2003

Tesina de grado

Facultad de Informática UNLP

Andrés Barbieri
Barbieri@lidi.info.unlp.edu.ar

Análisis e implementación de comunicaciones colectivas en NOWs

Ing. Armando De Giusti
Co-Director: Ms. en Informática Fernando G. Tinetti
Universidad Nacional de La Plata
Argentina
LIDI CeTAD

Tesina de Grado:
Análisis e Implementación de Comunicaciones Colectivas en NOWs
Facultad de Informática - UNLP

Andres Barbieri[1]

Director: Ing. Armando De Giusti
Co-Director: Ms. en Informática Fernando G. Tinetti
LIDI[2] - CeTAD[3]

6 de febrero de 2003

[1]barbieri@lidi.info.unlp.edu.ar
[2]Calle 50 y 115 - Fac. de Informática - Universidad Nacional de La Plata - (1900) La Plata, Argentina
[3]Calle 49 y 116 - Fac. de Ingeniería - Universidad Nacional de La Plata - (1900) La Plata, Argentina

Capítulo 1

Introducción

Para describir el estado actual y también la tendencia desde un tiempo atrás del cómputo científico se puede enunciar tres frases:

- Día a día la capacidad de procesamiento que ofrece un ordenador crece de forma acelerada.
- La factibilidad de acceder a los ordenadores más rápidos del momento sigue siendo una cuestión muy difícil.
- Los problemas con alta necesidad de cómputo se equiparan al mismo ritmo del avance de la tecnología.

De aquí se obtiene como resultado rendimientos casi constantes a lo largo del tiempo para la resolución de problemas con grandes requerimientos de cómputo utilizando un solo ordenador. La solución más factible para superar esta barrera en el ámbito informático sigue siendo la aplicación del procesamiento paralelo. En gran cantidad de publicaciones que tratan sobre el procesamiento paralelo se destacan las redes de estaciones de trabajo (NOWs - Networks of Workstations) haciéndose hincapié en su alta escalabilidad y su sorprendente relación costo beneficio. Es un hecho que son la plataforma más barata para la aplicación del procesamiento paralelo y además en muchos casos supera en un orden de magnitud a un supercomputador tradicional [CCWP00-1].
Las NOWs son una arquitectura distribuida para la cual las aplicaciones estarán compuesta del algoritmo fundamental que será el encargado de hacer el cómputo y de un overhead o sobrecarga de programa que será la porción de la comunicación o intercambio de datos entre los diferentes nodos. En consecuencia el tiempo total necesario para ejecutar el programa esta dado por la ecuación 1.1.

$$T_{total} = T_{computo} + T_{comunicaciones} \tag{1.1}$$

Si bien existen numerosas alternativas para el hardware de comunicaciones [CCWP00-2], este trabajo cuando se refiere a NOWs lo hace en el área de redes locales (LANs) estándares, en cuyo campo la técnica de control de acceso al medio más usada es la de CSMA/CD [Stalling] implementada por Ethernet o IEEE 802.3 [1] que abarcan la mayor cantidad de instalaciones.

[1] IEEE 802.3 es la estandarización de Ethernet

El punto más débil de las NOWs son las comunicaciones debido a que las redes tan difundidas no fueron diseñadas para el procesamiento paralelo, y, salvo muy pocas excepciones, la latencia o start-up y el ancho de banda obtenido son relativamente bajos si se los compara con hardware paralelo [LPP-HOWTO]. La forma de minimizar este inconveniente es tener optimizadas "al hardware" las operaciones de comunicaciones.
El modelo de comunicaciones más natural para esta arquitectura "débilmente acoplada" esta basado en el pasaje de mensajes introducido por CSP [CSP]. Las implementaciones de bibliotecas [2] más usadas de distribución libre para explotar a las NOWs como máquinas paralelas: PVM, LAM/MPI o MPICH, están basadas sobre este modelo.
Las aplicaciones que usan estas bibliotecas asumen que las rutinas de comunicaciones están optimizadas para aprovechar el hardware subyacente por lo que no se ocupan del rendimiento de comunicaciones. Sin embargo todas las características favorables que poseen las NOWs usadas como máquinas paralelas son significativas en el área de la computación de alto rendimiento si realmente las comunicaciones funcionan lo mejor posible (Al máximo de lo que permite el hardware), logrando minimizar la sobrecarga.
Dentro de las rutinas de comunicaciones existe un grupo denominado "comunicaciones colectivas" o *"aggregate functions"* las cuales involucran a un grupo de procesos o tareas. A partir de éstas se pueden resolver numerosos problemas del cómputo paralelo. Algunos ejemplos son los definidos en el área del álgebra lineal por PBLAS [PBLAS] (Subconjunto de las principales funciones de LAPACK [LAPACK] paralelizadas). Dentro de las operaciones matriciales una de las más cruciales es la multiplicación de matrices Esta puede ser implementada eficientemente con el uso de la operación colectiva *broadcast*. El *broadcast* es el mecanismo básico que usa Ethernet para transmitir la información, cuestión que da lugar a que pueda ser implementada eficientemente sobre NOWs. Es esta operación que se busca optimizar para que aproveche al máximo lo que ofrece el hardware subyacente.

1.1 Objetivos del Trabajo

Los objetivos del trabajo son medir el rendimiento de las comunicaciones colectivas en las algunas de las bibliotecas más usadas (PVM, LAM/MPI), analizar cuáles son los modelos más naturales de estas operaciones para las redes Ethernet e implementarlos para comparar los rendimientos encontrados en las implementaciones analizadas. El objetivo fundamental es optimizar el *broadcast* debido a que es una operación muy utilizada y su modelo es soportado directamente por el hardware.

1.2 Organización del Trabajo

- En la primera parte de este trabajo se analizan las colectivas sobre algunos de los *middlewares* más usados como soporte para la programación sobre NOWs y se intenta mostrar cuales son sus ventajas y cuales sus desventajas. Se analizan sus implementaciones y se mide su rendimiento.

[2] Conjunto de rutinas para resolver comunicaciones entre tareas que conforman la aplicación

- Se hace un análisis de los posibles modelos de implementaciones de algoritmos para resolver las comunicaciones colectivas.
- En la segunda parte se mencionan cuáles son las componentes e interfaz para una posible re-implementación y justificación del conjunto de comunicaciones colectivas. Se describe cual es la forma más natural de hacerlo.
- Se hace mayor énfasis en el *broadcast* y se tratan otras operaciones por completitud.
- Luego se muestra como se re-implementan y se las compara con las analizadas anteriormente.

1.3 Notas Aclaratorias

En el presente texto se hace referencia a proceso(s) o tarea(s) de forma indistinta, indicando programas secuenciales y autónomos en ejecución que se pueden comunicar entre sí y que en conjunto constituyen una aplicación o programa paralelo. En alguna de la bibliografía, como por ejemplo en la referente a PVM se usa el término tarea (task) y en otra proceso (process). Si se quiere ser más estricto se encuentran diferencias sutiles entre ambos: un proceso es un elemento abstracto concurrente y tarea una implementación de un proceso en un lenguaje dado, pero en ocasiones en la terminología de sistemas operativos se llama proceso a la implementación. De cualquier forma aquí los términos se utilizan de forma intercambiable.

Bibliografía

[LAPACK] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A Greenbaum, S Hammarling, A McKenney, D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. LAPACK Working Note 20, University of Tennessee, CS-90-105, May 1990. Home page: http://www.netlib.org/lapack

[PBLAS] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. LAPACK Working Note 100, University of Tennessee, May 1995.

[Stalling] Comunicación en Redes de Computadores. Willams Stallings - Prentice-Hall (5ta. Edición).

[CCWP00-1] Cluster Computing White Paper version 2.0. Editor Mark Baker University of Portsmouth, UK. An Introduction to PC Clusters for HPC: Thomas Sterling California Institute of Technology and NASA JPL, USA. 28th Dec 2000.

[LPP-HOWTO] Linux Parallel Processing HOWTO. Hank Dietz, pplinux@ecn.purdue.edu. v980105, 5 January 1998.

[CSP] CSP (Communicationg Sequential Processes) HOA78-Hoare, C.A.R., Communicating Sequential Processes, Communication of the ACM, Vol 21, Num 8, Agust 1978.

[CCWP00-2] Cluster Computing White Paper version 2.0. Editor Mark Baker University of Portsmouth, UK. Network Technologies: Amy Apon, University of Arkansas USA, Mark Baker. 28th Dec 2000.

Capítulo 2

Análisis de Grupos y Comunicaciones colectivas en PVM

2.1 Introducción

PVM[PVM1][PVM2] (Parallel Virtual Machine) fue desarrollado por el "Heterogeneous Network Project" con el objetivo de facilitar el cómputo paralelo heterogéneo[PVM3], área en la cual las NOWs conforman la base principal[Tntti]. En estos últimos años se ha incrementado la utilización de esta arquitectura debido que ha demostrado tener desemnpeños suficientemente satisfactorios, es un echo que la relación costo por performance de una máquina Beowulf[BWULF] es de 3 a 10 veces mejor que un supercomputador tradicional. PVM en algunos aspectectos de rendimiento se ha visto superado por otras alternativas, como implementaciones de MPI[MPI], pero hay que aclarar que la filosofía de PVM en estos entornos ha sido la de dar prioridad a la flexibilidad, simplicidad y tolerancia a fallas, tres aspectos de los cuales no se puede cuestionar ninguno. Además, hay que marcar que se encuentra el código y los binarios disponibles, para aprender de este y poder mejorar aquellos puntos donde sea necesario, sin olvidar la cantidad de programas que existen escritos utilizando esta biblioteca. Debido a esto es que se considera importante hacer el siguiente análisis.

2.2 Manejo de grupos

PVM soporta el manejo de grupos en una biblioteca (`libgpvm.a`) separada, la cual se ubica sobre el núcleo de PVM. Esta hace uso de la funcionalidad de comunicaciones aportada por la capa inferior. El *pvmd* no implementa estas operaciones, sino por el contrario son realizadas en una tarea PVM diferente llamada "GROUP SERVER" o simplemente *pvmgs* , que es automáticamente "leventada" cuando se invoca a la primera función que hace uso de grupos.
El manejo de grupos es completamente dinámico debido a la filosofía de PVM de sacrificar algo de eficiencia a beneficio de ganar en transparencia al usuario. Una tarea PVM puede incorporarse y salirse de los grupos cuando lo desee y

cuantas veces quiera sin tener que informar nada a nadie más que al *pvmgs* . Se puede enviar *broadcast* sin pertenecer al grupo destino, y en general las funciones colectivas pueden ser llamadas desde cualquier tarea, salvo `pvm_reduce(3PVM)`, `pvm_lvgroup(3PVM)` y `pvm_barrier(3PVM)` que, debido a su naturaleza requieren de forma necesaria que el remitente pertenezca al grupo destino.
Cuando se desea hacer uso de operaciones colectivas, los programas de usuarios deben linkeditarse con la biblioteca mencionada anteriormente (`libgpvm.a`).

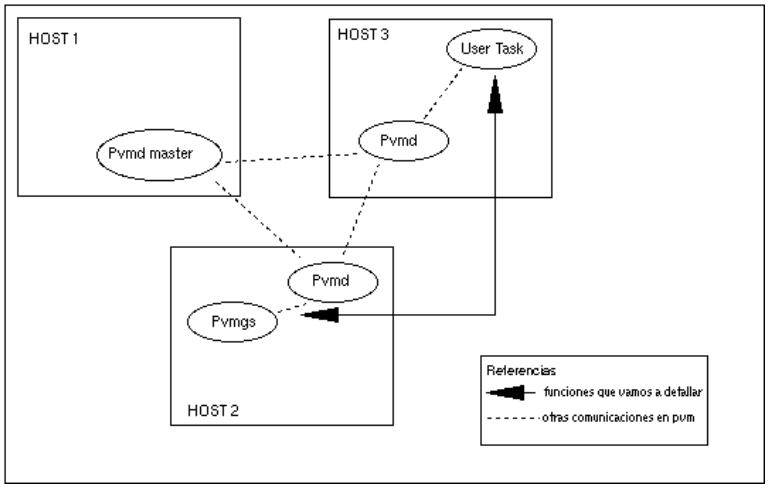

Figura 2.1: Espectro de comunicaciones que cubren las funciones a describir

2.2.1 Descripción de Funciones de Grupos

En esta sección se describen las funciones necesarias para armar, mantener y modificar los grupos de PVM. Luego más adelante se analizan las operaciones colectivas.

```
C          int inum = pvm_joingroup( char *group )
```

Agrega a la tarea que llama al grupo identificado con el nombre pasado como parámetro. Retorna el número de instancia de la tarea dentro del grupo, el cual comienza en 0. Una tarea puede estar simultáneamente en varios grupos. Los números de instancias de tareas son reusados por PVM cuando dicha tarea se sale del grupo y otra se quiere incorporar. Por esta causa es probable que una tarea que se sale y se reincorpora en un grupo obtenga un número de instancia diferente al anterior. Si el grupo no existe, éste es creado y luego se agrega la tarea. Como es de esperar, una tarea no puede incorporarse más de una vez a un mismo grupo sin salirse previamente.

```
C          int info = pvm_lvgroup( char *group )
```

Se desenrola a la tarea que llama del grupo indicado. Con respecto a los números de instancia se tienen las mismas consideraciones que con la función `pvm_joingroup(3PVM)`.

```
C          int size = pvm_gsize( char *group )
```

Retorna el número de miembros actualmente presentes en el grupo. Debido a que los grupos pueden cambiar dinámicamente esta rutina sólo asegura que el dato obtenido es válido en el instante de tiempo que es sacado de la tabla que maneja el *pvmgs*.

```
C          int inum = pvm_getinst( char *group, int tid )
```

Toma un string `group` y un entero `tid` y retorna el número de instancia de la tarea identificada por estos valores. Puede ser llamada por cualquier tarea.

```
C           int tid = pvm_gettid( char *group, int inum )
```

Devuelve el `tid` del proceso identificado con el número de instancia dado en el grupo indicado por el parámetro.

```
C          int info = pvm_freezegroup( char *group , int size)
```

Hace de un grupo dinámico (los grupos por omisión son dinámicos) uno estático. La información del grupo es "cacheada" por todos los integrantes. Esta función para completar debe ser llamada por todos los miembros del grupo que deseen pasar a ser estáticos. El parámetro `size` indica cuantos deben sincronizar, un valor de `-1`, indica todos los miembros del grupo. Todas las operaciones colectivas que puedan resolverse con la información "cacheada" se harán así. El *barrier*, hasta la última versión contemplada por el texto se manejaba mediante el *pvmgs* .

`pvm_freezegroup(3PVM):`

"... Subsequent operations then use the local information. Barriers are still arbitrated by the group server..."

"...Barrier are always arbitrated by the group server, even if the group has been made static with `pvm_freezegroup` ..."

Para liberar las estructuras asignadas que contienen la información del grupo localmente se debe llamar a `pvm_lvgroup(3PVM)`.
Si algún proceso deja el grupo estático donde hay otros esperando en una barrera se retorna un error.

2.2.2 Cómo Funcionan Internamente

El código para manejos de grupos y para las comunicaciones colectivas esta escrito en *C* y se encuentra en: `$PVM_ROOT/pvmgs/`. El que se muestra aquí pertence a la versión 3.3.11 y fue comparado con los fuentes de la última versión hasta el momento de la escritura del capítulo, versión 3.4.3, dando como resultado que no tiene prácticamente diferencias en la sección de manejo de grupos.

Estructura de grupos

El proceso denominado *pvmgs* mantiene una tabla de hashing en la cual tiene asentado todos los grupos que se han registrado sobre la PVM. (`pvmgs_core.c`)
Los grupos están almacenados en una estructura con la siguente forma:

```
...
typedef struct group_struct {
        char *name;         /* the name of the group */
        int  len;           /* length of the group name */
        int  ntids;         /* number of tids */
        int  *tids;         /* array of the tids */
        int  maxntids;      /* max number of tids before remalloc*/
        ...
    } GROUP_STRUCT, *GROUP_STRUCT_PTR;
...
```

(`pvmgs_ds.h`)
Los puntos al final significan que se suprimío una porción de la estructura o código por simplificación. Al final de cada porción de código se escribe el nombre del archivo fuente donde esta ubicado.
Como se observa en el campo `tids` se tiene un arreglo con los identificadores de tareas que conforman el grupo, los números de instancias son los índices derivados de la posición que ocupa en dicho arreglo cada tarea. La cantidad de tareas en el grupo se describe en el campo `ntids`. Además de estos valores, en la estructura se almacena información para sincronizar a las tareas, estado del grupo, etc.

