

DSM-PEPE: Un Sistema de Memoria Compartida Distribuida para Multicomputadores de Bajo Costo*

Alvaro E. Campos y Federico Meza
[acampos,fmeza]@ing.puc.cl
Departamento de Ciencia de la Computación
P. Universidad Católica de Chile
Casilla 306, Correo 22, Santiago, Chile
Fax +56-2-354-4444

Resumen

El paradigma de memoria compartida distribuida permite aprovechar la potencialidad de computación paralela que ofrecen los sistemas multicomputador. Estos sistemas son más escalables y menos costosos que los sistemas multiprocesador fuertemente acoplados. Su programación es simple, a diferencia de los sistemas distribuidos basados en paso de mensajes. En este artículo describimos el diseño y los detalles de implementación de DSM-PEPE, un sistema de memoria compartida distribuida que opera sobre multicomputadores basados en computadores personales con los sistemas operativos Windows y Linux. Con DSM-PEPE es posible ejecutar programas paralelos con semántica de memoria compartida, utilizando redes convencionales compuestas por estaciones de bajo costo. El sistema fue diseñado por niveles y utilizando orientación a objetos, lo cual facilita su modificación y portabilidad. De hecho, la mayoría del código es común para los dos sistemas operativos en los que opera. Se condujo una serie de experimentos con una aplicación paralela, y los resultados confirman la potencialidad de este tipo de sistemas como máquinas paralelas virtuales. Utilizando 8 procesadores, se consiguió una aceleración de 3,26 al multiplicar matrices de tamaño 2048×2048 , bajando el tiempo de ejecución en casi 20 minutos. Actualmente trabajamos en la incorporación de *multithreading* y migración de *threads*, a nivel de los programas del usuario, así como en la inclusión de protocolos de consistencia relajados.

Palabras clave: Memoria Compartida Distribuida, Programación Paralela, Sistemas Distribuidos, Sistemas Paralelos, Multicomputadores.

1. Introducción

La computación paralela tiene como objetivo disminuir el tiempo de ejecución de aplicaciones con grandes requerimientos de procesamiento, repartiendo el trabajo entre distintas unidades de procesamiento que pueden operar en forma simultánea pero coordinada.

En un multiprocesador los procesadores están acoplados a una memoria común. Los procesos se comunican en forma simple a través de la memoria compartida, haciendo escrituras y lecturas. Además, los programas requieren de sincronización para acceder a los datos compartidos y controlar el progreso individual de los procesos paralelos.

En un multicomputador la memoria se encuentra distribuida y la comunicación se lleva a cabo utilizando paso de mensajes. Los programas solicitan los datos compartidos que requieren y deben a su vez enviar aquellos datos que le son solicitados. Las primitivas de comunicación sincrónicas

*Financiado parcialmente por Fondecyt, mediante proyecto 2990074

permiten efectuar una coordinación implícita que controla el progreso de los procesos. La exclusión mutua se implementa con algoritmos distribuidos construidos sobre el paso de mensajes.

Ambos enfoques tienen ventajas y desventajas. Los multiprocesadores son costosos y poco escalables, pero la semántica que se usa para escribir programas es simple. Los multicomputadores utilizan *hardware* convencional y son escalables. Sin embargo, la comunicación es más lenta y los programas más complicados, pues debe controlarse el flujo de datos entre las memorias distribuidas.

En un punto intermedio, los sistemas de memoria compartida distribuida (MCD) [9] intentan sacar provecho de la escalabilidad y relativo bajo costo de los multicomputadores, pero aprovechando la facilidad de programación de los multiprocesadores de memoria compartida.

Un sistema de MCD implementa con *software* un espacio virtual de memoria compartida en un multicomputador con memoria distribuida. Los programas se apegan a una semántica de memoria compartida pero, dado que la memoria se encuentra distribuida, el sistema se encarga de la comunicación subyacente con paso de mensajes. En los últimos años se han desarrollado diversos sistemas de MCD. Entre ellos se encuentra *TreadMarks* [6], *CVM* [5], *Brazos* [13], y *Millipede* [3].

DSM-PEPE es un sistema de MCD desarrollado por nosotros y orientado a *hardware* de bajo costo. Toma su nombre de las siglas de *Distributed Shared Memory – Parallel Environment for Program Execution*. Se trata de un ambiente paralelo para la ejecución de programas con semántica de memoria compartida. El sistema permite la configuración de máquinas paralelas distribuidas en las que se ejecutan programas con semántica de memoria compartida. De esta forma es posible construir sistemas paralelos baratos, escalables y fáciles de programar. *DSM-PEPE* también es una plataforma de experimentación. Su arquitectura por niveles y su implementación orientada a objetos hacen que la modificación e incorporación de nuevos componentes sea sencilla.

El diseño de *DSM-PEPE* es orientado a objetos, lo que ayuda a que sea estructurado, configurable y fácil de modificar. La programación se hizo en el lenguaje C++ y actualmente puede utilizarse en diversas versiones de Windows (NT, 2000, XP) y en Linux, en todos los casos con procesadores de tipo Intel.

Los primeros resultados obtenidos al ejecutar programas paralelos sobre *DSM-PEPE* muestran su eficacia como ambiente de ejecución. Además, la mayor parte del código se mantiene intacto entre las distintas versiones, lo cual muestra su portabilidad, producto del diseño orientado a objetos.

