システムソフトウェアと
オペレーティング・システム　71－19
マルチメディア通信と分散処理　73－19
（1995．12．1）

レイヤ単位の並列処理を用いた通信プロトコルプログラムの実装と評価

佐藤 友実[†]　　加藤 聰彦[††]　　鈴木 健二[††]

電気通信大学[†]　国際電信電話（株）研究所[††]

伝送路の高速化に伴い通信プロトコルの高速処理方式が必要となっている。最近では、共有メモリ型マルチプロセッサ構成のワークステーションが広く普及しているため、このようなワークステーション上において、並列処理方式によりプロトコル処理を高性能化するための検討が重要であると考えられる。これに対し筆者らは、OSが提供するスレッドを用いて1つのレイヤのプロトコル処理を個別のプロセッサに割り当てる、レイヤ単位の並列処理方式を提案している。この検討に基づき、市販の共有メモリ型マルチプロセッサ上に、並列処理用ライブラリを実装するとともに、それを用いて、コネクションレス型およびコネクション型通信プロトコルによる評価用プログラムを開発し、その性能評価を行った。本稿では、これらの結果について述べる。

Implementation and Evaluation of Protocol Program using Processor-per-Layer Parallel Processing Method

Tomomi Sato[†]　　Toshihiko Kato[††]　　Kenji Suzuki[††]

Univ. of Electro-Communications[†]　　KDD R & D Laboratories[††]

As the network transmission bandwidth increases, the high performance processing method of communication protocols is required. Because shared memory multiprocessors workstations are widely used, it is important to study the implementation of the high performance parallel processing of communication protocols on such workstations. We have proposed an approach which assigns one processor to prcess one layer. According to this approach, we have developed a parallel processing library, and implemented evaluation programs for connectionless and connection oriented protocols. This paper describes the implementation of the parallel processing library and the evaluation programs, and the results of their performance evaluation.

1 はじめに

近年、100Mbps を越える超高速ネットワークの導入に伴い、それらに接続されるコンピュータにおいて、高性能な通信プロトコル処理の実現が求められている。一方、最近では、処理の高速化を目的として、メモリを共有するマルチプロセッサ構成を採用したワークステーションが広く普及している。従って、市販のワークステーション上で高速な通信プロトコル処理を行うためには、共有メモリ型マルチプロセッサを用いた通信プロトコルの並列処理方式の検討が重要である。

これに対し、筆者らは1つのレイヤのプロトコル処理を1つのプロセッサに割り当て、レイヤ単位の並列処理を行い、複数レイヤの処理をパイプライン的に行う方式を提案している [1]。本方式では、OS が提供するスレッドを用いて、各レイヤの処理を別個のプロセッサに割り当てるとともに、レイヤ間のインタフェースやレイヤ間で共有されるバッファの管理などにおいて、レイヤ間の同期のオーバヘッドをできるだけ減らす方法を採用している [1]。

これらの検討に基づき筆者らは、市販のワークステーション上で本方式を実現するための並列処理用ライブラリを実装した。さらに、本ライブラリを用いて、コネクションレスプロトコルと、コネクション型プロトコルをそれぞれ複数レイヤ積み重ねた評価用プログラムを作成し、その性能評価を行った。本稿では、その結果について報告する。以下本稿では、筆者らが提案している並列処理方式について2章で述べ、3章において作成した並列処理用ライブラリと、それを用いた通信プログラムの実装方法について述べる。4章で、評価用のプロトコル並列処理プログラムを用いた性能評価について示す。

2 並列処理方式

2.1 概要

レイヤ単位の並列処理を実現するために、以下のような方針を採用した。

- 1つのレイヤのプロトコル処理を、オペレーティングシステムでサポートされているスレッドにより実現し、各スレッドを個別のプロセッサに割り当てる。
- プリミティブや PDU(Protocol Data Unit) など情報は、ヒープ領域に確保されたグローバルバッファ上に作成し、複数のレイヤスレッドから参照可能とする。
- 各レイヤスレッドは、キューを介して、グローバルバッファ上のプリミティブへのポインタをやりとりする。
- レイヤ間のキューへのアクセス、グローバルバッファ上の情報へのアクセス、グローバルバッファの確保・解放などの管理において、各レイヤスレッドが他のレイヤスレッドとなるべく同期をとらずに動作するような方式を採用する。

