LAPI を用いた IBM SP2 への分散共有メモリの実装と性能調査

下野 靖史 Bernady O. Apduhan 浅図 達也 有田五次郎
九州工業大学知能情報工学科
〒 820-8502 飯塚市川津 680-4

概要

IBM RS/6000 SP2 システムで利用可能な LAPI (Low-level Application Programming Interface) は、低いレベルで High-Performance Switch を利用した、最適な単方向データ転送を提供する API である。LAPI は基本的に、ライブラリの開発や可搬性よりも性能を重視したプログラマに利用されるよう設計されている。
本稿では、HP Switch の高い通信性能を利用する LAPI を用いた分散共有メモリ (DSM) の実装について述べる。その後に、性能調査の為に今回行なった予備実験の結果について報告する。

Implementing a Distributed Shared Memory on LAPI for IBM SP2 — Early Experiences

Yasushi Shimono Bernady O. Apduhan Tatsuya Asazu Itsujiro Arita
Dept. of Artificial Intelligence, Kyushu Institute of Technology
680-4 Kawazu, Iizuka 820-8502

Abstract

The Low-level Application Programming Interface (LAPI), available in IBM RS/6000 SP2 System, is a low-level, high-performance one sided communication API designed to provide optimal communication performance on the IBM HP Switch. LAPI is primarily designed for use by libraries and power programmers where performance is a priority than code portability.
This paper discusses the implementation of a distributed shared memory on top of LAPI to exploit the high communication performance of the HP Switch. We describe the implementation method and some promising preliminary experiment results.

1 はじめに

現在 LAN 環境を利用したクラスタコンピューティングの研究が盛んに行なわれている。我々の研究室においても、分散共有メモリモデルに基づく並列処理環境である分散スーパーコンピューティング環境 (Distributed Supercomputing Environment, DSE) が研究・開発されてきた [2]。一方、専用のネットワークをもつ並列処理環境としてクラスタマシンというものがあるが、その 1 つに IBM RS/6000 SP2 システムがある [1]。SP2 システムは、各ノードがローカルなメモリを持つスケーラブルな分散メモリ型の並列処理システムであり、各ノードは POWER2 RISC System/6000 プロセッサを搭載し、その OS は AIX である。またノード間は High-Performance Switch (HPS) と呼ばれる高速なネットワークスイッチで接続されており、これを利用することで効率的で高速なノード間通信を行なうことができる [8][11]。

SP2 システムにおいて HPS を効率的に利用できる通信ライブラリとして LAPI (Low-level Application Programming Interface) がある [4][5]。LAPI は細粒度の通信に適した設計がされており、これを分散共有メモリモデルに適用することで、細粒度の通信が頻発するような状態にも耐え得る並列処理環境が構築できると考えられる。今回はそのための基礎調査として、LAPI を用いて SP2 システム上に分散共有メモリを構築した。さらに、将来的に LAPI を用いて SP2 システム上に分散共有メモリ型の並列処理環境を実装するために必要な点についての考察を行なった。

以下本稿では、第 2 章で LAPI の概要について簡単に述べ、3 章で我々の提案する分散共有メモリモデルの実装について述べる。そして 4 章では、今回実装した分散共有メモリモデルの基本的な性能を実験により評価し、5 章において関連研究について述べる。最後に 6 章で、本稿についてのまとめと今後の課題について述べる。

2 LAPI の概要

LAPI は、HPS を用いて効率的で高速なノード間通信を提供するよう設計された API である。

LAPI はユーザに対し柔軟な並列プログラムの記述を提供するために、3 つの大きな特徴を持つ。まず第 1 に、LAPI は HPS 上で高効率な通信を提供する。例えば、LAPI では UNIX のシステムコールにみられるような高価な通信インターフェースによるオーバーヘッドを極力排除したため、細粒度の通信においても低いレイテンシを実現している。第 2 に、LAPI は共有メモリ型プログラミングモデルにおける load/store のような、リモートメモリに対する一方的で柔軟なアクセスを許す。これはメッセージパッシングモデルにおける send/receive よりも扱いが容易である。また LAPI は HPS 上で、標準的なメッセージパッシングモデルの API である MPI[6] や MPL[7] と比べて、よりプリミティブなインターフェースを提供する。第 3 に、LAPI はユーザにアクティブメッセージ [9] 形式のインターフェースを提供する。これは、他ノードにメッセージが到着した時に、その場でユーザが定義したハンドラを呼び出すことができるという機能である。これにより、様々なアプリケーションや環境において、ユーザは自分の望むようにアプリケーションの通信機能をカスタマイズすることができる。