En este artículo describimos las principales características de *DSM-PEPE*, se tratan aspectos relacionados con la portabilidad entre sistemas operativos y se muestran los resultados de la experimentación llevada a cabo hasta ahora.

2. Características de DSM-PEPE

DSM-PEPE es un sistema de MCD para la ejecución de programas paralelos sobre un multicomputador compuesto por computadores personales de bajo costo. El sistema operativo puede ser Windows o Linux, pero debe ser homogéneo.

El diseño orientado a objetos de *DSM-PEPE* hace que sus componentes cuenten con interfaces limpias, que permiten su modificación o reemplazo en forma simple. Con *DSM-PEPE* es posible experimentar en forma flexible con los protocolos de consistencia, protocolos de comunicación y, a corto plazo, con políticas de migración de *threads*.

Los programas que ejecuta el sistema deben escribirse en el lenguaje C o C++. La semántica de memoria compartida se suministra a través de funciones de biblioteca. De esta forma, programas previamente escritos para un multiprocesador fuertemente acoplado pueden ser portados fácilmente a *DSM-PEPE*. Los programas paralelos dentro del sistema utilizan un paradigma SPMD, un único programa que actúa en forma paralela sobre distintos datos.

La memoria se solicita en bloques llamados regiones de MCD, en la misma forma en que se utiliza un *heap*. De esta forma la semántica es familiar al programador y los programas se portan con facilidad. Cada región es un fragmento de memoria que es administrado en forma dinámica

e independiente. En particular, los datos almacenados en una misma región son manejados por un mismo protocolo de consistencia. Regiones distintas pueden tener protocolos de consistencia distintos.

En DSM-PEPE, tanto las estructuras de datos que utiliza el sistema para su administración, como los elementos activos o *threads* que lo componen, son instancias de clases de objetos. La Figura 1 muestra los principales objetos participantes en un proceso en ejecución en uno de los nodos del sistema.

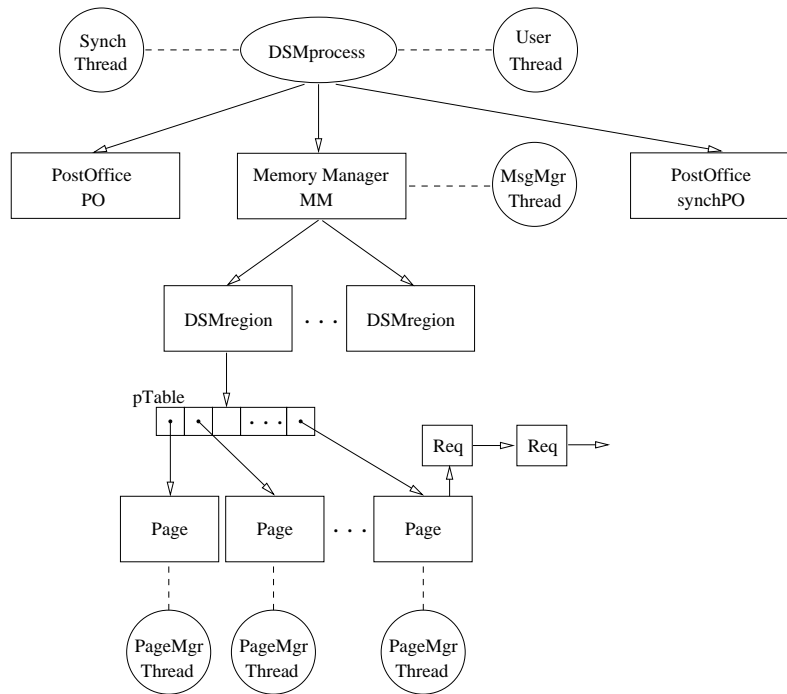


Figura 1: Objetos Dentro de un Proceso

Además del *thread* que ejecuta el código del usuario, existe un *thread* que está encargado de la recepción de mensajes desde los otros nodos del sistema, y un *thread* para cada página de memoria, encargado de garantizar su consistencia. Adicionalmente existe un *thread* que maneja en forma independiente las operaciones de sincronización.

El objeto administrador de memoria, o **MemoryManager**, tiene una instancia única, **MM**, en cada procesador. Su función es controlar el espacio global de direcciones compartidas para el proceso que se encuentra ejecutando. Además, es quien ofrece la interfaz de memoria compartida para el programador de aplicaciones, a través de funciones para la creación de regiones de memoria compartida y para la asignación de memoria dentro de una región. Cada una de las regiones creadas en el programa se representa como una instancia de la clase **DSMregion**.

Cada región tiene asociada un subconjunto del espacio lineal de direcciones compartidas, en la forma de un arreglo de páginas de memoria virtual. Este arreglo se representa en cada objeto **DSMregion** mediante una instancia de la clase **PageTable**. A su vez, cada página es representada dentro del sistema como una instancia de la clase **Page**, como se ilustra en la Figura 1. Cada página que ha sido instanciada dentro de una región de memoria es administrada por un *thread* individual.

Un objeto **Page** contiene información sobre una de las páginas compartidas. Por ejemplo, si la página tiene una copia válida en el nodo, o si su protección permite escritura o sólo lectura. También tiene una cola de requerimientos pendientes, locales o remotos, concernientes a la página en cuestión. De acuerdo al modelo de consistencia implementado, puede ser necesario mantener alguna otra información adicional.

