通信状態の可視化によるメッセージ通信型並列プログラムのデバッグ支援ツール

渡邉淳平
平石裕実

あらまし

近年のIT技術の進展に伴い，安価で高性能なコンピュータが開発され，これらのコンピュータを複数台用いてクラスタ・システムを構築することが容易になってきている．クラスタ・システム上での並列プログラムは，主としてMPIと呼ばれるメッセージ通信型のライブラリを用いて開発される．メッセージ通信型の並列プログラムは，シーケンシャルプログラムに比べると，デッドロックや同期不良の問題があるため，開発が困難である．本研究で開発した可視化ツールは，各MPIのメッセージ通信関数に対して，その関数へのパラメータや関数の実行結果の状況，関数の開始と終了時刻等に関する情報をログファイルに書き出す機能を付加し，これらのログファイルを収集して解析することにより，並列プログラムの実行に際して実際に発生したメッセージ通信の状況を可視化することが出来る．これにより，メッセージ通信の全体的な状況な把握が容易となり，デッドロックや同期不良が発生した場合に，その原因追求に役立つと考えられる．

1 はじめに

安価で高性能なパーソナルコンピュータ（PC）が開発されるのに伴い，複数のPCを結合したクラスタ・システムが構築されるようになってきている．これらのクラスタ・システムは，スーパーコンピュータに比べると極めて安価に構築することが出来，コストパフォーマンスが優れている．このような背景の下で，Beowulf[1]のような，複数のLinuxコンピュータをクラスタ化して，仮想並列スーパーコンピュータを形成する技術も提案され，ますます容易にクラスタ・システムを構築できるようになり，サーバのようにアクセスが集中するマシンの負荷分散を行うための並列処理や大規模科学技術計算における並列プログラムの開発がさかんに行われている．

クラスタ・システム上で動作する並列プログラムを開発するのは，一般に逐次型プログラムの開発より複雑になる傾向がある．並列プログラムでは，プログラム相互の通信部分が重要になり，複数台のマシンで協調して動作する必要がある．一台でも予期しない動作を行うと並列

プログラム全体が停止することがある．また，逐次型プログラムに比べると，バグの発生箇所を見つけるのも，より困難になることが多い．そこで，本研究では，Message Passing Interface (MPI)[11,12]を用いた並列プログラムの開発・デバッグを容易にするためのデバッグ支援ツールの開発を行った．本ツールは，並列プログラムの通信状態を可視化することにより通信の構造を把握しやすくしようとするものである．

2　MPIと開発環境

2.1　MPI

MPI[11,12]は分散メモリ型並列処理用に作られたFortranとC言語のライブラリ群である．PDS（Public Domain Software）から商用版まで多くの種類がある．並列処理の分野では広く使われており，クラスタマシンの普及とともに多くのユーザーに支持されている．

現在MPIはMPI−1とその拡張版のMPI−2がある．MPI−2は，動的プロセス生成，単方向通信，外部インターフェース，拡張集団通信，並列I/O，C++およびFortran90インターフェースなど，非常に広範囲の拡張が行われている．本大学の環境では商用版のMPI/Pro[5]が使われており，規格としてはMPI−1にのみ対応している．よって，以下でMPIといえばMPI−1を意味することにする．またMPI/ProはTCP/IPネットワークのほかに米EMULEX（旧GigaNet社）[6]のcLanシリーズに対応している．cLanシリーズはVIA（Virtual Interface Architecture）をハードウェア上で実装しており，TCP/IPをはるかにしのぐパフォーマンスを実現している．VIAは大規模クラスタマシンではよく使われるプロトコルであり，データ転送時のCPUの負荷率やデータ転送の遅延を大幅に小さくするとともにデータ転送速度が飛躍的に向上する．

MPIは基本的にメッセージパッシングと呼ばれる方式を使う．この方式は，プログラマが自らデータの配置やデータの転送を記述し，チューニングがしやすく，使い方次第ではシステムの性能をかなり引き出せる方式である．分散メモリ型の並列マシンでよく使われ，各プロセッサが個々のメモリ空間を持っているので，マシン同士でデータをやり取りするために送受信が使われる．一対一の通信のほかに集団通信を実装しており，行列の計算などで威力を発揮する．またタイミングの制御や非同期通信など便利な機能が多く組み込まれている．しかし，一方でメッセージパッシングを使ったプログラミングは設計・開発するのが難しく，「並列処理のアセンブラ」と呼ばれることがある．複数台のマシンで計算を進めていくため，プログラミングの構造が複雑になりやすく，ときには全体像が見えにくくなる．

2.1.1　1対1通信

MPIでの1対1通信では，メッセージ追い越しは発生しない．すなわち，送信側が2つのメッセージを続けて同じ送信先に送信し，両方の送信が同じ受信と対応する場合には，第1メッセ

ージがまだ保留状態にある間は，この受信操作は第2メッセージを受信することができない．受信側が2つの受信を続けて発行し，両方が同じメッセージに対応する場合には，第1受信がまだ保留状態にある間は，第2受信操作はこのメッセージに対応できない．これはシステムがメッセージをバッファリングするような場合にも通用される．これにより，プロセスがシングルスレッドでしかも受信側においてワイルドカードMPI_ANY_SOURCEやMPI_ANY_TAGが使用されていない場合に，メッセージ通信の対応関係が明確になる．

図1は実際のMPI通信をモデル化している．このように送信Aは受信Aで，送信Bは受信Bで受信され，メッセージの順序は保存される．

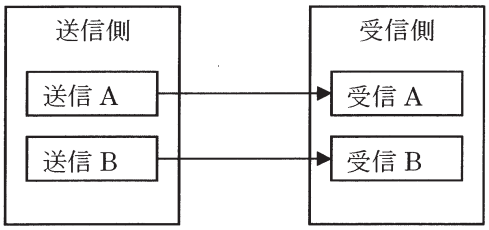


図1　メッセージの順序の保存

図2はメッセージの追い越しを表しているが，MPIでは，このようなことは生じないことが保証されている．

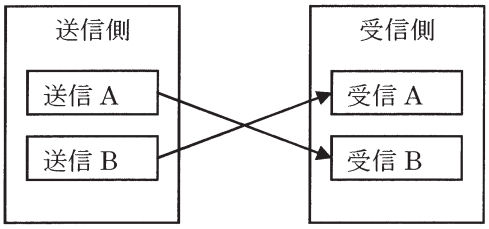


図2　メッセージの追い越し

1対1の通信に種類があってブロッキング／ノンブロッキングに分けられる．

(a) ブロッキング通信

メッセージは，対応する受信バッファに直接コピーされるか，もしくは，一時的なシステムバッファにコピーされる．メッセージのバッファリングは，送信操作と受信操作を独立なものとする．ブロッキング送信は，たとえ対応する受信が受信側で実行されていなくても，メッセージがバッファリングされるとすぐに完了することができる．一方で，メッセージバッファリングは，新たなメモリ間コピーを伴い，また，バッファリングのためのメモリアロケーションを必要とするのでコストが高くなるかもしれない．MPIはいくつかの通信モードの選択肢を提供しており，これにより，通信プロトコルの選択を制御することができる．

