ハイパフォーマンス
コンピューティング 74－9
（１９９８．１２．１１）

NT クラスタ環境におけるメッセージ通信とオブジェクト管理の実現

月川　淳　川瀬賢二　吉永　努
大津金光　馬場敬信

宇都宮大学 工学部 情報工学科

近年、WindowsNT によるクラスタ環境の普及は目覚しいものがある。本研究では、このNT クラスタ上に高性能な並列オブジェクト指向言語を提供する事を目的とする。性能向上のために、トランスポート層からその上位オブジェクト間のオーバーヘッドを低減したオブジェクトのプロセス内ロードモデルを提案する。さらに、NT の特徴を生かした実装を行う事により並列オブジェクト指向言語のメッセージを効率よく実現する。これらの工夫により、トランスポート層の持つ性能に対して 35%のオーバーヘッドで言語レベルのメッセージパッシング性能を達成した。

Implementation of Message Passing and Object Management for NT Clusters

Atsushi TSUKIKAWA, Kenji KAWASE, Tsutomo YOSHINAGA
Kanemitsu OOTSU and Takanobu BABA

Department of Information Science, Faculty of Engineering,
Utusnomiya University

Recently, there has been a remarkable increase in the popularity of the Windows NT cluster environment. The purpose of this research is to implement a high performance Parallel Object-Oriented Language system on an NT cluster. We propose an In-Process Object Loading Model which decreases the overhead between the transport layer and high-level user objects. In addition, we employ special characteristic of the NT system to implement an efficient messaging model for a Parallel Object-Oriented Language. Experimental results on the proposed system indicate that, at the language level, the system's message passing overhead is reduced to only 35% of that of the transport layer.

1. はじめに

近年、IBM PC/AT 互換アーキテクチャをもつパーソナルコンピュータ(PC)上で動作するWindowsNT(以下 NT)の普及には目覚しいものがあり、100Base/T クラスのネットワークインターフェースを持つ NT クラスタ環境が容易に利用可能になってきた。

クラスタ環境でのメッセージパッシングライブラリとしては MPI, PVM などが存在する。これらのライブラリを使用して並列プログラミングを行う場合、ユーザは常にメッセージの送受信を意識し、明示的に行う必要がある。

これに対し、我々の研究室では並列プログラミングを行う上で、言語レベルでのメッセージパッシングと並列性を持つ並列オブジェクト指向言語の

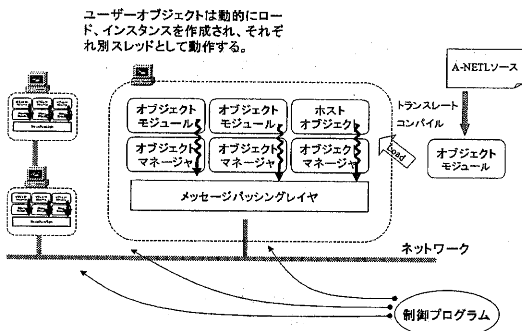

図1 システム概要

優位性を唱え、これまでに並列オブジェクト指向言語 A-NETL[1]を PVM を利用することによりワークステーションクラスタ上に実装[2]してきた。A-NETL は、中粒度の並列処理記述を目的とし、過去型、現在型、未来型のメッセージ送受信とともに、複数メッセージ受信、宣言型同期機構を持つ。

このような標準メッセージパッシング上に構築した言語処理系では、その処理系自体のポータビリティは保たれるものの、メッセージパッシングレイヤの持つ通信オーバーヘッドが問題となる。現在 NT 上で実装されている標準メッセージパッシングライブラリとして、MPICH/NT[3]、WMPI[4]、HPVM などがあるが、そのメッセージ遅延はトランスポート層として用いているソケットインターフェースの 3 倍から 40 倍にも及ぶ[5]。