La tabla de hashing es un arreglo donde las componentes son nodos con punteros a grupos además de referencias al grupo anterior y al siguiente dispersados en la misma estructura para resolver de forma secuencial las colisiones.

```
...
GROUP_LIST hash_list[HASHSIZE];
...
int nameindex;                      /* index of pvmgs-defined names
...
```

(pvmgs_core.c)

```
...
typedef struct group_list {
        struct group_list *prev, *next;
        GROUP_STRUCT_PTR group;
        } GROUP_LIST, *GROUP_LIST_PTR;
...
```

(pvmgs_ds.h)

Como inicia el servicio del *pvmgs*

El *pvmgs* una vez iniciado primero se registra como una tarea normal de PVM invocando a `pvm_mytid(3PVM)`, luego se registra en la base de datos de PVM con `pvm_putinfo(3PVM)` (versión 3.4) o `pvm_insert(3PVM)` (versión 3.3) indirectamente mediante la función interna `gs_register` para asegurar un único *pvmgs* . Siguiendo la secuencia de código inicializa la tabla de hashing y por último si fue todo bien entra en un loop infinito (`while (1)`) esperando leer algún mensaje de requerimiento y al recibirlo y lo resuelve y responde:

```
...
while (1)
 {
     if (pvm_recv(-1, -1) < 0)  /* receive a request  */

     ...

     /* get: length of message, message tag, and sending tid */

     if (pvm_bufinfo(pvm_getrbuf(), &len, &msgtag, &tid) < 0)
```

```
            ...

            /* Switch between different request types */
            switch(msgtag)
            {
             case (DIE):
            .....
             case (JOIN):
            ......
             case (LEAVE):
            ......
             case (DEADTID):  /* task in one or more groups has died */
            ......
             case (BARRIER):  /*  Handle barrier for a group */
            ......
             case (BARRIERV):
            ......
             case (BCAST):
            ......

            }

        ...

    }
    ...
```

(pvmgs_core.c)

La registración del *pvmgs* se hace a través de una base de datos que mantiene el *pvmd master*. Esta base de datos sirve para guardar y acceder a valores desde cualquier tarea. La información es referenciada mediante un par *(name, index)* donde *name* es un string terminado en null e *index* un entero, y, el tipo de valores que se pueden almacenar son enteros. Para especificar la primera entrada libre en *index* se usa el valor `-1`. La API para el acceso a este almacenamiento se hace a través de las siguientes funciones:

`pvm_insert(3PVM)`, `pvm_lookup(3PVM)`, `pvm_delete(3PVM)` (versión 3.3).

`pvm_putinfo(3PVM)`, `pvm_recvinfo(3PVM)`, `pvm_getmboxinfo(3PVM)`, `pvm_delinfo(3PVM)` (versión 3.4).

Estas operaciones no forman parte del manejo de grupos pero se utiliza como mecanismo subyacente para implementar algunos detalles.
A continuación se muestra cómo "bootea" el *pvmgs* , el paso anterior a que este exista:

```
...
gs_getgstid()
{
```

```
            ...
            mytid = pvm_mytid();
            if (gstid >= 0 && mytid == myoldtid)
                return (gstid);
           info = pvm_lookup(GSNAME, 0, &gstid);
           info = pvm_spawn(``pvmgs'', (char **)0, PvmMppFront,
                                    (char *)0, 1, &gstid);
          ...

   }
```

(pvmgsu_core.c)
Obtiene el `tid` del *pvmgs* , si no ésta lo inicia haciendo un `pvm_spawn`. Esta función es invocada desde `int_query_server` ante cualquier requerimento para localizar el *pvmgs* , por ejemplo para pedir un servicio de grupos o desde `gs_get_tidlist` para obtener una lista de `tids` de un grupo.
Una véz iniciada la tarea *pvmgs* ejecuta lo mencionado en la primera parte de la sección. El código de la registración que se indico que se hacia de forma indirecta mediante `gs_register` se muestra a continuación:

```
gs_register()
{
        .....

        cc = pvm_insert(name, -1, tid);

        .....
}
```

(pvmgs_func.c)

Cómo se brinda el servicio

Cada vez que el *pvmgs* recibe un requerimiento, chequea el tipo del mismo y llama a la función correspondiente para atenderlo, por ejemplo el caso de JOINGROUP:

(pvmgsu_core.c) (CLIENT SIDE: init request)

```
...
int
pvm_joingroup(group)
char *group;
{
    int x, gid;
        BGN_TRACE(pvmtoplvl, x, TEV_JOINGROUP0, group,(int *) NULL);
        int_query_server(group, JOIN, ``pvm_joingroup'', &gid, 0);
```

```
        END_TRACE(pvmtoplvl,x, TEV_JOINGROUP1, &gid);
        if (gid < 0)
            pvm_errno = gid;
        return gid;
}
...
```

(pvmgsu_core.c) (CLIENT SIDE: send request)

```
...
int_query_server(group, request, caller, rvalue, optarg)
...
{
     if (pvm_send(stid, request) < 0)
...
}
```

(pvmgs_core.c) (SERVER SIDE: recv request)

```
...
while (1)
        {
            if (pvm_recv(-1, -1) < 0)  /* receive a request */

........

 case (JOIN):     /* join a group with the lowest avail gid */
    pvm_upkstr(groupname);
    gid = gs_join(groupname, tid, hash_list, ngroups);
........
    SENDINTRESULT(gid,tid,msgtag,''gs_handle(join)'');
...
```

(pvmgs_func.c) (SERVER SIDE: process request)

```
...
int
gs_join(gname, tid, hash_list, ngroups)
```

```
char *gname;
int tid;
GROUP_LIST_PTR hash_list;
int *ngroups;
{
    int inst, thishost, hostindx;
    GROUP_STRUCT_PTR group;
        if (gname == (char*) 0) /* check for valid group name  */
            return(PvmNullGroup);
        group = gs_group(gname, hash_list, ngroups, CREATE);

........

}
...
```

2.3 Comunicaciones Colectivas

En esta sección se decriben las funciones colectivas soportadas en PVM.

2.3.1 Descripción de Funciones

```
C          int info = pvm_mcast( int *tids, int ntask, int msgtag )
```

El *multicast* fue una de las primeras funciones colectivas que soportó PVM y no forma parte de la biblioteca de manejo de grupos `libgpvm` sino que es parte del núcleo de PVM.
Esta función hace el envío del mensaje alojado en el buffer actual a las tareas indicadas por `ntask` (Cantidad de tareas) y por `tids` (arreglo de identificadores de todos los destinos). El mensaje no es enviado bajo ninguna circunstancia al emisor, ni aún éste figurando en la lista de receptores. Para la recepción se puede llamar a `pvm_recv(3PVM)`, `pvm_nrecv(3PVM)` o alguna otra de sus variantes. El *multicast* es asincrónico (asynchronous) y en las "man pages" indica que está basado en el cálculo de un "minimum spanning tree" entre *pvmds* , aunque los resultados que se obtienen parecen contradecirlo, la forma que parece funcionar es haciendo puntos a puntos para cada *pvmd* . Los *pvmds* una vez que tienen los datos los distribuyen entre las tareas integrantes del grupo que están ejecutando sobre el mismo equipo. Antes de esto el "multicaster" rutea de forma directa los mensajes con las tareas que tienen "PvmRouteDirect" como forma de ruteo (TCP).
La semántica es como la de `pvm_send(3PVM)`, se pueden ver documentadas en las páginas de los manuales (man pages):

`pvm_send(3PVM)`:

"...The `pvm_send` routine is asynchronous. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processor..."

`pvm_mcast(3PVM):`

"...Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processors. This is in contrast to synchronous communication, during which computation on the sending processor halts until the matching receive is executed by the receiving processor ..."

```
C          int info = pvm_bcast( char *group, int msgtag )
```

Fue también una de las primeras funciones colectivas que soportó PVM. Esta función hace envío del mensaje alojado en el buffer actual a todas las tareas miembros del grupo indicado por el parámetro `group`. Según la documentación a partir de la versión 3.2 en adelante el mensaje no es mandado al remitente y este no necesita ser parte del grupo para hacer el *broadcast*. Es también asincrónico (asynchronous) como el *multicast*. Funciona de la siguiente manera: primero determina los miembros del grupo chequeando sobre la "group database" mantenida en el *pvmgs* y luego realiza un *multicast*. Si el grupo es cambiado durante su ejecución los cambios no son reflejados.

```
C          int info = pvm_barrier( char *group, int count )
```

Siendo también una de las primeras funciones colectivas que se encontrarón en PVM. Bloquea a la tarea que llama hasta que `count` miembros del grupo invocán a la función. El que llama debe ser miembro del grupo. Si `count` es `-1` indica que se deben sincronizar todos los miembros. Esta función permite la sincronización entre todos o algunos miembros de un grupo. Durante cualquier llamada siempre se debe invocar por todos los participantes con el mismo valor para `count`.

```
C          int info = pvm_scatter(void *result, void *data,
                                  int count, int datatype,
                                  int msgtag,
                                  char *group, int rootginst)
```

Produce que desde una tarea designada `root` por el parámetro `rootginst`, un conjunto de datos sea dividido en partes iguales y luego se reparta entre todas las tareas de un mismo grupo. Todos los miembros deben llamar a `pvm_scatter(3PVM)` y cada uno recibirá su porción de datos. La información se distribuye de la siguiente forma: La enésima tarea según su número de instancia que se puede obtener mediante `pvm_getinst(3PVM)`, recibe la enésima porción, empezando la cuenta desde cero.
`result`: es un puntero a un área de memoria sobre la cual se desea recibir la porción de tamaño `count`.

`data`: solo tiene sentido en la tarea `root` y es un puntero al arreglo de datos a ser distribuido entre todos los miembros.
La tarea `root` se indicada mediante números de instancia y no mediante `tid`.

Ejemplo ilustrativo:

$$i\epsilon\{0,1,2,...n-1\}$$
$$Starting = offset + i * count$$
$$Length = count$$

Se Tienen los siguientes valores:

- Count = 4
- Members = 3
- Elements = 12

```
[0 1 2 3| 4 5 6 7|8 9 10 11]
```

```
Tarea 0 recibe [0  1  2  3] i=0
Tarea 1 recibe [4  5  6  7] i=1 (S = v + 1 * 4 = v + 4) (L = 4)
Tarea 2 recibe [8  9 10 11] i=2
```

Como esta operación no utiliza los buffers de PVM no tiene disponible las diferentes políticas de codificación como *PvmDataInPlace*, *PvmDataRaw* y *PvmDataDefault* y se deben usar los siguientes tipos:

C datatypes	FORTRAN datatypes
PVM_BYTE	BYTE1
PVM_SHORT	INTEGER2
PVM_INT	INTEGER4
PVM_FLOAT	REAL4
PVM_CPLX	COMPLEX8
PVM_DOUBLE	REAL8
PVM_DCPLX	COMPLEX16
PVM_LONG	—

Las "man pages" mencionan que el algoritmo actual es muy simple y robusto a costa de perder eficiencia. Se comprometen en futuras versiones a mejorar este aspecto con respecto a las diferentes arquitecturas.

```
C          int info = pvm_gather(void *result,  void *data,
                                 int count, int datatype,
                                 int msgtag,
                                 char *group, int rootginst)
```

Esta función puede ser considerada como la inversa de la anterior, pues su labor es recoger todas las porciones que forman un conjunto de datos entre los integrantes de un grupo. De la misma forma que `pvm_scatter(3PVM)` debe ser llamada por todos los miembros y la tarea que tenga el número de instancia indicado en el argumento `rootinst` será la llamada `root` y hacia este deben fluir los datos de la operación. A medida que arriban las porciones se van ubicando de acuerdo al identificador dentro del grupo, de la misma forma que se explico con el *scatter*.
`result`: solo tiene sentido en la tarea `root` y es un puntero a una dirección de memoria donde se acumularán los valores enviados desde los distintos miembros.
`data`: para cada miembro del grupo es un puntero local a los datos a ser recogidos. Esta función es no bloqueante como se indica en las "man pages":

> `pvm_gather(3PVM)`:
>
> "... `pvm_gather()` does not block. If a task calls..."

Con respecto a los mecanismos de codificación comparte las mismas consideraciones realizadas con `pvm_scatter(3PVM)`.

```
C          int info = pvm_reduce(void (*func)(),
                                 void *data, int count,
                                 int datatype, int msgtag,
                                 char *group, int rootginst)
```

`pvm_reduce(3PVM)` sirve para realizar operaciones globales tales como max, min, sum, product o alguna otra provista por el usuario la cual involucre una función que dado un conjunto de valores los reduzca a uno solo. Todos los miembros del grupo deben llamarla con arreglos locales de igual dimensión. La tarea `root` o `rootginst` obtiene como resultado la operación aplicada al total de los datos elemento a elemento (element-wise). Es necesario que cada una de las operaciones discrimine entre los diferentes tipos de datos.
El parámetro `func` debe tener la siguiente forma en *C*:

```
void (*func)(int *datatype, void *y, void *x, int *num, int *info)

void    func(int *datatype, void *x, void *y, int *num, int *info)
```

`x` e `y` son arreglos del tipo especificado por `datatype` con `num` entradas e `info` es el valor de retorno que devuelve la invocación de la misma mediante el cual se indica el estado con el cual concluyó.

Ejemplo ilustrativo:

Suponga que hay un grupo con 4 miembros, 0, 1, 2 y 3 y el `root` será 3:

```
Tarea 0  con los valores:    [  1   5   7 ]
Tarea 1        ``            [  5  12   7 ]
Tarea 2        ``            [  0   9   3 ]
Tarea 3        ``            [ 10  56  19 ]
```

Y se llama con `PvmMin` de tipo `PVM_INT` con tamaño 3, el resultado será:

```
Tarea 0  con los valores:     [  undefined ]
Tarea 1        ``             [  undefined ]
Tarea 2        ``             [  undefined ]
Tarea 3        ``             [  0   5   3 ]
```

Esta operación es no bloqueante y comparte la carencia de uso de buffers de PVM como sus pares `pvm_scatter(3PVM)` y `pvm_gather(3PVM)`. Si el grupo cambia durante su invocación los resultados son indefinidos.

`pvm_reduce(3PVM)`:

"...Caveat: `pvm_reduce()` does not block, a call to `pvm_barrier` may be necessary. For example, an error may occur if a task calls `pvm_reduce` and then leaves the group before the root has completed its call to `pvm_reduce`. Similarly, an error may occur if a task joins the group after the root has issued its call to `pvm_reduce`. Synchronization of the tasks (such as a call to `pvm_barrier`) was not included within the `pvm_reduce` implementation since this overhead is unnecessary in many user codes (which may already synchronize the tasks for other purposes)..."

2.3.2 Cómo Funcionan Internamente

Para esta sección se analiza la última versión disponible de PVM, la cual, para el momento es la 3.4.3.

Multicast

Esta funcionalidad es implementada por la operación `pvm_mcast(3PVM)` la cual es referenciada por varios archivos fuente pero los que interesan analizar son `.../src/lpvm.c, lpvmgen.c`.

- `lpvmgen.c`: (Funciones genéricas)

  Aquí se encuentra la definición de `pvm_mcast(tids, count, tag)` la cual realiza el siguiente algoritmo:

  1. Controla parámetros.
  2. Llama a `pvmmcast()` definida en `lpvm.c`

- `lpvm.c`: (El que da origen del núcleo,"libpvm", para ambientes Unix[UX])

  Aquí se encuentra la definición de `pvmmcast(mid, tids, count, tag)` :

1. Hace una lista ordenada de los destinos.
2. Saca duplicados.
3. Se remueve así mismo de la lista si se encuentra.
4. Envía sobre rutas directas usando comunicaciones punto a punto.
5. Envía al *pvmd* para que sea distribuido hacia el resto.

```
qsort(
      (void *)dst,
      (size_t)count,
      sizeof(int),
      int_compare);

/*
 * remove duplicates
 */
 j = 0;
 for (i = 1; i < count; i++)
              if (dst[i] != dst[j])
                  dst[++j] = dst[i];
 count = j + 1;

 ...
 for (i = 0; i < count; i++)
     {
      ...
      /* send message */
         if (cc >= 0)
            if ((cc = mroute(pvmsbuf->m_mid, pvmmytid |
                             TIDGID, tag, &ztv)) > 0)
      ...
      if (count > 0) {
      ...
   /* send message to pvmd */
      if (cc >= 0)
             mroute(...)
```

Broadcast

Es implementada por la operación `pvm_bcast(3PVM)` la cual esta definida en la biblioteca de grupos "libgpvm" y tiene sus fuentes en `.../pvmgs/pvmgsu_core.c`. Su funcionamiento es el siguiente:

1. Obtiene la lista de `tids` de las tareas en el grupo:

```
gs_get_tidlist(group, msgtag, &ntids, &tids, 0)
```

2. Recorre la lista de `tids` y se saca la tarea que llama:

```
for (i = 0; i < ntids; i++)
            ...
            if (tids[i] == mytid)
            ...
```

3. Hace un *multicast* con los restantes:

```
cc = pvm_mcast(tids, ntids, msgtag)
```

Barrier

Implementada por `pvm_barrier(3PVM)`.En esta operación se puede ver cómo se usa la precompilación condicional verificando el tipo de arquitectura sobre la que se esta ejecutando, por ejemplo se observa que si se corre sobre un equipo "Intel Paragon"[PGON] se usa las primitivas de bajo nivel soportadas. Sobre el "Paragon" las "NX" nativas son usadas si se dan las condiciones. El código fuente de esta operación se encuentra `.../pvmgs/pvmgsu_core.c`

```
#if defined(IMA_PGON)
      ...
#endif
```

Su funcionamiento es el siguiente:

1. Hace la llamada al *pvmgs* al cual le envía el "request":

```
int_query_server(group, BARRIER, ''pvm_barrier'', &rc, cnt);
        ....
```

2. El *pvmgs* la maneja mediante un case y llama a `gs_barrier()` (interna al *pvmgs*):

```
case (BARRIER):
        case (BARRIERV):
        ...
        cc = gs_barrier( groupname, msgtag, cnt, tid,
                         hash_list, ngroups );
        ...
```

3. En `gs_barrier()` se hace el siguiente manejo:
   (a) Chequea que el grupo exista.
   (b) Chequea que el que llama sea miembro del grupo.

