Rendimiento del paso de mensajes contiguos y no contiguos en Java usando MPI

Eric Gamess[(*,+)] y *Emilio Hernández*[(∇)]

[(*)] Universidad del Valle
Departamento de Ciencias de la Computación
Cali, Colombia

[(+)] Universidad Central de Venezuela
Postgrado en Computación
Caracas, Venezuela
E–mail: egamess@kanaima.ciens.ucv.ve

[(∇)] Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Caracas, Venezuela
E–mail: emilio@ldc.usb.ve

Resumen

El éxito creciente de Java hace surgir la posibilidad de una nueva alternativa a Fortran y C como lenguaje de alto desempeño. Java es un lenguaje que permite crear aplicaciones orientadas a objetos, seguras, portátiles, robustas y multihilos. Sin embargo, Java es también conocido por tener un bajo rendimiento debido a que es un lenguaje interpretado o ejecutado con técnicas de compilación de tipo "just in time", que no han alcanzado el nivel de optimización de las técnicas más tradicionales. El tiempo de ejecución de un programa paralelo depende del rendimiento en cómputo y comunicaciones. En este trabajo se estudia el rendimiento de Java en las comunicaciones, con énfasis en el paso de mensajes contiguos y no contiguos utilizando MPI.

Palabras claves: Computación de alto desempeño, Java, Message Passing Interface (MPI), mpiJava, Benchmarks, Parkbench.

1 Introducción

Por ser simple e independiente de la plataforma, Java es un lenguaje natural para desarrollar aplicaciones distribuidas y paralelas. Las distribuciones de Java vienen normalmente con dos paquetes de comunicación. El primer paquete es una adaptación de los sockets de la Universidad de Berkeley y el segundo llamado RMI (Remote Method Invocation) es una adaptación de los RPC (Remote Procedure Calls) de SUN Microsystem a Java introducido en JDK 1.1. Estos dos paquetes están muy bien adaptados para desarrollar aplicaciones cliente/servidor pero no son muy usados en el mundo de la computación paralela, donde los programas utilizan en su mayoría comunicaciones simétricas. PVM [1] y MPI [2] son bibliotecas de paso de mensajes que han tenido mucho éxito en el desarrollo de aplicaciones paralelas. Estas bibliotecas son de uso sencillo y soportan el modelo SPMD (Single Program Multiple Data) donde un grupo de procesos coopera en la resolución de una tarea ejecutando el mismo programa sobre datos diferentes. MPI ha sido normalizado por el "Message Passing Interface Forum" y cuenta con varias versiones de

dominio público de muy buena calidad, teniendo como consecuencia que MPI es la biblioteca de paso de mensajes más usada. MPI 1.1 cuenta con un conjunto de más de cien funciones, que incluyen funciones de comunicación punto a punto ejecutables en varios modos, bloqueantes o no bloqueantes. Otras funciones permiten operaciones colectivas como "broadcast", "barrier" y reducciones. El estándar 1.1 de MPI tiene soporte para C y Fortran 77. El estándar actual, MPI 2, cuenta con la creación dinámica de procesos, un conjunto de funciones para I/O en paralelo, el acceso a la memoria de un proceso remoto y tiene soporte para C++. En la actualidad, no hay versiones de MPI para Java propuestas por SUN Microsystem o normalizadas por el "Message Passing Interface Forum".

mpiJava [3] es un conjunto de clases Java que permite llamar a una implementación nativa de MPI desde Java. mpiJava implementa una interfaz para la mayoría de las funciones de MPI 1.1 y en su versión actual permite transferir arreglos de elementos primitivos. El paquete es de dominio público y se puede obtener por Internet [4]. Aunque existen otras interfaces de MPI y PVM para Java (ver sección 4), elegimos mpiJava por la disponibilidad de su código fuente, por el hecho de que el diseño de su interfaz se inspira en el estándar de MPI para C++ y porque su implementación a través de JNI (Java Native Interface) es limpia y eficiente.

