1E－3 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

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††

1. はじめに

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。そこで筆者らは、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検討している[1]。さらに、本検討に基づき、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、コネクション型通信プロトコルによる評価プログラムを開発した。本稿では、これらの結果について述べる。

2. 並列処理方式の概要

プロトコルの並列処理については、筆者らはProcessor-per-layerと呼ばれるレイヤ単位の並列処理方式を採用している[1]。その概要を以下に示す。

(1) １つのレイヤのプロトコル処理を１つのスレッドとし、各スレッドを個別のプロセッサに割り当てる。各レイヤスレッドはキューを介して、共通にアクセス可能なバッファ領域上に作成されたサービスプリミティブをやりとりする (図1参照)。

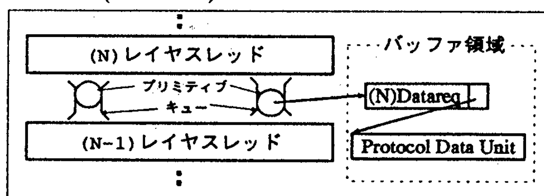

図 1: プロトコル並列処理プログラムの構成

(2) キューは固定長とし、読みだすスレッドと書き込むスレッドを１つとすることにより、双方が同期をとらずにキューにアクセス可能な方式を採用している。

(3) 各レイヤスレッドは、コネクション管理テーブルなどの１つのレイヤ内で使用されるデータを確保するためのローカルバッファ領域をもつ。

(4)PDU(Protocol Data Unit) やプリミティブは、複数レイヤから参照可能なデータを確保するためのグローバルバッファ領域上に作成する。さらに、レイヤ毎に、グローバルバッファ領域を設け、バッファを要求するスレッドは１つに特定し、１つの空き領域内でバッファを確保できる場合には、グローバルバッファ制御用データのロック無しに確保可能とした。

"A Study on Implementation of Protocol Program using Processor-per-Layer Parallel Processing Method"
Tomomi SATO†, Toshihiko KATO†† and Kenji SUZUKI††
The University of Electro-Communications†
KDD R&D Laboratories††

3. 実装方法

本方式を、SiliconGraphics 社の ONYX ワークステーション (OS:IRIX System V.4) を用いて実装した。この実装においては、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。以下に、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構成について述べる。

3.1 使用したシステムコール

並列処理用ライブラリを実装するために、IRIX が提供する以下のシステムコールを使用した。

- 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。
- 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。
- 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。

3.2 並列処理用ライブラリ

3.2.1 キューの管理

レイヤ間インタフェースに使用するキューの実体は、プログラム起動時に確保する。キューの作成とアクセスを行うために、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。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。これらの２つの関数によりキューへの読み書きが可能となる。

(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。キューがフルの場合は−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。キューが空の場合は NULL を返す。

3.2.2 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、ローカルとグローバルの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_malloc(引数:レイヤ番号、サイズ) ローカルバッファの確保を行い、確保したバッファのポインタを返す。

(2)p_galloc(引数:レイヤ番号、サイズ) グローバルバッファの確保を行い、確保したバッファのポインタを返す。

(3)p_free(引数:解放するバッファのアドレス) (1) または (2) で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必要がある場合は、スピンロックにより同期をとる。

3.2.3 スレッド管理

(1)p_startproc(引数:レイヤのメイン関数のアドレス) IRIX のシステムコール 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。

(2)p_wait(引数:レイヤ番号) 指定したレイヤスレッドをスリープさせる。レイヤスレッドの処理が無くなった場合に使用される。

(3)p_wakeup(引数:レイヤ番号) 指定したレイヤスレッドを起動させる。

3.3 プロトコルプログラムの構造

プロトコルプログラムは、上記のライブラリを使用して作成される。メインプログラムにおいて、p_startproc() によりレイヤのスレッドを起動する。各レイヤのメイン関数においては、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、メインループにおいて上位・下位からのプリミティブの処理を行う。メインループの構造を図 2に示す。

```
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 ) {
        p_wakeup( 上位レイヤ );  ・・・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 );
      if( 下位レイヤへ送信成功 ) p_wakeup( 下位レイヤ );
    }
     /** 受信処理 **/
    ・・・同様に受信処理を行う・・・
  }
  p_wait( このレイヤ );
}
```

図 2: レイヤプログラムのメインループ

4. 評価プログラムの作成

本方式の性能を評価するために、IP (Internet Protocol) を複数階層的に積み重ねた通信プログラムを試作した。本プログラムでは、各レイヤは同一のプロトコルを処理し、チェックサムの計算を含む IP ヘッダの設定、解析を行う。さらに、最上位のレイヤは、ユーザデータ用のバッファを連続的に確保し PDU を発生させ、また受信した PDU のユーザデータバッファを解放する。また、最下位のレイヤでは内部折り返しを行っている。ユーザデータ用バッファの確保および折り返しの処理では、値の設定やデータコピーを行っていない。なお、参考のために IP が 1 レイヤの場合も実現した。この場合は 1 つのスレッドでユーザデータの確保、IP ヘッダの設定、その解析、ユーザデータの解放を行っている。

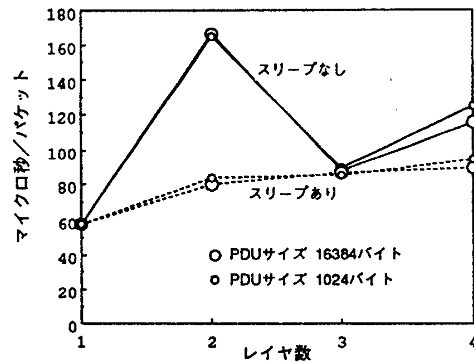

図 3: 評価結果 (プロトコル処理時間)

4 つのプロセッサを有するワークステーション上で、評価用プログラムを実行させ、プロトコル処理時間を計測した (図 3参照)。ここで、プロトコル処理時間とは、上位レイヤがある PDU を発生してから、一定数だけ後の PDU を最上位レイヤが解放するまでの時間を、その PDU 数で割った値とした。評価は、PDU サイズとレイヤ数を変化させて、スレッドのスリープする場合と、させない場合 (p_wait() と p_wakeup() を呼び出さない場合) について行った。その結果以下の評価を得た。

(1) 図 3に示すように、スレッドのスリープを行う場合と行わない場合とでは、プロトコル処理時間が異なる。これは、スレッドのスリープのオーバヘッドが大きいためであり、特に 2 レイヤの場合は、最上位と最下位のレイヤの処理負荷が異なるため、下位レイヤのスレッドが頻繁にスリープし処理時間が大きくなった。

(2) これに対してスレッドをスリープさせない場合の処理時間は、レイヤ数によらずほぼ一定である。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 レイヤの場合約 80μ秒となっている。これは、スレッドのスリープが無ければ、筆者らの方式の並列化のオーバヘッドが充分小さいことを示している。

(3) 複数のスレッドを 1 つのプロセッサに割り当てると、スレッドの切り替えのオーバヘッドがさらに増加することを考慮すると、各スレッドをスリープさせない構成を採用するのが適当であると考えられる。

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に関する検討結果を述べた。本検討により、筆者らが提案する並列処理方式は、並列化のオーバヘッドが少なく高速なプロトコル処理を実現できる見通しを得た。今後はより複雑なプロトコルプログラムを作成し、本方式の評価を進める予定である。

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] 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,September 1993.