本研究では、このようなトランスポート層からその上位オブジェクト間のオーバーヘッドを軽減するために、オブジェクトのプロセス内ロードモデルを提案する。また、NT の特徴を生かした実装を行う事により並列オブジェクト指向言語の過去・現在・未来型メッセージを効率よく実現する。

本稿では、現在広く利用可能な WinSock2 インターフェースをトランスポートレイヤとして用いた実装とその評価について述べる。

2. システム概要

図 1 にシステムの構成を示す。

- メッセージパッシングレイヤ
  システム内の各コンピュータ上で動作し、オブジェクトモジュールのロード、オブジェクトモジュール間のメッセージ転送の制御を行う。
- オブジェクトモジュール
  ユーザーが記述した A-NETL ソースプログラムは、トランスレート、コンパイルにより、ダイナミックリンクライブラリ(DLL)形式の実行ファイルであるオブジェクトモジュールに翻訳する。これには構造化例外処理(後述4.3)を利用したフューチャートラップ機構、メソッドの同期機構のコードが含まれる。
- オブジェクトマネージャ
  オブジェクトマネージャは DLL として作成した関数の集合体で、オブジェクトモジュール内メソッドのコンテキストチェンジや、リターンメッセージの処理が必要な現在・未来型メッセージの機能を提供する。
- ホストオブジェクト
  ユーザーインターフェースを持つ特殊なオブジェクトモジュールである。
- 制御プログラム
  メッセージパッシングレイヤに対して、オブジェクトのロード、アンロードなどのコマンドを発行する。

3. メッセージパッシングレイヤ

メッセージパッシングレイヤは、オブジェクトモジュール(以下オブジェクト)とメッセージトランスポートレイヤ間の通信オーバーヘッドを限界まで減らす方針で設計した。システム内での並列性を保つために、非同期・重複送受信を基本としている。送信中に生じたエラーについては、特殊なメッセージをオブジェクトに送ることにより通知する。オブジェクト間の同期通信は上位のオブジェクトマネージャで実現する。

3.1. プロセス内へのオブジェクトロード

メッセージパッシングレイヤとオブジェクトとのプロセス間通信によるオーバーヘッドを解消するために、オブジェクトをメッセージパッシングレイヤのプロセス空間内にロードし、それぞれを独立したスレッドとして実行する。NT の定義による「スレッド」とは、OS によりプリエンプティブにスケジューリングされる実行単位をさす。Solaris における Light Weight Process に等しい。

メッセージパッシングレイヤは、制御プログラムよりオブジェクトのロード要求を受け取ると LoadModule API によりオブジェクトを自プロセスのメモリ空間にマッピングする。その後、オブジェクトのメイン関数アドレスを取得しスレッドを作成する。

同種オブジェクトのインスタンスの生成は複数のスレッドを生成することにより行う。それぞれのインスタンスはシステム内でユニークなオブジェクト ID(ObjId)により識別される。

3.2. APCによるスレッド間通信

本システムは (a)送信スレッド (b)受信スレッド (c)複数のオブジェクトスレッド、により実行される。それらのスレッドはメッセージを受信するまではスリープ状態にあり無駄なCPU時間を消費しない。

高速なスレッド間通信を行うために、Asynchronous Procedure Call(APC)による非同期通知メカニズムを利用する。APC は、スレッドごとに作成されたAPCキューにデータをエントリすることにより、通知受付状態でスリープしているスレッドの任意のファンクションを起動する機構である。また、このキューはOSによりページアウトされないメモリ領域に確保されるため高速な動作が期待できる。

本システムでは、このAPC機構を用いてメッセージバッファのアドレスの受け渡しを行うことにより高速なスレッド間メッセージ通信を可能としている。

また、メッセージの不要なメモリコピーを極力避けるために、オブジェクトとメッセージパッシングレイヤではメッセージバッファを共用し、送信側でアロケートしたメッセージバッファは、送信終了後に送信スレッドで破棄する。同様に、メッセージ受信時に受信スレッドでアロケートしたバッファは、メッセージ受け手のオブジェクトがその利用を終了した時点で破棄する。

