並列アプリケーション実行におけるTCP/IP通信挙動の解析

高野了成[†,††] 石川裕[†,†††] 工藤知宏[†]
松田元彦[†] 児玉祐悦[†] 手塚宏史[†]

通信遅延が大きな環境でのTCP/IP通信において，ネットワーク上で輻輳が発生していないにも関わらず実効帯域が大きく低下する現象が観測されることがある．これは，送信側のインタフェース毎のバッファがあふれた場合に輻輳とみなす方式を用いているためで，これを輻輳とみなさないと実効帯域の低下を招かないことが知られている．並列アプリケーションの実行では同時に複数のコネクションで送信と受信を同時に行うことが多い．送信パケットを効率よく送り出すと同時に受信したパケットに対するACKを速やかに返すことが重要となる．しかし，インタフェース毎のバッファを輻輳とみなさない方式ではコネクション間の公平性が保たれず，ACKが長時間返されない現象が発生し実効性能が低下してしまう．本稿では，本現象を解析し，解決方法を考察する．解決方法として，ACKを優先的に返す機構や，バッファあふれが生じないようにユーザプログラムが送信完了を知ることにより送信量を制御することなどが考えられる．

The Analysis of TCP/IP Communication Behavior on Parallel Applications

Takano Ryousei,[†,††] Ishikawa Yutaka ,[†,†††] Kudoh Tomohiro ,[†]
Matsuda Motohiko ,[†] Kodama Yuetsu [†] and Tezuka Hiroshi [†]

When the bandwidth-latency product of a network is large, degradation of the TCP/IP communication performance is often observed even though no actual network congestions occur. This is because the current TCP implementation transits to the network congestion state in the case the interface queue of the sender becomes full. If the implementation does not transit to the congestion state in such case, the performance is not dropped. In a parallel application using TCP/IP, communication may involve several TCP connections at the same time. When a node simultaneously sends and receives data, it is important to send the acknowledge packets promptly as well as sending the data efficiently. However, when the queue full is not regarded as congestion, fairness among connections is not guaranteed. Therefore, the acknowledgment transmission is sometimes delayed, and the communication performance is degraded. This paper investigates this phenomena and discusses some solutions which include a new interface queue implementation which will transmit the acknowledgment packets promptly, and a new implementation scheme by which the user process may know the completion of the actual data transmission.

1. はじめに

近年，クラスタ，グリッド技術が注目され，LANやインターネット上での並列処理のニーズが高まっている[1)]．グリッド環境上での並列処理を実現するために，我々は，広域に分散して配置されたクラスタ計算機群をつないだグリッド環境で，MPIを用いて記述された並列アプリケーションを効率よく実行するGridMPI[2)]の開発を行っている．GridMPIではTCP/IPプロトコル標準を保ちながら通信性能を改良していくアプローチをとっている．

ネットワークの帯域遅延積が大きい場合，TCP/IP通信の実効帯域が低下することはよく知られている．これは，輻輳発生時に転送量が大きく低下することが原因で，1対1の単方向通信の遅延とパケットロスがある環境下における挙動についての報告は多い．

しかし，並列アプリケーションの実行では，1対1の単方向通信だけでなく，多数のプロセスが相互に複数のコネクションで通信を行うことがあり，通信の挙動が異なる．そこで我々は，遅延のある環境で並列アプリケーションを実行した際のTCP/IP通信の挙動を解析するために，ハードウェアにより通信路に遅延を挿入する装置を用いて実験環境を構築し，通信性能の測定と挙動の解析を行った．

本稿では，まず，遅延はあってもパケットロスがな

† 産業技術総合研究所 (National Institute of Advanced Industrial Science and Techinology)
†† 株式会社アックス (AXE, Inc.)
††† 東京大学 (University of Tokyo)

い場合の性能を測定し，ネットワーク上では輻輳が発生していないにも関わらず実効帯域が大きく低下する現象の観測結果を示す．これは，TCP/IP の実装において，送信側のインタフェース毎のバッファ(Interface Queue: IFQ) があふれた場合 (これを Send Stall と呼ぶ) を輻輳とみなす方式を用いているためである．1 対 1 の単方向通信では，Send Stall を輻輳と見なさないことにより実効帯域の低下を防ぐことができることを実験により確認した．

次に，同一プロセッサが複数のコネクションを介して同時に送受信する通信パターンでは，Send Stall を輻輳と見なさないと，実効通信性能が低下することを示す．この通信性能低下のメカニズムの解析を行った結果，異なるコネクションで受信と送信を同時に行う場合，送信パケットを効率よく送り出すと同時に受信したパケットに対する確認応答 (ACK) を速やかに返すことが重要であるにもかかわらず，Send Stall を輻輳と見なさないとコネクション間の公平性が保たれないために，ACK が長時間返されない現象が発生し，実効性能が低下してしまうことがわかった．

