄のほうが大きいと考えている。

2.3　ストリームと仮想ポインタ

モジュールインスタンス間を結ぶ通信路としてストリームと呼ばれる FIFO バッファを用いることができる。ストリームを用いる利点として以下のようなものが挙げられている [8]。

- データの流れに従って、アルゴリズムが素直に記述できる。
- 複数のモジュールをストリームで結合することにより、大きなシステムを構築できる。
- モジュールの入出力をストリームに限定することで、モジュールの再利用性が高くなる。

DPP モデルの初期の実装 [9] では、モジュールインスタンス間の通信路としてストリームを採用し、それを TCP/IP プロトコルで実現していた。しかし、実装の問題から 1 入力ストリーム 1 出力ストリーム以外の構成が困難であるという欠点があった。

本稿では、モジュールインスタンス間のデータ転送プロトコルとして仮想ポインタによる共有メモリ方式を用いる。仮想ポインタを用いることによって、次の様な利点が得られる。

- 複数の入出力路を持つモジュールが記述できる。
- ネットワーク I/O に関する詳細をアプリケーションプログラマから隠すことができる。
- 参照されるデータのみを必要に応じて転送するので、一部分のデータしか参照されない場合には効率がよい。
- 障害からの回復を行う際に、通信路上のデータを考慮しなくてよい。

具体的には、モジュールの入出力は次の様に定義される。

入力　呼び出しパラメータ (param1, param2, ...)
　　入力仮想ポインタ (in1, in2, ...)

出力　出力仮想ポインタ (out1, out2, ...)

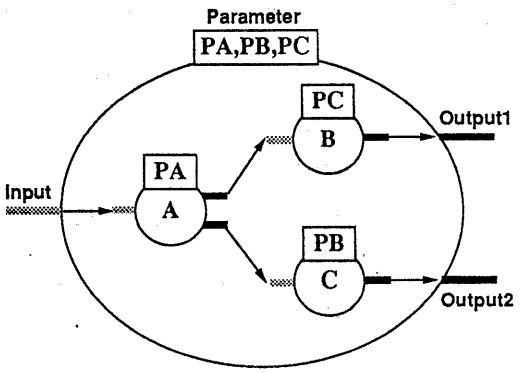

図 2　メタモジュールの例

呼び出しパラメータは、モジュールの細かな動作を決定するためのものである。入力仮想ポインタは実際に処理を行うべきデータを与え、出力仮想ポインタは処理結果を与える。例えば、濃淡画像を 2 値画像に変換するモジュールを定義する場合、呼び出しパラメータとして 2 値化の閾値、入力仮想ポインタとして濃淡画像、出力仮想ポインタとして 2 値画像を割り当てることができる。

2.4　パイプライン結合によるメタモジュールの構成

あるモジュールの出力を別のモジュールの入力に繋げることをパイプライン結合と呼ぶ。モジュールのパイプライン結合によって、複数のモジュール間のデータの流れを定義したメタモジュールを構成することが出来る。メタモジュールは、データフロー [10] によるプログラムの一種と考えることが出来る。

例えば、図 2のようなパイプライン結合によって、3 つのモジュールから構成される 1 入力ストリーム 2 出力ストリームのメタモジュールが定義できる。メタモジュールの呼び出しパラメータは、メタモジュール内の全てのモジュールに対する呼び出しパラメータを合わせたものとなる。

メタモジュールを利用する利点は次の様なものである。

- 小さなモジュールを組み合わせて大きなメタモジュールを構成することが出来る。
- メタモジュール内の各モジュールに制御を分散することにより、それらは自律的に動作する。

2.5　障害からの自律的な回復

何らかの障害によって、処理を実行中のモジュールインスタンスが異常終了した場合、DPP モデルでは、異常終了したモジュールを再起動することができる。この場合、異常終了したモジュールインスタンスとは異なるモジュールインスタンスが起動されることになるが、モジュールの関数的な性質と仮想ポインタの使用により、全体の処理の整合性が保たれる。