El sistema manipula los protocolos de consistencia a nivel de regiones de memoria. La consistencia de todas las variables que se almacenen en una misma región será garantizada por un mismo protocolo. Sin embargo, variables almacenadas en distintas regiones podrán ser administradas con distintos protocolos de consistencia. Esto facilita la experimentación con protocolos de consistencia, a la vez que permite diferenciar el tratamiento de los distintos patrones de acceso a los datos compartidos.

Para implementar un protocolo de consistencia es necesario derivar una clase de objetos a partir de la clase `Page`. Existe un conjunto de eventos ante los cuales el protocolo debe actuar [11]. Precisamente, para agregar un protocolo se debe escribir el código concerniente al tratamiento de estos posibles eventos relacionados con consistencia. Esto se hace creando una nueva clase y escribiendo una serie de funciones, hasta ahora virtuales, de la nueva clase derivada. De esta forma la inclusión de un nuevo protocolo es una tarea sencilla y el resultado es un código extremadamente organizado.

Cuando se ejecuta un programa paralelo en DSM-PEPE, las páginas son inicializadas únicamente en el `Nodo 0` del sistema, e inicialmente residen sólo en ese nodo. Los demás nodos reservan el espacio, pero marcan las páginas como *inválidas*, de modo que cualquier acceso que se haga a ellas genere una excepción de *acceso ilegal a memoria*. El sistema intercepta todas las excepciones referentes a la administración de la MCD. Una vez que se ha completado el manejo de una excepción, la instrucción que la provocó es reiniciada para que el programa del usuario continúe.

Las excepciones pueden provocarse por una operación de lectura o de escritura. Esto da lugar a dos tipos de excepción: *ReadFault* o *WriteFault*, respectivamente. El manejador de excepciones que protege el programa del usuario determina el tipo de excepción y la dirección involucrada. A continuación, genera un objeto de tipo `Request` que es insertado en la cola de requerimientos de la página asociada. El proceso del usuario es suspendido y el *thread* encargado de manejar los requerimientos para la página es señalizado.

El `PageMgrThread` asociado con una página es responsable de atender los requerimientos locales, como el descrito anteriormente, pero también los requerimientos remotos, generados por otros nodos dentro del sistema. Dichas peticiones remotas son colocadas en la cola de requerimientos por el `MsgMgrThread` que se está ejecutando en el nodo y que recibe los mensajes de los demás nodos del sistema.

Cuando el `PageMgrThread` atiende uno de los requerimientos de su cola, identifica el tipo de solicitud que está tratando. Dependiendo de esto, llama a la función apropiada dentro de la clase `Page` que tenga asociada. De esta forma, el protocolo de consistencia actúa en forma independiente del manejo normal de las excepciones, y se permite la coexistencia de varios protocolos en regiones diferentes de memoria compartida.

Las únicas dos funciones que son visibles al programador son `CreateDSMregion` y `DSMalloc`. La primera de ellas permite crear una región de MCD, a partir de un tamaño y un protocolo de consistencia que se aplicará a la región. La función retorna un identificador de región, de tipo `RegionID`, que debe usarse en lo sucesivo para solicitar memoria de la región creada.

Las regiones de MCD se administran como *heaps*. La función `DSMalloc` asigna una cantidad determinada de bytes de una región previamente creada mediante `CreateDSMregion`. La función retorna un puntero que sirve como mecanismo para que el programador pueda acceder a la memoria compartida recién asignada. La región está compuesta por una serie de páginas que se concentran en una tabla de páginas. Esta tabla es en realidad una instancia de la clase `PageTable`, la cual no es más que un arreglo con punteros a objetos de un tipo derivado de la clase `Page`. El tipo exacto de los objetos depende del protocolo de consistencia con que se creó la región.

Las páginas de memoria son el objeto alrededor del cual se construye la funcionalidad del sistema de MCD, y sus réplicas deben mantenerse consistentes a lo largo del sistema distribuido. Las páginas se representan mediante clases derivadas de una clase común llamada `Page`. Dependiendo del protocolo de consistencia asociado con una región de memoria compartida, el constructor de la

clase `PageTable` crea páginas de la clase derivada asociada con el protocolo. La clase `Page` mantiene un estado común a todos los tipos de páginas y sirve de enlace con el objeto `PageTable`.

El `MsgMgrThread` es el responsable de la recepción de mensajes por la red y su correcto encausamiento. Permanece bloqueado en espera de mensajes relacionados con consistencia. Ante la llegada de un mensaje, el *thread* extrae el identificador de región y de página asociados. Su siguiente acción depende del tipo de mensaje que se haya recibido. Si el mensaje es un requerimiento provocado por un *fault* remoto, ya sea de lectura o de escritura, generará un objeto de tipo `RemoteRequest`, y lo insertará en la cola de requerimientos de la página involucrada. Posteriormente señalará al `PageMgrThread` de la página para que atienda el requerimiento. Si el mensaje llega como respuesta a una petición que se generó previamente para otro nodo, se señalará al `PageMgrThread` que espera por dicha respuesta.

Por otra parte, un `PageMgrThread` está encargado de la atención de la cola de requerimientos de la página que tiene asociada. Mientras la cola se encuentre vacía, el *thread* permanece bloqueado en un semáforo del objeto `Page`. Los requerimientos son colocados en la cola por el manejador de excepciones, para el caso de requerimientos locales, y por el `MsgMgrThread`, para el caso de requerimientos remotos. En cualquiera de los casos, el `PageMgrThread` es señalado indicando la llegada del requerimiento. Luego, como la funcionalidad del protocolo de consistencia no está implementada en el objeto `Page` sino en una clase derivada de él, el `PageMgrThread` llama a la función asociada con el tipo de requerimiento que se está tratando. Dicha función será ejecutada por la instancia de la clase derivada de `Page`.