最後に，この問題の解決方法について考察する．ACK を優先的に返す機構や，バッファがあふれることがないように，送信の完了をユーザプログラムが知ることができるようにして送信量を制御することなどが考えられることを示す．

以下，2 節において MPI 通信ライブラリにおける通信機能および TCP/IP における実現方法を紹介する．3 節において MPI 通信ライブラリで利用される通信パターンの解析を行ない，4 節で考察を行なう．最後に 5 節でまとめを述べる．

2. MPI 通信機能と TCP/IP による実装

流体力学, 分子動力学などに代表される科学技術分野の並列アプリケーションでは, 問題領域を分割し, 各コンピュータは分割された一部分を計算し, その結果を他のコンピュータに送信する, という手順を繰り返す．すなわち, 周期的に 1 対 1, 全対全, 全対 1, 1 対全, などの通信でかつ一度に大量のデータが授受される．MPI 通信ライブラリは, このような通信パターンをサポートしている．

これらの MPI 通信機能を TCP/IP で実現した時の通信遅延と通信性能の挙動を解析するために, 解析用プログラムを開発した．解析用プログラムは MPI 通信ライブラリを使用せずに, TCP/IP を用いて MPI 通信機能と同様の通信パターンで通信するプログラムである．多くの MPI 通信ライブラリ[12),13)] では, 送信メッセージサイズに応じて受信側のバッファリングを制御するためにデータ授受のために複数のプロトコルが提供されているが, 解析プログラムでは以下のように簡素化する．

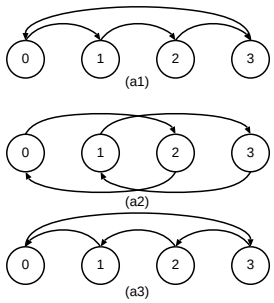

図 1 全対全通信アルゴリズム (マルチコネクション)

(1) 受信側は送信データを受信できるだけのバッファ領域が常に確保されている．
(2) 送信側は TCP 送信バッファに空きがある限り全ての送信データを一回の送信システムコールで処理する．

MPI 通信機能は, 大きく, 1 対 1 通信と複数のプロセスが通信に関与する集合通信の二つに分類される．以下, それぞれについて MPI 通信機能の概略と解析用プログラムの概要を説明する．

2.1 1 対 1 通信

1 対 1 通信機能は, 通信操作においてブロッキングするかしないかで大きく二つに分かれ, さらに, それぞれにおいて, 標準通信, 同期通信, レディ通信, バッファ通信という 4 種類の通信モードが提供される[11)]．

本稿では, 1 対 1 通信としてブロッキング標準通信 `MPI_Send` の実現を考える．ブロッキング標準通信では, ユーザが指定したデータ領域が送信されるまでブロッキングされる．

TCP/IP を使用して `MPI_Send` を実現する場合, write 関数によりデータが全て TCP 送信バッファにコピーされるまで当該関数から呼び出し元には制御は戻らない．

2.2 集合通信

集合通信には, 大きく以下の 4 タイプに分かれる．

- バリア同期
  全プロセスの実行同期
- 全対全通信
  全プロセスによるデータ交換
- 1 対全通信
  ブロードキャスト, データ配布 (Scatter)
- 全対 1 通信
  データ集め (Gather)

本稿では, 集合通信のなかから全対全通信 `MPI_Alltoall` を取り上げる．全対全通信が最も通信路のバンド幅を必要とする集合通信だからである．

集合通信の実現方法として**図 1** に示す通り, 一つのプロセスにデータ送信が集中しないように 1 対 1 の単一方向通信を繰り返す方法がある．例えば, 4 プロセスの全対全通信では, 図 1 の (a1), (a2), (a3) に示

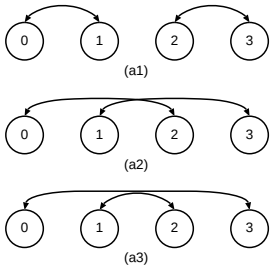

図 2 全対全通信アルゴリズム (シングルコネクション)

す通り, 3 フェーズで処理が完了する. このようにして, 各プロセスは一時期に一つのコネクションから受信, もう一つのコネクションから送信, を行ない, 特定プロセスへの通信の集中を避ける. 言い換えれば, 一つのプロセスからみると, 二つのコネクションを使用した送受信による通信パターンと言える.

プロセス数が 2 の冪乗の時, **図 2** に示す通り, TCP/IP の特徴である双方向通信機能を用いた通信方法もある. 先の通信方法と同様 3 つのフェーズで処理が完了する. 言い換えれば, 一つのプロセスからみると, 一つのコネクションの送受信による通信パターンと言える.