以下本章では、LAPI のアクティブメッセージとリモートメモリコピー (Remote Memory Copy, RMC) について説明する。

2.1 アクティブメッセージ

LAPI におけるアクティブメッセージの実行イメージ図 1 に示す [5]。

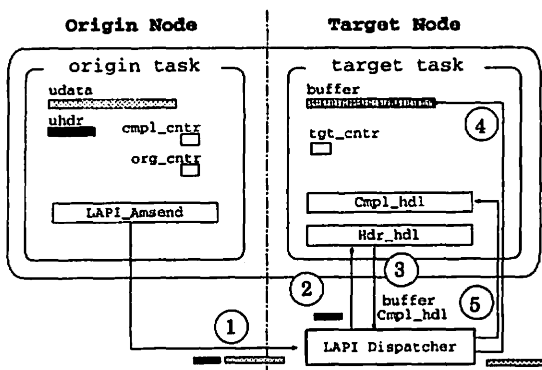

図 1: アクティブメッセージの実行イメージ

アクティブメッセージ処理の流れを図に従って説明する。アクティブメッセージの使用において、ユーザは二つのハンドラ — Header Handler と Completion Handler — を自由に定義することができる。あるプロセッサからアクティブメッセージの依頼があると、まずそのノード (Origin ノード) からユーザ定義のヘッダとデータが相手ノード (Target ノード) に送られる (ステップ 1)。ヘッダには処理すべきデータの他に相手ノードで呼び出されるハンドラの情報等が含まれる。Target ノードでは、送られて来たヘッダとデータを受け取った後に LAPI dispatcher が呼ばれ、ヘッダは Header Handler に渡される (ステップ 2)。Header Handler はヘッダをユーザ定義に従って処理すると、データを LAPI dispatcher を通して Completion Handler に渡す (ステップ 3,4)。最後に LAPI dispatcher によって Completion Handler が呼ばれ (ステップ 5)、アクティブメッセージ処理は終了する。

Origin ノードと Target ノード上の処理は非同期に行われるが、各々の終了を知りたいときは、LAPI に標準で用意されている各種カウンタを用いる。カウンタは、それぞれに対応する処理が終了した時点でインクリメントされるように設計されている。例えば図 1 において、org_cntr カウンタは Origin ノード上の処理 (ここではアクティブメッセージ処理) の終了を示し、tgt_cntr カウンタは Target ノード上の処理の終了を示し、cmpl_cntr カウンタは Completion Handler の終了を示す。特に cmpl_cntr カウンタは Origin ノードのプロセスから Completion Handler の終了を調べたいときに用いられ、これにより二つの処理の終了同期をとることが可能となる。

2.2 リモートメモリコピー

RMC では、Origin ノードが自ノード上の変数 と Target ノード上の変数を指定し、それに対して指定の処理を行う。処理の終了同期は、アクティブメッセージと同じく各種カウンタを調べることで可能となる。RMC には 2 種類の処理があり、ひとつは Target ノードにデータを送る PUT オペレーションであり、もうひとつは Target ノードからデータを取ってくる GET オペレーションである。PUT は、Origin ノードがデータを送信し資源の再利用が可能になると Target ノード上で受信データに対してなんらかの処理を行う前にすぐに次の処理に移ることができる。GET は、Target ノードから受信したデータが Origin ノード上の指定した変数上に完全に安定するまで次の処理を行うことができない。いわば PUT は非同期的処理であり、GET は同期的処理である。ただし、PUT オペレーションにおいて cmpl_cntr カウンタを参照すれば、処理の同期をとることが可能である。

その他の LAPI のプリミティブを表 1 に示す [3]。

3 分散共有メモリの実装

今回、分散共有メモリの実装を IBM RS/6000 SP Thin-66-2 上で行なった。以下に実装した DSM の構

表 1: LAPI functionalities

Operations	Functions
Initialize	LAPI_Init
Terminate	LAPI_Term
Active Message	LAPI_Amsend
Data Transfer	LAPI_Put
Data Transfer	LAPI_Get
Mutual Exclusion	LAPI_Rmw
Set the counter	LAPI_Setcntr
Wait the counter	LAPI_Waitcntr
Get the counter	LAPI_Getcntr
Ordering	LAPI_Fence, LAPI_Gfence
Address Exchange	LAPI_Address_init
Environment Query	LAPI_Qenv
Environment Setup	LAPI_senv