(c) Si el `count` es `-1` pone el `count` a la cantidad de miembros del grupo.

(d) Chequea si la llamada al *barrier* es la primera para resetear todo o incrementa `barrier_reached` (información mantenida por el *pvmgs* para cada grupo).

(e) Al llegar a la cuenta se hace un *multicast* para dejar seguir a los participantes y se asigna `barrier_count = -1`.

(f) Si el `tag` del mensaje es `BARRIERV` usa un coordinador por host. Esto es en el caso de `PGON` (Intel Paragon).

(g) El llamador se había bloqueado hasta recibir el mensaje del *multicast*.

```
        ....
pvm_recv(stid, request)

        /* Aca sigue despues de recibir el multicast */
        ....
```

Scatter

Las siguientes funciones fueron adicionadas más tarde a partir de la versión 3.3 y en su ejecución ninguna es manipulada por código del central de PVM sino que se encuentran en `pvmgsu_aux.c` donde se definen rutinas auxiliares a la biblioteca de grupos.

1. Determina si es `root`:

```
        ...
if (myginst == rootinst)
        ...
```

2. Si lo es, reparte la información a todos y se copia su parte localmente

```
...
for (i=0; i<gsize; i++)
  {
      if (i == myginst)
        BCOPY((char *) data + i*datasize*count,
              (char *) result, datasize*count);
      else
      {
           ...
           pvm_send( tids[i], msgtag)
           ...
```

3. Sino recibe su porción:

```
        ...
pvm_recv( roottid, msgtag )
        ...
```

Gather

1. Determina si es `root`:

```
        ...
if (myginst == rootinst)
        ...
```

2. Si lo es, recibe la información desde todos y copia su parte localmente:

```
      ...
      for (i=0; i<gsize; i++)
        {
            if (i == myginst)
            {
             BCOPY((char *) data, (char *) result +
                   i*datasize*count, datasize*count);
            }
            else
            {
             ...
             pvm_recv( tids[i], msgtag )
             ...
```

3. Sino envía su porción:

```
        ...
pvm_send( roottid, msgtag)
        ...
```

Reduce

La operación reduce se implementa de forma un tanto extraña. Por cada máquina que integra la operación hay una tarea designada coordinadora, que será la encargada de hacer la parte del *reduce* correspondiente a todos los participantes ejecutando en el mismo equipo. Una vez ejecutada la parte del *reduce* envía los datos hacia la tarea `root`, la cual combinará todas las porciones recibidas. Si solo existe una tarea por máquina participando en la operación dará como

resultado que no se ejecute en forma paralela ya que cada participante solo enviará los datos a `root` el cual será el encargado de hacer todo el cómputo (En esta situación todas la tareas son coordinadoras en su equipo).

1. Chequea si es coordinador y hay más de una tarea en el equipo, de ser verdad realiza una parte de la operación recibiendo las porciones desde las tareas locales

```
if ((pvmmytid==coordinator) && (nmembers_on_host>1))
     {
         /* recv data from other group members on
            same host, perform func */
         for (cnt = nmembers_on_host-1; cnt>0; cnt--)
         {
             if ((cc = pvm_recv(-1, msgtag) )<PvmOk)
                goto done;
             ...
             (*func)( &datatype, data, work, &count, &cc );
             ...
         }
     }
```

2. Si no es coordinador, entonces existe al menos otra tarea más en el mismo equipo que cumple el rol de coordinador, por lo cual envía sus datos a esta

```
     else if (pvmmytid != coordinator)
     {
         /* send data to the data coordinator on this
            same host */
         ...
         if ((cc = pvm_send( coordinator, msgtag))<PvmOk)
            goto done;
     }
```

3. Si es coordinador y no es `root` envía los datos a `root`

```
if ((pvmmytid==coordinator) && (pvmmytid != roottid))
    {
        /* send data to the roottid for the reduce
           operation */
        ...
        if ((cc = pvm_send( roottid, msgtag))<PvmOk)
           goto done;
    }
```

4. Si es `root` recibe primero del coordinador local si es que esta función fue delegada a otra tarea, luego recibe desde los demás coordinadores y efectúa la reducción final.

```
if (pvmmytid == roottid)
  {
      /* if root isn't the host coordinator,
         receive from coordinator 1st */
      if  (pvmmytid != coordinator)
      {
          if ((cc = pvm_recv(coordinator, msgtag) )<PvmOk)
             goto done;
          if ((cc = (*unpackfunc)( data, count, 1))<PvmOk)
             goto done;
      }

      if (nhosts_in_group-- <= 0) goto done;

      /* recv data from other group members on
         diff host, perform func */
      for (cnt = nhosts_in_group; cnt>0; cnt--)
      {
          if ((cc = pvm_recv(-1, msgtag) )<PvmOk)
             goto done;
          if ((cc=(*unpackfunc)( work, count, 1))<PvmOk)
             goto done;
          (*func)( &datatype, data, work, &count, &cc );
          if (cc < PvmOk) /* error flag from func */
             goto done;
      }   /* end */
```

Bibliografía

[PVM1] PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing , Al Geist , Adam Beguelin , Jack Dongarra , Weicheng Jiang , Robert Manchek and Vaidy Sunderam - The MIT Press, Cambridge, Massachusetts - 1994.

[PVM2] PVM 3 user's guide and reference manual. Technical - A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[PVM3] Recent Enhancements to PVM (www.netlib.org/utk/papers/pvm-ijsa/ijsa.html) - Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Vaidy Sunderam, Dec 1994.

[Tntti] Cómputo Paralelo en Redes de Estaciones de Trabajo para Aplicaciones Basadas en Algebra Lineal -Fernando G. Tinetti -Rep. Técnico PP003-01 - Julio 2001.

[BWULF] Beowulf HOWTO - Jacek Radajewski and Douglas Eadline -v1.1.1, 22 November 1998

[MPI] MPI: The Complete Reference - Marc Snier, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra - The MIT Press, Cambridge, Massachusetts - 1996

[UX] An Overview of UNIX - Jehan-François Pâris - Computer Science Department - University of Huston

[PGON] Intel Paragon - Supercomputer Systems Division - Intel Corporation - Paragon XP/S Product Overview - 1991

Capítulo 3

Análisis de Grupos y Comunicaciones colectivas en MPI

3.1 Introducción

El concepto clave en comunicaciones colectivas es el grupo, pero las rutinas en MPI no tiene uno de estos como como argumento, en su lugar lleva un comunicador, en inglés *communicator*, por esta razón es util pensar en el *communicator* como el grupo.
Antes de comenzar con la descripción de los grupos en MPI es conveniente dar un panorama sobre su filosofía. Lo primero que es necesario destacar es el hecho de que cualquier comunicación, sea punto a punto o colectiva, debe efectuarse a través de un *communicator*. En general, el manejo de grupos en MPI es bastante poco natural, característica que se acentúa más si el lector proviene de un entorno como PVM. Los *communicators* son objetos que abarcan un grupo de procesos y donde cada uno de estos son identificados con valores llamados `ranks` que valen entre 0..size-1. Siempre un proceso al iniciar pertenece a un grupo global que esta contenido dentro del *communicator* `MPI_COMM_WORLD` y a partir de este es que se derivan los nuevos *communicators*. Cada ejecución es independiente y posee un espacio de nombres global (`ranks en MPI_COMM_WORLD`) disjunto, es decir que se tiene diferentes mundos para diferentes aplicaciones que pueden ejecutar al mismo tiempo. Esto contrasta con PVM que maneja la asignación de `tids` de forma global para la misma máquina virtual.
MPI[MPI][MPI2] (Message Passing Interface) es un sistema portable que define la sintaxis y semántica de un conjunto de rutinas para construir aplicaciones paralelas/distribuidas basadas en el paradigma de pasaje de mensajes. A diferencia de PVM[PVM1], que es un estándar de facto, MPI fue estandarizado formalmente en un proceso que involucró 80 personas de 40 organizaciones internacionales diferentes que diseñaron y definieron una interfaz, dando lugar a distintas implementaciones. Desde junio de 1994, MPI ha ido ganando terreno en el área de las SPCs (Scalable Parallel Computers) con memoria distribuida, y sobre las NOWs (Networks of Workstations) alcanzando tantos adeptos

como los tiene PVM. Sus mayores logros han sido la portabilidad que ofrece y la compatibilidad de ejecutar transparentemente en sistemas heterogéneos, cabe destacar que este último ítem depende de la implementación. Otra característica no menos importante es que su dieño admite implementaciones eficientes[MPI], dando lugar a comunicaciones más rápidas. Es de esperar que MPI sea más veloz que PVM sobre grandes multiprocesadores, además de poseer más funciones para las comunicaciones colectivas y punto a punto[PVM-MPI1].

3.2 Manejo de grupos

Todos los mensajes en MPI poseen un envoltorio (en inglés envelope) básico que acompaña a los datos al momento de ser enviados, los atributos de este son:

- origen (source)
- destino (destination)
- etiqueta (tag)
- comunicador (communicator)

Este último es un objeto local que representa un domino de comunicaciones, es una abstracción que abarca el concepto de grupo, una estructura distribuida que indica qué proceso puede comunicarse con cuál y permite el intercambio efectivo de mensajes. Las aplicaciones a la hora de manejar grupos y comunicar procesos mediante operaciones colectivas deben recurrir a estos recursos. Debido a esto la primera parte de este capítulo introduce y describe este concepto.
Formalmente un *communicator* es un objeto opaco con una serie de atributos, además de reglas que regulan su ciclo de vida. Existen dos clases de *communicators*:

Intracommunicators: son aquellos que sirven para lograr comunicación dentro de un único grupo y tienen dos atributos: el grupo de procesos y la topología. Estos son usados tanto para operaciones punto a punto como colectivas entre sus miembros.

Intercommunicators: son aquellos que sirven para realizar comunicaciones punto a punto[1] entre procesos de grupos diferentes. Los atributos son los dos grupos que comunica que en general se los referencia como *left* y *right* o *local* y *remote*.

Además de los atributos mencionados un *communicator* puede tener otros agregados por el usuario o valores asociados a los mecanismos de "caching".
Su manejo es bastante estricto dando como resultado un esquema estático que contrasta fuertemente con el dinamismo provisto por PVM. En realidad los grupos comparados con los de PVM son realmente diferentes objetos aunque tienen similitudes superficiales[PVM-MPI2]. La falta de flexibilidad de MPI se debe a que el rendimiento tuvo más prioridad en su diseño[PVM-MPI1], de hecho, específicamente fue diseñado estático para lograr mejores resultados además de asegurar que las operaciones sean seguras (safety) y evitar los conflictos entre mensajes de otros módulos.

[1] en MPI-2 se admiten colectivas

3.2.1 Descripción de Funciones de Grupos

El objetivo de esta sección no es tratar exhaustivamente todas las funciones sobre grupos y *communicators* provistas por MPI, lo que se pretende es estudiar como trabajar con estas abstracciones y describir aquellas operaciones que se crean útiles para entender su funcionamiento.
Es necesario primero enumerar qué funcionalidad se requiere y, luego a partir de esta lista ver cómo lo resuelve MPI. Se necesitan operaciones para: crear grupos, obtener información de sus propiedades, tales como la cantidad de miembros y liberarse de un grupo.
Primero se ven los grupos propiamente dichos y luego los *communicators* y sus relación con los primeros. La sintaxis usada es la del lenguaje *C*[Kernighan 78][2].

Grupos

Constructores Los constructores de grupos sólo pueden ser usados a partir de otros ya existentes para crear nuevos mediante operaciones de conjuntos. No existen mecanismos para crear grupos de la nada. Los grupos son construidos de forma local, lo que evita la comunicación para la distribución de los datos. Estas funciones deben ser llamadas por todos los participantes del programa, aún estos no estando contemplados como miembros del nuevo grupo.

```
C           int MPI_Comm_group(MPI_Comm comm, MPI_Group *pgroup)
```

Devuelve un manejador `pgroup` de grupo para el *communicator* `comm`. A partir de éste es que se pueden construir los nuevos grupos. En general se pide el manejador del `MPI_COMM_WORLD`, que es un *default communicator* que define el domino inicial para todos los procesos que participan en el cómputo.
Además de `MPI_COMM_WORLD` existen en MPI otras constantes, para *communicators*: `MPI_COMM_SELF` que es un *communicator* pre-definido donde el único integrante del grupo es el proceso que lo referencia y `MPI_COMM_NULL` que indica que el objeto es inválido debido a que a sido liberado o creado de forma incorrecta. Para grupos los objetos pre-definidos son: `MPI_GROUP_NULL` y `MPI_GROUP_EMPTY`, siendo el primero un grupo inválido y el segundo un grupo vacío respectivamente.

```
C           int MPI_Group_union(MPI_Group g1, MPI_Group g2,
                                MPI_Group *pgu)

C           int MPI_Group_difference(MPI_Group g1, MPI_Group g2,
                                     MPI_Group *pgd)
```

[2]MPI esta definido además para Fortran y *C++*

```
C          int MPI_Group_intersection(MPI_Group g1, MPI_Group g2,
                                      MPI_Group *pgi)
```

Estas son operaciones tradicionales sobre conjuntos pero con la diferencia que acá trabajan sobre grupos donde el orden de los elementos es importante y el cual es conservado mediante los `ranks`[MPI]. La primera da como resultado un grupo `pgu` donde sus miembros son: primero los de `g1` y luego los de `g2`. La segunda genera el grupo `pgd` donde los elementos son los de `g1` pero que no están en `g2` respetando el orden de `g1` y la última la intersección de ambos grupos también respetando el orden. Las dos últimas funciones podrían devolver el grupo `MPI_GROUP_EMPTY`.

```
C          int MPI_Group_incl(MPI_Group g, int n, int *ranks,
                              MPI_Group *png)

C          int MPI_Group_excl(MPI_Group g, int n, int *ranks,
                              MPI_Group *png)
```

Estas funciones sirven para que, a partir de un grupo base se cree uno nuevo incluyendo o sacando miembros. La primera función toma solo los miembros de `g` indicados por el parámetro `ranks`. La segunda toma todos los miembros pero saca los señalados por `ranks`.

Consultores (Getters - Accessors) Hasta ahora todas las funciones que se mencionarón no tienen su igual en PVM, esto se debe a que todos los grupos en MPI son manejados localmente, pero también existen funciones que tienen sus similares como es el caso de los "accessors".

```
C          int MPI_Group_size(MPI_Group group, int *psize)
```

Esta función retorna en `psize` el número de procesos en el grupo. Su similar en PVM sería `pvm_gsize(3PVM)`.

```
C          MPI_Group_rank(MPI_Group group, int *prank)
```

Esta otra función retorna en `prank` el "ranking" del proceso invocador dentro del grupo. El "ranking" en MPI cumple el rol de número de instancia y `tid` en PVM. Si el proceso no es miembro se retorna `MPI_UNDEFINED`. Su similar en PVM sería `pvm_getinst(3PVM)`.

Destructor Cuando un manejador de grupo no se usa más es necesario explícitamente liberarlo esto se hace llamando a la función:

```
C           int MPI_Group_free(MPI_Group *grp)
```

Una vez invocada el grupo es marcado para ser liberado y el manejador es asignado a `MPI_GROUP_NULL`. Se podría comparar con `pvm_lvgroup(3PVM)`, pero difiere en el hecho de que el evento producido no se propaga a los demás miembros del grupo, solo se resuelve localmente. Para cerrar la sub-seccíon se muestra un ejemplo práctico:

Ejemplo ilustrativo:

Supongase que se tienen nueve procesos "rankeados" del 0 al 8 y se quiere asignarlos a tres grupos de la siguiente manera:

```
Grupo A  con los procesos:    [  0   3   6 ]
Grupo B        ‘‘             [  1   4   7 ]
Grupo C        ‘‘             [  2   5   8 ]
```

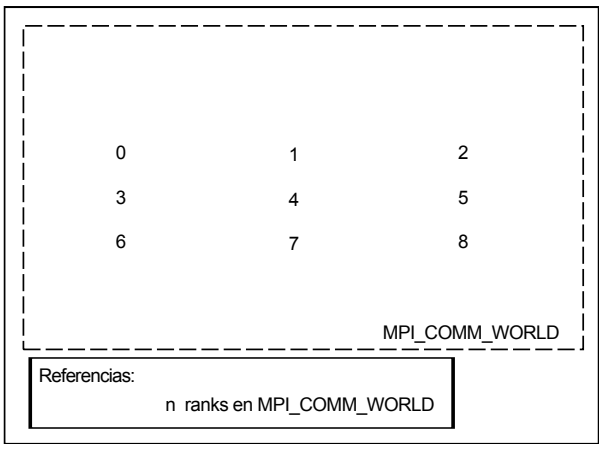

Figura 3.1: Estructura antes de ejecutar las primitivas de grupos.