3. 解　　析

3.1 解析用プログラム

2 節で紹介した MPI 通信ライブラリの通信パターンに対して，通信遅延，Send Stall を輻輳とみなすかどうかによる挙動の違いを解析した.

(1) 単一コネクションにおける単方向通信
(2) 二つのコネクションを使用した送受信
(3) 単一コネクションにおける送受信

解析用プログラムは MPI ライブラリを使わず，ソケットによって実装し，各ホスト毎に 1 プロセスを起動することで，TCP コネクションによる通信を行なっている.

3.2 実 験 環 境

図 3 に示すように，スイッチ間をハードウェアネットワークエミュレータ GNET-1[4] で接続した 4 ホストによるネットワークを用いて，実験評価を行なった. GNET-1 はハードウェアロジックにより正確で細粒度な通信遅延をエミュレートすることが可能であり，32ns 単位，最大で往復 256ms の通信遅延を設定することが可能である. 以下，通信遅延は双方向に均等に遅延がある往復通信遅延を指すこととする.

各ホストのハードウェア環境の諸元を**表 1** に，ソフトウェア環境を**表 2** 示す.

カーネルは Linux 2.4.20 をベースに Web100[10] パッチを適用した. Web100 は高性能ネットワーク環境における TCP バッファチューニングと性能解析ツールの

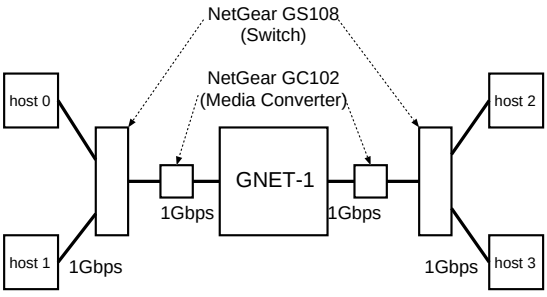

図 3 実験ネットワーク構成

表 1 ホストのハードウェア仕様

Processor	Pentium4 2.4GHz
Main Memory	FSB800 512MB
Mother Board	Intel D865GCL
NIC	Intel PRO/1000 (82547EI)
Bus Interface	CSA (Bus Bandwidth 2Gbps)

表 2 カーネル設定

Kernel	Linux 2.4.20 + Web100 2.2.1
Intel PRO/1000 Driver	5.0.43
Socket Buffer	2MB

提供を目的としたプロジェクトである．Web100 カーネルパッチはプロトコルスタックにフックを挿入することで，TCP コネクション単位での輻輳ウィンドウ，スロースタート閾値，広告ウィンドウなどの統計情報を取得し，proc ファイルシステムを介してユーザアプリケーションに提供する．本評価では TCP プロトコルスタックの内部動作を調査するため，この情報を利用した.

また，Intel PRO/1000 のデバイスドライバは 5.0.43，カーネルパラメータの設定としてソケットバッファサイズを 2MB とした.

3.3 単一コネクションの単方向通信

輻輳が発生しなければ，輻輳制御アルゴリズムに関わらず十分な実効帯域が得られるはずである．例えばギガビットイーサネットのネットワークインタフェースを持つ計算機同士が，ギガビット以上の利用可能な帯域を持つネットワークで接続されている場合，ネットワーク上で輻輳は発生しないので，実効帯域は十分に大きいはずである.

ところが，実際には，遅延がある場合，ネットワーク上でパケットのロスが発生していないにもかかわらず，実効帯域が小さくなることがある．**図 4** に，メッセージサイズと通信遅延を変えながら，TCP による単方向通信の性能を測定した結果を示す．この環境ではネットワーク上でのパケットロスは発生していない. 低遅延の場合は約 117.4MB/s と物理性能の約 94%の帯域利用率が出ているが，8ms では 70〜90MB/s と

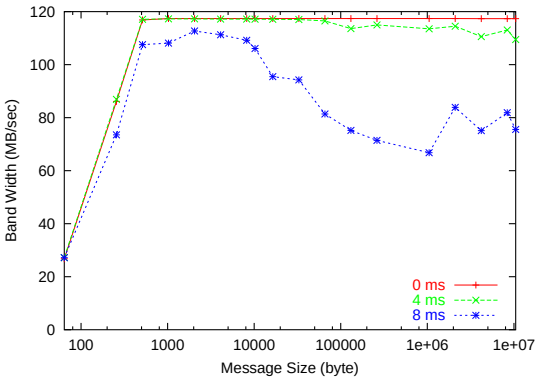

図 4 単一コネクションにおける単方向通信 (Send Stall 時 輻輳制御あり)

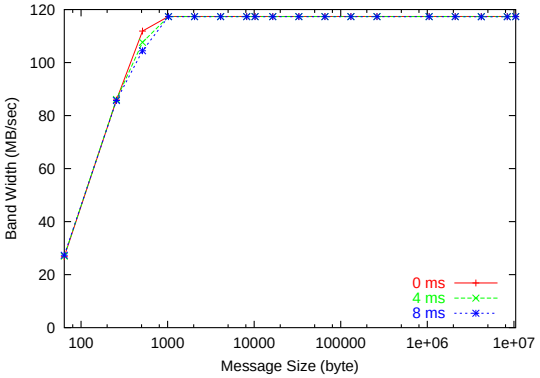

図 5 単一コネクションにおける単方向通信 (Send Stall 時 輻輳制御なし)

帯域利用率が上がらない.