オブジェクト間のメッセージパッシングの手順を以下に示す。(図 2)

(1) 送信元オブジェクトは送信用バッファを確保し、送信データを格納する。

(2) 送信スレッドの APC キューにデータをエンキューする。

(3) サスペンド状態にあった送信スレッドが起動する。

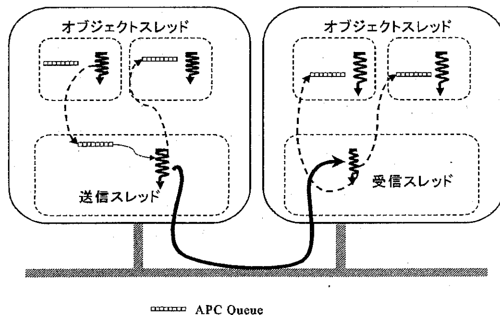

図 2 メッセージパッシング

(4) 送信先 ObjID が同一プロセス内に存在する場合、そのオブジェクトスレッドのAPCキューにデータをエンキューする。

(5) 送信先が他コンピュータの場合には、WinSock2 重複 I/O を用いて送信を行う。

(6) 送信中にエラーが生じた場合には、送信元オブジェクトに対してエラーメッセージを送信する。

(7) さらにエンキューされているデータがある場合には送信スレッド送信処理を行う。

(8) 送信スレッドは OS より送信の完了通知を受け取るとバッファ領域を開放する。

(9) 受信側コンピュータで待機中の受信スレッドが起動する。受信スレッドは送信されたデータ分のバッファを確保しデータを受信する。

(10) 受信スレッドは送信先オブジェクトのAPCキューにデータをエンキューする。

(11) 待機中の送信先オブジェクトのメソッドが起動される。データを受信したオブジェクトはバッファが不要になった時点でそれを開放する。

3.3. 複数ヒープによるスレッド並列性の確保

本システムのようにマルチスレッドで動作するプログラムでは、メモリ管理時に生じるシリアライズが、その並列性を損なわせる原因となる。これを解消するために、すべてのオブジェクトインスタンスに対して独立したヒープエリアを設けている。

独立したヒープはオブジェクトインスタンスが作成時に初期化され、そのハンドルはオブジェクトインスタンス固有データ(後述 4.2)に保存する。

3.4. グローバルイベントオブジェクト

オブジェクト間で効率的な同期をとるために、メッセージパッシングレイヤはグローバルイベントオブジェクトを装備している。グローバルイベントオブジェクトはトランスポートレイヤのブロードキャストメッセージによりセット／リセットが行われる。

4. オブジェクトマネージャ

4.1. ファイバによるコンテキスト切り替え

オブジェクトの持つ各メソッドは、そのコンテキストチェンジを高速に行うためにファイバを用いて実装する。ファイバとはユーザーレベルでスケジューリング可能な実行単位であり、SwitchToFiber API によりその制御を切り替える。

このAPIはスタック、レジスタ及び構造化例外フレームの待避、復元をユーザーモードで行うルーチンであり、インテル版の場合には約60マシン命令で実行される。また、ファイバコンテキストの保存には約200バイトを要する。

メソッド起動を高速に行うために、オブジェクトマネージャはメソッドに対応するファイバコンテキストをロード時に作成する。実行中のメソッドが多重起動された場合には、新たにファイバコンテキストを作成する。

オブジェクトマネージャは、一つのメッセージキューを持つ。メッセージパッシングレイヤより配送されたメッセージは、APC機構によりオブジェクトスレッドがスリープしている間に、このキューに入れる。メッセージキューとオブジェクトマネージャは、Win32同期機構のひとつであるイベントオブジェクトにより同期をとり、キューにメッセージが到着するとオブジェクトマネージャが即座に起動し、メッセージに対応するメソッドにコンテキストチェンジを行う。