El presente artículo se estructura como sigue. En la sección 2 hacemos una introducción a algunos benchmarks conocidos, utilizados como base para ver el impacto de Java sobre el paso de mensajes contiguos y no contiguos. En la sección 3 explicamos detalladamente los experimentos realizados, damos los resultados y discutimos sobre ellos. La sección 4 describe brevemente otras interfaces de MPI y PVM para Java. Terminamos el artículo con las conclusiones y los trabajos futuros que planeamos realizar.

2 Benchmarks para probar enlaces de comunicación

Parkbench [5] es un conjunto de benchmarks para plataformas paralelas diseñado gracias a un esfuerzo combinado de algunos investigadores y de varios fabricantes importantes de computadoras paralelas. En la distribución de los benchmarks de Parkbench [6], se encuentran algunos benchmarks de bajo nivel entre los cuales están **comms1** y **comms2**.

El benchmark **comms1** permite medir parámetros de una red de computadoras realizando un "pingpong" entre dos nodos. Un mensaje de tamaño variable es enviado por el proceso maestro al proceso esclavo e inmediatamente regresado por el esclavo. La mitad del tiempo del intercambio es registrado como el tiempo para enviar un mensaje de un nodo al otro. Al variar el tamaño del mensaje, se obtienen diferentes puntos (n, t) en el plano que son ajustados por mínimos cuadrados por la recta de la fórmula 1. Así, **comms1** calcula r_∞ y $n_{1/2}$.

$$t = \frac{n + n_{1/2}}{r_\infty} \tag{1}$$

En la fórmula 1, r_∞ representa la tasa de transferencia asintótica y $n_{1/2}$ es el tamaño del mensaje para el cual se obtiene una tasa de transferencia igual a $r_\infty/2$.

El benchmark **comms2** también involucra un par de procesos y permite medir parámetros de una red de computadoras como r_∞ y $n_{1/2}$ para enlaces "full duplex". Es similar a **comms1** con la diferencia que aprovecha los enlaces full duplex en caso de que la red disponga de ellos.

Cada nodo envía un mensaje al otro y recibe el mensaje que le fue enviado. Esta operación se realiza dos veces. Un cuarto del tiempo del intercambio es registrado como el tiempo para enviar un mensaje de un nodo al otro. Al variar el tamaño del mensaje, se obtienen diferentes puntos en el plano que son ajustados por mínimos cuadrados por la recta de la fórmula 1.

3 Descripción de los experimentos

Los experimentos descritos a continuación fueron realizados sobre un cluster de Pentium I, 133 MHz con 32 MB de RAM. Los equipos estaban interconectados por una red Myrinet [7] y corrían RedHat 5.2[1] como sistema operativo. Se utilizó la versión de JDK 1.1.7[2] modificada para Linux, mpiJava 1.1 y g77 versión 1.0.3 de la Cygnus[3].

La red Myrinet es una red de área local de alto rendimiento desarrollada por Myricom[4] que permite conectar un cluster de estaciones de trabajo o de PC. Ella se caracteriza por enlaces full duplex de 1.28+1.28 Gigabit/s, un tiempo de latencia corto y permite interconectar miles de computadores. Myrinet se hizo realidad gracias a (1) canales de comunicaciones robustos con control de flujo, (2) switches y (3) tarjetas de comunicaciones programables que dinámicamente establecen el mapa de la red, seleccionan la ruta y controlan el tráfico de los paquetes.

Se utilizó como implementación nativa de MPI la versión de MPICH [8] modificada por Myricom con el fin de aprovechar al máximo la red Myrinet. Esta implementación está desarrollada por encima de GM, un sistema de paso de mensajes de Myricom para redes Myrinet. El objetivo de GM es proveer un sistema de poco sobrepeso con un ancho de banda grande.

3.1 Experimentos para paso de mensajes contiguos

Los códigos fuente de **comms1** y **comms2** se encuentran en la distribución de Parkbench para Fortran 77 utilizando MPI o PVM. Escribimos versiones de **comms1** y **comms2** utilizando Java y mpiJava con el fin de poder medir el impacto producido por este ambiente sobre el paso de mensajes contiguos.