この原因について解析する. メッセージサイズ 1MB, 通信遅延が 0ms と 8ms の時点における帯域, 輻輳ウィンドウ, 広告ウィンドウの変化をそれぞれ**図 7**, **図 8** に示す.

帯域が 125MB/s, 遅延が 8ms のネットワークにおける帯域遅延積は 1MB となるので, 輻輳ウィンドウが 1MB あれば十分な帯域を得ることができる. しかし, 図 8 では輻輳ウィンドウが 1MB を下回り続けている. これは測定開始直後と, 2000ms 経過地点で輻輳が発生したと見なして, 輻輳ウィンドウが小さくなっているからである.

パケットロスが発生していないにも関わらず輻輳ウィンドウが小さくなっているのは, Linux 等☆の TCP 実装で用いられているフロー制御の仕組みのためである. 通信バッファに着目した TCP 実装の概略図を**図 6** に示す. コネクション毎にソケットレベルで送信および受信バッファが確保され, TCP プロトコルレベルでの送信および受信バッファとなる (以降, TCP バッファと呼ぶ). さらにネットワークインターフェイス毎に送信バッファ (IFQ), 受信バッファ (図における backlog) が使用される.

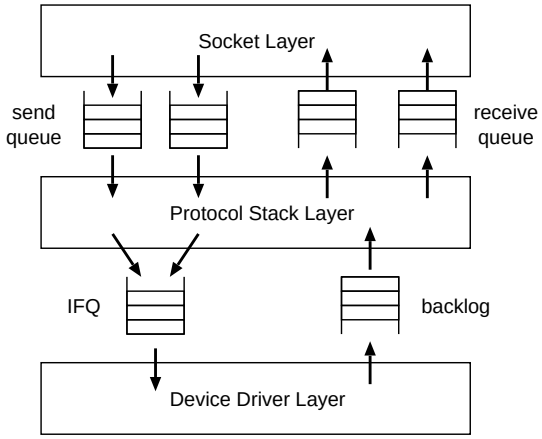

図 6 プロトコルスタック

輻輳制御アルゴリズムによるフロー制御は, コネクション単位でソケットレベルのバッファから IFQ への流量を制御することになる. 一方, IFQ からの送出はネットワークインタフェースが接続されている物理リンクの帯域容量により制限されるし, PAUSE(IEEE802.3x) やコリジョンなどの物理層のフロー制御機構によっても制限される. このため, TCP バッファから IFQ への送出量が, IFQ からネットワークへの送出量を上回ると IFQ があふれることになる. IFQ☆☆がフルで, TCP 層から IFQ に送れない状況を Send Stall と呼ぶ.

Linux 等では Send Stall を輻輳とみなすことにより TCP バッファから IFQ への送出量を制限する. この場合, ネットワーク上での輻輳と異なりパケットロスは生じないが, 輻輳ウィンドウは小さくなってそのコネクションの送信量が減らされる. ところが, 帯域遅延積が大きなネットワーク環境では, いったん小さくなった輻輳ウィンドウサイズの回復には時間がかかるため, ネットワークの物理帯域を有効利用できず実効帯域が小さくなってしまうのが帯域低下の原因である. 輻輳が発生しても図 7 に示すように, 遅延が小さく, 輻輳ウィンドウが遅延帯域積を上回っている場合には性能に影響しない.

Web100 プロジェクトで開発されたカーネルパッチを用いると, Send Stall を輻輳と見なさないようにすることができる. この手法を用いることにより, 1 対 1 の単方向通信では遅延が大きい場合の実効帯域低下を避けることができる. **図 5** にこのパッチを用いて Send Stall を輻輳と見なさないようにした場合の, 1 対 1 の単方向通信の実効帯域への影響を示す. 遅延がある場合でも実効帯域の低下はほとんど見られないこ

☆ BSD 系 Unix でも同様[5].