Una posible declaración del programa que debe ser ejecutado en todos los miembros sería:

```
#include <stdio.h>
#include <mpi.h>

#define NTASKS    3

int main(int argc,char *argv[])
```

```
{
        int                 i,rank,j;
        int                 ntasks = NTASKS;
        int                 ranksA[NTASKS];
        int                 ranksB[NTASKS];
        int                 ranksC[NTASKS];
        int                 size;
        MPI_Group           group,grp_A,grp_B,grp_C;

        /** Initialize MPI **/
        MPI_Init( NULL, NULL );

        /** Get rank **/
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        if ( rank<0 )
        {
             MPI_Finalize();
             perror(``Error at getting rank'');
             exit(1);
        }

        /** Get Global group handler **/
        MPI_Comm_group(MPI_COMM_WORLD, &group);

        /** Build ``A'', ``B'' and ``C'' member's ranks array **/
        j=0;
        for(i=0;i<ntasks;i++)
         {
           ranksA[i] = j;
           j++;
           ranksB[i] = j;
           j++;
           ranksC[i] = j;
           j++;
         }

         /** Create first new group (A) **/
         MPI_Group_incl(group, ntasks, ranksA, &grp_A);
         /** Create second new group (B) **/
         MPI_Group_incl(group, ntasks, ranksB, &grp_B);
         /** Create last new group (C) **/
         MPI_Group_incl(group, ntasks, ranksC, &grp_C);

         if (rank==0)
         {
         /** get new groups size **/
         MPI_Group_size(grp_A,&size);
         printf(``A members %d from rank %d \n'',size,rank);
         MPI_Group_size(grp_B,&size);
         printf(``B members %d from rank %d \n'',size,rank);
```

```
        MPI_Group_size(grp_C,&size);
        printf(''C members %d from rank %d \n'',size,rank);
        }

        /** Free groups **/
        MPI_Group_free(&grp_A);
        MPI_Group_free(&grp_B);
        MPI_Group_free(&grp_C);

        /** Bye, bye **/
        MPI_Finalize();
        return(0);
}
```

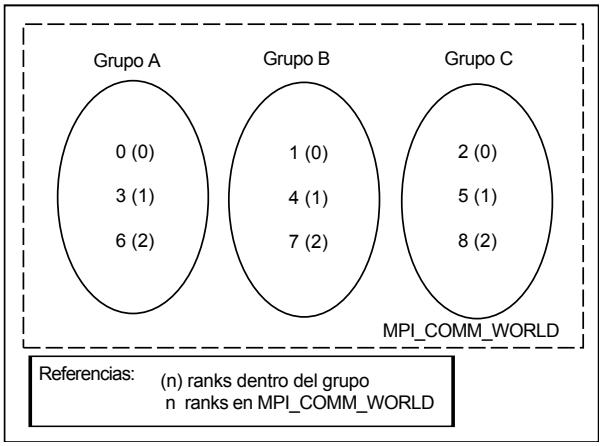

Figura 3.2: Estructura después de ejecutar las primitivas de grupos.

Comunicadores (*Communicators*)

Hasta ahora se vio como manejar grupos, pero la aplicación para poder iniciar una comunicación necesita hacerlo mediante *communicators*, para esto es necesario poder armar estos objetos y combinarlos con los grupos. Las operaciones que acceden a los *communicators* son locales como las de grupos pero las de creación son colectivas y requieren comunicación entre procesos. Primero se describe las funciones sobre *intracommunicators* y más tarde se extienden a *intercommunicators*.

Constructores

```
C           int MPI_Comm_dup(MPI_Comm comm, MPI_Comm* newcomm)

C           int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
                                MPI_Comm *newcomm)
```

La primera duplica un *communicator* existente con toda su información "cacheada", sus atributos (grupo(s), topología) y los valores adicionados por el usuario. A partir de `comm` genera `newcomm`. El nuevo objeto define un nuevo dominio de comunicación. Como lo define el estándar esta es una llamada colectiva sobre todos los procesos en `comm`. En general se implementa como una operación sincrónica para evitar el problema de tener mensajes que llegan a *communicators* que aún no han sido creados. Aplicable tanto a *intracommunicators* como a *intercommunicators*.
La segunda crea un nuevo *intracommunicator* a partir de `comm` con los miembros definidos en `group`. A diferencia de la anterior los atributos no son propagados. Si el proceso que llama no está en el grupo `group` devuelve `MPI_COMM_NULL`, la llamada es errónea si los parámetros con los cuales es invocada no son los mismos en todos los miembros o si `group` no es un subconjunto del grupo asociado a `comm`. Válida solo sobre *intracommunicators*.
Otro constructor que solo se menciona es `MPI_Comm_split` que crea nuevos *communicators* particionando uno existente mediante la asignación de valores enumerativos, llamados colores y claves. Solo aplicable a *intracommunicators*.

Consultores (Getters - Accessors)

```
C           int MPI_Comm_group(MPI_Comm comm, MPI_Group *pgroup)

C           int MPI_Comm_size(MPI_Comm comm, int *psize)

C           int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

La primera ya fue tratada con en la sub-sección de grupos.
`MPI_Comm_size` devuelve la cantidad de miembros que tiene el grupo asociado con `comm`. Es equivalente a obtener el grupo de `comm` accederlo mediante `MPI_Group_size` y luego liberarlo con `MPI_Group_free`. Aplicable a *intracommunicators*.
`MPI_Comm_rank` indica el `rank` dentro del grupo asociado a `comm` del proceso que llama. El valor devuelto esta entre `0..size-1`. En general se llama al principio del programa `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` obteniendo el "rank" absoluto del grupo global de `MPI_COMM_WORLD`. Es equivalente a obtener el grupo de `comm` accederlo mediante `MPI_Group_rank` y luego liberarlo con `MPI_Group_free`. Aplicable a *intracommunicators*.

Destructor

```
C           int MPI_Comm_free(MPI_Comm *comm)
```

Cuando un *communicator* no se usa más debe ser liberado, esto se hace llamando a `MPI_Comm_free`. Esta función marca el objeto para ser liberado y asigna al

manejador `MPI_COMM_NULL`. Es aplicada a *intracommunicators* y *intercommunicators.*

Extensión a *intercommunicators* Si bien todo lo descripto hasta ahora alcanza como herramienta para trabajar con las operaciones colectivas, se describe esta extensión por completitud ya que esta definida en MPI. Los *intercommunicators* involucran comunicaciones desde un proceso en un grupo con otro(s) procesos en otro grupo. las operaciones disponibles son:

Constructores

```
C                  int MPI_Intercomm_create(MPI_Comm lcomm,
                                            int lleader,
                                            MPI_Comm pcomm,
                                            int pleader,
                                            int tag,
                                            MPI_Comm *newcomm)
```

Crea un *intercommunicator* a partir de dos *intracommunicators.* La llamada es colectiva sobre la unión de los dos grupos. Por cada grupo de *intracommunicator* se obtiene un *intercommunicator* diferente, estos son bidireccionales (se pueden usar tanto para enviar como para recibir). Debido a que cada grupo obtiene uno diferente para lograr la comunicación se deben crear de a pares uno para cada lado (*left* y *right*). Los parámetros son:
`lcomm`: *intracommunicator* local.
`lleader`: `rank` del proceso denominado líder en el grupo local de `lcomm`.
`pcomm`: *intracommunicator* remoto o *bridge.* Debe ser un objeto que permita relacionar a los dos grupos, por ejemplo `MPI_COMM_WORLD`.
`pleader`: `rank` del proceso líder en el grupo remoto.
`tag`: tag del mensaje para permitir múltiples creaciones de *intercommunicators* con los mismo líderes de forma segura (safe).
`newcomm`: nuevo *intercommunicator* (manejador). Para realizar un enlace bidireccional se deben crear dos *intracommunicators.* Todos los proceso deben llamar con el mismo *intracommunicator* local (no el mismo manejador, cuestión que sería imposible en espacios de memoria diferentes) y líder local. Los dos proceso líderes deben especificar el mismo *bridge* e idénticas etiquetas.

```
C                  int MPI_Intercomm_merge(MPI_Comm comm, int high,
                                           MPI_Comm *newcomm)
```

Esta función en realidad es un constructor de *intracommunicator.* Toma los dos grupos que conecta el *intercommunicator* `comm` y produce un *intra* nuevo a partir de la unión de estos. El estándar define que debe ser bloqueante y colectiva en la unión de los dos grupos.

high es un valor lógico, todos los procesos dentro del mismo grupo deben usar el mismo valor para este parámetro. si los dos grupos proveen el mismo valor para high el orden es arbitrario, sino se ponen primero los que tengan high=false (0).

Consultores (Getters - Accessors)

```
C           int MPI_Comm_test_inter(MPI_Comm comm, int *flag)

C           int MPI_Comm_remote_size(MPI_Comm comm, int *psize)

C           int MPI_Comm_remote_group(MPI_Comm comm,
                                      MPI_Group *pgroup)
```

La primera función indica si el *communicator* es *inter* o *intracommunicator*.
La segunda devuelve el tamaño del grupo en el *communicator* remoto.
La última función devuelve el grupo del *communicator* remoto.
Para terminar con esta sub-sección se ve un ejemplo continuando con el dado anteriormente.

Ejemplo ilustrativo:

En el ejemplo se muestra una forma de comunicar los 3 grupos construidos anteriormente a través de *intercommunicators*.
Hasta ahora se tiene a los nueve procesos agrupados en tres grupos nombrados A, B y C. Antes de poder hacer algo con estos se debe crear y asignarles sus respectivos *communicators*:

```
...
/** intras **/
MPI_Comm comm_A, comm_B, comm_C;
/** inters left side **/
MPI_Comm left_1_inter, left_2_inter;
/** inters right side **/
MPI_Comm right_1_inter, right_2_inter;

...
/** Create intra-Comms **/
MPI_Comm_create(MPI_COMM_WORLD, grp_A, &comm_A);
MPI_Comm_create(MPI_COMM_WORLD, grp_B, &comm_B);
MPI_Comm_create(MPI_COMM_WORLD, grp_C, &comm_C);
...
```

Las llamadas son hechas por todos los procesos, aún estos no perteneciendo a los grupos. En este caso el valor asignado al manejador del *intracommunicator* es MPI_COMM_NULL. Luego cada grupo de procesos creará el *intercommunicator* de forma colectiva.

```
...
/** Each group create inter-comms **/
if ((rank%3)==0)
{
   MPI_Intercomm_create(comm_A, 0, MPI_COMM_WORLD, 1, 1,
                        &left_1_inter);
}
else if ((rank%3)==1)
{
   MPI_Intercomm_create(comm_B, 0, MPI_COMM_WORLD, 0, 1,
                        &left_2_inter);
   MPI_Intercomm_create(comm_B, 0, MPI_COMM_WORLD, 2, 2,
                        &right_2_inter);
}
if ((rank%3)==2)
{
   MPI_Intercomm_create(comm_C, 0, MPI_COMM_WORLD, 1, 2,
                        &right_1_inter);
}
...
```

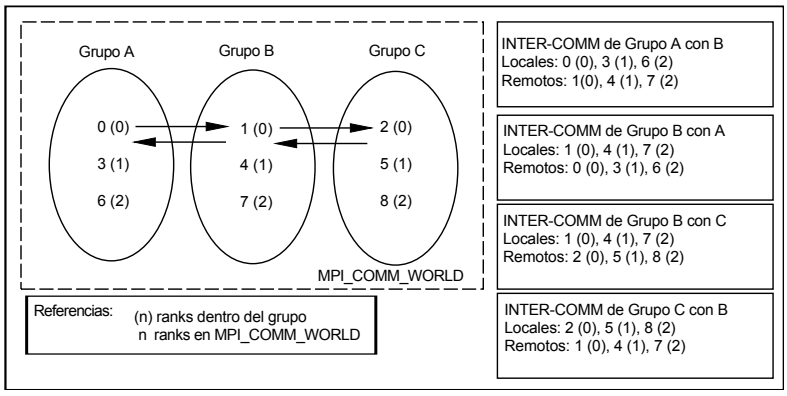

Figura 3.3: Estructura después de ejecutar las primitivas de *intercommunicators*.

3.2.2 Cómo Funcionan Internamente

Debido a que MPI es una definición y no una implementación, a la hora de analizar el código se debe elegir entre una de las tantas que son de dominio público. Los dos más usados son el LAM/MPI[LAM] y el MPICH[MPICH]. En el trabajo de decidió analizar LAM debido a que la organización del código es bastante modular y desde las páginas de los manuales (man pages) se hace referencia a dónde esta desarrollada cada operación.

Estructura de grupos

La implementación de los grupos es mediante entidades locales, lo que los hace muy sencillos. En LAM/MPI esta diseñado así:

```
(mpi.h)

...
typedef struct _group           *MPI_Group;
...

(mpisys.h)

...
struct _group {
        int             g_nprocs;               /* # processes */
        int             g_myrank;               /* my local rank */
        int             g_refcount;             /* reference count */
        int             g_f77handle;            /* F77 handle */
        struct _proc    **g_procs;              /* processes */
};
...
```

Los puntos al final y al principio significan que se suprimió una porción del código.

Cómo se resuelven las operaciones

Se mostrará solo una función de grupo debido a que el resto son similares en cuanto a que el manejo de la información es local.

```
(cgroup.c)

...
int MPI_Comm_group(MPI_Comm comm, MPI_Group *pgroup)
{
        lam_initerr();
        lam_setfunc(BLKMPICOMMGROUP);

        if (comm == MPI_COMM_NULL) {
                return(lam_errfunc(MPI_COMM_WORLD,
                        BLKMPICOMMGROUP, lam_mkerr(MPI_ERR_COMM, 0)));
        }

        if (pgroup == 0) {
                return(lam_errfunc(comm,
                        BLKMPICOMMGROUP, lam_mkerr(MPI_ERR_ARG, 0)));
        }
```

```
        *pgroup = comm->c_group;
        comm->c_group->g_refcount++;

        lam_resetfunc(BLKMPICOMMGROUP);
        return(MPI_SUCCESS);
}
```

La operación es bastante simple, controla que el manejador sea válido, si es así incrementa la cantidad de referencias al *communicator* y pone en pgroup.

Estructura de *communicators*

```
(mpi.h)

...
typedef struct _comm *MPI_Comm;
...

(mpisys.h)

...
struct _comm {
int c_flags; /* properties */
#define LAM_CINTER 0x10 /* intercommunicator? */
#define LAM_CLDEAD 0x20 /* local group dead? */
#define LAM_CRDEAD 0x40 /* remote group dead? */
#define LAM_CFAKE 0x80 /* fake IMPI comms */
```

Contexto y cantidad de manejadores

```
int c_contextid; /* context ID */
int c_refcount; /* reference count */
```

Grupo local y si es inter, grupo remoto

```
MPI_Group c_group; /* local group */
MPI_Group c_rgroup; /* remote group */
```

Tabla de hash para los atributos de usuario

```
HASH *c_keys; /* keys cache hash table */
```

Información acerca de la topología

```
int c_cube_dim; /* inscribing cube dim. */
int c_topo_type; /* topology type */
int c_topo_nprocs; /* # topo. processes */
int c_topo_ndims; /* # cart. dimensions */
int c_topo_nedges; /* # graph edges */
int *c_topo_dims; /* cart. dimensions */
int *c_topo_coords; /* cart. coordinates */
int *c_topo_index; /* graph indices */
int *c_topo_edges; /* graph edges */
```

Atributos extras

```
int c_f77handle; /* F77 handle */
MPI_Win c_window; /* window (if any) on comm */
MPI_Errhandler c_errhdl; /* error handler */
char c_name[MPI_MAX_OBJECT_NAME];
  /* This is not #if WANT_IMPI'ed out so that we can stay binary
     compatible with non-IMPI compiled programs */
        MPI_Comm        c_shadow;               /* shadow com for IMPI comms */
        long c_reserved[4]; /* for expansion */
};
...
```

Cómo se resuelven las operaciones

Solo se estudia una función para mostrar y destacar que los constructores se resuelve de forma colectiva involucrando a todos los procesos. En el caso de LAM/MPI se usan las mismas primitivas de MPI para comunicar a los participantes y se "deja la puerta abierta" mediante una directiva al precompilador para lograr la interoperabilidad entre diferentes implementaciones con IMPI[IMPI].
Para analizar mejor se divide el código en porciones:

(ccreate.c)

```
...
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
    MPI_Comm *newcomm)
{
int mycid; /* local max context ID */
int cid; /* global max context ID */
int rank; /* process rank */
int err; /* error code */

lam_initerr();
lam_setfunc(BLKMPICOMMCREATE);
```

Chequea los argumentos

Controla que el *communicator* origen no sea inválido o que no sea un *inter-communicator*.