3. Soporte del Sistema Operativo

En la actualidad es posible ejecutar DSM-PEPE en Windows y Linux sobre plataformas Intel. En ambos casos es necesario contar con cierto soporte del sistema operativo que permita la implementación del sistema.

Un sistema de MCD debe ofrecer a los programas un espacio virtual de memoria que es compartido por todos los procesadores. Para conseguir esto, debe tener la capacidad de manipular los espacios de memoria locales a cada procesador.

En DSM-PEPE el código del usuario se enlaza con una biblioteca. Cuando se crea una región de memoria compartida, ésta se replica en todos los computadores de la máquina paralela, pero se deja accesible sólo en uno de ellos. En el resto, la memoria se invalida para que no pueda ser accedida. Cualquier procesador que intenta acceder a la memoria compartida cuando se encuentra inválida va a generar una excepción que será interceptada por el sistema.

En Windows esto se consigue protegiendo el código del usuario con un manejador de excepciones. En Linux, se reemplaza el manejador de excepciones para la señal relacionada (`SIGSEGV`).

Para que este esquema funcione es necesario garantizar que el programa se cargue en las mismas direcciones virtuales en todos los computadores. Los programas distribuidos de DSM-PEPE cumplen con la característica SPMD (*single program, multiple data*). Los programas que se ejecutan en cada computador son idénticos. Esto garantiza que el sistema operativo los va a cargar en las mismas direcciones virtuales por lo que el espacio de memoria compartida distribuida va a ser el mismo en cada caso.

Los múltiples *threads* que se ejecutan se implementan en forma diferente en las versiones de Windows y Linux. En Windows, se utilizaron los *threads* y los mecanismos de sincronización del sistema operativo. En Linux, se utilizó una biblioteca de *threads* de nivel de usuario [2] recientemente portada por nosotros.

La ventaja de utilizar una biblioteca de nivel de usuario es que se tiene mayor flexibilidad en la manipulación de los *threads*. Esto es de vital importancia para implementar un mecanismo de migración, por lo que actualmente estamos trabajando en portar la biblioteca a Windows.

Windows ofrece un conjunto de funciones para manipular el estado de las páginas de memoria (*e.g.*, `VirtualQuery`, `VirtualProtect`, y `VirtualAlloc`). En Linux utilizamos los servicios del sistema para el manejo de la memoria: `mprotect`.

4. Programación

El núcleo de DSM-PEPE lleva a cabo las inicializaciones pertinentes y construye los objetos globales que son necesarios para la ejecución del programa.

Los programas del usuario tienen una estructura simétrica. Las regiones deben crearse en todas las instancias del proceso y las asignaciones de memoria también deben replicarse en forma distribuida. Para que cada proceso pueda conocer su identidad y trabajar en forma asimétrica, la función `main` recibe un parámetro entero, llamado `nodeID`. Este identificador permite diferenciar los distintos procesos que forman el programa paralelo y se asigna desde 0 en adelante.

En la actualidad el programador dispone únicamente de dos funciones provistas por la biblioteca de DSM-PEPE para el manejo de la MCD y cuyos prototipos se muestran en la Figura 2. El tipo de datos `RegionID` se utiliza para representar los identificadores de región de MCD. En la implementación actual el único protocolo de consistencia disponible es el secuencial.

```
RegionID CreateDSMregion(int size, Protocol protocol);
char *DSMalloc(RegionID region, int size);
```

Figura 2: Interfaz de Programación

La función `CreateDSMregion` define una región de memoria compartida cuya consistencia será garantizada por el protocolo especificado. El tamaño deseado para la región debe incluirse como primer argumento a la función, especificado en bytes. Si la región se pudo definir con éxito, la función devolverá un identificador de tipo `RegionID`. Si se produce algún error y la región no puede asignarse, la función retorna la constante `INVALID_REGION`.

La función `DSMalloc` asigna memoria dentro de una región de MCD previamente creada. El parámetro `region` debe ser el valor retornado por un llamado previo a `CreateDSMregion`. El parámetro `size` se refiere a la cantidad de bytes que se está solicitando asignar. La función retorna un puntero al área asignada, o la constante `NULL` en caso de error.

La sincronización distribuida se implementa a través de dos clases de objetos: `Barrier` y `Lock`, para barreras y *locks*, respectivamente. Las barreras permiten sincronizar a todos los procesos distribuidos en un punto determinado del programa. Los *locks* implementan exclusión mutua para el acceso a variables compartidas.

El código de la Figura 3 muestra la forma en que puede declararse un arreglo de números enteros en el espacio de memoria compartida distribuida. Se define una única región de MCD, cuyo tamaño se estima para que permita almacenar un arreglo de tamaño `N` de números enteros. Posteriormente se solicita memoria para el arreglo, el cual es referenciado utilizando un puntero a `int`. El nodo 0 inicializa el arreglo. Todos los demás procesadores esperan en una barrera que la inicialización finalice. Posteriormente todos despliegan en la pantalla el contenido del arreglo.