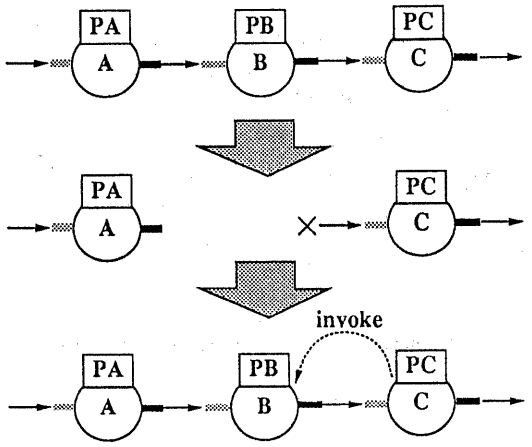

図 3　障害からの自律的回復

例えば、図 3のようにパイプライン結合した三つのモジュール A,B,C があるとする。これらのモジュールインスタンスが稼働中である時に、モジュールインスタンス B(以下、単に B) が動作していたプロセッサが故障したとしよう。この場合、B の出力仮想ポインタを受け取っているモジュールインスタンス C(以下、単に C) は、TCP/IP の機能により B との接続が切断されたことを知る。そこで、C はモジュール B を再起動して処理を継続するのである。

このように、C がモジュール B を再起動するには、B の起動パラメータと B の入力仮想ポインタ (A の出力仮想ポインタ) を C に与えておくだけでよい。これらの情報は、C を起動する際の起動パラメータの一部として与えることが出来る。

3　仮想ポインタによるメモリ共有

3.1　分散システムのための通信手段

異なるプロセス間でデータの授受を行うには、主として二通りの方法が存在する。一つは、共有メモリを介したものであり、もう一つは、メッセージ交換によるものである。

共有メモリを介したデータの授受の場合、ポインタを介した参照呼びだしを行うことで大量のデータ

を効率的に引き渡すことができる。しかし、ネットワークで接続された分散システムでは、共有メモリの実現そのものが非常に困難である。また、共有メモリを介してデータの授受を並列に行う場合には、何らかの同期を取る手段が必要となる。

一方、メッセージ交換によるデータの授受は、ネットワーク技術をそのまま利用して実現することができる。また、同期の問題に関しては、FIFO による転送を行うことによって解決出来る。よって、分散システムでは、通常、メッセージ交換によるデータの授受を採用する。しかしながら、アプリケーションプログラム作成の容易さ、大量のデータ中の一部のみを利用する場合の効率に関して、メッセージ交換方式は共有メモリ方式に劣ると考えられる。

そこで、本章では、仮想的な共有メモリ機能を実現するための仮想ポインタ (Virtual Pointer, 以下 VP) を提案する。VP は、参照のみが許される読みだし専用の共有メモリ機能を提供する。読みだし専用という制限を課したことによって、VP は既存のネットワークプロトコルで容易に実現可能となり、メッセージ交換によるデータの授受と同等の効率でデータを転送することができ、同期を取る必要もない。また、メッセージ交換方式とは異なり、VP が指すメモリはユーザから見れば通常の内部メモリと同等とみなせるので、ネットワーク I/O を意識することなく共有メモリ空間に対してランダムなアクセスが可能である。よって、VP を用いれば、既存のプログラムに多少の修正を加えるだけで分散システムにおける並列実行を行うことが出来る。

3.2 分散システムにおけるポインタ

ポインタとはメモリの位置を指すものである。単一プロセッサのシステムでは、ポインタはそのプロセッサの内部アドレスとして表現できる。それに対して、分散システムにおけるポインタは、ネットワーク内のあるプロセッサ上の内部アドレスを指し示すものである。そこで、そのような分散システムにおけるポインタを仮想ポインタ (VP) と呼ぶ。VP は、ネットワーク上でのプロセッサの位置とそのプロセッサの内部アドレスによって表現される。

VP によって指されるメモリを持つプロセッサを VP マスタ、VP を用いて VP マスタ上のメモリを参照するプロセッサを VP スレイブと呼ぶ。VP によるメモリ参照の概念図を図 4に示す。

3.3 読みだし専用の共有メモリ

VP によって共有されたメモリは読みだし専用である。言い換えれば、VP が指し示す共有メモリは、

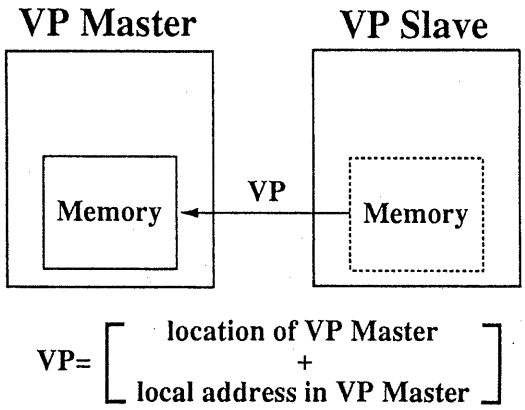

図 4 仮想ポインタによるメモリ参照

VP マスタ上のプロセスが一度だけ書き込みを行うことができるが、それ以外には参照することしか出来ないことを意味する。そこで、このような読みだし専用の共有メモリの利点と欠点について考察してみる。