メソッドが(a)終了、(b)現在型メッセージを送信、(c)未来型メッセージ送信後にフューチャートラップが発生、すると制御はオブジェクトマネージャに戻り、メッセージ待ちの状態でサスペンドする。

4.2. オブジェクトインスタンス固有データ

オブジェクトモジュール中のデータセクションは、すべてのオブジェクトインスタンスで共用されるために、インスタンス固有のデータの格納領域として利用できない。インスタンスデータ領域はオブジェクトの初期化時に確保し、そのアドレスをスレッドローカルストレージ[6]に保存することにより使用する。

4.3. 構造化例外処理によるフューチャートラップ

未来型メッセージを用いた場合、その返値を参照する以前に、それが利用可能になっているか検査する必要がある。これはフラグの参照で実行することができるが、返値が巨大なループ内などで何回も参照される場合にはそのオーバーヘッドが問題となる。

本システムでは、このオーバーヘッドを削減するために、Win32構造化例外処理機能(SEH)[6,7]を用いてフューチャートラップを実装する。SEHは、特定のブロックで生じたソフトウェア、ハードウェア例外を指定した例外ハンドラで処理する機能である。この方法では返値への検査を全く行わないため、返値が有効な場合には検査のオーバーヘッドを全く伴わない。

その実装方法を以下に示す。

(1) メソッドは起動時に、その例外ハンドラのアドレスをスレッド情報ブロックに設定する。インテル版の場合、スレッド情報ブロックはFSレジスタにそのアドレスが格納されている。

(2) 未来型メッセージを送信すると、返値を指すポインタを無効な値としておく。

(3) 返値を参照するとアクセス違反例外が発生する。例外が発生すると、CPUレベルでそれがトラップされ、OSから、先ほど設定した例外ハンドラに制御が移る。

(4) 例外ハンドラではメソッドをサスペンド状態とし、制御をオブジェクトマネージャに移す。

(5) オブジェクトマネージャは、リターンメッセージを受け取るとメソッドに制御を戻す。その後、例外ハンドラ内より実行は再開される。

(6) 例外ハンドラでは、アクセス違反が生じたレジスタを返値へのポインタで修正する。

(7) アクセス違反が生じた命令から再びメソッドが実行される。

なお、例外によるトラップと実行再開はマシンコードレベルで行われるため、フューチャートラップの実装にはインラインアセンブラにより直接レジスタを制御する。

5. 評価と考察

実装した本システムの評価を行うために、表1の環境において実験を行った。

表1 実験環境

PC	450MHz PentiumII Processor 256 MBytes Memory
OS	NT Workstation4.0 SP3
Network	100BaseT Switching HUB(Full Duplex) 100BaseT Non-Intelligent HUB(Full-Duplex)

5.1. 同一ホスト内での基本性能

この節では、同一ホスト内での基本性能を調べるための実験について述べる。測定にはすべてPentiumIIプロセッサのもつパフォーマンスカウンタを用いた。また一回の測定が同一のタイムスライス内で終了するように、Sleep APIによりタイムスライスを一旦放棄し、新しいタイムスライスが割り当てられた時点で測定を開始している

5.1.1. オブジェクト間のメッセージ遅延時間

同一ホスト上にロードされたオブジェクト間でのメッセージ通信時間は、約35μsecであった。これはメッセージサイズに関わらず一定である。またオブジェクト内でのメソッドコンテキスト切り

