ハイパフォーマンス
コンピューティング 81－5
(2000. 6. 2)

メッセージプーリング:通信と計算を重ね合わせるクラスタコンピューティング方式の設計

大鎌 広　石澤 祐介　藤原 祥隆
北見工業大学

1クライアントマルチサーバ型のPCクラスタの新たな並列処理方式としてメッセージプーリング方式を提案・設計している. 提案方式は, リモート関数呼び出しを発行する逐次処理プログラムとしてユーザプログラムを記述し, 同期や通信と計算の重ね合わせを明示的に記述することなしに通信時間を隠蔽をしながら並列処理を実行するシステムを目指している. このメッセージプーリング方式の特徴についてメッセージパッシングやリモートプロジジャーコールと対比しながら論じている.

Message Pooling: Design of Cluster Computing System to enable Computation and Communication Overlapping

Hiroshi OHKAMA　Yusuke ISHIZAWA　Yoshitaka FUJIWARA
Kitami Institute of Technology

Message Pooling is designed for one of parallel computational models of one-client multi-server PC clusters. The *Message Pooling* aims that its user-program is described as a sequential processing program issuing many remote functions, and parallel processing with computation and communication overlapping is enabled without description of synchronization and position of computation and communication overlapping. The *Message Pooling* is compared with the message passing and the remote procedure call.

1　はじめに

筆者らは計算機クラスタの新たな並列処理方式としてメッセージプーリング方式(*Message Pooling*)を提案し開発を続けている [7, 8, 9, 10]. 本稿ではメッセージプーリング方式の設計について説明する. 提案方式は1クライアントマルチサーバ型のPCクラスタによる高性能計算用のシステムで

- 自動的に通信と計算を重ね合わせ, 通信レーテンシを隠蔽をする
- ユーザプログラムはクライアントの逐次処理プログラムとして記述する
- クライアントで発行された多数のリモート関数を多数の計算サーバで並列処理することで速度を向上する

ことを目指している.

コストパフォーマンスの高いPCクラスタを構築するためには安価に広く普及している一般品のPCおよびネットワーク機器 (コモディティハードウェア) を構成要素とすることになるが, この場合, PC内のメモリ-プロセッサ間の通信に比して, PC間の通信のバンド幅は狭く, レーテンシは大きい. このためPCクラスタの性能を上げるためには, 計算と通信の重ね合わせと通信レーテンシの隠蔽を実現できる機構が必要である.

PCクラスタで良く利用されるMPI[1, 2, 3] はメッセージパッシングモデルに基づいたプログラミング環境を提供する. MPIでは, ノンブロッキング通信の利用によって, 計算と通信の重ね合わせと通信のレーテンシを隠蔽をできるように設計されている [2].

MPIのノンブロッキング通信を利用して計算と通信を重ね合わせる場合, `MPI_Isend`, `MPI_Irecv`, `MPI_wait` を使いプログラマがどの部分の計算と通信が重ね合わせ可能かを明示的に指定する必要があるため, プログラマにその負荷を強いることになる. さらに同じMPIを使っていても計算性能と通信速度は個々のシステムや計算の状況よって違うため, どの範囲の計算と通信の重ね合わせが可能かをプログ

ラム作成時に予測することは困難である．このためプログラム自身の目的そのものに起因してはいないプログラミングの負荷や困難を課すことになる．

メッセージパッシングモデルのプロセスおよびプロセス間通信と逐次処理モデルの関数および関数間のデータ受渡しを対比させて考える．関数は処理の開始/終了時点でのみ caller-callee 間の通信が可能なのに対し (大局変数の書き換えなどの副作用はないものとする), メッセージパッシングモデルはプロセス実行中に通信を行う柔軟性がある．それゆえメッセージパッシングモデルは込み入った相互作用を表現しやすく粒度の小さい並列処理を実現できる利点がある．しかしながら, 関数は相互作用の少なさゆえに再利用のための単位として取り扱いやすいという利点を持つ．それに対し, 込み入った相互作用を表現したメッセージパッシングモデルの各プロセスは他のプログラムへの再利用の単位としては不適切である．

提案するメッセージプーリング方式は,

- 1クライアントマルチサーバ型のPCクラスタによる高性能計算のための並列計算モデルで,
- プログラマが計算と通信の重ね合わせが可能な場所や同期を明示的に記述することなしに, 計算と通信を重ね合わせて, 通信を隠蔽すること
- 各プロセッサが複数のメッセージを蓄積することで効率良くプロセッサの使用率を上げること
- プログラマに関数と同様のインターフェースを提供し, 関数の持つ理解の容易さ, 再利用のしやすさを実現すること
- ユーザプログラムはクライアント上の逐次プログラムとして記述すること
- クライアントから発行された多くのリモート関数を多数の計算サーバで並列に処理することで速度を向上すること

を目指している．このメッセージプーリング方式を実現した計算システムを *Clop* (Cluster of Pooling) と呼ぶことにする．

2 メッセージプーリングとは?