5. Experimentación y Discusión de Resultados

Para probar la potencialidad del sistema de MCD que desarrollamos, se condujo una serie de pruebas en dos plataformas diferentes. La aplicación seleccionada es una multiplicación de matrices

```

main(int nodeID) {
    RegionID Rid;
    int *arreglo;
    Rid = CreateDSMregion(N*sizeof(int));
    if (Rid == INVALID_REGION)
        perror("Error: No se pudo crear la region\n");
    arreglo = (int *)DSMalloc(Rid, N*sizeof(int));
    if (arreglo==NULL)
        perror("Error: No se pudo obtener memoria para el arreglo");
    if (nodeID == 0) {
        for (i=0; i<N; i++)
            arreglo[i] = i;
    }
    WaitInBarrier();
    for (i=0; i<N; i++)
        printf("%d", arreglo[i]);
}

```

Figura 3: Programa Ejemplo

cuadradas de números enteros. Se utilizó dos multicomputadores compuestos por hasta 8 computadores idénticos. Se llevaron a cabo pruebas con 2, 4 y 8 procesadores activos. En los experimentos se utilizó matrices cuadradas de distintos tamaños (número de filas y columnas): 256, 512, 1024, 1536, y 2048. Como medida de referencia se ejecutó una versión secuencial del programa de multiplicación de matrices, en un único procesador y sin enlazar la biblioteca de MCD. Para minimizar el impacto de eventuales errores inducidos por factores externos, las mediciones se repitieron 10 veces, en ambiente dedicado, para luego considerar valores promedio.

En el primer entorno se cuenta con procesadores Intel Pentium III de 1 GHz, 256 MB de memoria RAM, y sistema operativo Windows XP. La red que comunica los computadores es una Ethernet de 10 Mbps.

El segundo ambiente está compuesto por procesadores Intel Pentium III de 550 MHz, 128 MB de memoria RAM, y sistema operativo RedHat Linux 7.1. En este caso, los computadores se encuentran unidos por un *switch* de 100 Mbps, que garantiza varias comunicaciones simultáneas entre pares de computadores.

La multiplicación de matrices es un proceso iterativo que puede paralelizarse fácilmente, pues se puede acceder a las matrices que se están multiplicando sólo para lectura. La escritura del resultado puede distribuirse entre los múltiples procesadores, eliminando así las condiciones de competencia. Para la implementación de este algoritmo se eligió un particionamiento de las matrices de tipo *striped* [8], o por rebanadas.

Debe recordarse que DSM-PEPE inicializa toda la memoria compartida en el primer computador. Para el caso del experimento, todas las matrices involucradas residen en primera instancia en un único computador, y deben viajar por la red conforme son requeridas por los otros computadores para hacer su trabajo.

El valor medido es el tiempo de ejecución de los programas, es decir, el tiempo que le toma al programa paralelo obtener la solución global. Estos valores se compararon con el tiempo obtenido por un programa secuencial especialmente escrito para un monoprocesador. De esta forma se obtuvo la aceleración o *speedup* que se consigue con la versión paralela. Se consideró también la eficiencia

Configuración	MonoProc	MCD 4p	Aceleración	Eficiencia
256 × 256	0,37	0,54	0,69	0,17
512 × 512	5,02	5,68	0,88	0,22
1024 × 1024	108,59	56,40	1,93	0,48
1536 × 1536	367,84	187,52	1,96	0,49

Cuadro 1: Multiplicación de Matrices (Windows XP, 4 procesadores, distintos tamaños)

Configuración	MonoProc	MCD 4p	Aceleración	Eficiencia
256 × 256	0,93	0,32	2,85	0,36
512 × 512	9,83	2,93	3,36	0,42
1024 × 1024	182,45	54,94	3,32	0,42
1536 × 1536	578,33	176,00	3,29	0,41
2048 × 2048	1602,48	491,91	3,26	0,41

Cuadro 2: Multiplicación de Matrices (Linux, 8 procesadores, distintos tamaños)

obtenida, en función del número de computadores utilizados. Estos parámetros se calcularon con las siguientes fórmulas:

$$\text{Aceleración} = \frac{\text{Tiempo Secuencial}}{\text{Tiempo Paralelo}} \qquad \text{Eficiencia} = \frac{\text{Aceleración}}{\text{Número de Procesadores}}$$

El Cuadro 1 muestra una comparación del tiempo de ejecución para distintos tamaños de matriz en la plataforma Windows, con 4 procesadores. La primera columna, **MonoProc**, se refiere al programa secuencial. La segunda columna, **MCD 4P**, se refiere al programa paralelo. La tercera columna muestra la aceleración alcanzada con el multiprocesador. Finalmente, la cuarta columna muestra la eficiencia del multiprocesador en el uso de las 4 unidades de procesamiento.

Puede observarse cómo, para matrices pequeñas, la versión distribuida obtiene un desempeño pobre, comparado con la versión monoprocesador, incluso provocando una desaceleración para las primeras dos ejecuciones. El problema que se presenta en estos casos es que el *overhead* introducido por la administración de la MCD y la comunicación por la red, es significativamente alto comparado con las mejoras de desempeño provocadas por el uso de 4 procesadores en paralelo. En otras palabras, el problema no es lo suficientemente complejo en términos de computación como para que se justifique su ejecución en el multicomputador.