成とそのアクセス手法について説明する。

3.1 分散共有メモリの構成

DSM の構成について説明する。まずクラスタ内の利用される各ノード上に任意の量のメモリ空間を確保し、それらを仮想的な共有メモリと見なして利用する。つまり DSM のメモリ空間は、各ノードで確保されたメモリ空間の総和となる。DSM のアドレスは、LAPI のタスク ID (ノード番号に対応) とそのノード上に確保されたメモリ内アドレスを組み合わせて、2 次元的に表される。また DSM へのアクセスは、LAPI の PUT、GET オペレーションを用いることで、利用するどのノードからでも自由に行なうことができる。メモリへのアクセスについては次に詳しく述べる。

3.2 分散共有メモリへのアクセス

DSM へのアクセスは、**READ**、**WRITE**、**WRITE_B** の 3 種類のオペレーションを用意している。ただし、**WRITE** は non blocking write であり、**WRITE_B** は blocking write である。**WRITE_B** は、データが完全に書き込まれるのを待って処理を続けたいときに用いられる。

DSM へのアクセスのそれぞれの処理は、**READ** は GET オペレーション、**WRITE** は PUT オペレーションを用いて実現される。**WRITE_B** は、LAPI が提供するカウンタの一つである completion counter を利用した PUT オペレーションによって実現される。それぞれの処理は以下の様な形式で呼び出される。

```
void READ(node,mem_addr,data,length)
{
  if(メモリ保護違反発生){
    エラー処理;
    終了;
  }
  else{
    LAPI_Get();
    LAPI_Waitcntr(origin_cntr);
  }
}

void WRITE(node,mem_addr,data,length)
{
  if(メモリ保護違反発生){
    エラー処理;
    終了;
  }
  else{
    LAPI_Put();
    LAPI_Waitcntr(origin_cntr);
  }
}

void WRITE_B(node,mem_addr,data,length)
{
  if(メモリ保護違反発生){
    エラー処理;
    終了;
  }
  else{
    LAPI_Put();
    LAPI_Waitcntr(origin_cntr);
    LAPI_Waitcntr(completion_cntr);
  }
}
```

引数 node は LAPI で提供されるタスク ID と 1 対 1 対応しており、メモリが存在するノードを指定する。引数 mem_addr には指定するノード上に確保されたメモリ内のアドレスを指定する。引数 data は DSM に書き込むまたは DSM から読み出すデータを保持する変数であり、**READ** では DSM から読み出すデータを保持するための変数として使われる。また、引数 length は扱うデータの長さを Byte 単位で指定する。

なお **WRITE**、**WRITE_B** においても呼び出しの形式は同様であり、data が読み出しの為に使われるか書き込みの為に使われるかで異なる。またどの処理においても、カウンタは必要最小限のものだけを使用する。

次に、DSM へのアクセス手順について説明する。まず最初に、DSM として確保された各ノードのメモリの先頭アドレスを LAPI のプリミティブである LAPI_Address_init を用いて調べ、アドレスバッファに保存する。この情報は全ノードが持つことになるので、これにより、DSM を持つどのノードのメモリでも参照することが可能となる。次にこの情報と、

READ、**WRITE**、**WRITE_B** の呼出しで得られるノード番号、メモリ内アドレス、データ長の情報を用いて、PUT または GET オペレーションを呼び出す。このとき、ターゲットノード番号として node を、ターゲットアドレスとして アドレスバッファの情報と mem_addr を加えたものを、また扱うデータ長として length を、それぞれ LAPI のオペレーションに与える。これによりユーザの希望する処理を行なうことができる。処理要求が **WRITE_B** の場合は、LAPI の提供するカウンタである completion counter を利用する。つまり、書き込み処理が Target ノードで終了してから completion counter がインクリメントされるので、このカウンタがインクリメントされるのを待つことで **WRITE_B** の処理を実現することができる。

今回の実装では、DSM の同一アドレスに対して複数のアクセスが同時に行なわれた場合、競合解消は全て LAPI に依存している。しかしデータ入力の順番が LAPI による競合解消に影響を与えるので、ユーザは DSM にデータを格納する順番を意識してアプリケーションをプログラムしなければならない。