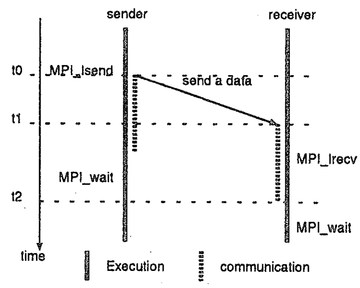

図 1: MPI nonblocking communication.

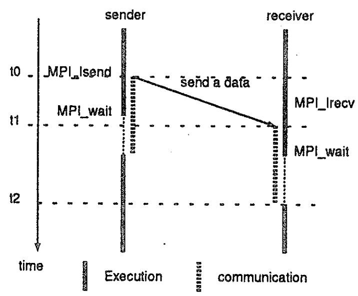

図 2: Blocking with `MPI_Isend` and `MPI_Irecv` when `MPI_wait` has been issued before the end of the communication.

2.1 MPI のノンブロッキング通信の問題点

メッセージパッシングの例として MPI のノンブロッキング通信 [1, 2, 3] を考える．MPI のノンブロッキング通信は `MPI_Isend`, `MPI_Irecv`, `MPI_wait`, `MPI_test` の通信プリミティブで構成される．対応する送受信を決定するマッチング, 通信のバッファリングを送信側受信側のどちらで行なうように実装されているかにより, 実際の通信開始/終了, 通信プリミティブの発行の順序は異なる [4]．図 1に, 単純のためマッチングのプロトコルを省略し, 十分な大きさの通信バッファが受信側にあるとしたときの時系列ダイアグラムを示す．図 1の受信側で `MPI_Irecv` を発行する時間は `MPI_wait` の前であれば `MPI_Isend` 前でも良く, `MPI_Isend`, `MPI_Irecv` 共にブロックせず通信と計算の重ね合わせを可能としている．さらに `MPI_Irecv` を先行発行するときにはゼロコピー通信に切替える効率的実装 [4] も可能である．

しかしながら, `MPI_Isend`, `MPI_Irecv` の発行時にはブロックしないものの, `MPI_wait` をバッファの

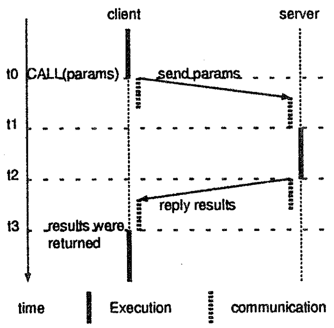

図 3: RPC blocks the execution from "call" to "return" .

再使用が可能になる前に発行するとバッファの再使用が可能になるまでブロックする (図 2). これを防ぐにはユーザプログラムを作成するプログラマが適切な位置に MPI_wait を挿入する必要があるが, これは下記の理由で困難な仕事である.

- 1回目の MPI_Irecv を発行した後すぐに 2 回目の MPI_Irecv を発行しなければならないとき, 2 回目の MPI_Irecv の前に MPI_wait を発行しなければならないが, MPI_wait が早過ぎると受信側でブロックする.
- 通信が終了する時刻を決定するのは, ユーザプログラムで決まる通信データ量および計算量と計算システムで決まるデータ転送速度および計算速度が関係する.

データ転送速度および計算速度はユーザプログラムそのものには無関係であり, 通信が終了する時刻を決定することはユーザプログラムのプログラマに負わせるべきではない. とくにポータブルなプログラムを書く場合や複数のノードを使い通信データ量および計算量が動的に変化する時は通信が終了する時刻を決定するのは極めて困難である.

MPI_test でポーリングすることで, ある程度効率が良くポータブルなプログラムを作成することは可能ではあるが計算システムの制限でプログラマに余分な負荷をかけていることには変わりはない.