La situación cambia notoriamente conforme se complica el problema aumentando el tamaño de las matrices. Ya para matrices de tamaño 1024 × 1024, la versión multiprocesador obtiene un mejor tiempo de ejecución que la versión monoprocesador, con una aceleración de 1,93. El elevado costo computacional que tiene este problema de mayor dimensión hace que el *overhead* introducido por la MCD se vea opacado ante la incorporación de mayor poder computacional. Para el caso de matrices 1536 × 1536 el sistema distribuido también obtiene menor tiempo de ejecución, y la eficiencia más alta en el uso de los 4 procesadores.

El Cuadro 2 muestra resultados similares para el caso de la plataforma Linux, con 8 procesadores, llegando a matrices de mayor tamaño. Los procesadores en esta configuración son un poco más lentos que en el caso anterior y la red es más rápida. Esto provoca que el sistema paralelo obtenga aceleración para todos los problemas, respecto a la versión monoprocesador.

Para ayudar en la comparación, la Figura 4 resume las aceleraciones alcanzadas por las dos versiones de DSM-PEPE, para distinto número de procesadores, manteniendo constante el tamaño de las matrices: 1024 × 1024. El eje horizontal representa el número de procesadores: 2, 4 y 8, y el eje vertical la aceleración obtenida, respecto a la versión monoprocesador.

Comparación de Aceleración entre Windows y Linux
Multiplicación de Matrices 1024×1024

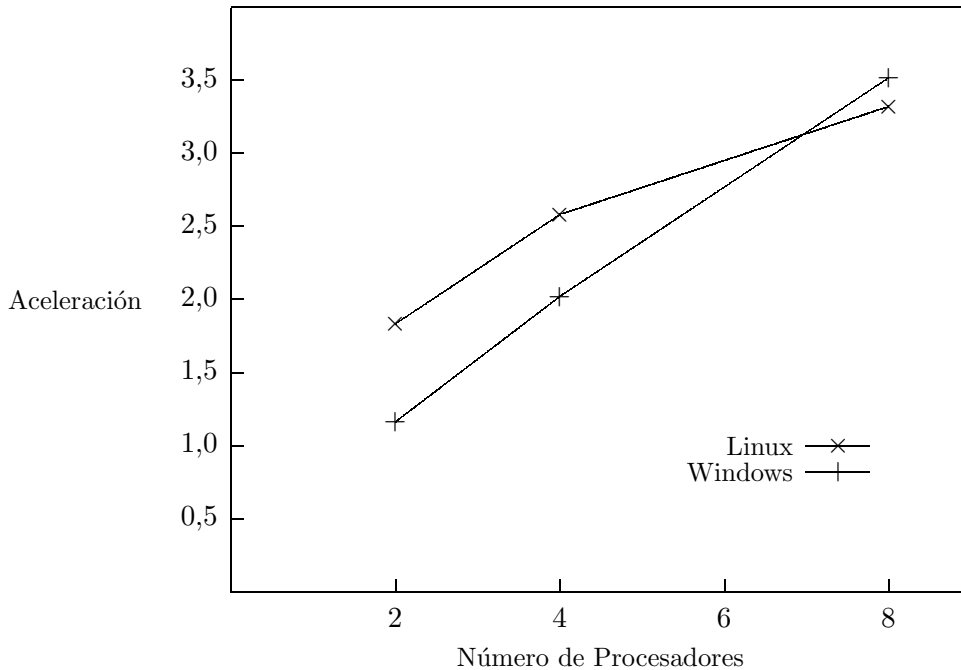


Figura 4: Aceleración en Ambas Plataformas

Analizando la figura puede verse que en Linux el sistema obtuvo mayor aceleración, al menos en los casos de 2 y 4 procesadores. Esto puede deberse a que el medio de comunicación es más rápido, y conmutado por un *switch* en el caso de Linux. De esta forma, el retardo de la red puede disimularse mejor. Para el caso de 8 procesadores la versión de Windows muestra una mejor aceleración. Por el momento no tenemos una explicación justificada para este comportamiento. Se esperaría que la curva de Windows fuera decreciendo en forma similar. Una posible explicación es que las máquinas Windows son más rápidas, por lo que se requeriría de problemas aún más grandes para saturarlas. Esto puede verse también como que la red Linux es más rápida y permite mejores aceleraciones hasta que los procesadores se saturan.

La curva correspondiente a Linux muestra el comportamiento esperado. La aceleración va aumentando conforme se incrementa el número de procesadores. Sin embargo, la pendiente es cada vez menor, pues el trabajo asignado a cada procesador es menor, por lo que el *overhead* provocado por la comunicación por la red va opacando cada vez más el beneficio obtenido por el paralelismo.

La Figura 5 compara los resultados de aceleración que obtuvo el sistema en Linux, para distintos tamaños de las matrices que se multiplican. Cada una de las series de datos representa los resultados para una configuración particular del multicomputador: 2, 4 y 8 procesadores. La línea recta correspondiente al monoprocesador se incluye como referencia.

La configuración con 2 procesadores muestra la curva con el comportamiento más predecible. La aceleración empieza a crecer conforme se aumenta el tamaño de las matrices, pues la complejidad del problema implica mayor trabajo en cada procesador. Sin embargo, de un punto en adelante la curva empieza a caer. Conforme las matrices se hacen cada vez más grandes, cada procesador debe encargarse de más trabajo secuencial. Además, la cantidad de páginas que deben viajar por la red es cada vez mayor. Al mantenerse constante el número de procesadores y aumentar el tráfico de datos por la red, la aceleración disminuye.