Se corrieron los siguientes programas:

- la versión de **comms1** de Parkbench para Fortran 77 y MPI
- la versión de **comms1** desarrollada para Java y mpiJava
- la versión de **comms2** de Parkbench para Fortran 77 y MPI
- la versión de **comms2** desarrollada para Java y mpiJava

La figura 1 muestra los resultados de **comms1** y **comms2** de Parkbench para Fortran 77 y MPI. Los resultados para los benchmarks desarrollados en Java y mpiJava son casi exactamente los mismos, lo que muestra un sobrepeso muy pequeño de mpiJava en el paso de mensajes, con respecto a su implementación más eficiente. La razón reside en que mpiJava está implementado

[1] http://www.redhat.com
[2] http://www.blackdown.org
[3] http://www.cygnus.com
[4] http://www.myri.com

sobre una biblioteca nativa, usando el JNI y no se realiza ninguna copia adicional del mensaje entre el área de memoria de Java y el área de memoria que manipulan las funciones nativas.

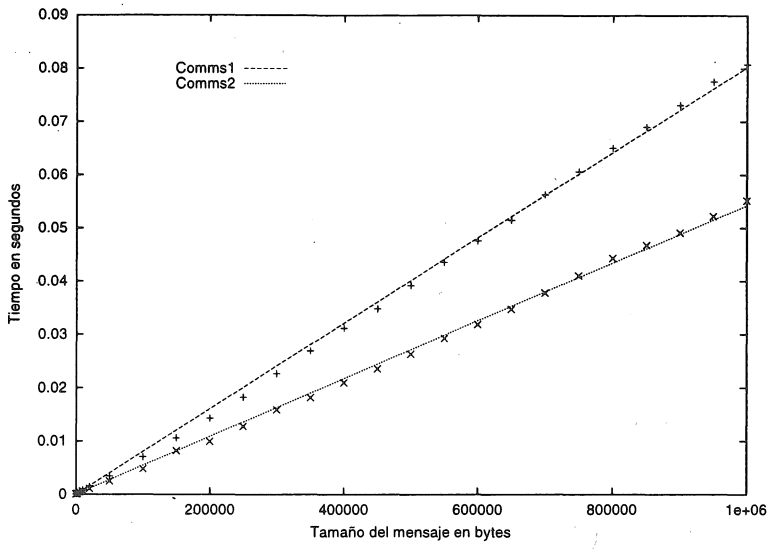

Figura 1: Gráfica de comms1 y comms2

La tabla 1 nos da los valores de r_∞, $n_{1/2}$ y del "Startup time" medido como proyección de la recta definida por r_∞ y $n_{1/2}$ sobre el eje de las ordenadas. Estos valores muestran que el valor de r_∞ es el mismo en **comms1** como en **comms2** para Fortran 77 y Java. Se puede observar que el valor del "Startup time" es mayor en el caso de los experimentos de Java y se debe al tiempo adicional que introduce la llamada nativa. Esto significa que el comportamiento de programas que realizan comunicaciones de grano fino (muchos mensajes de tamaño pequeño) puede verse afectado en una implementación en Java con mpiJava. Sin embargo, un programa que realiza comunicación de grano grueso (muchos mensajes de tamaño grande) no debe tener un rendimiento en comunicaciones significativamente inferior al mostrado por su similar en Fortran 77 y MPI.

Experimentos	r_∞ (MByte/s)	$n_{1/2}$ (Byte)	Startup time (μs)
comms1 (g77)	12.48	559.21	44.81
comms1 (java)	12.47	2742.23	219.91
comms2 (g77)	18.31	1138.45	62.17
comms2 (java)	18.29	7147.23	390.77

Tabla 1: Resultados de comms1 y comms2

El valor de r_∞ para **comms2** es superior al valor de r_∞ para **comms1**, lo que muestra que la red Myrinet es full duplex. Los valores obtenidos están muy lejos del "peak performance" anunciado por Myricom, lo cual puede tener diversas causas relacionadas con el protocolo de comunicación implementado por GM, el software de bajo nivel suministrado por Myricom.

3.2 Experimentos para paso de mensajes no contiguos

La mayoría de los benchmarks evalúan la tasa de transferencia entre nodos para datos contiguos, pero eso no refleja siempre el tipo de comunicación necesario en las aplicaciones científicas. Por ejemplo, es frecuente tener que pasar una o varias filas de una matriz almacenada por columnas.