逐次処理における副作用のない関数はお互いの独立性が高く, 入出力が関数の起点と終点に限定されることの利点として, これまで多くの再利用可能なコードが作成され使われて来た. 効率の良い並列

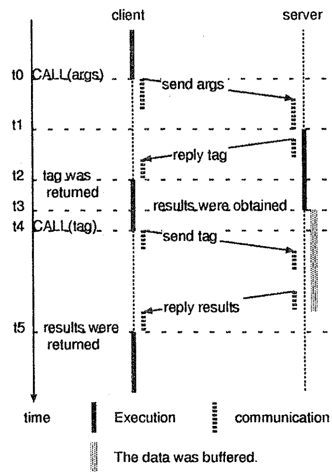

図 4: "Early reply" enables the execution from "tag return" to "tag call" while processing the remote procedure .

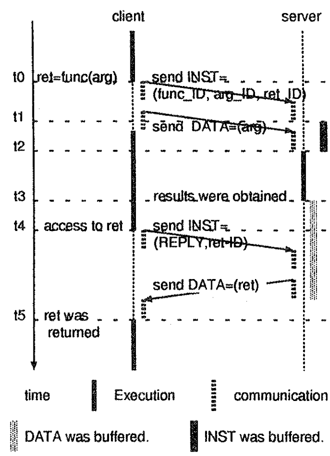

図 5: "Message-Pooling" enables the execution from "call" to "access to ret" while processing the remote procedure .

処理のコードをかくことはそれほど容易なことでなく, 同期の必要な理由を理解し, 対象とする計算システムに精通しかつプログラムの持つ意味だけでなくそのシステムでのコードの実際の動作を予測できることが必要とされる. 作成に時間がかかるならば再利用できるように作成すべきであるが MPI で効

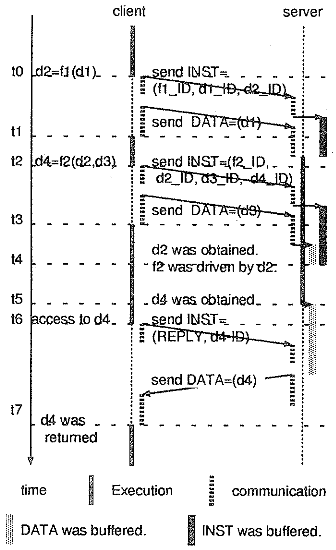

図 6: "Message-Pooling" enables reuse of data without return.

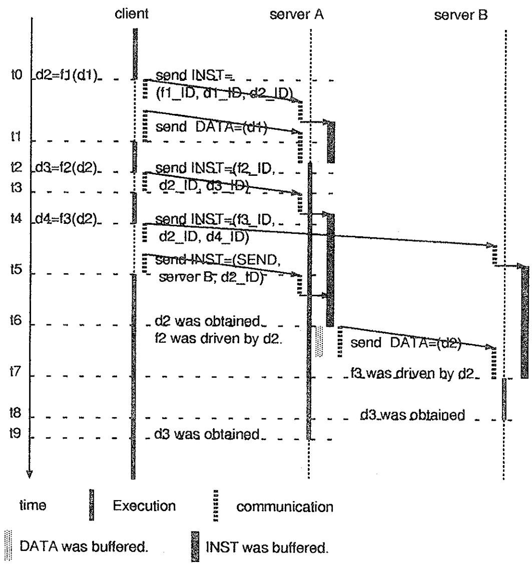

図 7: "Message-Pooling" enables sending data between servers.

率良く並列処理を実装するために，込み入った相互作用を各 MPI プロセスで行なっていると，その通信パターンが一致したプロセス同士でしか通信できないために，MPI プロセスはひとつの単位として再利用しにくい．

2.2 Remote Procedure Call の問題点

分散処理で関数と同様のインターフェースを持つ方式として Remote Procedure Call(RPC) [5, 6] が知られている．シンプルな RPC の場合 [5, 6] の時系列ダイアグラムは図 3 になり，関数を呼び出してから結果が返されるまで計算がブロックされる．この RPC では高性能なサーバを非力な複数のクライアントで利用する状況には適しているが，1 クライアントマルチサーバ型の PC クラスタによる高性能計算には適用出来ない．

クライアントでのブロックを短縮するため，RPC の変種で Early reply [6] と呼ばれる方法が使われるときがある (図 4)．Early reply では本来ひとつの関数の呼び出しを入力引数を関数にを与えるための呼び出しと関数から結果を回収するための呼び出しの二つに分離する．入力引数を与えるための呼び出しに対し，サーバが実際の計算をはじめるはじめる前に，即座に結果の ID 番号となる tag を返し，クライアントは tag が戻るとブロックを外す．その後，tag を指定して結果の回収のための呼び出しを行なう．これによりサーバの計算中もクライアントは計算をすることが可能になる．