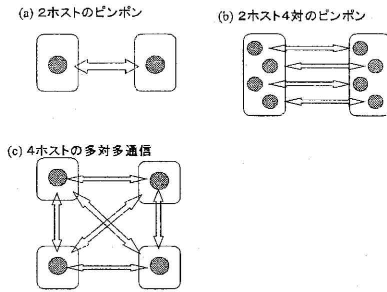

図3 実験で用いたオブジェクト配置

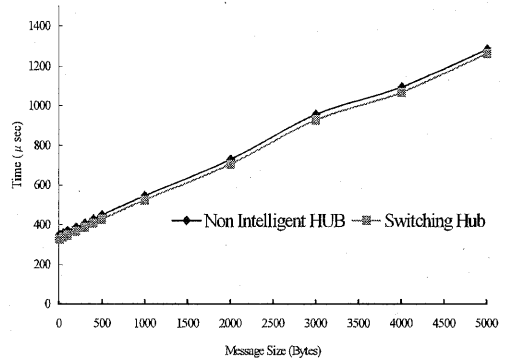

図4 ピンポン時間

替えにかかる時間は約 15μsec であった。C の標準ライブラリ関数である malloc 関数での呼び出しにかかる時間は約 14μsec であるので、これと同程度か 2 倍程度の時間でコンテキストチェンジとホスト内メッセージ転送実現していることがわかる。

5.1.2. フューチャトラップに要する時間

フューチャートラップにかかる時間として、フューチャ変数にアクセスによる例外発生から、例外ハンドラに制御が移るまでの時間を測定したところ、約 24μsec であった。これに対してフラグによる返値の検査に要する時間は数十 nsec である。メソッド中で大規模ループが存在するなどして、フューチャー変数への参照が 1000 回を越えるような場合には構造化例外処理によるフューチャートラップの効果が得られると考えられる。

5.1.3. 複数ヒープの効果

複数ヒープによる効果を調べるために、単独ヒープを用いたシステムとその実行時間を比較した。使用したアプリケーションは、リストに 100 万件の要素を追加・削除を行うオブジェクトで、これを同一ホスト上に 4 つのインスタンスを作成してその実行時間を調べた。表 2にその結果を示す。

表2 複数ヒープの効果(msec)

複数ヒープ	単独ヒープ
3595	4166

使用したアプリケーションはメモリ確保、開放を頻繁に行うものであるが、複数ヒープを用いることで約 15%の性能向上がなされている。

5.2. ホスト間通信における性能

複数のホストにまたがったオブジェクト間での通

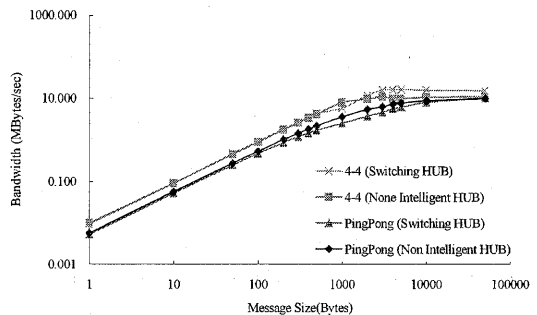

図5 2ホスト間での通信性能

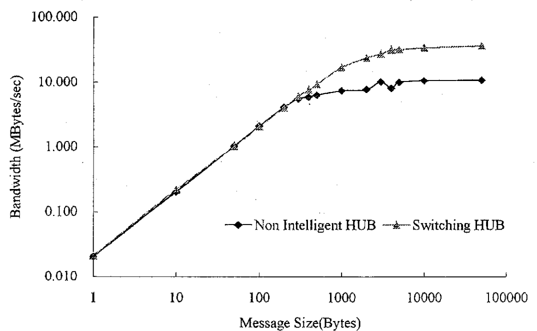

図6 4ホストでの多対多通信性能

信性能を評価するために、に示す３つの条件でメッセージサイズを変えながら測定を行った。(図 3)

(a) 2ホスト1対のピンポン時間とバンド幅(図 4,5)

(b) 2ホスト4対のピンポンによるバンド幅(図 5)

(c) 4ホストによる多対多通信でのバンド幅(図 6)

ネットワーク機器として Non Intelligent HUB と Switching HUB を用い、それぞれ測定した。

測定には GetTickCount API(精度 11msec) を使用し、各メッセージサイズに応じて 4 桁以上の有効数字が得られるまでの回数を繰り返し行い、全体の実行時間とメッセージ転送量よりバンド幅を算出した。