標準送信モード（MPI_Send）では，送出メッセージがバッファリングされるかどうかを決定するのはMPIの実装に任せている．MPIは，送出メッセージをバッファリングするかもしれない．このような場合には，送信呼び出しは，対応する受信が起動される前に完了することができる．一方で，バッファ領域が使用できないかもしれないし，MPIが性能上の理由から，送出メッセージをバッファリングしないことを選択するかもしれない．この場合には，送信呼び出しは，対応する受信が発行され，そして，データが受信側に移動してしまうまでは完了しない．

このように，標準モードでの送信は，対応する受信が発行されているかどうかにかかわらず，開始することができる．これは，対応する受信が発行される前に完了することもある．標準モード送信はノンローカルである，つまり，送信操作が成功完了するかどうかは，対応する受信が発生するかどうかに依存する．

図3はシステムのバッファリングが無いときの同期通信で，受信側が関数の準備ができていないならば送信動作は完了できない．受信の準備ができるまで送信側のプロセスは動作を停止しなければならない．

図3　同期通信（バッファリング無し）の通信待ち

図4は図3の次の段階の動作で，受信の準備が整ったならば送信を開始できる．

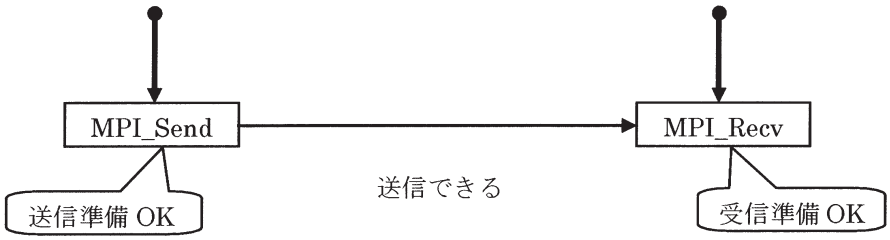


図4　同期通信の通信成立

図5はシステムがバッファリングを許可する場合の通信である．送信側は受信側の準備が整っていなくてもデータをシステムバッファに送信し，通信動作を完了する．システムバッファ内のデータは受信側が準備を完了した段階で，システムバッファから受信側に送信される．なお，ハイテクリサーチクラスタシステムではバッファリングされる．しかし，バッファには制限があり大きすぎるデータの場合バッファから溢れてしまい，プログラムが正常を保てない可

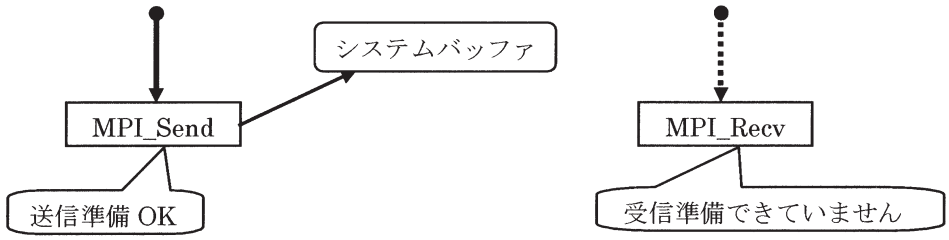


図5　同期通信のバッファリング

能性がある．

(b) ノンブロッキング通信

ノンブロッキング通信開始が呼ばれると，送信操作が起動されるが，完了はしない．送信バッファからメッセージがコピーされる前に送信開始の呼出しは戻ってくる．通信を完了させるため，すなわちデータが送信バッファからコピーされたことを確認するためには，別の送信完了の呼出しが必要となる．同様に，ノンブロッキング受信開始呼出しは受信作業を起動するが完了はしない．この呼出しはメッセージが受信バッファに格納される前に戻される．受信操作を完了させ，データが受信バッファに受け取られたことを確認するためには，別の受信完了の呼出しが必要である．

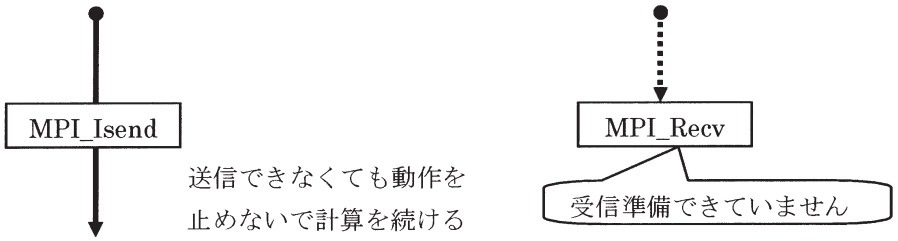


図6　非同期通信

図6は送信側が非同期通信で，受信側が同期通信の通信である．送信は受信の準備ができていなくても終了することができ，通信動作は即座に終わる．しかし実際の通信がいつ行われるかはユーザーには判らない．ユーザーがこの時点までには通信が終わって欲しいと思うところにMPI_Wait関数を置いて動作を確認することができる．

2.1.2　集団通信

集団通信は，グループの中の全プロセスが同一の引数をもって通信ルーチンを呼び出すことで同期的に実行される．集団通信の構文および意味は，1対1通信の構文および意味と同じように定義される．したがって，一般のデータタイプが利用可能であり，このタイプは送信側プロセスと受信側プロセスとで一致しなければならない．重要な引数の1つに，通信に参加する

プロセスのグループを定義し，通信のためのコンテキストを与えるコミュニケータがある．ブロードキャストやギャザといった幾つかの集団通信ルーチンでは，1つのプロセスだけがメッセージを発信したり受信したりする．そのプロセスはルートと呼ばれる．集団通信関数の引数には，“ ルートでのみ意味を持つ ” ものがあり，ルートを除く全参加プロセスでは無視される．

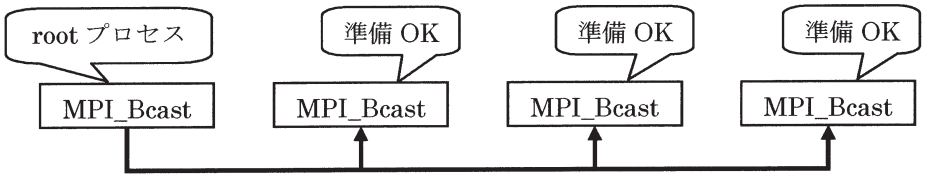


図7　集合通信（MPI`_Bcast）

図7はMPI_Bcastの例でrootプロセスから他のプロセスにデータを送信している．集合通信の場合同じコミュニケータを持つ全プロセスで同期的に通信が行われるので，全プロセスの準備が完了していないと通信は行われない．