Early reply は 1 クライアントマルチサーバの PC クラスタで利用できる可能性があるが次の問題がある．

- tag を reply する間はクライアントはブロックする．

コモディティハードウェアのみで構成された PC クラスタではノード間のレーテンシが大きいため，ブロックされていなければ実行できる計算量が無視できない．計算サーバ数が多くなるほどブロック時間の総和は大きくなるので台数効果を低下させる原因となる．

- 結果をクライアントに戻さずに，その結果を次の処理に入力できない．

サーバ で得られた結果を再び他の関数の入力として同じサーバで処理する時，データがクライアントとサーバ間を往復するために非効率である．

- 本来ひとつであるべき関数を実装の都合で二つの副作用のある関数で表現している.

そのためプログラムの可読性を低下させ, プログラマへの負荷を増加させている. 再利用のためのインターフェースとしてはひとつの副作用のない関数より扱いにくい.

2.3 メッセージプーリングによる解決策

提案するメッセージプーリング (*Message Pooling*) 方式の時系列ダイアグラムは図 5になる.

Message Pooling はデータおよびサーバへの命令 (サーバで起動する関数) の全てにシステム全体でユニークな ID 番号をクライアントがつける. この ID 番号はメモリでなくデータに付けられた ID 番号である. サーバでのリモート関数の起動順序はデータ/命令の ID 番号により決定され, クライアントでのリモート関数の発行順序を維持はしない.

このことにより, *Message Pooling* は次の特徴を有する.

- サーバからの ID 番号の応答をクライアントが待つ必要がない (図 5).

クライアントからリモートの関数を起動する時に, Early reply はサーバが tag を発行するため, サーバからの tag の応答をクライアントが待つ必要があるが, *Message Pooling* はクライアントがデータ/命令の ID 番号をつけるので tag 応答のための往復の通信レーテンシーがない.

- データは ID 番号で識別され, サーバ側で保存されるので, クライアント側にすぐ送り返す必要はない.

そのためサーバ側で計算された結果をクライアントに送り返さずに同じサーバで別の関数の入力に再利用できる (図 6). さらに 2 台のサーバ A, B があるとするとサーバ A で計算した結果 d2 がサーバ B での計算に必要だとすると サーバ A からサーバ B に直接転送することも可能である (図 7). その場合, クライアントが計算に必要な入力データの位置を元に, クライアントがサーバ A からサーバ B への転送命令を必要に応じて発行する.

- プログラマは通常のローカルで副作用のない関数と同じに扱える.

結果の回収は関数の発行とは別の時間に行なわれる点では Early reply と同じであるが, クライアントへの結果はデータの内容にクライアントがアクセスした時に自動的に回収されるので, 明示的にプログラマが結果の回収を指示する必要がない. データの ID 番号を指定してリモートの関数を発行している限り結果を回収せず, そのまま他の関数に入力として渡しても良い. C 言語のポインタとの類推で説明すると, *Message Pooling* において ID 番号で識別されたデータを引数に渡してリモートの関数を起動することと C 言語でポインタを引数に指定して関数を起動することに対応させると, 結果の回収は ポインタの参照はずし (dereference) のときに自動的に発行されることに相当する.

- 通信と計算の重ね合わせが可能な場所の指定なしに, 重ね合わせが計算サーバで実現されている.

それに対して MPI のノンブロッキング通信では通信と計算の重ね合わせを実現するためには重ね合わせ可能な場所を指定する必要がある.

現時点の実装ではクライアントはシングルスレッドで実装しているので通信と計算の重ね合わせは起こらない. しかしクライアント側でデータの送受信のバッファリングを行う実装も可能である. その場合, クライアントでのデータ内容のアクセスでデータ受信を駆動するのでなくバッファからのコピーを駆動し, データのクライアントへの送信を促す命令をあらかじめ発行することが考えられる.

3 メッセージプーリングシステムの機構

Message pooling を実現したシステム *Clop* (Cluster of Message Pooling) の構成を図 8 に示す. *Clop* はクライアント用のライブラリ (*Clop library*) と計算サーバから構成される.

Clop library は *Clop* で通信される全てのデータをネットワーク透過なバイトストリームに変換/復元する機能を実現し, その各データにユニークな ID 番号をつけそのデータの位置を管理する機能を持つ.