また、(a)本システム、(b)WinSock2 インターフェースによって記述したもの、(c)MPICH/NT[3]を使用したもの、におけるピンポン時間を同一条件で測定した結果を表 3に示す。WinSock2 による測定は、TCP_NODELAY オプションを指定したソケットを用いて、メッセージサイズが小さい場合にもバッファリングによる遅延が生じないようにしてある。

本システムのピンポンを定常的に繰り返した場合のオーバーヘッドは、ホスト間のトランスポートとして用いている WinSock2 に対して、100 バイトのデータ転送時に約 100μsec (片道で 50μsec) であった。これはソケットインターフェースの遅延時間の 35%である。メッセージサイズが小さくなるにつれてオーバーヘッドが大きくなっているのは、本システムが内部処理用に付加している 20 バイトのパケットヘッダの影響と思われる。

メッセージサイズ 10 バイトという小さいメッセージサイズでのバンド幅性能は、実験(1) で 0.052 MBytes/sec、実験(2)で 0.10MBytes/sec、実験(3)で 0.21 MBytes/sec となっており、 WinSock を直接使用して書いたものとほぼ同一の結果が得られている。

実験(3)における最大バンド幅は、Non Intelligent HUB の使用で 11.1MBytes/sec、Switching HUB の使用で 37.5MBytes/sec である。それぞれメッセージサイズが 2KByte、10KByte を越えた部分で最大バンド幅を示している。

表3 ピンポン時間の比較(μsec)

Message Size(Bytes)	1	100	500	1000
(a)本システム	349.5	369.5	448.6	544.8
(b)WinSock	197.8	269.0	388.4	514.8
(c)MPICH/NT	2256.0	2352.0	7551.0	12841.0

6. むすび

本稿では NT クラスタ上で既存のトランスポート上に並列オブジェクト指向言語を実装し、その評価を行った。プロセス内にオブジェクトのロードを行い、APC 機構を用いることで、オブジェクト間のメッセージ遅延をトランスポート層に対して 35%ほどのオーバーヘッドで実現した。また、複数ヒープの使用や、構造化例外処理によるフューチャートラップの有効性を示した。

今後の課題としては、メッセージサイズが小さい場合でのバンド幅の向上、アセンブラレベルでの実装の最適化などが挙げられる。前者は、パケットヘッダサイズの見直しによる改善を検討中である。さらに、A-NETL 特有の同期機構及び型なし変数の効率的な転送を実現するとともに、アプリケーションレベルでの評価を行っていく予定である。

謝辞

本研究は一部文部省科学研究費 基盤研究(C)課題番号 09680324、基盤研究(B) 課題番号 10558039、奨励研究(A) 課題番号 09780237 の援助による。

参考文献

[1] T. Baba, T. Yoshinaga, "A-NETL: A Language for Massively Parallel Object-Oriented Computing", Proc. Programming Models for Massively Parallel Computers(PMMP'95), pp.98-105, 1995.

[2] 斎藤宣人,古田貴寛,月川淳,馬場敬信,吉永努,並列オブジェクト指向トータルアーキテクチャ A-NET －WS クラスタへの A-NETL の実装－,情報処理学会第 52 回全国大会予稿集 2L-6,1996

[3]`http://www.erc.msstate.edu/mpi/mpiNT.html`

[4] `http://alentejo.dei.uc.pt/w32mpi/`

[5] M. A. Baker, MPI on NT: The Current Status and Performance of the Available Environments, EuroPVM/MPI98, Liverpool, UK, September, 1998,`http://www.sis.port.ac.uk/~mab/Papers/PVM-MPI-98/`

[6] Microsoft Corporation, "Platform, SDK and DDK Documentation", Microsoft Developer Network

[7] Matt Pietrek, "A Crash Course on the Depths of Win32 Structured Exception Handling", Microsoft System Journal,Vol12 No1,1997