2.2　現状のデバッガ

現在ある並列プログラム用デバッガは，動作の仕方で二種類に分けられる．TotalView[4]，MPIGDB[7] およびdbxR−II[8]に代表されるような，実行自体をブレイクポイントなどで制御して，並列プログラムに対してステップ単位でデバッグする方式がある．TotalViewはデータの視覚化機能などを備えており，並列プログラムをステップ実行しながら組み合わせ判定が未解決のメッセージに関して通信依存をMessage Queue Graph（MQG）として視覚化できる．MPIGDBは逐次デバッガGDBを基にしていて，Multi-Purpose Deamon（MPD）を通じてプロセスごとにアタッチしたGDBを制御できる．dbxR-IIは非決定的な並列プログラムに対して，同じ実行動作を繰り返し得る再宴法を実現しサイクリックなデバッグを可能にする．このタイプのデバッガは次に述べるログファイルを取るデバッガに比べてより詳細にプログラム実行を制御でき，逐次デバッガGDBのようにメモリダンプなどによりプログラムの内側から調べることができる．Vampir[3]，XMPI[9]およびATEMPT[10]などに代表される実行時にログファイルを作成・保存して，計算・通信の状態を時系列に視覚化してプロセスの挙動を把握する方式がある．このタイプのデバッガはログをファイルに保存するためのオーバーヘッドがMPIプログラムの実行に対して生じるので取り扱いには気を付けなければならない．しかし，ステップ単位のデバッガに比べて全体像が把握しやすく，どういう通信が起こっているのを容易に理解できる．プロファイラに近い部分がある．[13]

2.3 ハイテクリサーチクラスタシステム

本論文で使用したクラスタシステムの概要を表 1に示す．これらはすべて100Base-Tx Ethernetおよび1Gbps cLanのネットワークに繋がれており，共にTCP/IPでの通信ができ，cLanではVIAによる通信もできる．各マシンはデュアル・プロセッサ構成であり，hr000～hr100のOSはRedHat 6.2Jでカーネルは2.2.24-6.2.3smpである．hr101～hr104は他のマシンよりかなり後に追加導入されたので他に比べてスペックに差がある．この4台はOSはRedHat 9でカーネルは2.4.20–18.9smpである．

表1 ハイテクリサーチクラスタのスペック表

ホスト名	CPU	Memory	Disk
hr000	Dual PentiumIII 866MHz	1GB	9.2GB × 3
hr001～hr060	Dual PentiumIII 750MHz	512MB	9.2GB × 2
hr061～hr100	Dual PentiumIII 866MHz	768MB	9.2GB × 2
hr101～hr104	Dual Xeon 2.4GHz	1GB	18.2GB × 2

2.4 GTK+

GTK+（GIMP ToolKit）[14]は，グラフィカルユーザーインターフェイス（GUI）によってアプリケーションを開発するためのライブラリであり，Linuxのアプリケーション開発に広く用いられている．C言語で書かれたオブジェクト指向のライブラリであり，Ada95+，C++，Eiffel，Objective-C，Pascal，Perl，Pythonなど多くの言語で書かれたアプリケーションをサポートする．本研究で開発したデバッグシステムはC言語を用いて作られている．Linux用GUIアプリケーションを構築する際のツールとして標準なのはXlibであるが，アプリケーションを迅速に開発するには不向きな低レベルのライブラリである．GTK+は描写にあたっては常にGDK（GIMP Drawing Kit）を用い，Xlibなどのグラフィック APIと直接やり取りするのでなくGDKを通じてアクセスする．

本ソフトウェアでは，gtk+–1.2.6–7を使って開発している．現在gtk+は2.2.xまでバージョンアップされており，各種ウィジェットは高性能になっているが現時点で安定しているgtk+-1.2.x系を使うことにする．

3 hrcデバッガ

3.1 システム概要

本ソフトhrcデバッガは並列プログラムを視覚化することによってデバッグする．その手順は図8にあるように，まずソースファイルを変換プログラムにかける．変更したソースファイルにオブジェクトファイルtmpi.oをリンク・コンパイルし，実行ファイルを作る．できた実行

ファイルを各マシンに配り，実行させると各マシンにログファイルが作成される．これを集めて一つのログファイルとして結合する．それをhrcデバッガを使い視覚化すると，通信状態を見ることができる．ユーザーはこれを見ながらまたプログラムを修正する．このように循環した使い方をすることでデバッグに利用する．

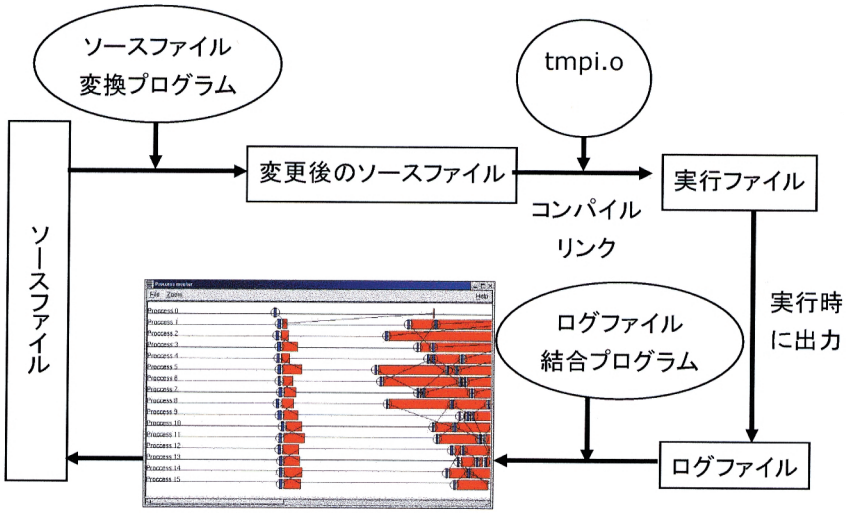


図8　システム概要

3.2　ログ情報

通信を視覚化しユーザーに情報を提供するためにはどんな情報が必要になるのか考察する．まずどのMPI関数が呼ばれたか関数名を記述する必要がある．通信の開始された時間と終了した時間は視覚化するため必要な情報である．MPIは時間を計測するための関数を用意している．MPI_Wtime関数は1970年1月1日からの経過時間を，秒数を表す倍精度実数で返す．これをMPI通信関数の前後で起動し時間を計測する．MPI関数を呼び出すときの引数もログに記述する必要がある．1対1通信と集団通信では引数の種類が違う．1対1通信の場合，送信元と送信先を指定し通信タグも指定しなければならない．集団通信の場合，グループの中の全プロセスが同一の引数をもって通信ルーチンを呼び出すことで実行される．集団通信はルートと呼ばれるプロセスだけが送信したり受信したりするので，引数に送信元，送信先および通信タグを指定する必要がない．