プロトコルの並列処理を行うプログラムの全体構成は、図1に示すようになる。

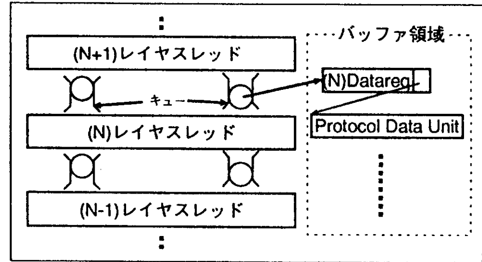

図 1: プログラムの全体構成

本並列処理方式では、図2に示すように、プロトコルはレイヤ毎にパイプライン的に処理される。あるレイヤが1つのプリミティブの処理を終了し、隣接レイヤにプリミティブを送信する。プリミティブを受信した隣接レイヤはその処理を開始する。この間、プリミティブを送信したレイヤは次のプリミティブを処理する。

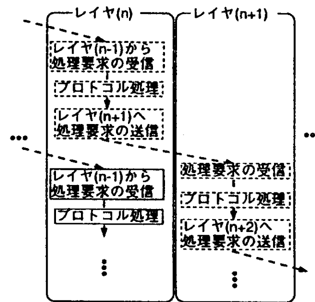

図 2: 並列処理の流れの概要

2.2 レイヤ間のキュー

本方式では、レイヤ間のキューに対して、キューへの受信と送信を行うスレッドをそれぞれ1つに限定し、以下のような構造のキューを用いることにより、キューへのアクセスのための同期を不要としている。すなわち、キューは、図3に示すように、プリミティブのポインタが格納される固定長 (QUEUE_MAX) の配列と、キューの先頭と最後を指し示す `head` 変数と `tail` 変数から構成される。キューへの書き込みには `tail` のみ参照し、その要素が '0' ならば、プリミティブのポインタを書き込み、`tail` を1進める (`tail` に `((tail+1)%QUEUE_MAX)` を代入する)。一方読み込みでは、`head` を参照し、それが示す要素にプリミティブがつながれているときはプリミティブを読みだし、その要素に '0' を代入して `head` を更新する。

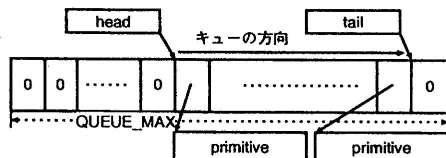

図 3: キューの構造

2.3 グローバルバッファの管理

プリミティブやPDUなどのためのグローバルバッファは、複数のレイヤスレッドにより確保・解放される必要がある。本並列処理方式では、その際のスレッド間の競合をなるべく避けるために、グローバルバッファの管理方法に対して、以下の方針を採用した。

- レイヤ毎にグローバルバッファ用の領域を用意し、1つの領域からバッファを確保するスレッドを特定する。
- グローバルバッファ領域上には、空き領域と確保されたバッファとが混在するが、1つの空き領域内でバッファを確保できる場合は、確保の処理を解放の処理と同期をとることなく実行する。
- バッファの解放の処理においては、同期をとる。

これらの方針に基づいて、以下のような管理方法を採用した。

(1) グローバルバッファ領域の空き領域を、図4に示すFACB(Free Area Control Block)のリストによって管理する。FACBは連続した空き領域の先頭と最後部へのポインタ(`TopPtr`,`BottomPtr`)と、他のFACBとの間で双方向リンクを構成するためのポインタを含む。