利点 VP スレイブは一度参照したメモリを全てキャッシュすることができる。このため、不必要なトラフィックは全く起こらず、効率的なデータ転送を行うことができる。また、共有メモリのアクセス制御の問題が生じないので、同期を取る必要がない。

欠点 並行して協調動作を行うプロセス群を記述することが困難である。

複雑な協調動作を必要とする並行アルゴリズムは、通信速度の遅いネットワーク結合による分散システム環境にはもともと適しておらず、密結合のマルチプロセッサ向きであるといえる。本稿で提案する分散並列処理のモデルは、対等な関係の協調ではなく、主従関係のあるサーバクライアント型のアルゴリズムから並列性を引き出すことを目的としている。この目的には、読みだし専用という制限を課した共有メモリであっても有効に機能する。

3.4 仮想ポインタの実現

本節では、VP の機能を実現するための詳細について述べる。VP は、VP マスタのインターネットアドレスとその内部アドレスから構成されることは既に述べた。実際の実装では、さらに、VP への読みだし要求を受け付ける TCP ポート番号と参照可能

なメモリの大きさの情報を付加した次の様な構造体によってVPは表される。

```
struct vp {
    int addr;
    int port;
    void *magic;
    int size;
}
```

ここで、addrはVPマスタのインターネットアドレスを表し、portはVPマスタ上に割り当てられたTCPポート番号である。magicはVPマスタの内部アドレスであり、sizeは参照可能なメモリの大きさが格納されている。

また、VPスレイブでは、VPマスタとの接続や読みだし状況を管理するためにさらにいくつかの情報を付加した以下のような構造体を用いてVPを表す。

```
struct vp_clt {
    struct vp vp;
    int fd;
    long *map;
    char *body;
}
```

vpはVPマスタから受け取ったVPであり、fdはVPマスタと接続しているソケットである。mapは既にマスタVPから読み込みを行ったメモリをページ単位で記憶している32ビット整数の配列である。1ページ当り1ビットで記憶するので、1ページの大きさを P バイトとすれば、map配列の大きさは、

$$\left\lfloor \frac{\left\lfloor \frac{size+P-1}{P} \right\rfloor + 31}{32} \right\rfloor$$

ワードとなる。bodyは、ユーザプログラムが実際に参照を行うためのVPスレイブの内部アドレスである。VPのユーザから見れば、bodyからsizeバイトの領域が仮想的にVPマスタのmagicからsizeバイトの領域と同一であるとみなせる。以上の情報を加えて図4を書き直したものが図5である。

次に、実際にVPを介してメモリを参照する様子を順を追って説明する。

1. VPは、他のプロセッサからの参照を可能としたいメモリの先頭アドレスとその大きさを引数として、VPマスタライブラリで提供されるvp_svr_init 関数を呼び出すことにより作成される。

   ```
   struct vp *p = vp_svr_init(magic, size);
   ```

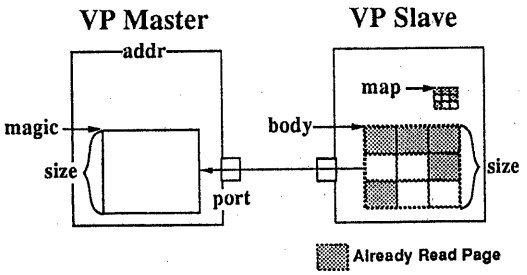

図5 仮想ポインタの実装

VPを作成したプロセッサはVPマスタとなり、空いているTCPポートがVPマスタ用に割り付けられ、VPスレイブからの接続を待つ。

2. 作成されたVPは、何らかの手段で他のプロセッサに渡される。VPを他のプロセッサに渡すためには、VP以外の通信プロトコルも必要となる。

3. VPを受け取ったプロセッサはVPスレイブとなり、VPマスタで提供されたメモリを参照することが出来る。まず、VPマスタから受け取ったVPを引数として、VPスレイブライブラリで提供されるvp_clt_init関数を呼び出す。これにより、VPマスタとVPスレイブの間にTCPプロトコルによる接続が確立する。