3.3　ログファイルのフォーマット

この節では，ログファイルのフォーマットについて記述する．ログファイルは一行に一イベントを含んでおり，データはコンマで区切られている．一イベントとは，プログラム中でMPI通信関数が一回呼ばれたときの情報である．ログファイルの先頭には，個々の通信ログデータ

```
1:  --MPI_Start--,1071742765.355226
2:  --MPI_End--,1071742765.402195
3:  --MPI_Proc_Num--,8
4:  --Proc_Comm_Num--,0,12
5:  --Proc_Comm_Num--,1,12
6:  --Proc_Comm_Num--,2,12
7:  --Proc_Comm_Num--,3,12
8:  --Proc_Comm_Num--,4,12
9:  --Proc_Comm_Num--,5,12
10: --Proc_Comm_Num--,6,12
11: --Proc_Comm_Num--,7,12
12: --MPI_Send--,0,2048,6,1072442568.822679,1,0,1072442568.822851
    ......
```

図9　ログファイルの例

の前にプログラム全体に関する情報を記述している．

図9にログファイルの例を示す．--MPI_START-- はプログラム全体の開始時間（秒），--MPI_END– はプログラム全体の終了時間（秒）を表す．–MPI_Proc_Num– はプログラムが何台のマシンによって動いているかを表し，この例では8台のマシンで実行されている．--Proc_Comm_Numm– は各プロセスでの通信イベントの発生数を表しており，この例では，プロセスランク0～7までの全てのマシンで各々12回通信が行われていることを示している．–MPI_Send– は，MPI_Send関数を使ったときに残されるログ情報である．各情報はコンマで区切られている．左から通信イベント名，自分のランク，通信データ数，通信データ型，通信開始時間，通信相手のランク，通信タグ，通信終了時間が記録される．

3.4　ソースファイル変換プログラムtransform

元のソースファイルのMPI関数を変更するプログラムであり，図10に示すように“MPI_”で始まるMPI関数を“MPI_T”に置き換える．それとログファイルに関する関数，MPI_Logfile_Open関数とMPI_Logfile_Close関数を付け加える．MPI_Logfile_Open関数は引数に自分のランクを取る．例えば，“transform before.c after.c”と入力する場合，before.cが元の

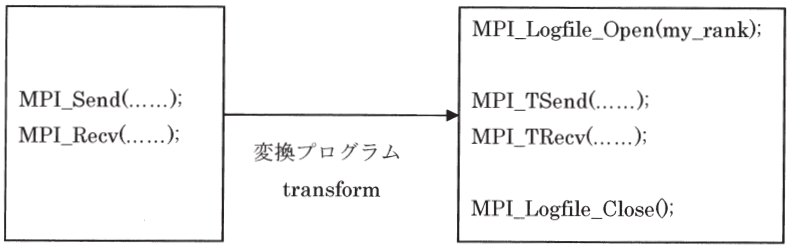


図10　“transform”による変換

ソースファイル，after.cが変換後のソースファイルである．

3.5　MPI_T関数

MPI_T関数は，MPI関数に対してログを記述するように変更した関数群である．変換プログラムtransformによってMPI関数はMPI_T関数に変換され実行される．3.7.4節で説明しているが，タイマースレッドとメインスレッドに分かれ，メインスレッドではまずMPI_Wtime関数を使ってプログラムの開始時間が記録される．そしてユーザーがMPI関数を呼び出すときに指定した引数を記録する．次に実際のMPI関数を使って通信が行われ，通信が正常終了するとMPI_Wtime関数を使ってプログラムの終了時間が記録される．

3.6　ログファイル結合プログラムpreparation

生成された複数のログファイルを一つにまとめるプログラム．10台でプログラムを動かした場合，logfile0.log～logfile9.logが各マシン上に作られる．各マシンからhr000にログファイルを集めて，結合プログラムを実行する．例えば，“ preparation sendrecv.log ”と入力した場合，同じディレクトリにあるlogfile*.logを全て結合し，sendrecv.logという名前で一つにまとめたログファイルをつくる．

各ログファイルに“ --MPI_Start-- ”という項目があり，各マシンでプログラムが起動された時間が書いてある．プログラムが起動される時間は多少誤差があるので最も初めに起動されたマシンの時間を結合ファイルに記述する．次に，“ --MPI_End-- ”という項目があり，各マシンでプログラムが終了された時間が書いてある．“ --MPI_Start-- ”と同じように誤差があるので最も終わりに終了したマシンの時間を結合ファイルに記述する．次に何台のマシンでプログラムを動かしたかを調べる．これはログファイル数を調べたらわかることである．そして，各マシンが何回通信を行ったかを調べる．結合前のログファイルは自分のプロセスの通信の事しか記述されていないので，それを数えてプロセスxは何回通信をしたかを記述する．

3.7　通信イベントの対応付け

ログファイルには各通信の情報が記述されているが肝心なところの情報がない．それは送受信の対応関係が判らないという点である．1対1通信の場合は，送信元は通信相手のランクは判るが，どの時点での対応する受信が自分のパートナーか調べなければならない．集団通信の場合は，同じコミュニケータ内の全プロセスが同じ集合通信を呼び出すのでパートナーを調べるのではなくrootプロセスがどのランクなのかを調べる必要がある．rootプロセスがどのランクか判れば，rootプロセス以外のランクはrootプロセスに対して何らかの通信をしているのでパートナーとして設定する．

図11に1対1通信の場合の対応付けの様子を示す．2.1.1節で述べたようにメッセージは追い

越し禁止であるので，各プロセスの通信イベントを先頭から見て行き，通信相手のランク情報から対応する可能性のあるイベントのうちで，最初に現れる対応付けが行われていないもの同士を対応付ければ良い．