Multiplicación de Matrices Cuadradas
 Comparación de aceleración para distintos tamaños del problema
 (Plataforma Linux)

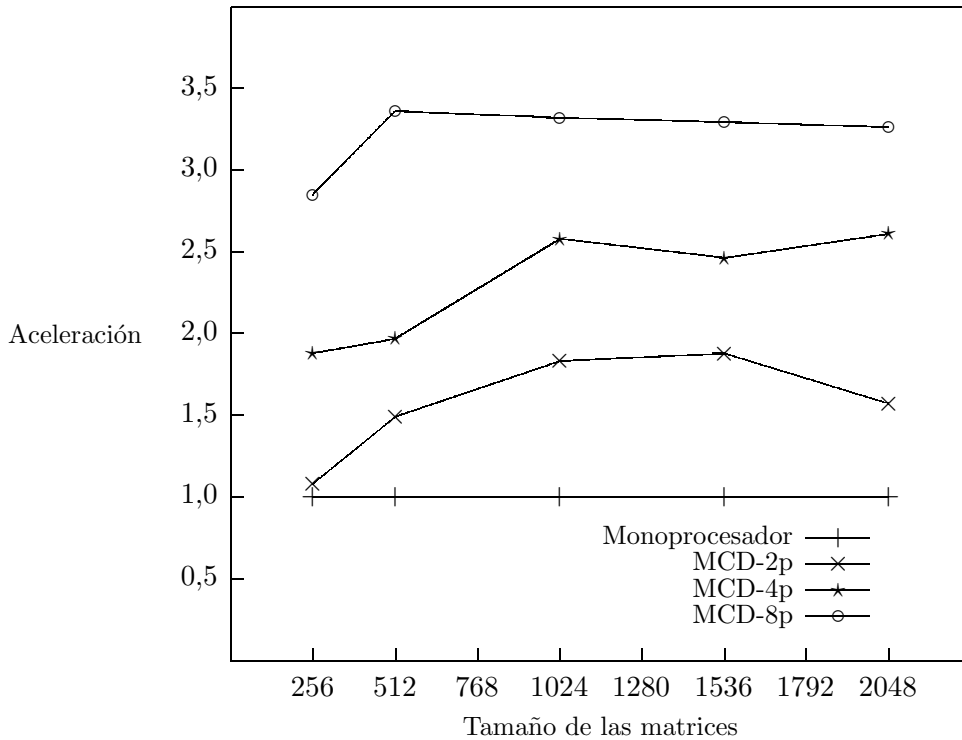


Figura 5: Aceleración en Plataforma Linux

En el caso en que se usan 4 procesadores, el comportamiento del sistema es más irregular y por lo tanto difícil de analizar. Cuando se utilizan 8 procesadores, el sistema obtiene la mejor aceleración con las matrices de 512×512 , empezando a decaer a partir de ese momento con una pendiente constante.

Finalmente, vale la pena resaltar algo que es evidente a partir de la Figura 5. Conforme se agregan procesadores al sistema, la aceleración es mucho mayor. Esto puede notarse por la separación que tienen entre sí las distintas curvas.

En la Figura 6 se puede comparar los tiempos de ejecución para el programa, para distintos tamaños del problema. Cada una de las series de datos representa los resultados para una configuración particular del multicomputador: 2, 4 y 8 procesadores. Aquí también se incluye la versión monoprocesador como referencia.

Esta figura confirma el análisis que hicimos previamente. En todos los casos el tiempo de ejecución va creciendo conforme se incrementa el tamaño de las matrices. Dada la complejidad computacional del problema, la curva que representa el tiempo de ejecución es una parábola. Puede verse como, cuando se usan muchos procesadores, la curvatura es más leve.

Finalmente, es importante notar las diferencias absolutas en los tiempos de ejecución. Para las matrices más grandes la diferencia de tiempo de ejecución es de aproximadamente 1,100 segundos, es decir, casi 20 minutos. La versión monoprocesador consumió 1,600 segundos en resolver el problema, mientras que DSM-PEPE poco menos de 500.

Multiplicación de Matrices Cuadradas
 Comparación de tiempo de ejecución para distintos tamaños del problema
 (Plataforma Linux)