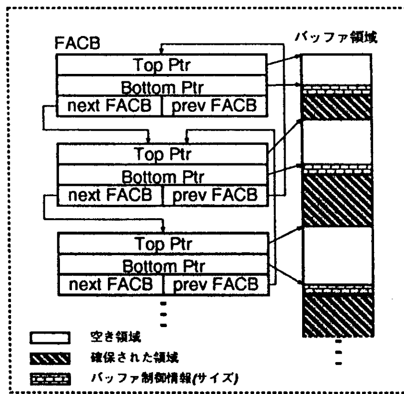

図4: グローバルバッファの構成

(2) 確保されたバッファは、先頭にそのサイズを持たせる。
(3) バッファを要求するスレッドは、FACBリストにロックをかけることなしに、先頭のFACBの`TopPtr`と`BottomPtr`を参照する。参照した時点での空き領域のサイズ(`TopPtr-BottomPtr`)が、要求されたバッファサイズよりも大きい場合は、先頭のFACBの`TopPtr`を必要とするサイズだけずらし、バッファを確保する。
空き領域のサイズが要求されたサイズよりも小さい場合は、FACBリストをロックし、現在参照しているFACBを最後に列べかえ、次のFACBからバッファを確保しようと試みる。バッファが確保できるまでこの手順を続ける。FACBリストの管理する空き領域からバッファを確保できない場合は、さらに別の空き領域を割り当てる。
(4) バッファの解放は、まずFACBリストにロックをかけ、FACBリストを先頭からたどり`BottomPtr`、`TopPtr`を参照し、解放されるバッファの挿入位置を決める。このとき、解放される領域が、前後のFACBが管理する空き領域と隣接しているか判断しFACBリストを更新する。

3 実装方法

本方式を、SiliconGraphics社のONYXワークステーション(OS:IRIX R5.3 System V, 4 CPU)を用いて実装することとした。ここでは、前述の並列処理方式を実現する並列処理用ライブラリを作成し、それらを用いてプロトコルプログラムを開発するというアプローチを採用した[2]。このアプローチによって、並列処理用ライブラリの作成のみで、他機種への通信プログラムの移植が可能となる。以下に並列処理用ライブラリが含む関数と機能、および、それを用いた通信プログラムの構成について述べる。

3.1 並列処理用ライブラリ

3.1.1 使用したシステムコール

本ライブラリの作成においては、以下に示すIRIXシステムコールを使用した。なお、IRIXにおいては、スレッドという概念が明確には定義されておらず、アドレス空間を共有するプロセスがスレッドに対応する。

- **sproc()** アドレス空間共有プロセスを生成する。
- **sysmp()** マルチプロセシング制御を行う。スレッドを特定のプロセッサに割り当てるために使用する。
- **ussetlock()** スピンロックを行う。競合したときはロックを獲得するまでビジーウエイトする。
- **usunsetlock()** スピンロックを解除する。
- **uspsema()** ロックを行う。競合したときはプロセスがスリープする。
- **usvsema()** ロックを解除する。

3.1.2 レイヤ間キュー

レイヤ間インタフェースに使用するキューの領域は、プログラム起動時に確保する。その領域をキューとして使用するために、以下の関数を設けた。

1. **p_cremsgq_r**(引数:キューの名前)
   プリミティブ受信用キューを作成する。作成したキューのIDを返す。
2. **p_cremsgq_s**(引数:キューの名前)
   プリミティブ送信用キューを設定する。キューのIDを返す。
3. **p_putmsg**(引数:キューID、プリミティブ)
   プリミティブをキューイングする。キューがフルで書き込みできない場合は–1を返す。
4. **p_getmsg**(引数:キューID)
   指定されたキューにプリミティブが存在する場合はプリミティブのポインタを返す。キューからプリミティブを読み込みできない場合はNULLを返す。

3.1.3 グローバルバッファ管理

各レイヤのスレッドは、起動時に malloc() を用いて、グローバルバッファ領域のためのメモリブロックを確保する。この場合、スピンロック (ussetlock()) を用いて、各スレッドで同期をとる。バッファ領域が確保された後は、各スレッドは以下の関数を用いて、バッファの確保・解放を行う。

1. p_galloc(引数:レイヤ番号、要求するサイズ)
   グローバルバッファを確保し、確保したバッファのポインタを返す。
2. p_free(引数:解放するバッファのアドレス)
   確保したバッファを解放する。ただし、p_galloc() や p_free() により、グローバルバッファ制御用データにアクセスするときにロックの必要がある場合は、スピンロックにより同期をとる。

3.1.4 スレッド制御

1. p_startproc(引数:レイヤのメイン関数のアドレス)
   sproc() を使用し、レイヤスレッドをアドレス空間共有プロセスとして生成し、そのレイヤ番号を返す。