MPI provee un conjunto de funciones para crear tipos derivados en tiempo de ejecución. La función MPI_Type_struct permite crear nuevos tipos donde los datos son contiguos pero de diferentes tipos y la función MPI_Type_vector permite crear nuevos tipos donde los datos son del mismo tipo pero no contiguos. El programador crea un nuevo tipo de datos especificando la forma y el tamaño de los datos a transferir antes de utilizarlo en operaciones de envío o recepción. El prototipo de la función MPI_Type_vector es:

```
MPI_Type_vector(count, blocklen, stride, old_type, newtype)
```

Esta función permite construir un nuevo tipo de datos que consiste en "count" bloques, donde cada bloque está compuesto por "stride" elementos del tipo básico, de los cuales se requieren solamente "blocklen" elementos. La figura 2 permite aclarar el concepto.

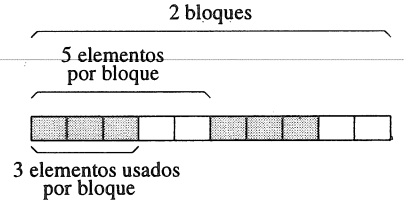

Figura 2: Uso de MPI_Type_vector (count=2, blocklen=3, stride=5)

Se desarrollaron programas (**commByte** y **commDouble**), tanto en Fortran 77 y MPI como en Java y mpiJava, inspirados en **comms1** de Parkbench para hacer experimentos sobre el paso de mensajes no contiguos. Por medio de dos parámetros (**smMaster** en el caso del maestro y **smSlave** en el caso del esclavo) se puede especificar el modo de envío/recepción de los datos. El modo CONTIGUOUS se utiliza para especificar que se requiere enviar datos que se encuentran en forma contigua, STRIDE para datos no contiguos que son enviados creando un nuevo tipo con la función MPI_Type_vector y LOOP para datos no contiguos que son copiados en forma contigua en un buffer antes de ser enviados. Gracias a estos parámetros se pueden realizar diversos experimentos como:

- Datos no contiguos en el maestro ni en el esclavo y los dos trabajan en modo STRIDE. El esquema del código se encuentra en la figura a continuación.

```
/* Maestro */                    /* Esclavo */

MPI_TYPE_VECTOR( ... )           MPI_TYPE_VECTOR( ... )
...                              ...
MPI_SEND( ... )                  MPI_RECV( ... )
MPI_RECV( ... )                  MPI_SEND( ... )
```

- Datos no contiguos en el maestro ni en el esclavo y los dos trabajan en modo LOOP. El esquema del código se encuentra en la figura a continuación.

```
/* Maestro */

DO I=1, N
    BUFFER(I)=MATRIZ(fila, I)
ENDDO
MPI_SEND( ... )
MPI_RECV( ... )
DO I=1, N
    MATRIZ(fila, I)=BUFFER(I)
ENDDO
```

```
/* Esclavo */

MPI_RECV( ... )
DO I=1, N
    MATRIZ(fila, I)=BUFFER(I)
ENDDO
DO I=1, N
    BUFFER(I)=MATRIZ(fila, I)
ENDDO
MPI_SEND( ... )
```

- Datos contiguos en el maestro y en el esclavo y los dos trabajan en modo CONTIGUOUS. El esquema del código se encuentra en la figura a continuación.

```
/* Maestro */

MPI_SEND( ... )
MPI_RECV( ... )
```

```
/* Esclavo */

MPI_RECV( ... )
MPI_SEND( ... )
```

La figura 3 muestra los resultados de la adecuación del modelo $(r_\infty, n_{1/2})$ a diferentes casos de estudio del paso de mensajes contiguos y no contiguos. Se realizaron los siguientes experimentos con el programa **commByte**:

- El maestro y el esclavo envían mensajes contiguos formados por bytes y trabajan en modo CONTIGUOUS. Los resultados obtenidos con Fortran 77 y Java son muy similares y están representados por la recta de leyenda "Contiguous (Fortran 77 & Java)"
- El maestro y el esclavo envían mensajes no contiguos formados por bytes y trabajan en modo STRIDE con "stride=64" y "blocklen=1". Los resultados obtenidos con Fortran 77 y Java son muy similares y están representados por la recta de leyenda "Stride (Fortran 77 & Java)"
- El maestro y el esclavo envían mensajes no contiguos formados por bytes y trabajan en modo LOOP con "stride=64" y "blocklen=1". Los resultados obtenidos con Fortran 77 están representados por la recta de leyenda "Loop (Fortran 77)" y los de Java por la recta de leyenda "Loop (Java)"