☆☆ Linux におけるデフォルトサイズは 100 パケットである

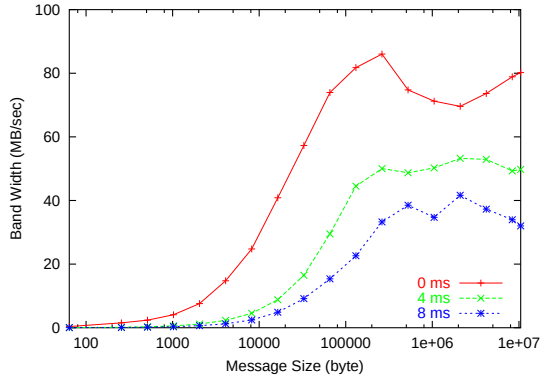

図 10 二つのコネクションを使用した送受信 (Send Stall 時 輻輳制御あり)

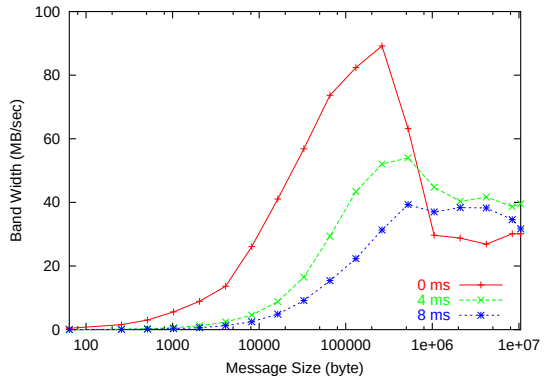

図 11 二つのコネクションを使用した送受信 (Send Stall 時 輻輳制御なし)

とがわかる．**図 9** に示すように，この場合輻輳ウィンドウのサイズが小さくなることはない．

3.4 二つのコネクションを使用した送受信

図 1 に示した通信パターンである二つのコネクションを使用した送受信の性能を測定した．Send Stall を輻輳とみなす場合とみなさない場合の結果をそれぞれ**図 10**，**図 11** に示す．

この結果よりメッセージサイズが 256KB より大きい場合に，Send Stall を輻輳とみなさなければ，大きく性能が劣化してしまうことがわかる．GNET-1 により観測したメッセージサイズ 1MB での実効帯域幅の変化を**図 12** に示す．これより約 200ms 間，送信が止まる現象が見られる．さらに ACK シーケンス番号を調べた結果，この間 ACK が送出されていないことがわかった．200ms は遅延 ACK タイマの起動時間であり，このタイミングで ACK が送出されている．

これは Send Stall を輻輳とみなさないため，TCP レベルでの流量制限が働かず，Send Stall し続け，ACK パケットも Send Stall により送信されなくなった結果だと考えられる．この場合，コネクション間の公平性を実現する機構がないため，送信用コネクションの

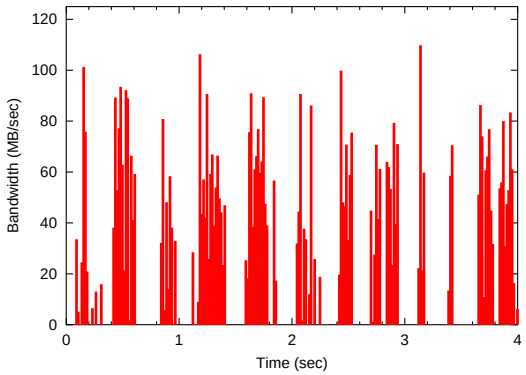

図 12 図 11 における 1MB 転送時の帯域幅の変化

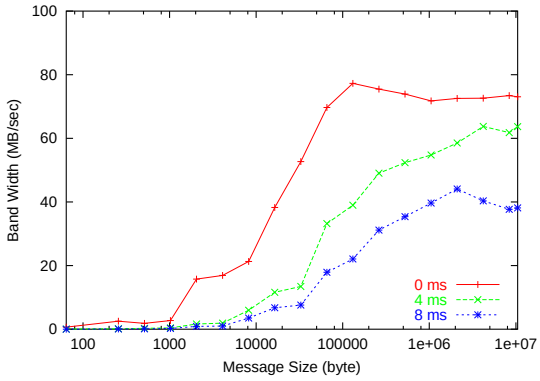

図 13 単一コネクションによる送受信 (Send Stall 時 輻輳制御あり)

データパケットはキューイングされるが，受信用コネクションの ACK パケットはキューイングされないという現象が起こり得る．TCP バッファは ACK を受信するまで解放されないので，アプリケーションは続くデータを write できずに送信が止まってしまう．

3.5 単一コネクションによる送受信

最後に図 2 に示した通信パターンである単一コネクションによる送受信の性能を測定した．Send Stall を輻輳とみなす場合とみなさない場合の結果をそれぞれ**図 13**，**図 14** に示す．

遅延の大きな場合，1 対 1 通信と同様に Send Stall を輻輳とみなさない方が性能がよい．また，二つのコネクションを用いた場合に見られた Send Stall 時の性能劣化は表われない．これは単一コネクションでは，遅延 ACK により ACK をデータパケットにピギーバックするため，ACK が返っているからだと考えられる．