```
/*
 * Check the arguments.
 */
if (comm == MPI_COMM_NULL) {
return(lam_errfunc(MPI_COMM_WORLD, BLKMPICOMMCREATE,
lam_mkerr(MPI_ERR_COMM, 0)));
}

if (LAM_IS_INTER(comm)) {
return(lam_errfunc(comm, BLKMPICOMMCREATE,
lam_mkerr(MPI_ERR_COMM, 0)));
}

if (group == MPI_GROUP_NULL) {
return(lam_errfunc(comm, BLKMPICOMMCREATE,
lam_mkerr(MPI_ERR_GROUP, 0)));
}

if (newcomm == 0) {
return(lam_errfunc(comm, BLKMPICOMMCREATE,
lam_mkerr(MPI_ERR_ARG, 0)));
}
/*
 * Create the new context ID using MPI_Allreduce().
 * Processes not in group participate but do not affect the context ID.
 */
rank = group->g_myrank;

...
```

Ejecuta la colectiva all-reduce

Esta es una operación de *reduce* seguida por un *scatter*, todos obtienen el resultado al final. Ver más adelante operaciones colectivas.

```
err = MPI_Allreduce(&mycid, &cid, 1, MPI_INT, MPI_MAX, comm);
if (err != MPI_SUCCESS) {
```

```
LAM_TRACE(lam_tr_cffend(BLKMPICOMMCREATE, -1, comm, 0, 0));
lam_resetfunc(BLKMPICOMMCREATE);
return(lam_errfunc(comm, BLKMPICOMMCREATE, err));
}

...
```

Crea el nuevo manejador

Crea la nueva estructura en memoria.

```
/*
 * Create the new communicator.
 */
*newcomm = 0;
if (lam_comm_new(cid, group, MPI_GROUP_NULL, 0, newcomm)) {
return(lam_errfunc(comm, BLKMPICOMMCREATE,
lam_mkerr(MPI_ERR_OTHER, errno)));
}

group->g_refcount++;
(*newcomm)->c_errhdl = comm->c_errhdl;
comm->c_errhdl->eh_refcount++;

if (!al_insert(lam_comms, newcomm)) {
return(lam_errfunc(comm, BLKMPICOMMCREATE,
lam_mkerr(MPI_ERR_INTERN, errno)));
}

...

return(MPI_SUCCESS);
}
```

3.3 Comunicaciones Colectivas

MPI aporta gran cantidad de funciones para las comunicaciones colectivas, solo se analizarán los pares de las que están en PVM por tres razones: simplicidad, comparación, y porque el resto se puede implementar con las base que se muestra, aunque tal vez no de forma eficiente.
Antes de los detalles para las funciones se dan características generales de todas las colectivas:
El "matching" de parámetros en las comunicaciones colectivas es más restrictivo que en las punto a punto, debido a que la cantidad de datos especificados debe ser igual en las llamadas de todos los procesos aunque el tipo de datos no

necesariamente debe ser el mismo. No se usa `TAG` en la llamadas. Solo existen en versión bloqueante[3] y estándar[4]. El modo estándar permite una semántica local dejando que la llamada termine tán pronto como su participación no sea más necesaria, lo que no indica que todas hallan terminado, aunque este es un aspecto libre a la implementación. Cabe destacar que en MPI no existe el *multicast.*
MPI soporta las siguientes funciones colectivas:

- All-to-all
  - all-gather
  - all-to-all/complete exchange
  - all-reduce
  - reduce-scatter
- All-to-One
  - reduce
  - gather
- One-to-all
  - broadcast
  - scatter
- Other
  - barrier
  - scan
  - exclusive scan

Si un error se produce dentro de cualquiera de las funciones de MPI, el manejador de error corriente es llamado. El manejador por omisión aborta el trabajo (job). Este puede ser cambiado llamando a `MPI_Errhandler_set`. MPI no garantiza que un programa pueda seguir ejecutando luego de producido un error.

Algunos tipos soportados son:

generic types	MPI datatypes in *C*	MPI datatypes in *Fortran*
`Character`	MPI_CHAR	MPI_CHARACTER
`Short Integer`	MPI_SHORT	Not supported
`Integer`	MPI_INT	MPI_INTEGER
`Long`	MPI_LONG	Not supported
`Byte`	MPI_BYTE	MPI_BYTE
`Real`	MPI_FLOAT	MPI_REAL
`Double`	MPI_DOUBLE	MPI_DOUBLE_PRECISION
`Boolean`	MPI_INT	MPI_LOGICAL
`Complex`	Not supported	MPI_COMPLEX
`Packed`	MPI_PACKED	MPI_PACKED

[3]Al menos hasta MPI-1
[4]En MPI hay modo: standard, buffered, synchronus y ready

El algoritmo en LAM/MPI funciona de la siguiente forma: para cuatro o menos procesos, se usa un algoritmo lineal enviando desde 0 a size-1. Si son más los procesos involucrados un algoritmo basado en un árbol es usado. El árbol se construye calculando la dimensión máxima del cubo que se puede obtener con respecto a la cantidad de procesos y luego para cada `rank` se hace un cómputo:

```
...
/* Calc'ula la dimensi'on del cubo de acuredo */
/* a la cantidad de miembros del grupo        */

dim = lam_cubedim(l_group->g_nprocs);

/* En este ejemplo el que dirige es el rank=0 */

root = 0;

/* Determina par'ametros extras para el c'alculo */

vrank = (rank + size - root) % size;
hibit = lam_hibit(vrank,dim);

/* Calc'ula de qui'en recibe */

from  = ( (vrank & ~(1 << hibit)) + root ) % size;

/* Solo lo muestra */

printf(``%d recibe de %d\n'', rank, from );
...
```

3.3.1 Descripción de Funciones

```
C          int MPI_Bcast(void *buff, int count,
                         MPI_Datatype datatype,
                         int root, MPI_Comm comm)
```

Hace un envío de un mensaje desde el proceso con `rank=root` a todos los procesos del grupo en el *communicator* usado, incluido `root`. El parámetro `root` debe ser el mismo en todos los invocadores.
Los parámetros son:
`buff`: buffer/mensaje a enviar.
`count`: número de elementos del tipo `datatype` a enviar.
`datatype`: manejador del tipo de dato a enviar.
`root`: `rank` del proceso que envía el *broadcast*, debe ser el mismo en todos los participantes.
`comm`: *communicator* a partir del cual se hace el envió del mensaje.

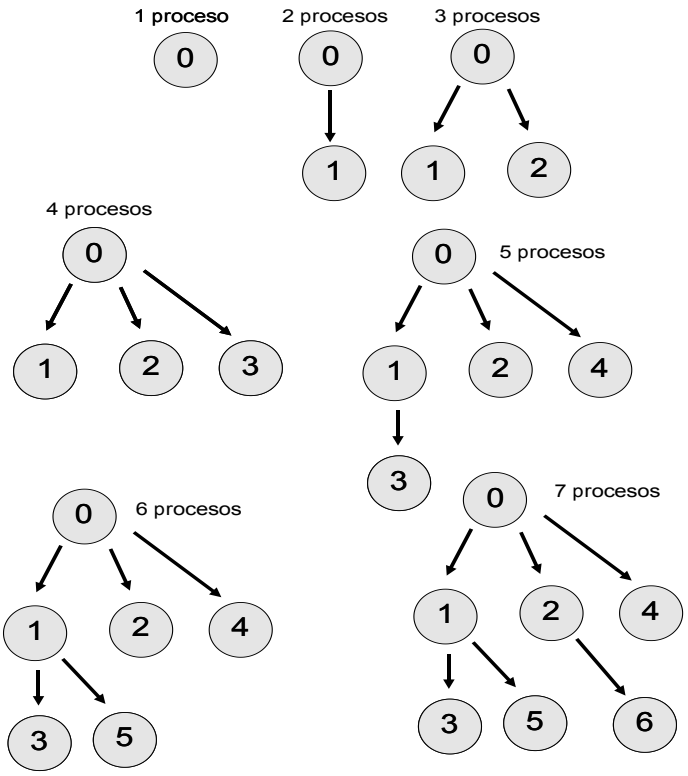

Figura 3.4: Estructura de los árboles para las colectivas.

Esta operación es extendida en MPI-2 a *intercommunicators*. La sintáxis de esta operación como también para *gather*, *scatter* y *reduce* es: un grupo, por ejemplo *left*, es el que impone el `root`, todos los miembros de *left* llaman con `MPI_PROC_NULL` como `root`, salvo el mismo `root` que la llama con `MPI_ROOT`. Todos los procesos en *right* la llaman con el `rank` del `root`. La semántica es: todos los procesos en *right* reciben el broadcast desde el `root` en *left*.

```
C          int MPI_Barrier(MPI_Comm comm)
```

Bloquea al llamador hasta que todos los miembros del grupo invocarón la función. La llamada termina en cada proceso después que todos entrarón en la invocación. Los parámetros son:
`comm`: *communicator* a partir del cual se hace la sincronización.

Esta operación es extendida en MPI-2 a *intercommunicators*. La sintáxis es

igual que para *intracommunicators*. La semántica es: todos los procesos en *left* se bloquean hasta que todos los procesos en *right* hayan entrado al *barrier*.

```
C             int MPI_Gather(void *sbuf, int scount,
                             MPI_Datatype sdtype,
                             void *rbuf, int rcount,
                             MPI_Datatype rdtype,
                             int root, MPI_Comm comm)
```

El proceso denominado `root` recibe mensajes de todos los procesos en el grupo, inclusive el mismo. Los mensajes recibidos los almacena en el orden de los `ranks`.
Los parámetros son:
`sbuf`: dirección del buffer de envío.
`scount`: cantidad de elementos del buffer de envío.
`sdtype`: tipo de los elementos del buffer de envío.
`rbuf`: dirección del buffer de recepción, solo tiene significado en el proceso `root`.
`rcount`: cantidad de elementos de una recepción, tamaño de una porción sola, no del total de los datos a recibir. Solo tiene significado en el proceso `root`.
`sdtype`: tipo de los elementos a recibir, solo tiene significado en el proceso `root`.
`root`: rank del proceso que recibe las porciones, debe ser el mismo en todos los participantes.
`comm`: *communicator* a partir del cual se hace el envió de los mensajes.

Existe una variante de esta función llamada `MPI_Gatherv` permitiendo que la cantidad enviada por cada proceso sea variable.

Esta operación es extendida en MPI-2 a *intercommunicators*. La semántica es: todos los procesos en *right* reciben una porción desde el `root` en *left*.

```
C            int MPI_Scatter(void *sbuf, int scount,
                             MPI_Datatype sdtype,
                             void *rbuf, int rcount,
                             MPI_Datatype rdtype,
                             int root, MPI_Comm comm)
```

Es la invera de `MPI_Gather`. El proceso denominado `root` envia mensajes a todos los procesos en el grupo, inclusive a si mismo. Los mensajes los resuelve a partir de un área de memoria que particiona en tamaños regulares enviandolos en el orden de los `ranks`. Los parámetros son:
`sbuf`: dirección del buffer de envío. Solo tiene significado en el proceso `root`.
`scount`: cantidad de elementos de un solo envío. No es la cantidad total. Solo tiene significado en el proceso `root`.
`sdtype`: tipo de los elementos del buffer de envío. Solo tiene significado en el proceso `root`.

`rbuf`: dirección del buffer de recepción.
`rcount`: cantidad de elementos a recibir en una recpción.
`sdtype`: tipo de los elementos a recibir.
`root`: rank del proceso que envía las porciones, debe ser el mismo en todos los participantes.
`comm`: *communicator* a partir del cual se hace el envió de los mensajes.

Existe una variante de esta función llamada `MPI_Scatterv` permitiendo que la cantidad enviada para cada proceso sea variable.

```
C              int MPI_Reduce(void *sbuf, void* rbuf,
                              int count,
                              MPI_Datatype dtype,
                              MPI_Op op, int root,
                              MPI_Comm comm)
```

Produce una reducción global sobre todos los miembros del grupo aplicando una operación predefinidas:

MPI_MAX: máximo

MPI_MIN: mínimo

MPI_SUM: suma

MPI_PROD: producto

MPI_IAND: and lógico

MPI_BAND: and a nivel de bits

MPI_IOR: or lógico

MPI_BOR: or a nivel de bits

MPI_IXOR: xor lógico

MPI_BXOR: xor a nivel de bits

El usuario puede definir también sus propias operaciones las cuales debe definir para los diferentes tipos de datos. La operación debe ser asociativa y las predefinidas también son conmutativas, pero esta última no es una restricción. El orden de evaluación es de acuerdo a los números de `rank`.
El proceso `root` obtiene como resultado la operación aplicada al total de los datos elemento a elemento (element-wise). El proceso denominado `root` envía mensajes a todos los procesos en el grupo, inclusive a si mismo. Los mensajes los resuelve a partir de un área de memoria que particiona en tamaños regulares enviandolos en el orden de los `ranks`. Los parámetros son:
`sbuf`: dirección del buffer de envío.
`rbuf`: dirección del buffer de recepción.
`count`: cantidad de elementos del buffer, tanto de envío como de recepción. Debe tener el mismo valor en todos los participantes.

`dtype`: tipo de los elementos del buffer, tanto de envío como de recepción.
`op`: operación a aplicar, una de las definidas anteriormente, debe ser la misma en todos los participantes.
`root`: `rank` del proceso que recibe el resultado, debe ser el mismo en todos los participantes.
`comm`: *communicator* a partir del cual se hace el envió de los mensajes.

Esta operación es extendida en MPI-2 a *intercommunicators*. La semántica es: todos los procesos en *right* tienen los datos para hacer el reduce con el `root` impuesto por *left*. Aquí todos los procesos de ambos grupos participan.

3.3.2 Cómo Funcionan Internamente

Para ver como funcionan se analiza de la "suit" de colectivas del LAM/MPI solo el *broadcast*, ya que el resto lo hace de una forma similar.

Análisis del Broadcast

Los comentarios agregados por el autor del texto están en castellano y tienen el siguiente formato:

```
/*
 *--------------------------
 * ...
 *--------------------------
 */
```

Los pasos que realiza són:

1. Chequea los argumentos.
2. Llama a `MPI_Group_size`.
3. De acuredo al size decide que algoritmo usar si el lineal o el basado en el árbol.
4. Llama al algoritmo correspondiente.

```
int MPI_Bcast(void *buff, int count, MPI_Datatype datatype,
      int root, MPI_Comm comm)
{


    ...

   /*
    * Check for invalid arguments.
    */
    if ((comm == MPI_COMM_NULL) || ...

    MPI_Comm_size(comm, &size);
```

```
    ...

    /*
     * Decide which algorithm to use.
     */
     if (size <= 1)
     {
        lam_resetfunc(BLKMPIBCAST);
        return(MPI_SUCCESS);
     }
     /*
      *-------------------------------------------
      * El precompilador decidi'o si usa
      * o no el IMPI (Interoperable MPI)
      * Protocolo para lograr la interoperabilidad
      * entre diferentes implementaciones.
      *-------------------------------------------
      */

     #if LAM_WANT_IMPI
     else if (LAM_IS_IMPI(comm))
     {
        return IMPI_Bcast(buff, count, datatype, root, comm);
     }
     #endif
     else if (size <= LAM_COLLMAXLIN) /** LAM_COLLMAXLIN == 4 **/
     {
     /*
      *-----------------------------------------------------
      * Usa el algoritmo lineal, con daemon o client2client
      *-----------------------------------------------------
      */
         return(RPI_SPLIT(bcast_lin_lamd, bcast_lin,
                          (buff, count, datatype, root, comm)));
     }
     else
     {
     /*
      *-----------------------------------------------------
      * Usa el algoritmo basado en 'arbol, con daemon o client2client
      *-----------------------------------------------------
      */
          return(RPI_SPLIT(bcast_log_lamd, bcast_log,
                           (buff, count, datatype, root, comm)));
     }
}
```

El lineal

1. Determina si es `root` o no para enviar o recibir.
2. Si es `root` contruye una lista virtual de hijos y manda a cada uno a través de un "handler".
3. Primero construye todos los "handlers" y luego envía.

```
static int bcast_lin(buff, count, datatype, root, comm)

void *buff;
int count;
MPI_Datatype datatype;
int root;
MPI_Comm comm;

{

    ...

    lam_mkcoll(comm);

    /*
     * Non-root receive the data.
     */
    if (rank != root)
    {
       err = MPI_Recv(buff, count, datatype, root,
                      BLKMPIBCAST, comm, MPI_STATUS_IGNORE);
    ...