計算サーバは命令/データが到着すると命令バッファ/データバッファにプールし, 必要なデータが揃い次第命令に対応した関数を駆動する. 関数の駆動を起こすイベントは, 新データの到着, 一つの関数の計算終了, 新命令の到着である. 多くの命令とデータをプールし, データが揃い実行可能条件を満たし

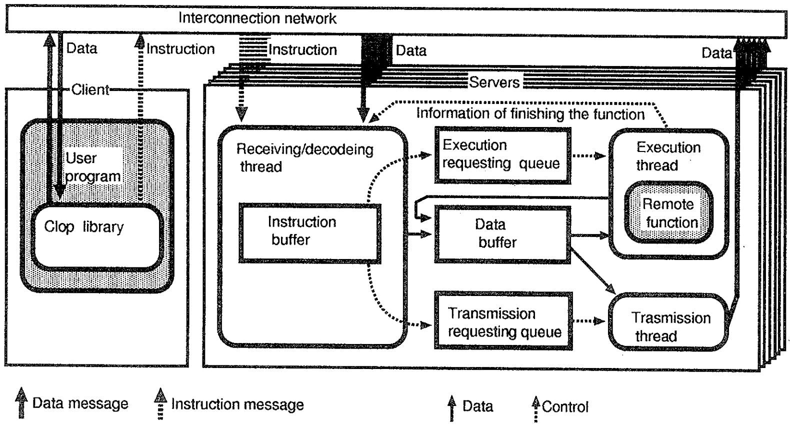

図 8: Clop system.

た命令が常に存在する状況にすることでプロセッサの使用率を上げることができる.

クライアントはリモート関数の呼び出しをサーバへの命令列にして各サーバに送る. データにより関数が駆動されることはデータフロー型計算システムと同じであるが, 関数とデータの依存関係はクライアントの逐次処理からのリモート関数の発行で生成された半順序であり, 繰り返しそのものを関数とデータの依存関係として記述することはない. データ ID はメモリに対してでなくデータに対してユニークに付けられるのでクライアントで $d = f(d)$ のように破壊代入をするように見える関数を実行したとしても, 入力引数の d と左辺の d には異なる ID 番号が付けられる. この意味で データ ID はグローバルポインタとは異なっている.

4 まとめ

1 クライアントマルチサーバ型の PC クラスタによる高性能計算用の並列処理方式としてメッセージプーリング方式を設計・提案した. MPI のノンブロッキング通信のように通信と計算の重ね合わせを指示せず, RPC のようにリモート関数呼び出し発行でブロッキングせずに, 自動的に通信と計算を重ね合わせ, 通信レーテンシを隠蔽をし, ユーザプログラムはクライアントの逐次処理プログラムとして記述し, クライアントで発行された多数のリモート関数を多数の計算サーバで並列処理することで速度を向上することがメッセージプーリングでは可能となることを議論した.

参考文献

[1] Message Passing Interface Forum:“MPI: A Message-Passing Interface Standard”, (1995)

[2] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI—The Complete Reference, Volume 1, The MPI Core second edition, The MIT Press (1998)

[3] Gropp, W., Lusk, E., Skjellum, A.: Using MPI—Portable Parallel Programming with the Message-Passing Interface, The MIT Press (1994)

[4] 建部 修見, 児玉 祐悦, 関口 智嗣, 山口 喜教: リモートメモリ書き込みを用いた MPI の効率的実装, 情報処理学会論文誌, Vol.40, No.5,pp.2246-2255 (1999)

[5] Birrell, A. D., Nelson, B. J. : Implementing Remote Procedure Calls, ACM Transactions on Computer Systems, Vol. 2, No. 1, pp.39-59, (1964)

[6] Wilbur, S., Bacarisse, B. : Building distributed systems with remote procedure call, IEE Software Journal, pp.148-159 (1987)

[7] 松村 博光, 大鎌 広, 藤原 祥隆 : 通信ブロックの軽減を考慮した大規模行列における分散処理システムの設計, 情報処理学会研究報告,HPC-70-4, pp.19-24(1998-3)

[8] 松村 博光, 大鎌 広, 藤原 祥隆 : メッセージキャッシング型 PC クラスタにおけるスレッドの効果, 情報処理学会研究報告,HPC-73-3, pp.13-18(1998-10)

[9] 小林 賢一, 大鎌 広, 藤原 祥隆 : PC クラスタ “Clop” におけるリアルタイムログシステムの設計, 情報処理学会研究報告,HPC-80-26, pp.149-154(2000-3)

[10] 石澤 祐介, 大鎌 広, 藤原 祥隆 : PC クラスタ “Clop” の LINPACK Benchmark による性能評価, JSPP2000 論文集ポスター論文, 印刷中 (2000-6)