4. 考　　察

前節の測定結果を元に，Send Stall 時に，TCP 層において輻輳制御を行なう場合と行なわない場合の，

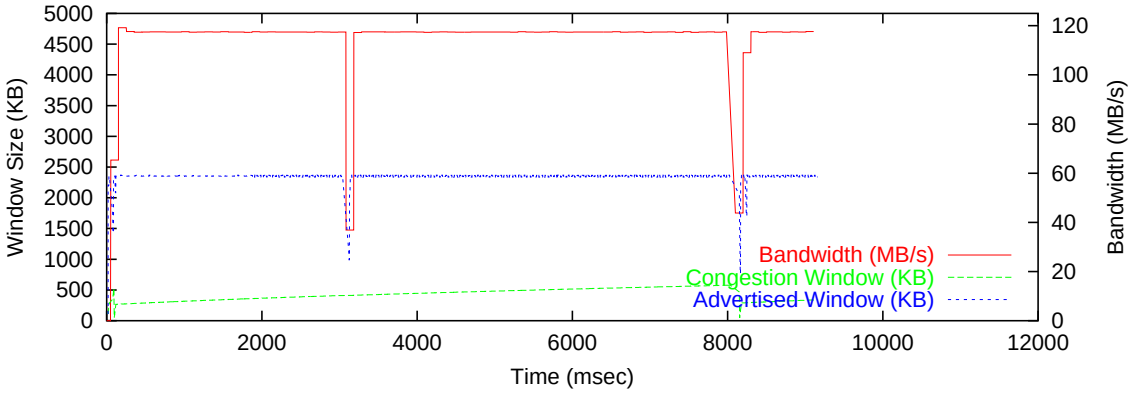

図 7 ウィンドウサイズの変化 (遅延 0ms，メッセージサイズ 1MB, Send Stall 時 輻輳制御あり)

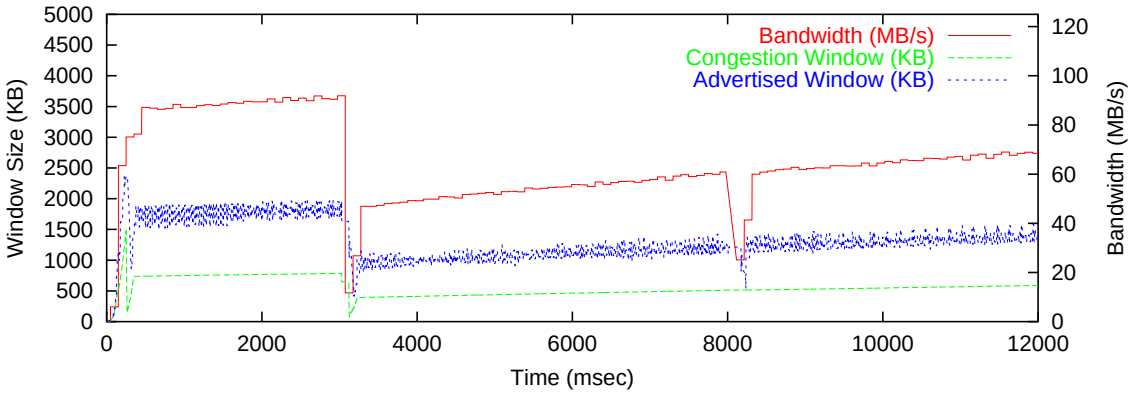

図 8 ウィンドウサイズの変化 (遅延 8ms，メッセージサイズ 1MB，Send Stall 時 輻輳制御なし)

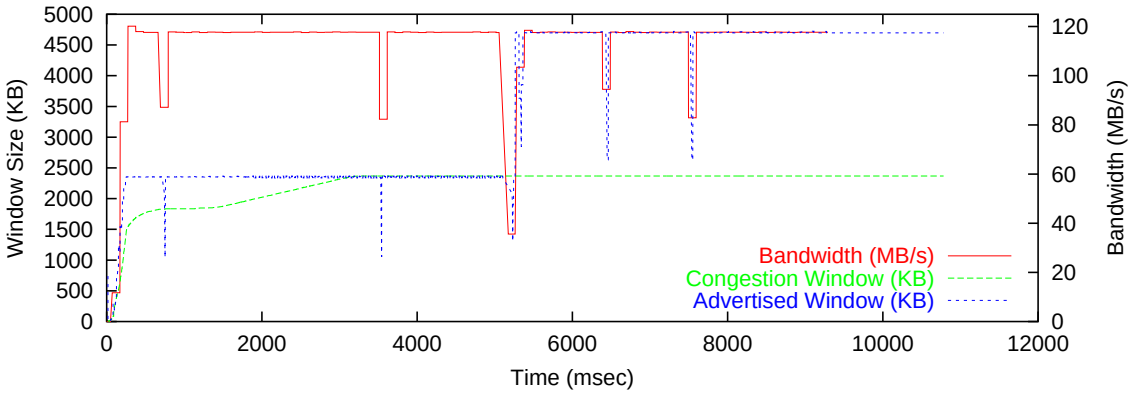

図 9 ウィンドウサイズの変化 (遅延 8ms，メッセージサイズ 1MB，Send Stall 時 輻輳制御なし)