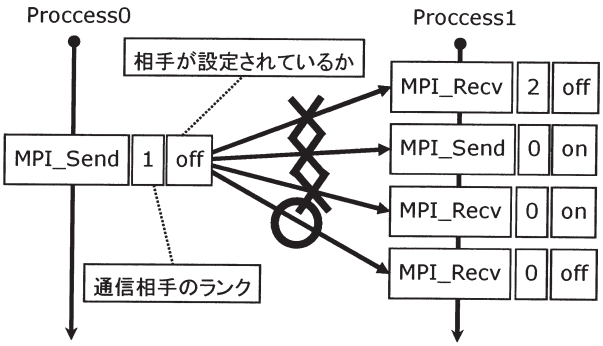


図11　1対1通信の対応付け

図12は集合通信の対応付けの様子である．集合通信の場合，全てブロッキング同期通信であるので，各プロセスの通信イベントを先頭から調べて行き，同一の集合通信同士を対応付ければ良い．このときrootランクを調べることにより通信の方向を決めることが出来る．

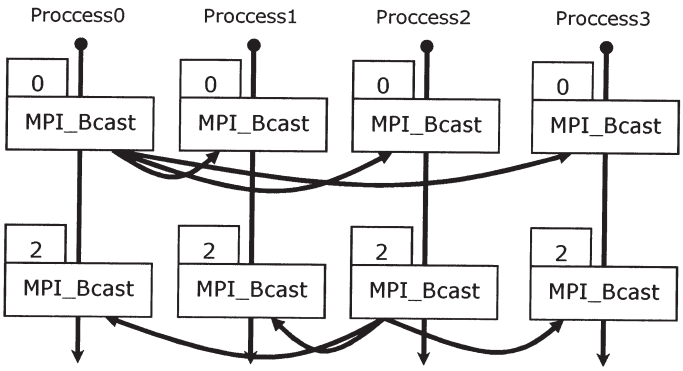


図12　集合通信の対応付け

3.8　タイマースレッド

MPI_T関数にはタイマーが組み込まれている．これは通信が一定時間内に終わらない場合，MPI_Abort関数を使ってプログラム全体を終了してしまうものである．MPI_T関数は開始されると図13に示すように2つのスレッドに分かれて実行され，メインスレッドは通信を行い，タイマースレッドはタイマーを動かす．タイマーが設定した時間内に通信が終了したら，何もせずに終了する．設定した時間内に終わらなかった場合，MPI_Abort関数を使ってプログラムを強制終了する．このときログファイルに“ –MPI_Abort– ”を記述し，MPI_Abortが起動した時

刻を記す．これによりどのプロセスの通信がユーザーが決めた時間内に終了しなかったかが判り，デッドロックなどで動作が止まった場合にもMPI_Abortを起動して終了してくれる．

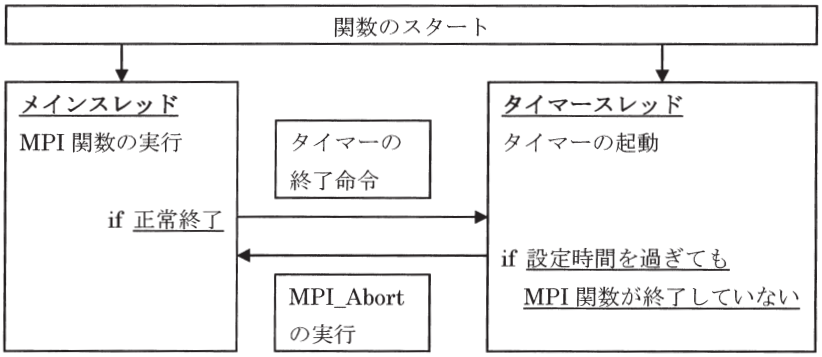


図13　タイマースレッドの動作説明

しかし，プログラムによっては，この機能が邪魔になる場合もある．そのために設定を選べるようにした．その方法は表2に示す環境変数を使う．タイマー機能を使うかどうかは環境変数MPI_ABORT_SWITCHでオン／オフの切り替えができる．タイマー機能を使うなら，その通信が反応のない状態が何秒続いたら強制終了するのかも設定できる．環境変数MPI_ABORT_TIMEでその時間を設定する．デフォルトではタイマー機能は使わない設定になっている．

表2　タイマースレッド用の環境変数

MPI_ABORT_SWITCH	MPI_ABORT_TIME
0：オフ　1：オン	非負整数　マイクロ秒単位

4　動作概要

4.1　使い方

まず，図14に示すように，元となるソースファイルに対して変換プログラムを実行する．こ

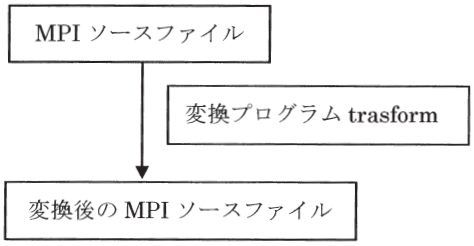


図14　ソースファイルの変換

れはMPIの関数を自分で作成した新しい関数に置き換えるためである．

次に，図15に示すように，変換したプログラムを，ライブラリにリンクしてコンパイルし全てのマシンに配布する．

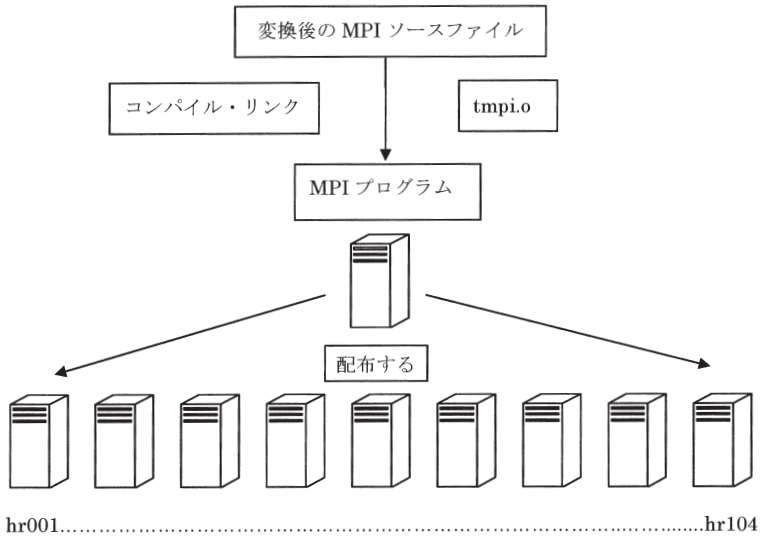


図15　プログラムのコンパイルと配布

このプログラムをhr000から実行すると，各マシンにログファイルが作られる．そして，各マシンからhr000にログファイルを集める．ログファイルは実行したマシン数だけあるのでそれ