3.2 プロトコルプログラムの構造

前節の並列処理用ライブラリ関数を使用したプロトコルプログラムは、メイン関数で p_startproc() によりレイヤのスレッドを起動する。各レイヤのスレッドは、1つのプロセッサに割り当てられた後は、スリープすることなく以下の処理を繰り返す。スレッドは、メモリブロックの確保、キューの作成などの初期化処理を行った後、図5のような構造を持つメインループに入る。メインループでは、上位・下位レイヤスレッドからのキューを監視し、プリミティブある場合、それを取得しプロトコル処理を行う。処理が終了すると、必要なプリミティブを作成して隣接レイヤへ送信する。このとき、隣接レイヤへのキューがフルである場合は、レイヤ内に保持しておき、キューにプリミティブが送信可能となった時点で送信を行う。

```
for( ;; ) {
   while( キューにプリミティブを送信可能||
                        キューから受信可能 ) {
      /** 送信処理 **/
      p_getmsg( 上位レイヤからのキュー  );
      if( サービスプリミティブ取得成功 ) {
        ・・・プロトコル処理・・・
      }
      if( 下位レイヤへのサービスプリミティブあり ) {
         p_putmsg( 下位レイヤへのキュー );
     }
      /** 受信処理 **/
     ・・・同様に受信処理を行う・・・
   }
}
```

図 5: レイヤのプロトコルプログラムのメインループ

4 性能評価

4.1 性能評価の方法

以下のような方針により、提案する並列処理方式の評価用プログラムを作成し、性能評価を行った。

- 並列処理の効果を明確化するためは、各レイヤのプロトコルの処理量の違いの影響を除く必要がある。このため、評価用プログラムでは、同一のプロトコルを複数レイヤ積み重ねた構成を用いる。
- プロトコル処理のオーバヘッドを明確化するため、内部折り返しによるデータの転送を採用する。
- 評価は、プロトコル処理量が小さいコネクションレスプロトコルと、受信確認やフロー制御などをサポートするコネクション型プロトコルの双方を用いて行う。
- 比較のために、1つのレイヤの場合についても評価する。

このような方針に基づいて、図6に示すような構成の評価用プログラムを作成した。評価用プログラムでは、各

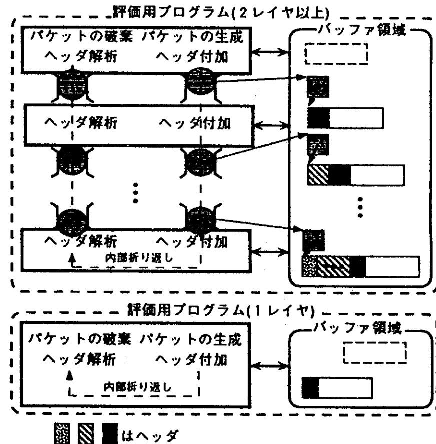

図 6: 評価用プログラムの構成

レイヤにおいて、データの送信時における PDU のヘッダの付加と、データ受信時における PDU のヘッダの除去、およびそれらに対応するプロトコル処理を行う。これに加え、最上位のレイヤにおいては、送信するユーザデータ用のバッファの確保と、受信したユーザデータのバッファの解放を行う。また、最下位レイヤでは、内部折り返しの処理を行う。ユーザデータ用のバッファの確保および内部折り返しの処理においては、値の設定やデータコピーを行っていない。また、1レイヤの処理においては、並列処理用ライブラリは使用せず、1つのスレッドでユーザデータの確保、送信処理、内部折り返し、受信処理、ユーザデータの解放を行っている。

性能評価を行うために、最上位のレイヤが一定数のユーザデータの受信に要した時間をその PDU 数で割った値(以下プロトコル処理時間と呼ぶ)を使用することとした。この値は、最上位レイヤが生成したユーザデータ1つ当

たりに要する、すべてのレイヤのプロトコルの処理時間を表す。

4.2 コネクションレスプロトコルを用いた評価

コネクションレスプロトコルの例として IP(Internet Protocol) を用い、評価用プログラムを作成した。

4.2.1 実装したプロトコル機能

source/destination の IP アドレスなどの IP ヘッダの付加・除去のみを行っている。その際、IP アドレス等のパラメータについては固定値を代入する。また、パケットの全体長 (total length) はその都度与えられたユーザデータ長から計算する。さらにヘッダチェックサムの代入・検査もパケット毎に行っている。ただし、IP におけるフラグメンテーションは行っていない。