    }
    /*
     * Root sends data to all others.
     */
    for (i = 0, preq = reqs; i < size; ++i)
    {

       if (i == rank) continue;
       /*
        *-------------------------------------------------------
        * Crea un request/handler para hacer un send est'andar
        * Que m'as tarde ejecutar'a
        *-------------------------------------------------------
        */
        err = MPI_Send_init(buff, count, datatype, i, BLKMPIBCAST,
         comm, preq++);
```

```
        if (err != MPI_SUCCESS)
        {
            lam_mkpt(comm);
            return(lam_errfunc(comm, BLKMPIBCAST, err));
        }
    }

        /*
         *-----------------------------------------
         * Procesa requests/hanlders iniciados en for
         *-----------------------------------------
         */

    ...

    return(MPI_SUCCESS);
}
```

Basado en árbol

1. Calcúla la dimensión máxima del cubo a partir de la cual construye el árbol virtual.
2. Si no es `root` recibe.
3. Si es `root`, o algún intermedio que debe enviar construye una lista virtual de hijos y les envía a cada uno (Primero construye todos los handlers y luego envía).

```
static int bcast_log(buff, count, datatype, root, comm)

void *buff;
int count;
MPI_Datatype datatype;
int root;
MPI_Comm comm;

{

    ...

    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    lam_mkcoll(comm);

    /*
     *---------------------------------
     * Calcula la dimensi'on del cubo
     *---------------------------------
     */
```

```
vrank = (rank + size - root) % size;

dim = comm->c_cube_dim;
hibit = lam_hibit(vrank, dim);
--dim;

/*
 * Receive data from parent in the tree.
 */
if (vrank > 0)
{
    peer = ((vrank & ~(1 << hibit)) + root) % size;

    err = MPI_Recv(buff, count, datatype, peer,
                   BLKMPIBCAST, comm, MPI_STATUS_IGNORE);
    if (err != MPI_SUCCESS)
    {
        lam_mkpt(comm);
        return(lam_errfunc(comm, BLKMPIBCAST, err));
    }
 }
/*
 * Send data to the children.
 */
preq = reqs;
nreqs = 0;

for (i = hibit + 1, mask = 1 << i; i <= dim; ++i, mask <<= 1)
{

    peer = vrank | mask;
    if (peer < size)
    {
        peer = (peer + root) % size;
        ++nreqs;

/*
 *-------------------------------------------------------
 * Crea un request/handler para hacer un send est'andar
 * Que m'as tarde ejecutar'a
 *-------------------------------------------------------
 */
 err = MPI_Send_init(buff, count, datatype, peer, BLKMPIBCAST,
                     comm, preq++);
 if (err != MPI_SUCCESS)
 {
         lam_mkpt(comm);
         return(lam_errfunc(comm, BLKMPIBCAST, err));
 }
```

```
    }
   }

     /*
      *-----------------------------------------
      * Procesa requests/handlers iniciados en for
      *-----------------------------------------
      */

      ...

      return(MPI_SUCCESS);
}
```

Bibliografía

[MPI] MPI: The Complete Reference - Marc Snier, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra - The MIT Press, Cambridge, Massachusetts - 1996.

[MPI2] MPI-2: Extension to the MPI - MPI Forum - University of Tennesse Knoxville - 1997.

[PVM1] PVM: Parallel Virtual Machine A Users'Guide and Tutorial for Networked Parallel Computing , Al Geist , Adam Beguelin , Jack Dongarra , Weicheng Jiang , Robert Manchek and Vaidy Sunderam - The MIT Press, Cambridge, Massachusetts - 1994.

[PVM-MPI1] PVM and MPI: a Comparison of Features - Al Geist, J. A Kohl P. M. Papadopulos - Calculateurs Paralleles (Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Departament of Energy) - May, 30 1996.

[PVM-MPI2] Why are PVM and MPI So Different ? - William Group, Ewing Lusk - Mathematics an Computer Science Division Argonne National Lab. - June 1997.

[Kernighan 78] The *C* Programming Language - Kernighan, B. W., and D. M. Ritchie - Prentice Hall Editorial, Englewood Cliff, NJ - 1978.

[LAM] LAM/MPI - University of Notre Dame (http://www.mpi.nd.edu/lam) - 1998-2001.

[MPICH] MPICH - Mathematics and Computer Science Division, Argonne National Laboratory (http://www.mcs.anl.gov/mpi/mpich) - Sep. 2000.

[IMPI] IMPI - Interoperable Message Passing Interface http://impi.nist.gov.

Capítulo 4

Modelos y Métricas

4.1 Introducción

En este capítulo se describen cuales son los diferentes algoritmos o modelos para resolver las operaciones colectivas analizadas con PVM y MPI en los capítulos anteriores. Para cada algoritmo se evalua el costo y cual es su factibilidad para implementarlo sobre NOWs.

4.2 Modelos

Si bien todos los modelos consideran a los dos factores más importantes que determinan el tiempo de comunicación:

Ancho de Banda: (bandwidth) se define como la máxima cantidad de datos que pueden ser transmitidos por unidad de tiempo.

Latencia: (start-up) es el mínimo tiempo requerido para transmitir cualquier objeto, está ligado a la velocidad de transmisión de la señal sobre el medio e incluye cualquier overhead de send/receive por software. Este factor determina la granularidad útil mínima.

los modelos propuestos para las comunicaciones colectivas son altamente dependientes de la implementación por software y del hardware subyacente, esta es la razón de la diversidad de modelos existentes. En esta sección se presentan algunos modelos teóricos y se marcan aquellos que son más apropiados con el objetivo del estudio del trabajo, las NOWs.

4.2.1 Modelos Punto a Punto

Las comunicaciones punto a punto son fundamentales para todo el subsistema de comunicaciones y en cierta medida muchas primitivas de IPC (Inter-Process Communication) colectivas se desarrollan sobre las punto a punto. Se cree que es fundamental conocer estos modelos para luego desarrollar y comprender los de las colectivas.
El modelo base es descripto por la ecuación 4.1 [Foster95] [Ttti02]:

$$T(n) = \alpha + \beta(n) \tag{4.1}$$

T(n) es el tiempo total de la comunicación necesario para transmitir n bytes, α es tiempo de latencia o start-up y β es el tiempo necesario para la transmisión dado por ancho de banda de la red. Se asume que no hay diferencia entre comunicar cualquiera de los procesadores o nodos, ni existe contención por ganar acceso el canal (channel contention). Este modelo es refinado en [Don96] dando lugar a la ecuación 4.2:

$$T(n) = \alpha + \beta(n) + (h-1)\gamma \tag{4.2}$$

Donde se considera que los costos de comunicaciones entre nodos son diferentes de acuerdo al camino a recorrer. Se agrega el factor γ, que significa el delay por salto (hop), que se multiplica por cantidad de hops, *(h - 1)* [1], que el mensaje debe atravesar. En la mayoría de los casos estos factores dependen altamente de las características de bajo nivel de las comunicaciones, como lo son: topología, políticas de ruteo (data-routing functions), modo de switching (switching methodology), modo de operar (sincrónico o asincrónico), etc. [Hwang93] [Briggs84].

Otro modelo refinado de las comunicaciones punto a punto es el propuesto por Ian Foster [Foster95] donde se tiene en cuenta la contención por ganar acceso al medio, factor que es de suma importancia en redes basadas en buses como Ethernet, estandarizado por la *IEEE* como *802.3* [802.3].

$$T(n) = \alpha + \beta(n)(1/S) \tag{4.3}$$

En estas arquitecturas típicamente solo se puede transmitir un paquete por vez, por lo tanto el acceso se secuencializa, En la ecuación 4.3 este hecho es representado por S que indica la cantidad de nodos que necesitan enviar concurrentemente sobre el mismo dominio de colisiones. Este factor muestra que el ancho de banda efectivo para cada nodo es de *1/S* del ancho de banda ofrecido por el medio.

4.2.2 Modelos de las Colectivas

Ahora, en el área de interés del texto, se encuentra que cada una de las funciones para comunicaciones colectivas posee un modelo distinto, que como se dijo anteriormente, depende de la implementación tanto de software como de hardware. Muchas implementaciones son resueltas a nivel de software usando directamente las punto a punto o, indirectamente, primero construyendo un *spanning tree* (los cuales pueden tener diferentes formas). En [Sathish] se presentan variados algoritmos basados en árboles binomiales, árboles binarios, listas secuenciales, etc.
Una de las operaciones colectivas más usada es el *multicast* o en su omisión el *broadcast* y será sobre esta que se hará más hincapié. Algunos modelos pueden ser:

[1] h es la cantidad de nodos intermediarios más el remitente y el receptor

Broadcast

Secuencial - Lineal

$$T(n) = K(\alpha + \beta(n)) \tag{4.4}$$

Basada en árbol binario con envíos concurrentes desde un mismo nodo

$$T(n) = log_2(K)(\alpha + \beta(n)) \tag{4.5}$$

Basada en árbol binario con punto a punto desde un mismo nodo

$$T(n) = 2log_2(K)(\alpha + \beta(n)) \tag{4.6}$$

Usando broadcast físico ideal

$$T(n) = \alpha + \beta(n) \tag{4.7}$$

Usando broadcast físico con retransmisiones y protocolo de sync.

$$T(n) = K\alpha + \beta(n) + R(n, K-1, ...) \tag{4.8}$$

En estas ecuaciones, α es la latencia, β el ancho de banda, K la cantidad de nodos que por comodidad se lo puede considerar $K = 2^m - 1$ y R es una función de retransmisión que depende de la cantidad de nodos receptores, la cantidad de bytes transmitidos y de la calidad de servicio del medio. Estos valores son correctos dependiendo del hardware como mínimo, por ejemplo los algoritmos basados en árbol 4.5 y 4.6 no se comportarían de acuerdo al modelo si se los ejecuta sobre una red de estaciones de trabajo conectadas mediante un bus (ya sea lógico con un hub + 10BaseT, o físico usando 10Base5 o 10Base2 [Tan88]).

Se está considerando un modelo en el cual el emisario también recibe los datos, si no fuese así al factor K se le deberá sustraer 1. Cabe destacar que el costo de enviarse a si mismo en general es menor que sacar la información fuera del proceso (se supone un modelo que mapea un proceso por nodo).

No es discusión de este trabajo cuál es la estructura óptima para construir el *spanning tree*, tema que ya es presentado en [Bani] y en [Bhat]. De todos los modelos el más adecuado para las NOWs es el 4.8, broadcast físico con retransmisiones. Debe tenerse en cuenta que esta afirmación se hace tomando como base LANs tradicionales. Hasta PVM versión 3.4 parece usarse el modelo 4.4 y LAM/MPI usa los modelos de árboles para más de 4 procesos [2].
Algunos de los modelos existentes para el resto de las colectivas son:

Scatter

En este caso al total de información a transmitir,n bytes, se divide entre todos los pares (peers) n/K.

Secuencial - Lineal

$$T(n) = K(\alpha + \beta(n/K)) \tag{4.9}$$

[2]Los árboles que se usan son binomiales

Basada en árbol binario con envíos concurrentes desde un mismo nodo

$$T(n) = log_2(K)\alpha + \beta((n/K)(K-1)) \tag{4.10}$$

Basada en árbol binario con punto a punto desde un mismo nodo

$$T(n) = 2log_2(K)\alpha + \beta(2(n/K)(K-1)) \tag{4.11}$$

Usando broadcast físico ideal

$$T(n) = K\alpha + \beta(n) \tag{4.12}$$

Cuando la cantidad de información es poca se puede hacer un broadcast mandándole todo a todos. Este algoritmo es útil cuando la implementación del broadcast es eficiente, con un start-up muy bajo y aprovecha al máximo las prestaciones del hardware.
Otro algoritmo posible es el encadenado o *pipelined* en el cual el proceso P_i recibe del P_{i-1}. Con este modelo el proceso P_i deberá esperar *i-1* pasos hasta recibir.

Las ecuaciones 4.10 y 4.11 surgen del siguiente análisis:

- Se supone que hay un árbol binario lleno, K nodos donde $K = 2^m - 1$
- Los nodos serán P_0, P_1, ... , P_{K-1}
- Se tiene un arreglo A de K porciones de tamaño M cada una ubicado en el proceso P_0 (proceso *root*).

$$A_0 \quad A_1 \quad ... \quad A_{K-1} \tag{4.13}$$

- P_0 se quedará con A_0 y enviará hacia el hijo izquierdo $A_1..A_{(K-1)/2}$ y hacia el derecho $A_{((K-1)/2)+1}..A_{K-1}$.
- Luego cada receptor si no es hoja (nodo sin hijos) enviará usando el mismo algoritmo pero con un K reducido a la mitad del valor que tenía en su nodo padre.
- Si se instancia el algoritmo con diferentes valores para K se obtiene:

m	$K = 2^m - 1$	Paso 1	Paso 2	Paso 3	Paso 4	Total
1	1					0M
2	3	2 * 1M				2M
3	7	2 * 3M	2 * 1M			8M
4	15	2 * 7M	2 * 3M	2 * 1M		22M
5	31	2 * 15M	2 * 7M	2 * 3M	2 * 1M	52M

En cada paso se describe la cantidad de información transmitida por nivel del árbol. Se multiplica por 2 debido a que cada nodo no hoja tiene 2 hijos. Se supone que los envíos desde un mismo nodo son punto a punto y los nodos del mismo nivel lo hacen de forma concurrente.

- Si se define la función de costo del algoritmo de acuerdo a la tabla anterior como F donde representa la cantidad de información a ser transmitida de acuerdo a la cantidad de nodos y al tamaño de la porción se obtiene:

$$F(K,M) = \sum_{i=0}^{log_2(K+1)-1} 2(2^i - 1)M \tag{4.14}$$

Si se instancia/aplica la función se obtiene los mismos valores de la tabla:

$$F(1,M) = \sum_{i=0}^{0} 2(2^i - 1)M = 2(2^0 - 1)M = 0M \tag{4.15}$$

$$F(3,M) = \sum_{i=0}^{1} 2(2^i - 1)M = 0M + 2(2^1 - 1)M = 2M \tag{4.16}$$

$$F(7,M) = \sum_{i=0}^{2} 2(2^i - 1)M = 0M + 2M + 2(2^2 - 1)M = 8M \tag{4.17}$$

$$F(15,M) = \sum_{i=0}^{3} 2(2^i - 1)M = 0M + 2M + +6M + 2(2^3 - 1)M = 22M \tag{4.18}$$

- Reescribiendo sin usar el símbolo de sumatoria:

$$F(K,M) = \sum_{i=0}^{log_2(K+1)-1} 2(2^i - 1)M = \tag{4.19}$$

$$F(K,M) = 2M \sum_{i=0}^{log_2(K+1)-1} (2^i - 1) = \tag{4.20}$$

$$F(K,M) = 2MG(K) \texttt{ donde } G(K) = \sum_{i=0}^{log_2(K+1)-1} (2^i - 1) \tag{4.21}$$

Se obtiene luego que:

$$G(K) = \sum_{i=0}^{n-1}(2^i - 1) \texttt{ donde } n = log_2(K+1) \tag{4.22}$$

$$G(K) = \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} 1 = \tag{4.23}$$

$$G(K) = (\sum_{i=0}^{n-1} 2^i) - (n) = \tag{4.24}$$

$$G(K) = (2^n - 1) - (n) = \tag{4.25}$$

$$G(K) = (2^n) - (n+1) \tag{4.26}$$

Nos queda:

$$F(K,M) = 2M[(2^n) - (n+1)] \texttt{ donde } n = log_2(K+1) \tag{4.27}$$

$$F(K,M) = 2M[(2^{log_2(K+1)}) - (log_2(K+1)+1)] \tag{4.28}$$

- Escribiendo la ecuación del modelo agregando la latencia se tiene:

$$T(K,M) = 2log_2(K)\alpha + \beta(F(K,M)) = \tag{4.29}$$

La latencia multiplica por 2 debido a que el costo se presenta por cada nodo hijo.

$$T(K,n) = 2log_2(K)\alpha + \beta(2M[(2^{log_2(K+1)}) - (log_2(K+1)+1)]) \texttt{ donde } n{=}KM \tag{4.30}$$

- la ecuación 4.30 puede ser aproximada por:

$$T(K,n) = 2log_2(K)\alpha + \beta(2M(K-1)) \texttt{ donde } n{=}KM \tag{4.31}$$

y si se suprime los 2 suponiendo que los envíos desde un nodo son simultaneos se tiene: [Luecke00]:

$$T(K,n) = log_2(K)\alpha + \beta(M(K-1)) \texttt{ donde } n{=}KM \tag{4.32}$$

De todas estas ecuaciones la más apropiada para las NOWs basadas en bus es la 4.9, sin embargo si se tiene capacidad de switching a nivel dos OSI (Open Systems Interconnection) los modelos 4.10 y 4.11 "calzan" mejor.

Gather

Los modelos para el *gather* son los mismos que los usados para el *scatter* debido a que las transmisiones necesarias son las mismas, solo cambia el orden y el sentido, por ejemplo: si se usa un algoritmo basado en árbol se comenzará desde los nodos hojas hasta llegar a la raíz. Solo el modelo de *broadcast* de los presentados no es aplicable.

Reduce

Los modelos para el *reduce* en general son el del *gather* más la operación. Otra forma de hacerlo es ir reduciendo a medida que se va transmitiendo la información siguiendo los mismos patrones que se presentaron anteriormente.

Gather + Operación

$$T(n) = (T_{gather}(n) + Op(n)) \tag{4.33}$$

Encadenado (Pipelined)

$$T(n) = (K-1)[(\alpha + \beta(n/K)) + Op(n/K)] \tag{4.34}$$

En este algoritmo todos los procesos menos el primero y el último reducen y envían. P_{K-1} (el primero) solo envía y P_0 (el último) solo reduce.

Basada en árbol binario con envíos concurrentes

$$T(n) = (log_2(K+1) - 1)[\alpha + \beta((n/K)) + Op(n/K)] \tag{4.35}$$

En este caso se obtiene que todos los nodos que no son hojas ni raíz envían y reducen, los nodos hojas solo envían y la raíz solo reduce. En total se tienen *(K-1) sends* (cantidad de aristas) y *[(K-1)/2] operations* (cantidad de nodos - cantidad de hojas), pero como se hacen de forma concurrente los valores se reducen al orden logarítmico.

En estas ecuaciones la cantidad de nodos también es considerada como $K = 2^m - 1$, donde m es cualquier entero positivo. El modelo que se cree más apropiado para las NOWs es 4.33 debido a su simplicidad y a la gran diferencia que existe entre cómputo y comunicaciones sobre estas arquitecturas.

$$T_{computo}(n) << T_{comunicaciones}(n) \tag{4.36}$$

Barrier

Por último se ven los modelos para el *barrier*. Exísten varias formas de desarrollar el algoritmo, las variantes presentadas son:

Gather + Broadcast

$$T(min) = (T_{gather}(min) + T_{broadcast}(min) \tag{4.37}$$

Primero se hace un *gather* recibiendo una señal de "listo y en espera" de todos los pares *(peers)* menos *root*, cada uno se bloquea hasta recibir la señal de "continuar". Una vez recibidas todas la señales por el *root* se los desbloquean mediante un *broadcast*.

Basado en Anillo

$$T(min) = K(\alpha + \beta(min)) \tag{4.38}$$

Todos los procesos se encuentran conectados lógicamente en forma de anillo. Cada un recibe y envía un *token* desde su predecesor y hacia su sucesor en la formación, el algoritmo termina cuando el iniciador o *root* recibe el *token*.

Lineal - Secuencial

$$T(min) = 2(K-1)(\alpha + \beta(min)) \tag{4.39}$$

En este algoritmo un proceso cumple rol de *root*, el cual envía la menor cantidad de datos a cada par *(peer)* y luego espera de recibir de estos. El problema que tiene este algoritmo es que puede producir *channel contention* en el momento en que todos los puntos *(peers)* envían la respuesta.

Butterfly [Andrews91]

$$T(min) = log_2(K)[2(\alpha + \beta(min))] \tag{4.40}$$

Es un *barrier* simétrico de K procesos que se va construyendo de a pares. Tiene $log_2(K)$ pasos, donde en cada uno, cada proceso sincroniza con un proceso diferente a distancia $2^{stage-1}$ (ver figura 4.1). Lo ideal sería que $K = 2^m$, si no es así puede ser construido usando la próxima potencia de 2 mayor que K y si al calcular con que par el proceso debe hacer el *barrier* obtiene un valor fuera del rango no hace nada y espera el próximo paso. La sincronización de a pares se debería hacer enviando y recibiendo un mensaje mínimo desde y hacia el par designado. El 2 multiplica debido a que hay dos envíos por par.

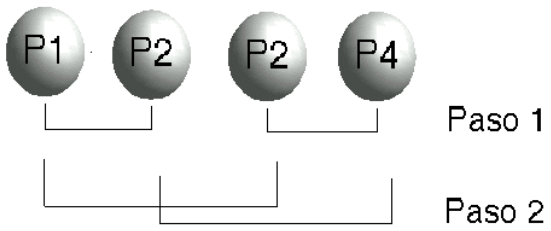

Figura 4.1: Butterfly con 4 procesos

En estas ecuaciones *min* es la cantidad mínima que se puede transmitir. El modelo que se selecciona es el 4.40.

4.3 Métricas

Las métricas son aquellos valores a medir para poder comparar las comunicaciones colectivas. Si bien son varios los parámetros que se pueden considerar, en este estudio solo se seleccionan los dos más relevantes:

- *latencia* o *start-up*
- *ancho de banda (bandwidth)* o *productividad del canal (channel throughput)*

Estos se derivan de los parámetros que componen los modelos presentados. En las comparaciones para el *barrier* solo se considera el tiempo de la operación, el ancho de banda se encuentra implícito en la duración total.
Para mensajes cortos y operaciones de sincronización la latencia es el parámetro más significativo, en cambio para grandes mensajes el ancho de banda es más importante. Otro parámetro importante pero no considerado aquí es el tamaño de mensaje para el cual se alcanza la mitad del máximo ancho de banda (peak bandwith) $n_{1/2}$, factor que indica la aceleración, pero que solo debe ser considerado en el contexto de α y β [Don96].
Es un hecho que para las operaciones colectivas existen otros parámetros que afectarán el rendimiento como los son el grado de concurrencia o la escalabilidad, pero se cree que los analizados son suficientes.

Bibliografía

[Sathish] Sathish S. Vadhiyar, Graham E. Fagg, Jack Dongarra. Automatically Tuned Collective Communications. Computer Science Department, University of Tennessee, Knoxville.

[Ttti02] Fernando Tinetti, Andres Barbieri, Análisis del Rendimiento de las Comunicaciones sobre NOWs, CACIC 2002, UNPA. Octubre del 2002.

[Don96] Jack Dongarra and Thomas H. Dunigan, Message-Passing Performance of Various Computers. Mathematical Science Section, Oak Ridge National Laboratory. Report ORNL/TM-13006, Feb 1996.

[Hwang93] Hwang, Kai. Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw-Hill, Inc. 1993.

[Briggs84] Fayé A. Briggs and Kai Hwang. Computer Architecture and Parallel Processing, McGraw-Hill, Inc. 1984.

[Foster95] Ian Foster. Designing and Building Parallel Programs, Addison-Wesley Inc., Argonne National Laboratory, and the NSF Center for Research on Parallel Computation 1995.

[802.3] Institute of Electrical and Electronics Engineers, Local Area Network - *CSMA/CD Access Method and Physical Layer Specifications* ANSI/IEEE 802.3 - 1985, IEEE Computer Society.

[Bani] Mohammad Banikazemi and Dhabaleswar K. Panda. Efficient Collective Communication on Heterogeneous Networks of Workstations. Dept. of Computer and Information Science The Ohio State University, Columbus.

[Bhat] Prashanth b. Bhat, C.S. Raghavendra and Viktor K. Prasanna. Efficient Collective Communication in Distributed Heterogeneous Systems. Dept. of EE-Systems, University of Southern California, Los Angeles.

[Tan88] Tanenbaum, A.S. (1988). Computer Networks. Second Edition Englewood Cliffs NJ: Prentice-Hall.

[Luecke00] G. Luecke , Jing Yuan , S. Spanoyannis, M. Kraeva. Performance and Scalability of MPI on PC Clusters - Jan 23, 2000.. Durham Center, Iowa State University.

[Andrews91] Andrews, Gregory R. (1991). Concurrent Programming: Principles and Practice. Addison Wesley Publisihing Company.

Capítulo 5

Entorno de Prueba

5.1 Introducción

En este capítulo se describen cuales son los entornos en los cuales se evaluaron las comunicaciones colectivas. Se mencionan las características de las componentes de hardware y de software. En el final se encuentra la sección más relevante, que es en la cual se describen las técnicas de medición, cuestión que no es trivial.

5.2 Configuración Experimental

5.2.1 Hardware y Sistema Operativo

Las pruebas se hicieron sobre dos redes diferentes:

- Una red reducida de máquinas heterogéneas del LIDI (Laboratorio de Investigación y Desarrollos en Informática) que forma parte de la Facultad de Informática de la UNLP.

- Un cluster de 8 PCs homogéneas pertenecientes al Laboratorio de Procesamiento Paralelo y Tiempo Real de la Facultad de Informática de la UNLP.

Una tercera red fue usada para realizar las pruebas con PVM para corroborar algunos de los resultados anómalos obtenidos.

- Una red reducida de máquinas heterogéneas del CeTAD (Centro de Técnicas Analógicas-Digitales) que forma parte de la Facultad de Ingeniería de la UNLP.

Las características técnicas son:

- Red heterogénea del LIDI:

Hostname	System Description	Operating System	Clock Rate	Main Memory
lidi34	PC-K6.2	Linux 2.2.12-20	500MHz	124 MB
lidi48	PC-Pentium MMX	Linux 2.2.12-20	233MHz	28 MB
lidi35	PC-Celeron	Linux 2.2.12-20	266MHz	128 MB
lidi47	PC-Pentium-S	Solaris 7 for Intel	100MHz	32 MB

Hostname	Network adapter
lidi34	NE2000-PCI
lidi48	NE2000-PCI
lidi35	NE2000-PCI
lidi47	NE2000-ISA

La LAN usada es de 10Mb/s con cableado UTP (10BaseT) con 2 hubs.

- Cluster homogéneo de 8 PCs:

Hostname	System Description	Operating System	Clock Rate	Main Memory
lidipar[X]	PC-PIII	Linux 2.2.12-20	700MHz	64 MB

Hostname	Network Adapter
lidipar[X]	Kingston DEC-Tulip 100BaseTX

Los valores para X son: 5, 6, 7, 8, 9, 12, 13, 14. La LAN usada es de 100Mb/s con cableado UTP Cat. 5 (100BaseTX) con 1 switch en modo HDX (Half duplex).

- Red heterogénea del CeTAD:

Hostname	System Description	Operating System	Clock Rate	Main Memory
Purmamarca	PC-PII	Linux 2.2.5-15	400MHz	64 MB
Maimara	PC-Am5x86	Linux 2.2.5-15	133MHz	16 MB
Tilcara	PC-Pentium	Linux 2.2.5-15	133MHz	16 MB
Cetadfomec1	PC-Celeron	Linux 2.2.5-15	300MHz	32 MB
Cetadfomec2	PC-Celeron	Linux 2.2.5-15	300MHz	32 MB
Kuntur	PC-AMD-K5	Linux 2.2.5-15	75MHz	16 MB

Hostname	Network Adapter
Purmamarca	NE2000-PCI
Maimara	NE2000-ISA
Tilcara	NE2000-ISA
Cetadfomec1	NE2000-PCI
Cetadfomec2	NE2000-PCI
Kuntur	NE2000-PCI

La LAN usada es de 10Mb/s y el cableado es de UTP con 4 hubs.

5.2.2 Middleware

Middleware es una palabra que en el ámbito de la informática se usa de gran cantidad de formas, en este caso se llama así a la capa de software que se encuentra entre el sistema operativo y la aplicación de usuario, tal como se describe en [CCWP]. En el trabajo se usa como capa de servicio a PVM 3.4.3[PVM] en la red de 100Mb/s y PVM 3.3.11[PVM] en las redes de 10Mb/s. Para MPI [MPI] se tiene un abanico amplio de posibles implementaciones, las más usadas sobre redes de estaciones de trabajo y de dominio público son LAM/MPI [LAM][LAMb], MPICH[MPICH][MPICHb] y CHIMP[CHIMP]. La elección fue LAM/MPI en la versión 6.5.2 por varias razones, el tamaño del tarball[1] para instalarlo, la simplicidad de instalación, y la más importante la experiencia previa que ya se tenía con esta distribución. Además de ésto se buscarón reportes que comparen dichas distribuciones sobre redes de estaciones de trabajos y la documentación que se encontró indicaba que la biblioteca de MPICH en modo directo seteaba las conexiones bajo demanda a diferencia de LAM/MPI que lo hacia en la inicialización [Nevin96], esto hacia un poco más complicado las benchmarks sobre el primero, y termino por definir.

5.2.3 Lenguajes y Compilación

Las bibliotecas usadas, PVM y LAM/MPI, están disponibles tanto para ser usadas en *C*, *C++*[Strous91][Weisk90] y *Fortran*[FTRAN]. La elección fue *C*. Para el compilador en todos los casos se uso gcc con licencia GNU, en su versión egcs-2.91.66. No se usaron opciones de optimización para el compilador.

5.3 Benchmarks

5.3.1 Metodología de Medición

Estudio Previo de las Técnicas de Medición

Para medir los parámetros descriptos en el capítulo de métricas existen varias propuestas, pero la mayoría está destinada a comunicaciones punto a punto. Se encontró en [Don96] la propuesta del "Echo test" donde existe un proceso llamado server que manda N veces datos al cliente, quien, por cada vez que recibe los reenvía de vuelta. El trayecto de ida y vuelta es conocido como *round trip* y el tiempo que este dura como *round trip time (RTT)*, para determinar el tiempo de un envío se debe dividir por dos, *RTT/2*. Además como se itera N se obtiene:

$$CommTime = \frac{(StopTime - StartTime)}{2N} \tag{5.1}$$

[1] Conjunto de archivos estructurados en directorios archivados con el comando tar (tape archiver) y comprimidos

Esta técnica también es conocida como "Ping-Pong" (ver figura 5.1)[Nupairoj].