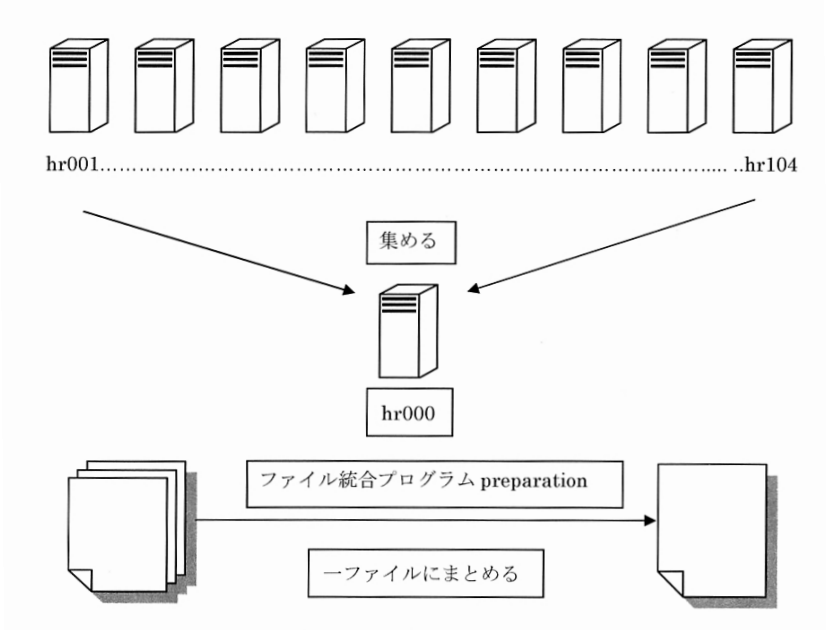


図16　ログファイルの統合

を一ファイルにまとめる（図16参照）．

これによってできたログファイルをグラフィカルソフトにかけると，図17に示すように通信状態を表示できる．図において，上から順にProccess 0，Proccess 1…というように昇順に並んでいる．横軸は時間を表している．この例では，青の矩形がMPI_Sendを表しており赤の矩形がMPI_Recvを表している．青の矩形から赤の矩形へ伸びている線は，データの通信を表している．データを送信しているイベントには○が付いている．

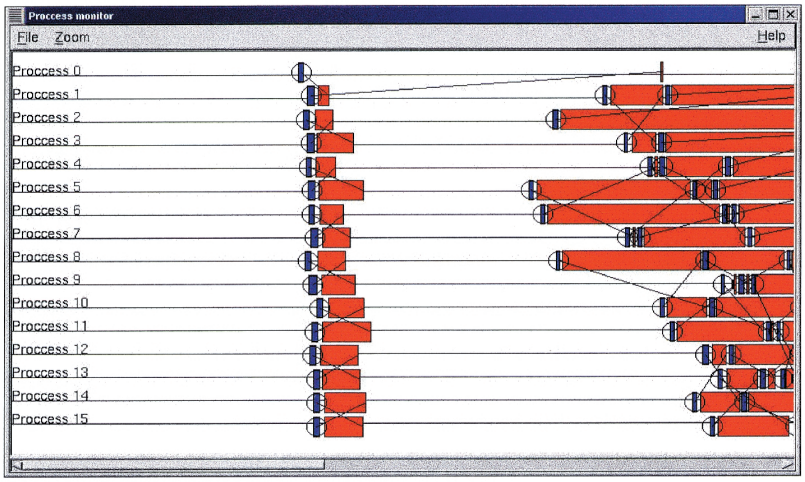
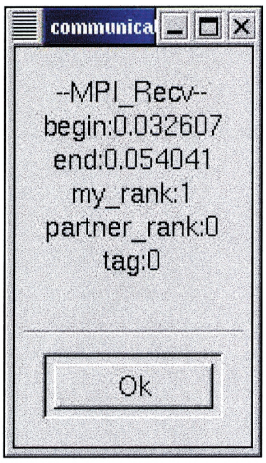


図17　通信状態の表示

ここで，MPI_Recvの矩形をクリックすると図18に示すように，MPI_Recvに関する情報がポップアップ表示される．



通信イベント名	--MPI_Recv--
通信開始時間※	0.032607
通信終了時間※	0.054041
自分のランク	1
通信相手のランク	0
通信タグ	0

※通信開始・終了時間はプログラムの開始された時点からの経過時間を表している

図18　通信イベントグラフのポップアップウィンドウ

4.2 例題

例として，通信の順序が実行時に決定するプログラムを視覚化してみる．ソースコードが同じでも通信状態が違ってくることを考察する．プログラムは3台のマシンで動かすものとし，そのソースコードの主要部分を図19に，また，プログラムの流れを図20に示す．

```
0: switch(rank){
1:   case 0:
2:     memset(buf2,64,0);
3:     MPI_TBcast(buf1,64,MPI_CHAR,0,MPI_COMM_WORLD);
4:     MPI_TSend(buf2,64,MPI_CHAR,1,tag,MPI_COMM_WORLD);
5:     break;
6:   case 1:
7:     MPI_TRecv(buf2,64,MPI_CHAR,MPI_ANY_SOURCE,tag,MPI_COMM_WORLD,
8:                 &status);
9:     MPI_TBcast(buf1,64,MPI_CHAR,0,MPI_COMM_WORLD);
10:    MPI_TRecv(buf2,64,MPI_CHAR,MPI_ANY_SOURCE,tag,MPI_COMM_WORLD,
11:                &status);
12:    break;
13:  case 2:
14:    memset(buf2,64,1);
15:    MPI_TSend(buf2,64,MPI_CHAR,1,tag,MPI_COMM_WORLD);
16:    MPI_TBcast(buf1,64,MPI_CHAR,0,MPI_COMM_WORLD);
17:    break;
18: }
```

図19 実行順序可変のソースコード

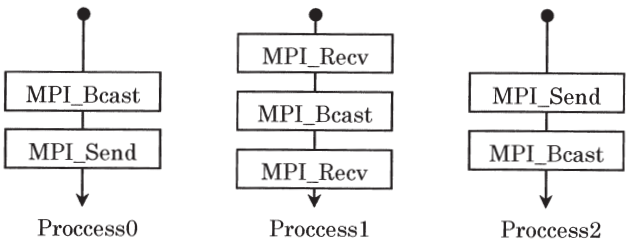


図20 プロセスの通信順序

この場合，Proccess1のMPI_RecvはProccess0かProccess2のどちらのMPI_Sendと対応するかは実行環境や実行タイミングにより違ってくる．Process0のMPI_Bcastのデータをシステムがバッファリングすればproccess1の最初のMPI_RecvはProccess0のMPI_Sendと対応することも可能である．