4 実験と評価

今回構築した DSM の性能を評価するために、TCP/IP と UDP/IP を用いて同様に DSM を構築し、それらの通信速度を比較する実験を行なった。実験内容は、**READ**、**WRITE**、**WRITE_B** をそれぞれ 1000 回試行し、その実行時間を比較した。その結果を以下に示す。

なお、本実験では IBM 社のクラスタマシンである RS/6000 SP Thin-66-2 を 2 ノード使用した。またネットワークには、RS/6000 に実装されている SP スイッチを使用した。

4.1 実験結果と考察

4.1.1 READ オペレーション

READ オペレーションにおいては、LAPI を用いた場合の処理速度は、TCP または UDP 用いた場合の処理速度よりも、データサイズが小さい場合は 8 倍程度、データサイズが大きい場合は 4 倍程度速かった。

READ オペレーションは、処理要求の送受信とそれに対するデータの送受信という 2 回の通信を行なわなければならない。そのため TCP や UDP ではメッセージの送受信毎に send と recv のような高価なシステムコールを呼ばなければならない。しかし LAPI は、カーネルを介すことなく他ノードのプロセスと通信を行なうことができる。また TCP や UDP では **READ** を呼び出す毎に処理要求の解析とデータ

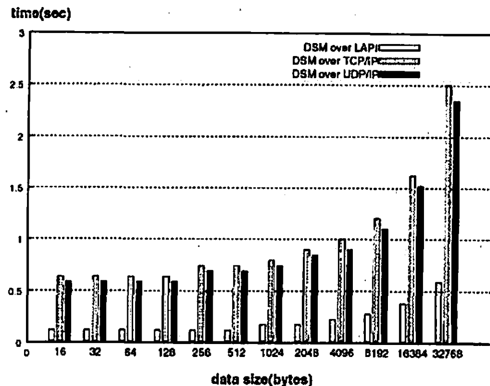

図 2: **READ** オペレーションにおける実験結果

の転送を行なわなければならないが、LAPI は 1 命令で処理の要求とデータの転送を行なうことができる。

さらに LAPI は SP スイッチ専用のライブラリなので、SP スイッチ上で非常に高速かつ効率的な通信を行なうことができる。

これらが、LAPI を用いた方が TCP や UDP を用いるよりも高速な処理を行なうことができた理由である。

4.1.2 WRITE オペレーション

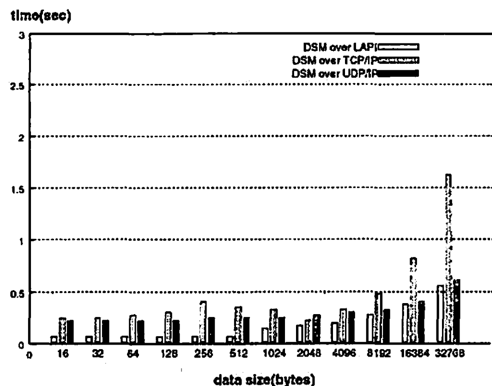

図 3: **WRITE** オペレーションにおける実験結果

WRITE オペレーションにおいては、LAPI を用いた場合と TCP または UDP を用いた場合とで、**READ** オペレーション程の処理速度の差は無かった。LAPI を用いた場合の処理速度は TCP や UDP と比べて 3 ～ 5 倍程度速かった。

WRITE オペレーションを連続して行なった場合、受信側のソケットバッファがあふれ、送信されたデータの一部が消失する可能性がある。その時 TCP では再送処理を行なうが、これが全体の処理に対して大きなオーバーヘッドとなる。実際データサイズが大きい

時、バッファあふれによるデータの消失が起こった可能性があり、処理時間が LAPI や UDP よりも 3 倍程度長くなっている。

またコネクションレスである UDP は、TCP のような通信の信頼性は無い。図 3 ではデータサイズが大きくなると LAPI を用いた場合と比べて遜色無い処理速度があるように見える。しかし実際は、受信側のソケットバッファの容量に限界があるため、受信側が送信された全てのデータを受け取っているとは限らない。この実験では、連続した書き込み要求によって受信バッファがあふれたため、多くのデータが消失した可能性が高い。よって、データサイズが大きい時の処理速度は LAPI を用いた場合と同程度であるが、データ消失率が高いので、信頼性のある通信とは言えない。これを解決するためにはデータの再送処理などを行ない通信に信頼性を持たせる必要があるが、そのためにパフォーマンスが低下することは避けられない。