   ```
   struct vp_clt *q = vp_clt_init(p);
   ```

   mapは全て0に初期化され、bodyにはsizeバイトのメモリが割り当てられる。

4. VPで指された共有メモリを参照したい場合、参照したいメモリのVPからの相対位置をposとして、次の様な参照要求を行う。ただし、pはvp_clt構造体へのポインタである。

   ```
   vp_clt_mapping(p, pos);
   ```

   vp_clt_mapping関数の内部では、map配列を調べて参照要求のあったメモリが既に読み込まれているかをチェックし、まだ読み込まれていない場合にかぎり、VPマスタに1ページ分のメモリ読みだし要求を行って、map配列を更新する。

   このように、1バイト単位で参照要求を行っていると関数呼びだしのオーバヘッドが大きいので、連続するメモリに対して一度に参照要求を出すことが出来る。

   ```
   vp_clt_maparea(p, pos, size);
   ```

vp_clt_maparea 関数では、pos から size バイトの領域を一度に参照可能にする。この際に、複数ページに跨るような参照要求の場合には、VP マスタへの読みだし要求は一回にまとめて送られる。

5. VP マスタ上では、VP として割り当てたメモリのどこまでを既に書き込んだかを次の様な関数で VP マスタライブラリに知らせる。

   vp_svr_mapping(p, pos);

   pos よりも後のメモリに対する読みだし要求がきた場合は、その要求はキューに入れられ、要求元の VP スレイブはブロックされる。この機能によって、VP を用いた FIFO を実現することが出来る。

6. VP マスタでは、VP スレイブからの要求を待つソケットには非同期 I/O によるシグナルが発生するように設定しておく [11]。非同期 I/O を用いたことで、通常の処理とは無関係に VP スレイブからの要求に答えることができる。

7. VP スレイブが VP を必要としなくなれば、以下の関数により、VP マスタとの接続を切断し、資源を開放する。

   vp_clt_finish(p);

   VP マスタは、VP スレイブとの接続が切れたことを検出し、その VP がどのプロセッサからも参照されなくなると、VP を開放する。以後、この VP に対する参照要求は受け付けられない。

8. VP マスタのユーザプログラムは終了するまえに、次の関数により VP の開放を待つ必要がある。

   vp_svr_finish(p);

   上記の関数内では、そのプロセスが提供している全ての VP が開放されるまで、非同期 I/O によるシグナル処理のみを行う。

上記の手順からわかるように、VP には、大量のデータ参照とランダムなデータ参照における効率の向上、VP マスタ側のメモリの自動開放といった特徴がある。

3.5 仮想ポインタの効率測定

以下の条件のもとで VP の実装に関する効率の測定を行った。

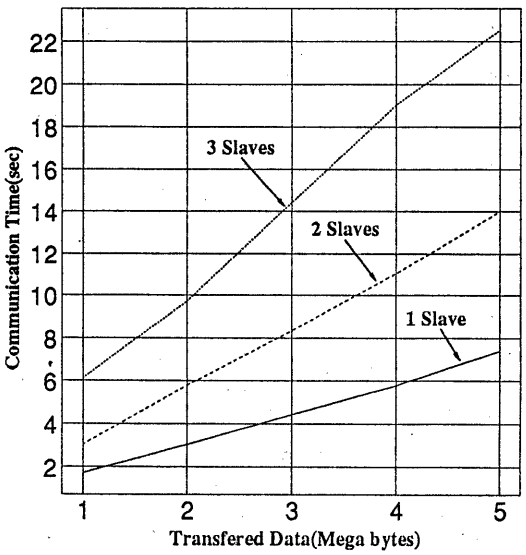

図 6 通信量と通信時間

- プロセッサとして、単一の Ethernet で結合された Sparc Station 4 台を用いる。
- あるプロセッサを VP マスタとして、UDP/IP によるブロードキャストパケットによって、同時に複数 (1-3 台) のプロセッサに VP を配る。
- VP を受け取った VP スレイブは、参照可能な全メモリを連続して参照する。
- VP マスタが起動されてから、VP スレイブによる共有メモリの参照が行われ、VP スレイブが VP マスタとの接続を全て切断して VP が開放されるまでの時間を実時間で測定する。

VP で参照可能なメモリの大きさ、VP スレイブの数、ページサイズをそれぞれ変化させて、上記の実験を行った。実験の結果を図 6および図 7に示す。

図 6の実験では、vp_clt_maparea 関数を用いているので、ページサイズに関係なく共有メモリの参照要求は一度しか行われない。そのため、実験の結果得られている通信速度は、VP スレイブの数が 1 の場合、TCP プロトコルの通信速度と同等の $600k - 700kbytes/sec$ 程度となっている。また、VP スレイブの数に比例して通信時間が増加していることから、大量のデータを複数のプロセッサに転送する場合には、VP マスタがボトルネックとなることが分かる。