通信遅延による通信性能の差異を表 3 にまとめた．単一コネクションにおいては単方向通信，同時送受信に関係なく，Send Stall 時に輻輳制御を行なわない方が良い結果となっている．

二つのコネクションによる送受信では，Send Stall 時に輻輳制御を行なわないと遅延がないときに性能が極端に劣化する．本通信パターンでは，あるノードに注目すると，一つのコネクションにはデータを送信し，もう一つのコネクションからはデータを受信している．Send Stall 時に輻輳制御が行なわれないため，送信しているコネクションの TCP バッファに溜ったデータは IFQ に空き領域がある限り IFQ に送られる．もう

表 3　Send Stall 時の輻輳制御方式の違いによる性能
1 Mbytes データ送信時の帯域幅 (MByte/sec)

通信パターン	Send Stall 時 輻輳制御あり		Send Stall 時 輻輳制御なし	
	遅延 0 msec	遅延 8 msec	遅延 0 msec	遅延 8 msec
単一コネクション単方向通信	117.4	66.8	117.4	117.3
単一コネクション送受信	71.6	39.7	74.6	48.7
2 コネクションによる送受信	71.2	34.7	29.7	37.0

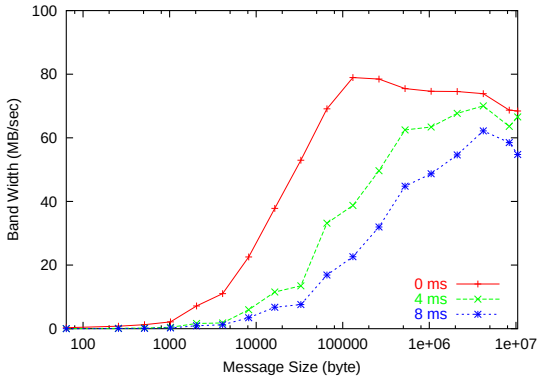

図 14　単一コネクションによる送受信 (Send Stall 時 輻輳制御なし)

一つのコネクションに届いたデータに対する ACK パケットを送信しようとしても，IFQ が一杯のため送信されないことが生ずる．前章で述べた通り，実験では，最大 200ms の遅延が観測されている．このような状態になる原因は，Send Stall 時に輻輳制御を行なわないことにより，IFQ に対する複数コネクションからのデータ送信要求を公平に扱えなくなるからである．

並列アプリケーションで使われる集合通信では，一般に二つのコネクションによる送受信パターンが多く使われる．本通信パターンにおける通信コストは並列アプリケーションの実行性能に大きく影響する．二つのコネクションによる送受信の性能向上には，Send Stall を起こさないようにするか，Send Stall 時でも各コネクションの公平性を保つかのいづれかの方法を用いることが考えられる．

4.1　Send Stall 回避手法

ユーザレベルで Send Stall を起こさないように MPI 通信ライブラリレベルで，IFQ のサイズ以上のデータを一度に送信しないようにする手法を考察する．

send 関数あるいは write 関数は，送信データが IFQ に送られなくても，TCP 送信バッファに空き領域ががあると TCP 送信バッファにコピーされて処理が呼び出し元に戻ってくる．すなわち，send あるいは write 関数から処理が戻ってきても，送信データが IFQ に送られたとは限らない．また，select 関数を使用しても，TCP 送信バッファに空き領域があることは分かってもバッファが空であるかどうかは分からない．すなわち，send, write, select 関数だけでは，Send Stall を引き起こさないように送信量を調整することは不可能である．

全てのコネクションの TCP 送信バッファの総量を IFQ サイズとすれば，IFQ がオーバフローすることはなくなる．しかし，コネクションの数，コネクション毎の帯域遅延積が動的に変わる場合に，IFQ サイズを見積もるのは困難である．

IFQ の空き容量を調べるシステムコールは一般に存在しない．Linux の場合，ioctl 関数で `TIOCOUTQ` コマンドを用いて TCP 送信バッファの未送信データサイズを得ることが可能である．本機能を使用して全てのコネクションにおける未送信データサイズが IFQ サイズ以下に保つようにプログラミングすれば，Send Stall は引き起こさない．しかし，ioctl 関数を呼ぶことによるオーバヘッドが生じる．