この様な場合，本プログラムを使ってログを取り視覚化するとプログラムの通信がどのよう

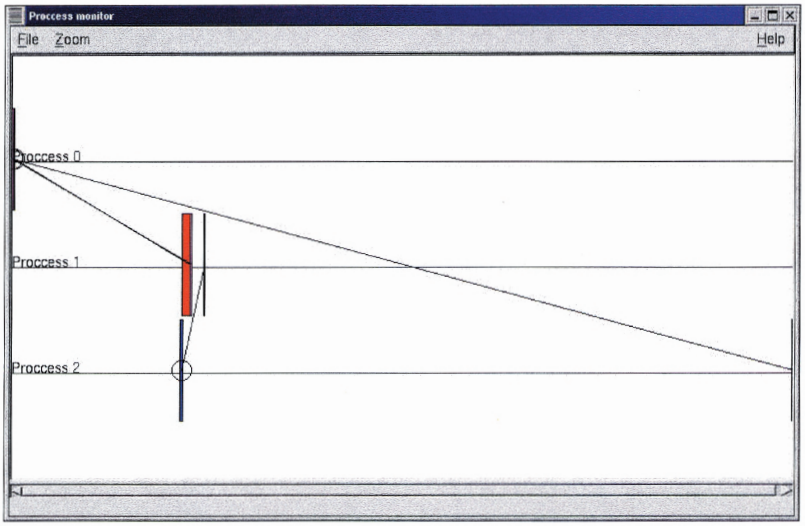


図21　hrcデバッガのイベントグラフ

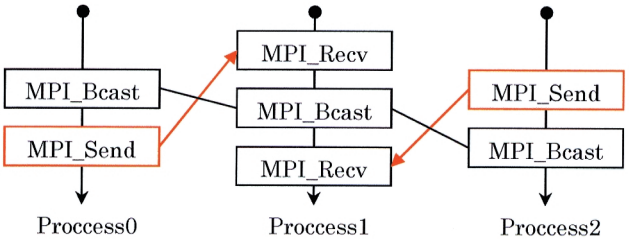


図22　図21の概念

に起こっているのかが判る．

例えば，図21のようなイベントグラフが得られたとする．図において，ピンクがMPI_Bcast，青がMPI_Send，赤がMPI_Recvである．縮小していて少し判りにくいが，図21はProccess1の最初のMPI_RecvにProccess0のMPI_Sendが対応している．この状況を図22に概念的に描いているが，この場合Proccess0のMPI_Bcastがシステムにバッファリングされ，次のMPI_Sendがすぐに開始されProccess1の最初のMPI_Recvと対応したと考えられる．

図23は，同じプログラムの別の実行例に対するイベントグラフであり，図24はその概念図である．この場合，Proccess1の最初のMPI_RecvにProccess2のMPI_Sendが対応している．このときはシステムがバッファリングしなかったのか，バッファリングしたが図21のときと異なりProcess0でのMPI_Sendを実行するタイミングが遅れたためにこのような通信になったと考えられる．

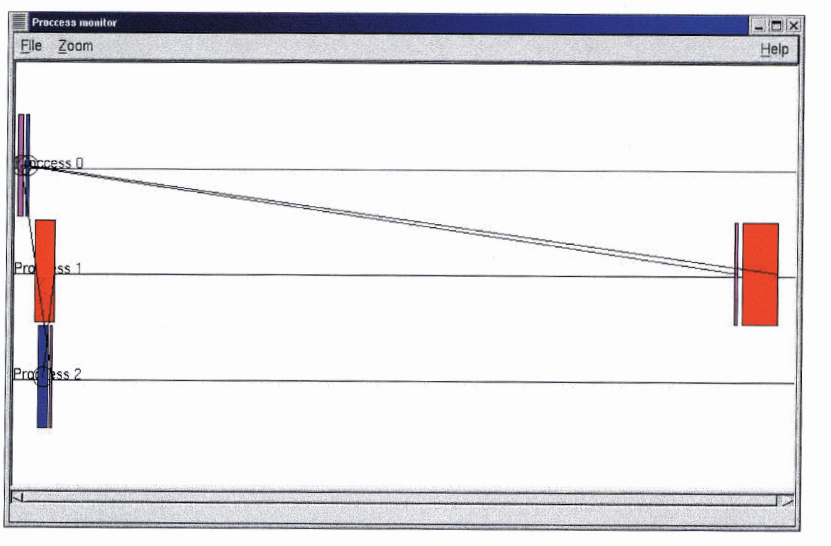


図23 別の実行例に対するイベントグラフ

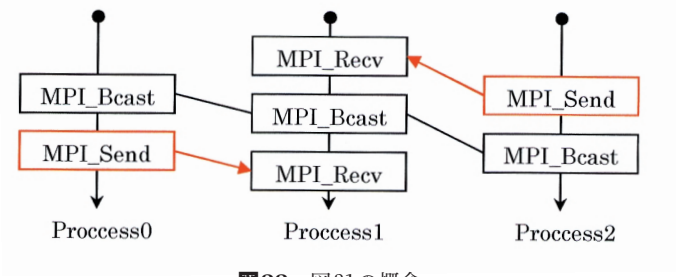


図22 図21の概念

4.3 オーバーヘッドの測定

プログラムのログ情報を取り出すことによるオーバーヘッドを調べた．一例として，バイトニックソートをMPI関数とMPI_T関数の両方で実行し動作への影響を調べた．このプログラムはMPI_SendrecvとMPI_Gatherを使って通信をしているので，この二つをMPI_TSendrecvとMPI_TGatherに変更した．変更後ソートプログラムを動かしプログラム全体の実行時間を測定した．2の26乗個のデータと2の27乗個のデータを1，2，4，8，16，32台のマシンでソートしたときの実行時間を図25に示す．これより，プログラムの実行時間への影響はほとんどないように思われる．

図26はMPI_T関数を使って動作させたバイトニックソートの実行時間（秒）からMPI関数を使って動作させたバイトニックソートの実行時間（秒）の差を示しているが，両者の差はほぼ±1秒以内に収まっておりMPI_T関数を用いることによるプログラムへの影響はほぼないと考えられる．