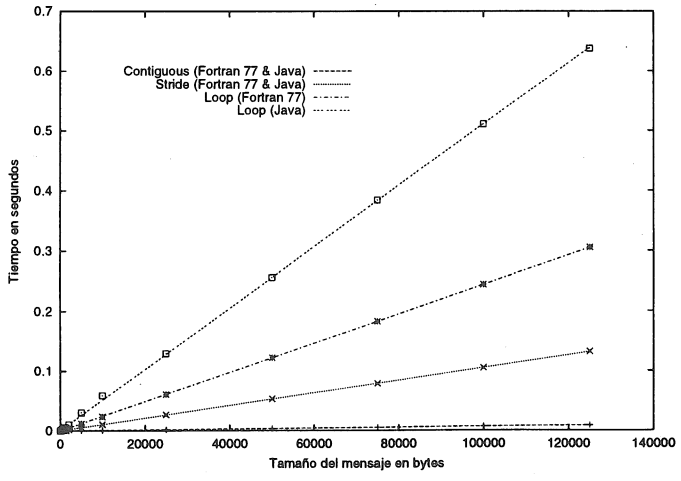

Figura 3: Gráfica de commByte

El stride utilizado en estos experimentos (64 bytes) es un valor suficiente para cargar una línea de cache diferente en la memoria cache para cada copia en esta arquitectura.

Se realizaron experimentos similares a los anteriores con el programa **commDouble** con la diferencia que los datos enviados eran reales en precisión doble (8 bytes) en lugar de ser bytes. En este caso "stride=8", de modo que el byte inicial de los reales de doble precisión a enviar coinciden posicionalmente con los bytes enviados en los experimentos de transmisión de bytes (stride=64). Los resultados obtenidos con Fortran 77 y Java son representados en la figura 4.

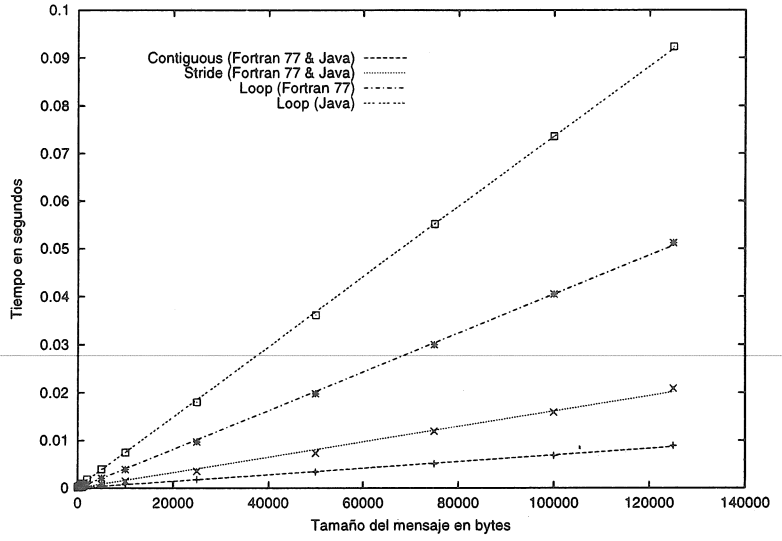

Figura 4: Gráfica de commDouble

Experimentos	Maestro	Esclavo	r_∞ (MByte/s)	Startup time (μs)
Fortran 77 (byte)	contiguous	contiguous	14.01	44.78
Fortran 77 (double)	contiguous	contiguous	13.51	44.74
Java (byte)	contiguous	contiguous	13.95	209.15
Java (double)	contiguous	contiguous	13.47	219.15
Fortran 77 (byte)	stride	stride	0.95	45.89
Fortran 77 (double)	stride	stride	0.62	45.89
Java (byte)	stride	stride	0.94	218.19
Java (double)	stride	stride	0.61	225.64
Fortran 77 (byte)	loop	loop	0.41	41.22
Fortran 77 (double)	loop	loop	0.24	40.48
Java (byte)	loop	loop	0.19	219.12
Java (double)	loop	loop	0.13	221.13