LAPI は通信に信頼性があり、かつ高速なデータ転送が可能である。TCP の再送処理のオーバーヘッドや UDP の通信におけるデータ消失率を考慮すると、本実験の環境では LAPI を用いる方が有利であると言える。

4.1.3 WRITE_B オペレーション

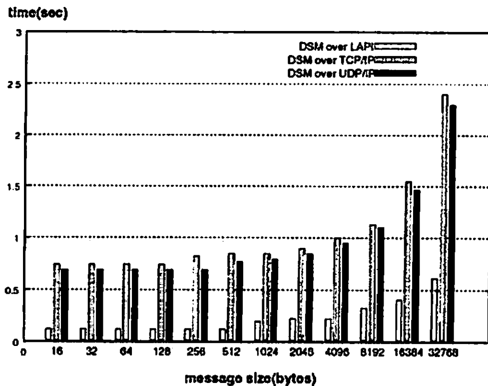

図 4: **WRITE_B** オペレーションにおける実験結果

WRITE_B オペレーションは、**READ** オペレーションのように処理要求の送受信とデータの送受信という 2 回の通信を行なう。処理速度の差は、TCP や UDP を用いた場合よりも LAPI を用いた場合の方が、データサイズが小さい場合は 7 倍程度、データサイズが大きい場合は 4 倍程度速かった。

これは **READ** の場合と同様に、システムコールの回数やメッセージ解析の処理などが原因と考えられる。

また TCP や UDP ではデータの書き込み確認処理は確認メッセージの送受信とその解析により行なうが、LAPI を用いた場合はカウンタ (completion counter) を利用することで簡単に実現することができる。

4.2 評価

本実験の環境では、LAN において標準的に使用される TCP や UDP を用いて DSM を実装するよりも、今回実装した DSM の方がどのオペレーションにおいても処理速度が速いことがわかった。

この理由としては、第 1 に使用したネットワークにある。今回使用したネットワークは SP スイッチであり、LAPI は SP スイッチ専用のライブラリなので、TCP や UDP よりも効率良く処理が行うことができるからである。

第 2 に、TCP や UDP では通信処理とメモリアクセスの処理は別々に行なわなければならなが、LAPI はその 2 つの処理を 1 命令で行なうことができる。よって LAPI の方が効率のよい分散共有メモリへのアクセス処理を実現することができる。また **WRITE_B** オペレーションにおける書き込み確認処理も、TCP や UDP ではメッセージの送受信とその解析を行なわなければならないが、LAPI を用いるとカウンタを利用するだけで簡単に実現できるという利点がある。

第 3 に、ソケットを使用する場合は send や recv などの高価なシステムコールを使用するので、カーネルを介すこと無く通信を行なうことができる LAPI の方が、本実験の環境では有利だからである。

これらから、LAPI は SP スイッチ上で、TCP の様に通信の信頼性が高く、かつ UDP のように高速な通信を行なうことができることが分かった。さらに PUT、GET オペレーションを用いて、リモートなメモリへのアクセスを簡単に行なうことができるので、今回構築した DSM へのアクセス機能は比較的容易に実装することができた。

5 関連研究

現在、SP2 システム上で LAPI の性能を利用した研究がいくつか行なわれている。

Pacific Northwest National Laboratory (PNNL) では Global Array(GA)[5] のパフォーマンスを最適化するために LAPI を用いた GA の実装を行なっている。GA とは、科学技術計算アプリケーションの並列化による高速化を目的として開発された、可搬性のある分散共有メモリ型の並列プログラミングモデルである。

また、オハイオ州立大学では MPI を LAPI を用いて実装する研究が行なわれている [10]。SP2 システム上の MPI では HPS を利用した高速な通信が可能だが、通信のときにユーザインターフェース部分と HPS の間で余分なバッファコピーが起こる。このユーザインターフェースと HPS の間を LAPI を用いて実装することで、バッファコピーのオーバーヘッ