```
P1:  /* Server */        P2:  /* Client */
Initialize               Initialize
Start timer
For I=1 to N             While (true)
 Send                     Recv
 Recv                     Send
End For
Stop Timer
```

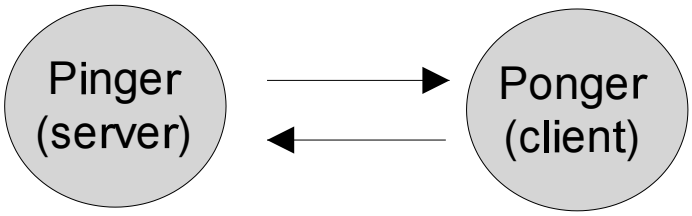

Figura 5.1: Modelo Ping-Pong de comunicaciones

Si bien estas técnicas son bien conocidas y usadas ampliamente no sirven para ser usadas directamente sobre comunicaciones colectivas.
Otra técnica para medir los tiempos de comunicaciones es el uso de un reloj global sincronizado [Ni]. Por ejemplo uno provisto por MPI, que se accede mediante la función `MPI_Wtime` [2], o a más bajo nivel sincronizar los hosts mediante el *timed (time server daemon)* [TIMED89] o alguna implementación del *NTP (Network Time Protocol)* [Mills91] como puede ser xntp, y luego acceder con la API tradicional ofrecida por el sistema operativo. El uso de relojes sincronizados genera intrusión y complejidad en la medición convirtiendo la técnica en poco apta para el propósito del trabajo.
El problema es atacado en [Nupairoj] donde se mide el *broadcast* y el *barrier*. La técnica empleada indica medir repetidamente el tiempo antes y despúes de la operación desde el proceso `root` [3], por supuesto el *broadcast* debe ser bloqueante, pues se debe considerar la duración de la operación como el tiempo total que tarda en completarse para todos los nodos, si esto no es así la técnica no sirve. Además no se menciona qué pasa con los tiempos de espera hasta que todos sincronizan en la operación. Esto último puede ser solucionado sincronizando a todos antes de comenzar a medir los tiempos mediante un *barrier* como se describe en [Luecke00].

[2]Para que todos los nodos participantes estén sincronizados el atributo MPI_WTIME_IS_GLOBAL debe estar configurado a true

[3]El barrier carece de proceso root por lo cúal se debe escoger alguno del los participantes

```
Pi:
Initialize
S = Get Timer
For I=1 to N
 Bcast
End For
E = Get Timer
T= (E - S) / N
```

Medición del *broadcast* bloqueante.

Las dificultades de la medición de las operaciones colectivas son enumeradas en [Ni] donde además se propone una técnica para la medición del *multicast*:

- La carencia de un reloj global en la mayoría de los sistemas de memoria distribuida (Las redes de estaciones de trabajo forman parte de esta categoría).
- La resolución de los relojes a menudo no es buena.
- El intervalo entre dos *multicasts* consecutivos no sirve como medición de la latencia (técnica que usa el ping-pong).
- Si se usa la técnica del ping-pong la respuesta de todos los receptores provocaría *channel contention* dando resultados incorrectos.
- La dificultad de encontrar cúal será el último nodo en recibir y que este se mantenga invariante a lo largo de todas las pruebas.

Las propuesta que se ofrece para medir el *multicast* tiene dos pasos. El primero consiste en predecir cúal será el último nodo en recibir los datos y el segundo es la ejecución de la operación en sí. La predicción se hace en base al algoritmo usado por la implementación del *multicast*. En cada paso se va cambiando el `root`, el último en recibir tendrá este rol en la próxima iteración, esto asegura que la siguiente prueba no comenzará hasta que no haya terminado la anterior. El cálculo de la duración se hace tomando el tiempo en el proceso `root` antes de la operación y luego tomándolo al recibir un *ack (acknowledge)* de un nodo arbitrario. A esto se le resta el tiempo del *ack* que es el costo de transmitir un byte (El *ack* se calcula usando la técnica clásica del ping-pong punto a punto).