Tabla 2: Resultados de commByte y commDouble

La tabla 2 nos da un resumen comparativo de los valores de r_∞ y del "Startup time" para los diferentes experimentos realizados. Como se esperaba, los resultados obtenidos con mensajes contiguos son mejores que los obtenidos con mensajes no contiguos. Sin embargo es de hacer

notar la enorme degradación de rendimiento exhibida por la transmisión de mensajes no contiguos, con respecto al rendimiento de la transmisión de mensajes contiguos. Tanto en el caso Fortran 77 y MPI como en el caso Java y mpiJava el paso de mensajes no contiguos con el modo STRIDE fue aproximadamente 14.7 veces más lenta que el paso de mensaje contiguos. En ambas implementaciones (Fortran 77 y Java) los resultados obtenidos con mensajes no contiguos son mejores en el caso del modo STRIDE que en el caso del modo LOOP. Esto refleja una buena implementación de bajo nivel de las copias de datos en la biblioteca MPICH. En el caso de mensajes no contiguos en modo LOOP, la versión de Fortran 77 es muy superior a la de Java y se debe a la copia del arreglo antes de enviarlo y a la copia del arreglo luego de su recepción. Estas copias son más rápidas en Fortran 77 que en Java.

Es de notar que la tasa de transferencia es superior en el caso de bytes que en el caso de reales en doble precisión, muy especialmente en el caso de datos no contiguos. Es importante hacer esta distinción porque el cómputo científico se basa principalmente en la manipulación de elementos de punto flotante. La relación de velocidades entre el paso de mensajes contiguos y mensajes no contiguos en modo STRIDE fue aproximadamente 21.8, una diferencia aún mayor que en el caso del paso de bytes.

Estas relaciones podrían disminuir con la utilización de procesadores más rápidos, dependiendo de la implementación de bajo nivel del MPI_Send y MPI_Receive de un objeto definido con MPI_Type_vector. Si la transferencia es realizada por el DMA de la tarjeta Myrinet no se espera una mejora significativa, mientras que si la transferencia es realizada por el procesador la mejora puede ser importante.

La tabla 3 muestra el sobrepeso de la llamada a las funciones de mpiJava, a través de la JNI. Como se puede ver con estos resultados, el sobrepeso de la llamada nativa a MPI es significativa solamente en el caso de envío de mensajes pequeños.

Modo del maestro	Modo del esclavo	Tamaño mensaje (Byte)	Tiempo (ms)	Sobrepeso llamada nativa	% que representa la llamada nativa
contiguous	contiguous	8	0.21	0.16	76.19%
contiguous	contiguous	1000	0.30	0.16	53.33%
contiguous	contiguous	50000	4.12	0.16	3.88%
contiguous	contiguous	100000	7.67	0.16	2.09%
stride	stride	8	0.24	0.17	70.83%
stride	stride	1000	1.18	0.17	14.41%
stride	stride	50000	52.75	0.17	0.32%
stride	stride	100000	106.50	0.17	0.16%
loop	loop	8	0.26	0.17	65.39%
loop	loop	1000	5.17	0.17	3.29%
loop	loop	50000	256.00	0.17	0.07%
loop	loop	100000	511.20	0.17	0.03%

Tabla 3: Sobrepeso de la llamada nativa en el envío de mensaje

Todos los programas desarrollados para este estudio se pueden obtener por Internet en la dirección: `ftp://kanaima.ciens.ucv.ve/pub/paper/panel99`.

4 Otras interfaces de MPI y PVM para Java

JavaMPI [9, 10] es otra interfaz que permite llamar al código nativo de su implementación de MPI desde Java. Ella fue desarrollada con la ayuda de JCI [11] (Java to C Interface generator), una herramienta que genera automáticamente la interfaz entre una biblioteca nativa existente y Java. JCI facilita el trabajo de implementación pero no llega a una API totalmente orientada a objetos.