図 7の実験では、$1Mbytes$ のデータに対して、ページサイズを変化させて 1 ページずつ参照要求を出した

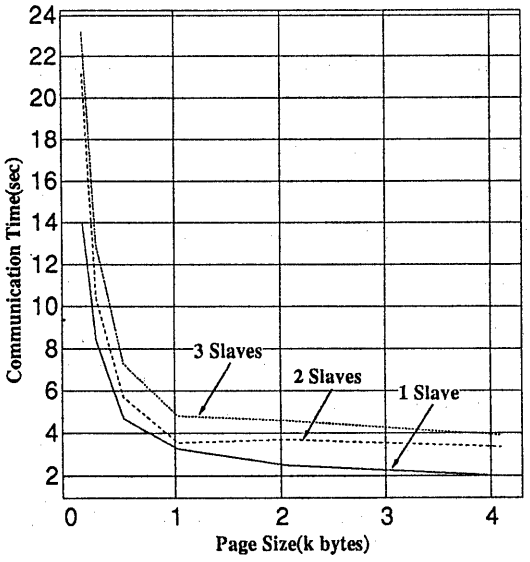

図 7　ページサイズと通信時間 (通信量 $1Mbytes$)

場合の通信時間を測定している。この結果から、ページサイズが $1kbytes$ 以下では、通信のオーバーヘッドが非常に大きく大量のデータを転送するのに適さないことがわかる。これは、Ethernet の最大パケット長が $1526bytes$ であることが原因であると考えられる。よって、効率のよいデータ転送を行うには、ページサイズを $1kbytes$ 以上にするのがよい。ページサイズを $1kbytes$ としたときの通信速度は $330kbytes/sec$ 程度である。

4　おわりに

モジュールの非同期呼び出しに基づく分散並列処理モデルを提案し、その詳細について述べた。本モデルの特徴は、動的な負荷分散を行えること、複数のモジュールを組み合わせたメタモジュールを構成しデータフローによる処理が行えること、障害発生時に自律的な回復が行えることなどである。また、モジュール間での通信手段として、仮想ポインタによる共有メモリ方式を提案し、その効率について実験と考察を行った。

今後は、DPP モデルに基づいたアプリケーションを作成し、実用上の問題について検討して行きたい。また、メタモジュールのビジュアルなエディットや、分散処理プログラムのデバッグ手法についても考察を行う予定である。

参考文献

[1] Birrell, A.D. and Nelson, B.J.: Implementing Remote Procedure Calls, ACM trans. on Computer Systems, vol.2, no.1, pp. 39-59(1984).

[2] Corbin, J.R.: The Art of Distributed Applications, Springer-Verlag(1991).

[3] Walker, E.F., Floyd, R. and Neves, P.: Asynchronous Remote Operation Execution in Distributed Systems, Proc. 10th International Conference of Distributed Processing, pp. 253-259(1990).

[4] Wang, Y., Morris, R.J.T.: Load Sharing in Distributed Systems, IEEE Trans. on Comput., vol.C-34, no.3, pp. 204-217(1985).

[5] Singh, A., Schaeffer, J. and Green, M.: A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations, IEEE Trans. on Parallel and Distributed Systems, vol.2, no.1, pp. 52-67(1991).

[6] Gifford, D.K. and Glasser, N.: Remote Pipes and Procedures for Efficient Distributed Communication, ACM Trans. on Computer Systems, vol.6, no.3, pp. 258-283(1988).

[7] Smith, R.G.: The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver, IEEE Trans. on Comput., vol.C-29, no.12, pp. 1104-1113(1980).

[8] 久世和資, 佐々政孝, 中田育男: ストリームによるプログラミングのための言語とその実現方式, 情報処理, vol.31, no.5, pp. 673-685(1990)

[9] 仙田修司, 美濃導彦, 池田克夫:"文書画像理解のための分散処理方式," 第 44 回情報処理学会全国大会, 1-233(1992).

[10] Davis, A.L. and Keller, R.M.: Data Flow Program Graphs, Computer, vol.15, no.2, pp. 26-41(1982).

[11] Stevens, W.R.: Unix Network Programing, Prentice Hall, 1990.