ドを回避している。どちらの研究においても、LAPIの高い通信能力が示されている。

今後我々は、我々の研究室で研究、開発されている分散並列処理環境であるDSE[2] をLAPIの上に実装する研究を行なう予定である。

DSEは分散共有メモリ型の並列プログラミング環境であり、可搬性を考慮してマルチプラットフォームで実装されている。GAとは異なり、並列処理を行なう場合の問題点を調べたり、並列処理動作をモニタリングすることによって問題の分析に必要なデータの収集を行なうことができる。

現在DSEはLAN上にTCP/IPを用いて実装されている。しかしノード間で通信するたびにsendやrecvなどの高価なシステムコールを使用するため、これらが処理のネックになっている。これを解決するために、DSEの並列処理機能をLAPIを用いて実現する予定である。

例えば、DSEでは複数のノードでプロセスを並列に実行させることで並列処理を行なうが、この機能をLAPIのアクティブメッセージの機能を利用することで実現することができる。また分散共有メモリの実現とそのアクセス機能は、今回実装したDSMを利用して実現することができる。

6 まとめと今後の課題

本稿では、将来的にクラスタマシン上に分散共有メモリ型並列処理環境を実装するための予備調査として、LAPIを用いてSP2システム上へのDSMの実装を行なった。そして実装したDSMの性能を調査するために、TCPとUDPを用いて同様に実装を行ない、それらの通信速度を比較する実験を行なった。その結果、LAPIを用いて実装したDSMの方がTCPやUDPを用いたものよりも通信性能において優れていることが分かった。また今回は、LAPIを利用する上での制約や、SPスイッチ上でのLAPIの高い通信性能を確認することもできた。

今回の実験結果から、SPスイッチ用にLAPIを用いてDSEを実装すると、ノード間通信のオーバーヘッドが削減されてDSEの処理能力が向上する見込みが得られた。今後は、今回の実験を基にDSEを構築し、DSEの並列処理能力の向上を目指す予定である。またその他に、DSEの基本機能である可変ネットワークトポロジーの実現や並列処理動作のモニタリング機能なども実現しなければならない。さらにその後には、不規則な通信パターンを持つアプリケーションを作成して再度システムの評価を行なう必要があると思われる。

参考文献

[1] T. Agerwala, J.L. Martin, J.H. Mirza, D.C. Sadler, D.M. Dias, M. Snir, SP2 System Architecture, *IBM Systems Journal*, 34(2), pp.152-184, 1995.

[2] Tatsuya Asazu, Bernady O. Apduhan, Itsujiro Arita, Towards a Portable Cluster Computing Environment Supporting Single System Image, In *Proc. ICPP'99-MMNS Workshop*, pp. Sept. 1999.

[3] IBM.*PSSP for AIX : Command and Technical References*, rel 2.4, document GC23-3900-05, IBM Corporation, 1998.

[4] IBM. *PSSP for AIX : Administration Guide: The Communications Low-Level Application Programming Interface*, rel 2.4, document GC23-3897-05, IBM Corporation, 1998.

[5] Gautam Shah, et al, Performance and Experience with LAPI – A New High-Performance Communication Library for the IBM RS/6000 SP, In *Proc. of the International Parallel Processing Symposium*, pp. 260-267, March 1998.

[6] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, March 1994.

[7] M. Snir, P. Hochschild, D.D. Fryre, K.J. Gildea, The Communication Software and Parallel Environment of the IBM SP2, *IBM Systems Journal*, 34(2), pp.205-221, 1995.

[8] C.B. Stunkel, et al, The SP2 High-Performance Switch, *IBM Systems Journal*, 34(2), pp. 185-204, 1995.

[9] T. von Eiken, D.E. Culler, S.C. Goldstein, K.E. Schauser, Active Messages: A Mechanism for Integrated Communication and Computation, In *Proc. International Symposium on Computer Architecture*, pp. 256-266, 1992.

[10] Mohammad Banikazemi, Rama K. Govindaraju, Robert Blackmore and Dhabaleswar K. Panda, Implementing Efficient MPI on LAPI for IBM RS/6000 SP System: Experiences and Performance Evaluation, In *Proc. International Parallel Processing Symposium*, pp.183-190, 1998.

[11] Jose Miguel, Agustin Arruabarrena, Ramon Beivide and Jose Angel Gregorio, Assessing the Performance of the New IBM SP2 Communication Subsystem, *IEEE Parallel & Distributed Technology*, pp. 12-22, Winter 1996.