DOGMA [12] (Distributed Object Group Metacomputing Architecture) es un ambiente para aplicaciones paralela en Java. Este ambiente contiene MPIJ que es una implementación de un subconjunto de MPI 1.1 escrito totalmente en Java e inspirado en la versión de C++.

MPI Software Technology [13] propone un producto comercial escrito en su mayoría en Java, que implementa las funcionalidades de MPI 1 y el manejo dinámico de procesos de MPI 2.

jPVM [14] es una interfaz para una biblioteca nativa de PVM para Java. Esta biblioteca fue inicialmente conocida con el nombre de JavaPVM y es desarrollada en el "Georgia Institute of Technology".

JPVM [15] es una implementación de PVM escrita totalmente en Java parecida a PVM para Fortran y C. Ella provee nuevas herramientas como los "safe threads" y varios "end-points" de comunicación por hilo.

5 Conclusiones y trabajo futuro

Los resultados expuestos en este artículo caracterizan el rendimiento de las comunicaciones de datos en "clusters" de componentes estándar (*commodity*), compuestos por estaciones de trabajo comunicándose a través de una red de alta velocidad, como Myrinet. El diseño de subsistemas de comunicación que mejoren algunos aspectos de las comunicaciones, como por ejemplo el problema de las copias de datos en memoria, así como las consideraciones que deben tener los programadores durante el diseño de sus algoritmos para minimizar las transferencias de datos no contiguos, son de vital importancia para obtener programas paralelos eficientes. En este trabajo se mostró el impacto que tienen estos aspectos usando Java y MPI, una solución al problema de portabilidad que se está vislumbrando como estándar en un futuro cercano.

El impacto de los manejadores de dispositivos (*drivers*) de bajo nivel en las comunicaciones es muy importante. La posibilidad de programar el protocolo de bajo nivel que ofrece la arquitectura de comunicaciones Myrinet ha dado lugar a la implementación de varios de estos protocolos, con diferentes niveles de confiabilidad y rapidez. El impacto de estas implementaciones de bajo nivel sobre el comportamiento de los benchmarks de comunicación y sobre la capacidad predictiva de éstos será una de las vías de investigación que desarrollaremos en el futuro.

Referencias

[1] A. Geist, A. Beguelin, J. Dongarra, and W. Jiang. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[2] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* University of Tennessee, Knoxville, Tennessee, June 1995.

[3] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and X. Li. mpiJava: A Java MPI Interface, 1998.

[4] http://www.npac.syr.edu/projects/pcrc/mpiJava.

[5] J. Dongarra and T. Hey. *The Parkbench Benchmark Collection*, volume 11 (2-3). Supercomputer, 1995. 94-114.

[6] PARKBENCH: PARallel Kernels and BENCHmarks. http://www.netlib.org/parkbench.

[7] N. Boden, D. Cohen, R. Felderman, A. Kulawik, and C. Seitz. Myrinet – A Gigabit-per-Second Local-Area Network. Technical report, Myricom, Arcadia, California, 1995.

[8] MPICH - A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich.

[9] S. Mintchev and V. Getov. Towards Portable Message Passing in Java: Binding MPI. In *Proceedings of EuroPVM-MPI*, pages 135–142, Kraków, Poland, November, 1997. Springer LNCS 1332.

[10] S. Mintchev. *Writing Programs in JavaMPI.* School of Computer Science, University of Westminster, 1997.

[11] S. Mintchev and V. Getov. Automatic Binding of Native Scientific Libraries to Java. In *Proceedings of ISCOPE*, pages 129–136, Marina del Rey, California, December, 1997. Springer LNCS 1343.

[12] Distributed Object Group Management Architecture. http://ccc.cs.byu.edu/DOGMA.

[13] G. Crawford III, Y. Dandass, and A. Skjellum. The JMPITM Commercial Message Passing Environment and Specification: Requirements, Design, Motivations, Strategies, and Target Users. http://www.mpi-softtech.com/publications.

[14] D. Thurman. jPVM. http://www.isye.gatech.edu/chmsr/jPVM.

[15] A. Ferrari. JPVM: Network Parallel Computing in Java. ACM 1998 Workshop on Java for High-Performance Network Computing. http://www.cs.ucsb.edu/conferences/java98.