4.2.2 評価結果と考察

IP のパケットサイズを変更して、プロトコル処理時間を計測した結果を図 7に示す。この結果では、1 レイヤの場合の処理時間が約 60μ秒であるのに対し、2 から 4 レイヤの場合、パケットサイズに関わらず約 80μ秒となっており、レイヤ (スレッド) 数よる処理時間の変化がほとんど見られない。この結果から、以下の考察を得た。

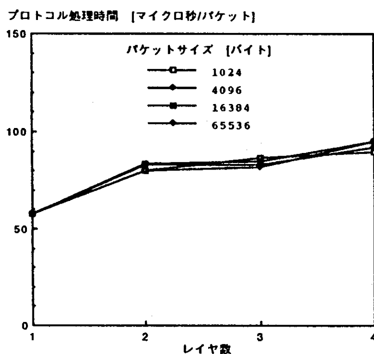

図 7: IP の評価結果

(1) 本実験においては、筆者らが提案するプロトコル並列処理方式のオーバヘッドが約 20μ秒程度であると考えられる。この値は、処理量の軽い IP プロトコルに比べても、充分小さいと考えられる。
(2) レイヤ数が 2 から 4 の場合に、パケット処理時間が増加していないことから、本実験においては、並列度を増すとその分パイプラインによる並列処理の効果が増大すると考えられる。
(3) データのコピーを行っていないため、パケット処理時間はパケットサイズに依存していない。

4.3 コネクション型プロトコルを用いた評価

コネクション型プロトコルの例として、OSI のトランスポートプロトコルクラス 4(TP4) [3] を用いた。

4.3.1 実装したプロトコル機能

実装した TP4 の機能を以下に示す。

- コネクションが確立された後のデータ転送フェーズのみをサポートした。
- データ転送フェーズの機能の内、AK TPDU を用いたデータの受信確認、クレジットを用いたフロー制御、再送用の TPDU の保持、チェックサムの設定と検査を実装した。
- AK TPDU はクレジットの半分の DT TPDU を受信した時点で、送信することとした。
- チェックサム計算を行う場合は、最上位のレイヤ (1 レイヤの場合はそのレイヤ) のみで実行させることとした。

4.3.2 評価結果と考察

ユーザデータサイズを 1024 バイトとし、クレジットおよびレイヤ数を変えて、ユーザデータ当たりのプロトコル処理時間を測定した。図 8および図 9に、どのレイヤにおいてもチェックサムの計算を行わない場合について、再送用の TPDU の保持を行わない場合と、保持を行う場合の結果をそれぞれ示す。また、図 10と図 11に、最上位のレイヤにおいてチェックサムの計算を行った場合について、再送用の TPDU の保持を行わない場合と、保持を行う場合の結果をそれぞれ示す。これらの結果から以下の考察を得た。
(1) 図 8の結果においては、クレジットの充分大きい範囲では、並列処理のオーバヘッドは約 30μ秒である。この実験においては、各レイヤのプロトコル処理の内容は異なるものの、並列処理ライブラリの使用法については、IP の場合とほぼ同様である。従って、IP と同定度のオーバヘッドの値が得られたと考えられる。
(2) 図 8と図 9を比較すると、再送のために TPDU を保持する場合においては、クレジットが大きくなるにつれて、プロトコル処理時間が増加していることがわかる。TPDU を保持する場合は、AK TPDU による受信確認がなされるまで PDU 用のバッファは解放されない。このため、クレジットが大きくなると使用されているグローバルバッファの数も増え、その確保・解放の処理のオーバヘッドが増加すると考えられる。
(3) 同様に、再送のために TPDU を保持する場合においては、レイヤ数が増加するにつれて、プロトコル処理時間が増加している。この理由は以下のとおりであると考えられる。このプログラムにおいては、PDU 用のバッファとして、PDU 本体とその制御用のディスクリプタを用いる方法 [4] を採用している。TPDU の保留を行う場合は、各レイヤで保留用のディスクリプタを確保し、AK TPDU を受信する毎に、各レイヤでそれを解放する。すなわち、TPDU を保留する場合は、グローバルバッファ領域への