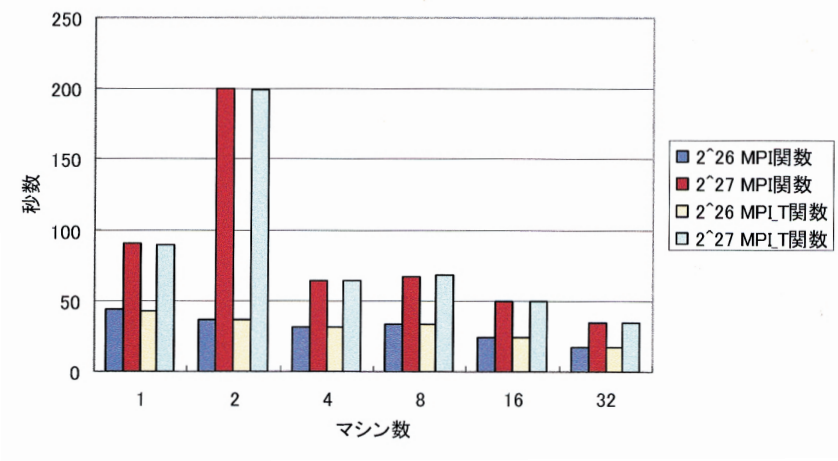


図25 MPI関数使用時とMPI_T関数使用時の実行時間

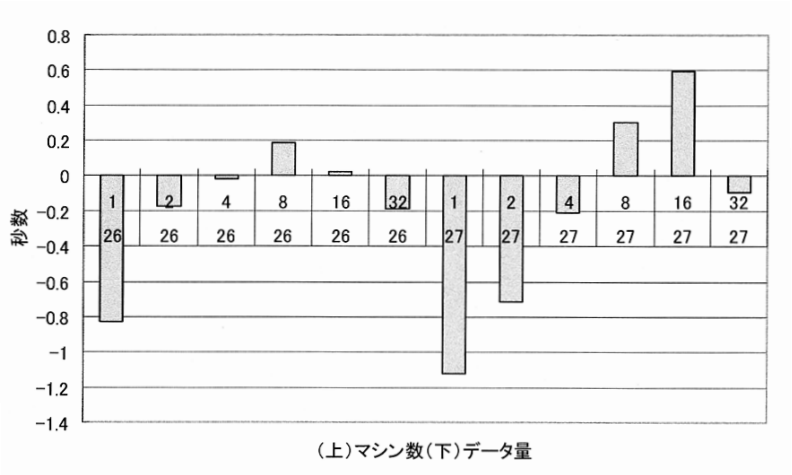


図26 MPI_T関数とMPI関数の実行時間の差分

5 ま と め

本ソフトhrcデバッガを使って通信状態を視覚化することには成功したが，実際に利用するためにはまだ機能が少なすぎる．実際に通信が上手く動作している場合はイベントグラフを描写してくれるが，プログラムの途中で通信が止まった場合などの異常終了が起こった場合はイベントグラフを上手く描写できない．これは致命的なことである．この部分の強化をしなければならない．

ログをはき出すのにかかる時間はほぼ無視できるので，デバッグされるプログラムへの影響はほとんど無いと思われる．しかし，プログラムがもっと大規模になれば影響も出てくると考えられるので，デバッグされるプログラムへの影響をより小さくしなければならない．

MPI通信の対応付けアルゴリズムは無駄がありまだ高速にできる余地があるので，もっと改良していかなければならない．

ユーザーに対して表示する情報が少ないので，より多くの情報を提供できるようにしていきたい．しかし，多すぎる情報はユーザーにとってデバッグ作業をより複雑にするので効果的な情報を提供していけるようにしたい．

本ソフトが対応しているのは通信グループMPI_COMM_WORLD内での通信に限られている．ユーザーが個別に通信グループを作成し，それを使いプログラムを構築した場合，このソフトでは対応しきれない．グループを分けるのはそれほど行われることではないが，今後これに対応していきたい．

本ソフトではまだ全MPI関数に対しての実装を完了していない．表 3は実装を完了したMPI関数の表である．

表3 実装したMPI_T関数

MPI 関数	実装した MPI_T 関数
MPI_Send	MPI_TSend
MPI_Ssend	MPI_TSsend
MPI_Rsend	MPI_TRsend
MPI_Bsend	MPI_TBsend
MPI_Isend	MPI_TIsend
MPI_Recv	MPI_TRecv
MPI_Irecv	MPI_TIrecv
MPI_Wait	MPI_TWait
MPI_Sendrecv	MPI_TSendrecv
MPI_Bcast	MPI_TBcast
MPI_Gather	MPI_TGather

MPI関数全てをMPI_T関数に変更することができなかった．時間的に間に合わなかった部分もあるが，構造的に見直す必要もある．ログ情報を記録する構造体typProccess_dataに全種類の通信の情報を記録させようとしたため，1対1通信では使うが集団通信では使わないような変数が出来てしまった．通信の種類に合わせたデータ構造体を作るべきだった．

通信に関する情報もまだ少なく，ユーザーに対して効果的に提示する方法も未熟であった．通信量が多いプログラムを可視化するとグラフが複雑になり，本来の目的であるプログラムの全体像を掴むことが難しくなってしまった．

今の段階では，ユーザーがイベントグラフを見て自分で異常と思うところを発見しなければいけない．この部分をソフトで自動的に検知するようにできれば，ユーザーの負担が少なくなり，通信に因って生じるバグを効果的に発見することができる．

参 考 文 献

1. Beowulf.Org, The Beowulf Cluster Site, http://www.beowulf.org/.
2. MPI Software Technology : Products, SeeWithinPro, http://www.mpi-softtech.com/products/cluster/seewithinpro/.
3. Pallas GmbH, http://www.pallas.com/.
4. Etnus, LLC. - Homepage for TotalView Multiproccess Debugger/Analyzer http://www.etnus.com/.
5. MPI Software Technology, Products : MPI/Pro, http://www.mpi-softtech.com/products/cluster/mpi_pro/.
6. Emulex Corporation, http://www.emulex.com/.
7. MPICH - A Portable MPI Implementation, http://www-unix.mcs.anl.gov/mpi/mpich/.
8. 三栄武, 高橋直久, PVMプログラムのための再演型デバッガの実現と評価, 情報処理学会論文誌, Vol.37, No.7, pp.1308-1319, 1996.
9. XMPI – A Run/Debug GUI for MPI, http://www.lam-mpi.org/software/xmpi/.
10. GUP Linz - ATEMPT, http://www.gup.uni-linz.ac.at/research/debugging/atempt/.
11. p.パチェコ著, 秋葉博訳, MPI並列プログラミング, 培風館, 2001.
12. MPIフォーラム, MPI日本語訳プロジェクト訳, MPI: メッセージ通信インターフェース標準（日本語訳ドラフト）, 1996年5月31日版, http://phase.hpcc.jp/phase/mpi-j/ml/mpi-j-html/contents.html.
13. 置田真生, 伊野文彦, 藤本典幸, 萩原健一, MPI-PreDebugger:通信依存解析に基づくメッセージ通信並列プログラム向けデバッグ支援ツール, 情報処理学会論文誌, ハイパフォーマンスコンピューティングシステム, Vol.43, No.SIG6(HPS5), Sep. 2002.
14. Eric Harlow著, アンク監訳, GTK+/GDKによるLinuxアプリケーション開発, 翔泳社, 1999.