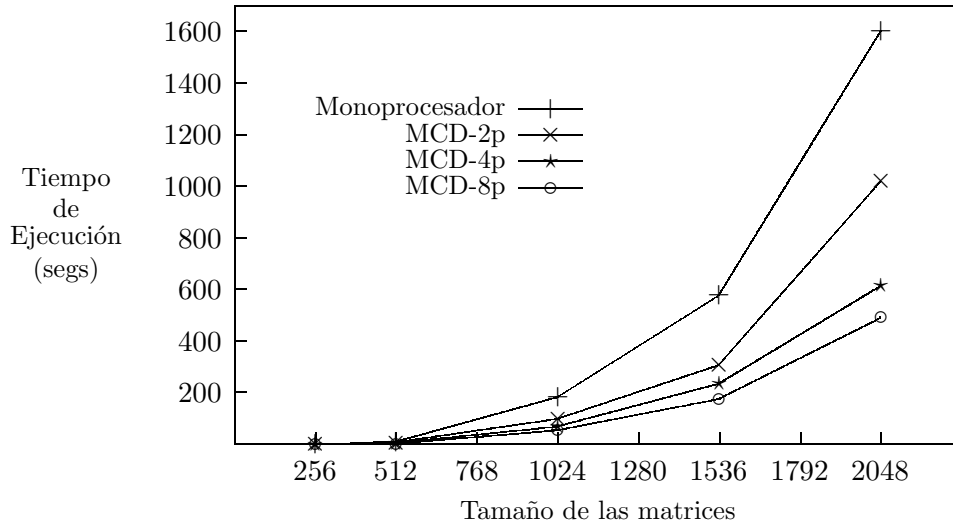


Figura 6: Tiempo de Ejecución en Plataforma Linux

6. Conclusiones y Trabajo Futuro

El diseño de DSM-PEPE, orientado a objetos, hace posible modificar el sistema e incluir nuevos componentes, por ejemplo, nuevos protocolos. La implementación del protocolo de consistencia secuencial es un buen ejemplo del procedimiento a seguir para incorporar un protocolo al sistema. Puesto que el sistema se encarga de manejar la ocurrencia de eventos que involucran acciones de consistencia y la comunicación misma, el protocolo tan sólo debe manejar la atención de los eventos y el tráfico de las páginas compartidas. En un futuro cercano se espera poder contar con implementaciones para otros protocolos de consistencia, de preferencia más relajados, por ejemplo, el modelo PRAM [10] o el modelo *Lazy Release* [7].

A pesar de que muchos de los factores que vale la pena evaluar en un sistema de MCD pueden estudiarse utilizando técnicas de simulación, la ejecución efectiva sobre un multicomputador ofrece mayor realismo, principalmente en aspectos como retardo de comunicación y competencia por el medio, en el caso de una **Ethernet**.

El desempeño es un aspecto primordial en un sistema de MCD, dado que la motivación para su uso es obtener mayor poder computacional. En el caso de DSM-PEPE se ha demostrado que, aún en problemas muy simples y con moderados requerimientos de computación, un pequeño computador paralelo puede obtener mejoras interesantes en el tiempo que toma obtener las soluciones.

Los resultados preliminares de la experimentación demuestran la factibilidad de los sistemas de MCD como computadores paralelos efectivos. Sin embargo, aunque estos resultados son alentadores, todavía quedan muchos aspectos por mejorar en la implementación, con el objetivo de reducir el *overhead* y acercar así el desempeño a los niveles conocidos para sistemas de paso de mensajes.

Actualmente se efectúa un conjunto de pruebas de mayor complejidad. Se pretende modificar problemas estándares, como los contenidos en **Splash** [12], para probar el potencial del modelo. Si la tarea de portar las aplicaciones se logra con un mínimo de esfuerzo, quedará demostrado que este tipo de sistemas es un sustituto potencial para los costosos multiprocesadores para los cuales fueron escritos los programas. Por ahora, queda la sensación de que la programación en DSM-PEPE es muy simple, y sus programas resultan muy familiares.

Durante la implementación del sistema, se pudo verificar que tanto **Windows** como **Linux** ofrecen las características necesarias para implementar un sistema de MCD compuesto por computadores personales de bajo costo. Con un diseño por niveles se logró aprovechar la mayor parte del código en ambas versiones de nuestro sistema.

Las mayores potencialidades de **DSM-PEPE** están por ser incorporadas. Está en proceso de programación una nueva versión del sistema que permitirá la incorporación de *multithreading* en los programas del usuario, junto a capacidades de migración de *threads*. Esto pretende aumentar la localidad de referencia [4] de los datos compartidos.

La implementación en desarrollo contiene réplicas en todos los nodos de cada uno de los *threads* del usuario. Esto permite su ejecución en una única ubicación mientras el resto de las copias permanecen bloqueadas. Ante una migración, el *thread* activo se bloqueará y su réplica en el nodo destino reiniciará la ejecución en el mismo estado en que se suspendió la anterior. Ya hemos logrado avances importantes en el mecanismo de migración, utilizando la biblioteca de *threads* a nivel de usuario.

Estas innovaciones abrirían nuevas posibilidades de investigación en aspectos como la relación entre el modelo de consistencia y la migración de *threads* [1], así como políticas de migración y balance de carga [4].

Referencias

- [1] Alvaro E. Campos and Federico Meza. Modelos de Consistencia y Migración de Threads en un Sistema de MCD. Technical Report RT-PUC-DCC-99-01, Departamento de Ciencia de la Computación, P. Universidad Católica de Chile, Santiago, Chile, Junio 1999.
- [2] Gordon V. Cormack. A micro-kernel for concurrency in C. *Software—Practice & Experience*, 18(5):485–491, May 1988.
- [3] Roy Friedman, Maxim Goldin, Ayal Itzkovitz, and Assaf Schuster. Millipede: Easy Parallel Programming in Available Distributed Environments. *Software: Practice and Experience*, 27(8):929–965, August 1997.
- [4] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *Journal of Systems and Software*, 42(1):71–87, 1998.
- [5] Peter Keleher. The CVM Manual. Technical report, Department of Computer Science, University of Maryland, 1996.
- [6] Peter Keleher. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.
- [7] Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *19th Annual International Symposium on Computer Architecture, ACM*, pages 13–21, May 1992.
- [8] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, California, 1994.
- [9] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, October 1986.
- [10] R.J. Lipton and J.S. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Department of Computer Science, Princeton University, September 1988.
- [11] Mukund Raghavachari and Anne Rogers. Ace: A Language for Parallel Programming with Customizable Protocols. *ACM Transactions on Computer Systems*, 17(3):202–248, August 1999.
- [12] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, 1991.
- [13] Evan Speight and John K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the First USENIX Windows/NT Workshop*, August 1997.