競合が行われ、このため、レイヤ数が増加するにつれ、並列処理のオーバヘッドが増加すると考えられる。
(4) 最上位のレイヤでチェックサムを計算している場合では、プロトコル処理時間が約 600μ秒に増加している。これは、最上位レイヤの処理時間がチェックサムの処理を含め約 600μ秒であり、全体のプロトコル処理時間が最も処理時間の大きいレイヤによって決定されるためであると考えられる。

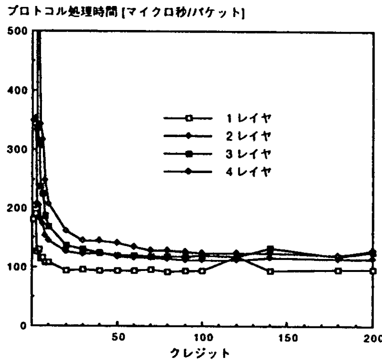

図 8: TP4 の評価結果
(チェックサムなし、TPDU 保持なし)

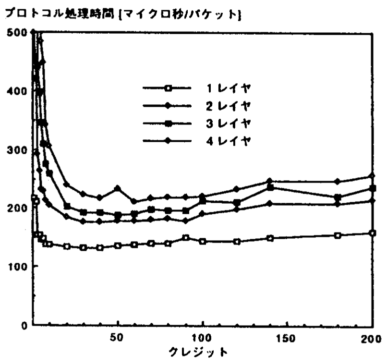

図 9: TP4 の評価結果
(チェックサムなし、TPDU 保持あり)

5 おわりに

本稿では、レイヤ単位のプロトコルの並列処理を実現するための、並列処理用ライブラリと、それを用いたプロトコルプログラムの実装方法について述べ、さらに、評価用プログラムを用いた性能評価について示した。並列処理用ライブラリを用いて、コネクションレスプロトコルと、コネクション型プロトコルを対象に、並列処理方式を評価した結果、小さいオーバヘッドでレイヤ毎のプロトコルを並列に実行させることが可能であるという評価を得た。今後は、本方式を用いた実際の通信プログラムの開発を行う予定である。

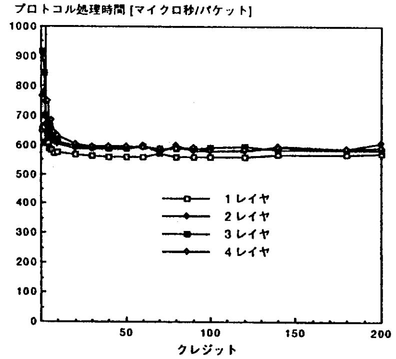

図 10: TP4 の評価結果
(最上位でチェックサム、TPDU 保持なし)

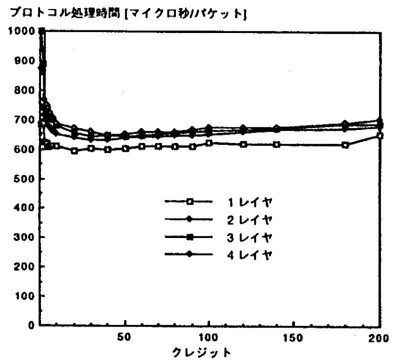

図 11: TP4 の評価結果
(最上位でチェックサム、TPDU 保持あり)

参考文献

[1] 加藤, 鈴木, “共有メモリ型マルチプロセッサを用いた通信プロトコルの並列処理方式,” 情処マルチメディア通信と分散処理研究会, 69-17, March 1995.

[2] 佐藤, 加藤, 鈴木, “レイヤ単位の並列処理方式を用いた通信プロトコルプログラムの実装に関する検討,” 情処第 51 回全国大会, 1E-3, Sept. 1995.

[3] CCITT X.224, “Transport Protocol Specification for Open Systems Interconnection for CCITT Applications.” 1988.

[4] 加藤, 井戸上, 鈴木, “OSI プロトコル実装のためのユーザデータをコピーしないバッファ制御方式,” 情処マルチメディア通信と分散処理研究会, 62-13, Sept. 1993.