4.2　Send Stall 時のコネクション間の公平性

Send Stall 時に輻輳制御がされ，輻輳ウィンドウが減少し送出量が減少，その結果として，他のコネクションが IFQ にデータを送信する機会を得ている．輻輳制御されたコネクションは，通信遅延が大きいネットワークを使用していると，ウィンドウサイズの回復が遅れる．

コネクション間の公平性を輻輳制御で行ないかつ輻輳制御されたコネクションのウィンドウサイズの回復を早めるような輻輳制御アルゴリズムを使用する．例えば，Scalable TCP[7]，HighSpeed TCP[8]，FAST[9] などの輻輳制御アルゴリズムを Send Stall 時にだけ適用する方法が考えられる．

5.　ま　と　め

本稿では，通信遅延が大きな環境での TCP/IP 通信において，ネットワーク上で輻輳が発生していないにも関わらず実効帯域が大きく低下する現象について並列アプリケーションにおいて使用される通信パターンを用いて解析し考察した．

並列アプリケーションで使われる集合通信にあらわれる 2 コネクションによる送受信パターンにおいて，通信遅延が大きくても通信性能をあげるためには，Send Stall を起こさないようにするか，Send Stall 時でも各コネクションの公平性を保つ必要があることを示した．前者は MPI 通信ライブラリの実装において TCP 送信バッファが空になることを確認しながら送信することにより実現できることを示した．後者については，Send Stall 時に特別な輻輳アルゴリズムを用

いて，減少したウィンドウサイズを高速に回復させる方法を示した．

本稿では，Linux カーネルを使用して計測した結果を示したが，TCP 実装における Send Stall 時の扱いは BSD 系 TCP/IP の実装においても同様であり[5]，BSD 系 Unix でも本実験と同様の結果が得られると考える．

本研究は，我々が開発を進めている GridMPI[2] 研究開発の一部である．グリッド環境上での TCP/IP 性能向上は，輻輳制御に留まらず，通信ライブラリ，カーネルインターフェイス，TCP/IP 実装，など多岐に渡って研究を行なう必要がある．論文[3] ではカーネルインターフェイスに着目して既存ソケットインターフェイスの問題点と解決方法を提案した．今後，本稿で述べた知見を基に TCP/IP 実装の改良を行ない，既研究成果とともに高性能 GridMPI 環境を構築する．

謝辞 本研究の一部は文部科学省「経済活性化のための重点技術開発プロジェクト」の一環として実施している超高速コンピュータ網形成プロジェクト（NAREGI：National Research Grid Initiative）および，新エネルギー・産業技術総合開発機構基盤技術研究促進事業（民間基盤技術研究支援制度）の一環として委託を受け実施している「大規模・高信頼サーバの研究」によるものである．

参 考 文 献

1) 関口智嗣, “最新グリッド事情 – グリッドでできること –,” インターネットコンファレンス 2002 招待講演, 2002.
2) 石川裕，松田元彦，工藤知宏，手塚宏史，関口智嗣: “GridMPI - 通信遅延を考慮した MPI 通信ライブラリの設計”, 情報処理学会，SWoPP 2003, pp.95-100, 2003.
3) 松田元彦，石川裕，工藤知宏，手塚宏史: “MPI 通信に適した通信 API の設計と実装”, 情報処理学会，SWoPP 2003, pp.101-106, 2003.
4) 児玉祐悦，工藤知宏，佐藤博之，関口智嗣: “ハードウェアネットワークエミュレータを用いた TCP/IP 通信の評価”, 情報処理学会，SWoPP 2003, pp.47-52, 2003.
5) G. Wright and W. Stevens: “TCP/IP Illustrated, Volume 2: The Implementation”, Addison Wesley, 1995.
6) M. Allman, V. Paxson, W. Stevens: “TCP Congestion Control”, RFC 2581, April 1999.
7) T. Kelly: “Scalable TCP: Improving Performance in Highspeed Wide Area Networks”, `http://www-lce.eng.cam.ac.uk/~ctk21/papers/scalable_improve_hswan.pdf`, Submitted for publication, December 2002.
8) S. Floyd: “HighSpeed TCP for Large Congestion Windows”, Internet Draft, `http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-highspeed-00.txt`, Work in progress, July 2003.
9) C. Jin, D. Wei, S. H. Low, G. Buhrmaster, J. Bunn, D. H. Choe, R. L. A. Cotterll, J. C. Doyle, W. Feng, O. Martin, H. Newman, F. Paganini, S. Ravot and S.Singh: “FAST TCP: From Theory to Experiments”, April 2003.
10) The Web100 Project, `http://www.web100.org/`.
11) Message Passing Interface Forum, “MPI: A Message-Pasing Interface Standard”, `http://www.mpi-forum.org/`, June 1995.
12) MPICH, `http://www-unix.mcs.anl.gov/mpi/mpich/`.
13) LAM/MPI, `http://www.lam-mpi.org/mpi/`.