```
P1:  /* Root */                  | Pn:  /* Last */
ACK = Previously Calculated      |
Initialize                       | Initialize
S = Get Timer                    |
 Bcast                           |  Bcast
 Recv Ack                        |  Send Ack
E = Get Timer                    |
T= (E - S) - ACK                 |
```

Un paso en la medición del *broadcast* usando predicción del último receptor.

Esta forma de medir parece bastante compleja, a esto se le agrega la necesidad de evitar la *channel contention* entre el *ack* y los datos enviados al último

por lo cual antes del *ack* se debe esperar un tiempo muerto mayor que el más lento (Se podría evitar esperando el *ack* del último).
Todo este estudio previo sirvió para resaltar la necesidad de diseñar e implementar técnicas más sencillas y eficaces para la medición, desde luego siguiendo como guía lo anterior:

Técnica de Medición Desarrollada

La primera diferencia con las técnicas anteriores es la ubicación de la iteración. La idea es poder medir también el costo que hay en el start-up, por lo cual el *for* se sube de nivel. Para poder saber cúal fue el costo de la operación se debe conocer el costo para cada participante, incluso `root`.

Primero se ejecuta la colectiva, luego se recogen los costos y se calcula cúal fue el mayor de todos.

Existe una pequeña especialización de la técnica de acuerdo a la operación a medir. Para el *broadcast*, el *multicast* y *scatter* no hay variación con lo explicado:

`P1: /* Root */`	`Pi: /* (i!=1) - Receiver */`
`For I=1 to N`	`For I=1 to N`
`Initialize`	`Initialize`
`Sync Barrier`	`Sync Barrier`
`S = Get timer`	`S = Get timer`
`Bcast/Mcast/Scatter`	`Bcast/Mcast/Scatter`
`E = Get timer`	`E = Get timer`
	`ACK = E - S`
`Recv all acks`	`Send ACK`
`M= MAX(E-S,ACK2,ACK3,...)`	
`Finalize`	`Finalize`
`End For`	`End For`

Medición del *scatter*, el *multicast* o el *broadcast*

Para el barrier es similar con la diferencia que se calcula el promedio y no el máximo, aunque podría igualmente hacerse de esta última forma. Se usa el promedio porque al ser una operación por naturaleza bloqueante la varianza es escasa y el promedio es un buen estimador de la media.
Para el *gather* y el *reduce* el valor usado es el tiempo que tarda en hacerse la operación desde `root`. Puede haber una diferencia de tiempo entre el *barrier* para sincronizar y la llamada al *gather*.

Medición punto a punto necesarias

Las pruebas punto a punto también fueron necesarias para conocer cuáles son los parámetros sobre los cuales están asentadas las colectivas. Se realizaron a tres niveles dentro del modelo OSI:
Las pruebas a más bajo nivel que se realizaron fueron hechas en la capa 3 del modelo OSI, la capa de Red (Network), la cual mapea a la capa de Internetworking sobre la suit TCP/IP. Estas fueron hechas utilizando el comando `ping (8)` [StevensI98]. Este es un sencillo programa que viene como herramienta de testeo para la mayoría de los S.O. que tienen soporte de TCP/IP. La herramienta

genera paquetes ICMP (Internet Control Message Protocol) de requerimiento de "echo" hacia una dirección IP determinada la cual debe remitir el "pong" (la respuesta). Cada vez que envía un paquete toma el tiempo y luego vuelve a tomarlo al recibir la respuesta, así calcula lo que se conoce como *RTT (Round Trip Time)*.
Las pruebas siguientes suben un nivel en el modelo OSI, son sobre la capa de Transporte (Transport), la que corresponde a TCP (Transmission Control Protocol) y UDP (User Datagram Protocol) en la suit desarrollada por el DoD. Los resultados fueron obtenidos usando el programa `netperf` [Netperf] [4] y con programas propios desarrollados sobre sockets [ComerIII96]. Una mejor alternativa a `netperf` es `netpipe` (Network Protocol Independent Performance Evaluator) [Netpipe] [5] que tiene licencia GNU. Estos programas se han estandarizado como herramientas para medir el rendimiento de la red usando un esquema cliente/servidor. Las pruebas con TCP se hicieron en dos modos, con la opción `TCP_NO_DELAY` y sin esta. Si ésta opción esta activada se deshabilita el *nagle algorithm* (piggy backing) [StevensI98].
Las últimas pruebas realizadas fueron usando PVM y MPI con la metodología de medición tradicional de punto a punto. Para LAM/MPI se uso el modo *client-to-client* (c2c) donde se rutea de forma directa y *daemon-to-daemon* (lamd) donde el envío de mensajes es a través de procesos servidores intermediarios. PVM aporta las mismas formas de ruteos llamadas *default* y PvmRouteDirect con las cuales también se experimentó.

5.3.2 Modo de las Pruebas

En todas las operaciones colectivas se uso una tarea por máquina, así en el caso de la primera red se tuvo 4 tareas máximas, en el segundo 8 y en el último 6. Todas las operaciones colectivas estudiadas, salvo el *barrier* necesitan tener una tarea denominada `root` que para las pruebas ejecutaron en *lidi34*, *purmamarca* y *lidipar14*. La elección de estas máquinas se debe a que tienen memoria suficiente para almacenar todos los tamaños de mensajes y son las más rápidas dentro de la red a la cual están conectadas.

5.3.3 Tamaños de Datos y Misceláneos

Tamaños de Datos

- Los tamaños usados para las pruebas de *multicast* y *broadcast* son: 8, 128, 1024, 16384, 130600, 1167200 Floats.

- Los tamaños de las porciones usadas para las pruebas de *scatter*, *gather* y *reduce* son: 8, 128, 1024, 16384, 130600, 783600 Floats.

- Se selecciono como unidad el Float debido a que el trabajo está orientado en general a aplicaciones de cálculo numérico y más precisamente del álgebra lineal, para las cuales este tipo de dato es un buen representante.

[4] Copyright ©1993 Hewlett-Packard Company ALL RIGHTS RESERVED
[5] Copyright ©1997, 1998, 1999 Iowa State University Research Foundation, Inc. with GNU General Public License as published by the Free Software Foundation

Pruebas realizadas

- Se realizaron 4 corridas de los benchsmarks de 16 tests (iteraciones) por cada middleware y configuración diferente. Se tomaron y analizaron los mejores resultados de una de las 4.
- Para la red de 10Mb/s del LIDI la secuencia fue:
  - 2 computadores desde *lidi34* a *lidi35*
  - 3 computadores desde *lidi34* a *lidi35* y *lidi48*
  - 4 computadores desde *lidi34* a *lidi35*, *lidi48* y *lidi47*
- Para la red de 10Mb/s del CeTAD la secuencia fue:
  - 2 computadores desde *purmamarca* a *maimara*
  - 3 computadores desde *purmamarca* a *maimara* y *cetadfomec1*
  - 4 computadores desde *purmamarca* a *maimara*, *cetadfomec1* y *2*
  - 5 computadores desde *purmamarca* a *maimara*, *cetadfomec1*, *2* y *kuntur*
  - 6 computadores desde *purmamarca* a *maimara*, *cetadfomec1*, *2*, *kuntur* y *tilcara*
- Para la red de 100Mb/s la secuencia fue:
  - 2, 3, ..., 8 computadores desde *lidipar14* a *lidipar5*, *6*, *8*, *11*, *12*, *13* de forma escalonada.
- Para PVM las combinaciones de configuraciones fueron:
  - *Multicast* y *broadcast*:
    - PvmDataDefault PvmDontRoute **d - u**
    - PvmDataDefault PvmRouteDirect **d - t**
    - PvmDataRaw PvmDontRoute **w - u**
    - PvmDataRaw PvmRouteDirect **w - t**
    - PvmDataInPlace PvmDontRoute **p - u**
    - PvmDataInPlace PvmRouteDirect **p - t**
  - *Scatter*, *gather* y *reduce*:
    - PvmDataDefault PvmDontRoute **d - u**
    - PvmDataDefault PvmRouteDirect **d - t**
  - *Barrier*:
    - PvmDontRoute **u**
    - PvmRouteDirect **t**
- Para MPI las combinaciones de configuraciones fueron:
  - *Broadcast*, *scatter*, *gather* y *reduce*:
    - Default lamd **d - u**
    - Default client to client **d - t**
    - Raw lamd **w - u**

    - Raw client to client **w - t**
- *Barrier*:
    - lamd **u**
    - client to client **t**

Para el reduce se selecciono la operación `min` (mínimo) tanto para PVM como para MPI. Los arreglos de datos fueron llenados con valores desde cero hasta la capacidad del arreglo menos uno *size-1*.

Bibliografía

[CCWP] Cluster Computing White Paper version 2.0. Editor Mark Baker University of Portsmouth, UK. Middleware, Mark Baker and Amy Apon, University of Arkansas, USA. 28th Dec 2000.

[PVM] PVM version 3.3.0 and version 3.4.0 are available on Netlib site (http://www.netlib.org/pvm) - 12 March 1999.

[MPI] MPI: The Complete Reference - Marc Snier, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra - The MIT Press, Cambridge, Massachusetts - 1996.

[LAM] LAM/MPI - University of Notre Dame (http://www.mpi.nd.edu/lam) - 1998-2001.

[MPICH] MPICH - Mathematics and Computer Science Division, Argonne National Laboratory (http://www.mcs.anl.gov/mpi/mpich) - Sep. 2000.

[CHIMP] R. Alasdair, A. Bruce, J. G. Mills, and G. Smith, CHIMP Version 2.0 User Guide. University of Edinburgh. Mar 1994.

[Nevin96] Nick Nevin, The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster, Ohio Supercomputer Center Technical Report OSC-TR-1996-4, Columbus, Ohio, 1996.

[FTRAN] X3J3 Subcommittee. American National Standard Programming Language Fortran (X3.9-1978). American National Standards Institute, 1978.

[Strous91] B. Stroustrup. The C++ Programming Language. Addison-Wesley, second edition, 1991.

[Weisk90] Keith Weiskamp, Bryan Flamig, The Complete C++ Primer. Academic Press, 1990.

[MPICHb] B. Gropp, R. Lusk, T. Skjellum, and N. Doss, Portable MPI Model Implementation, Argone National Laboratory, July 1994.

[LAMb] G. Burns, R. Daoud, and J. Vaigl, LAM: An Open Cluster Environment for MPI. Ohio Supercomputer Center, May 1994.

[Don96] Jack Dongarra and Thomas H. Dunigan, Message-Passing Performance of Various Computers. Mathematical Science Section, Oak Ridge National Laboratory. Report ORNL/TM-13006, Feb 1996.

[Nupairoj] Natawut Nupairoj and Lionel M.Ni. Performance Evaluation of Some MPI Implementations on Workstation Clusters. Department of Computer Science, Michigan State University.

[TIMED89] Gusella, R.and Zatti, S. (1989) The accuracy of clock synchronization achived by TEMPO in Berkeley Unix 4.3BSD. IEEE Trans. Software Engineering vol. 15, pp. 847-53.

[Mills91] Mills, D. L.(1991), Internet Time Synchronization: the Network time Protocol. IEEE Trans. on Comms, vol. 39, no. 110. pp. 1482-93. Electrical Engineering Department, University of Delaware.

[Luecke00] G. Luecke , Jing Yuan , S. Spanoyannis, M. Kraeva, Performance and Scalability of MPI on PC Clusters - Jan 23, 2000.. Durham Center, Iowa State University.

[Ni] Natawut Nupairoj and Lionel M. Ni, Benchmarking of Multicast Communication Services. Technical Report MSU-CPS-ACS-103. Department of Computer Science, Michigan State University.

[StevensI98] TCP/IP Illustrated, Volume 1, The Protocols. W. Richards Stevens. Addison Wesley Longman Inc. 1. 11th. Printing February 1998.

[ComerIII96] Internetworking with TCP/IP, Volume 3, Client-Server Programming and Applications. Douglas E. Commer and David L. Stevens. Prentice Hall Inc., 1996.

[Netperf] Netperf on the web http://www.cup.hp.com/netperf/NetperfPage.html.

[Netpipe] Netpipe on the web http://www.scl.ameslab.gov/netpipe/.

Capítulo 6

Pruebas Punto a Punto

6.1 Introducción

Antes de analizar los resultados de las pruebas usando las operaciones colectivas se debe caracterizar el rendimiento de las comunicaciones punto a punto, que, como se mencionó en el capítulo de *Modelos y Métricas* en muchas situaciones son la base de las colectivas mismas.
Nótese que la magnitud de los tamaños de datos para las pruebas de ICMP, TCP y UDP, en todos los gráficos está expresada en bytes y no en floats.

6.2 Resultados Obtenidos

6.2.1 Resultados ICMP

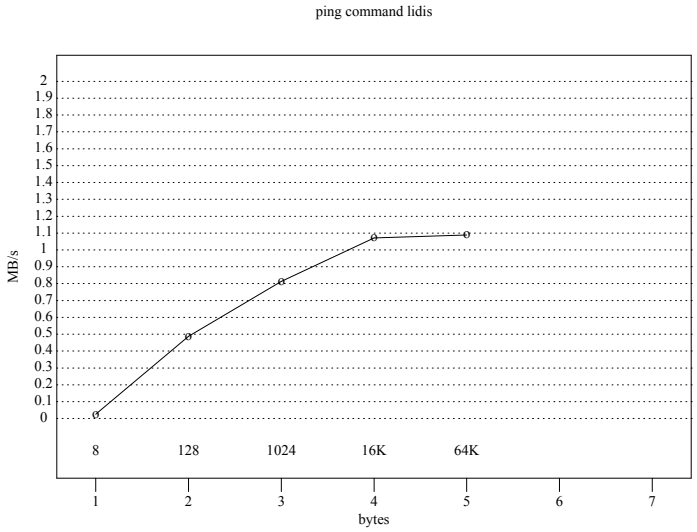

Figura 6.1: ICMP sobre red heterogénea del LIDI - 10Mb/s

Los gráficos 6.1 y 6.2 muestran los resultados del ancho de banda obtenido con el comando `ping (8)`. Los valores volcados en el gráfico se logran a partir de hacer el *RTT/2*. Las pruebas se hicieron desde las mismas máquinas seleccionadas para alojar los procesos `root` de las colectivas (Ver capítulo *Entorno de Prueba*). Los tamaños de datos usados llegan hasta 64KB debido a la limitación del tamaño del paquete ICMP [1]. En los gráficos se observa que el rendimiento es muy cercano al físico, esto se debe básicamente al mínimo overhead.

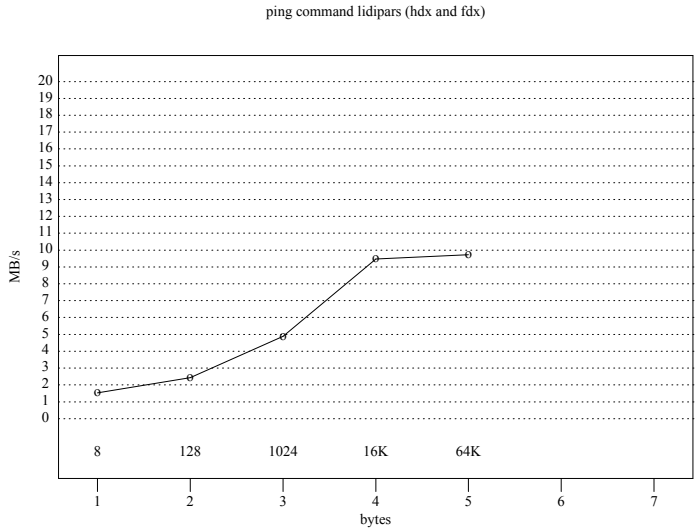

Figura 6.2: ICMP sobre cluster homogéneo - 100Mb/s

6.2.2 Resultados TCP/UDP

Las pruebas mostradas en las figuras 6.3, 6.4, 6.5 y 6.6 son sobre la capa de Transporte (Transport). En los gráficos 6.3 y 6.4 se observa que el start-up es mejor que la obtenido con ICMP, cuestión que no debería ocurrir. Esto sucede debido a que `netperf` no esta teniendo en cuenta este factor porque las pruebas realizadas son de *Streaming* o transferencia de volumen (en inglés bulk) donde se mide cuan rápido un sistema envía y cuan rápido un sistema recibe. Para tener en cuenta la latencia se deberían hacer pruebas de *Request/Response*.
El verdadero start-up se encuentra con las pruebas usando los programas de benchmarks desarrollados por el autor. Otra diferencia notoria entre los resultados con `netperf` es el alto ancho de banda alcanzado usando UDP y el start-up es más notorio usando este protocolo. Lo primero se debe a que las pruebas son unidireccionales y el protocolo posee menos overhead. Lo siguiente se debe a que UDP es un protocolo no confiable por lo que se deben analizar los resultados con más cuidado ya que para pocos datos el rendimiento para el envió es mucho más alto que para la recepción, debido a esto los reportes deben contemplar el valor obtenido del lado del receptor.

[1] Los paquetes ICMP se encapsulan en paquetes IP los cuales tienen un tamaño máximo de 64KB en IPv4

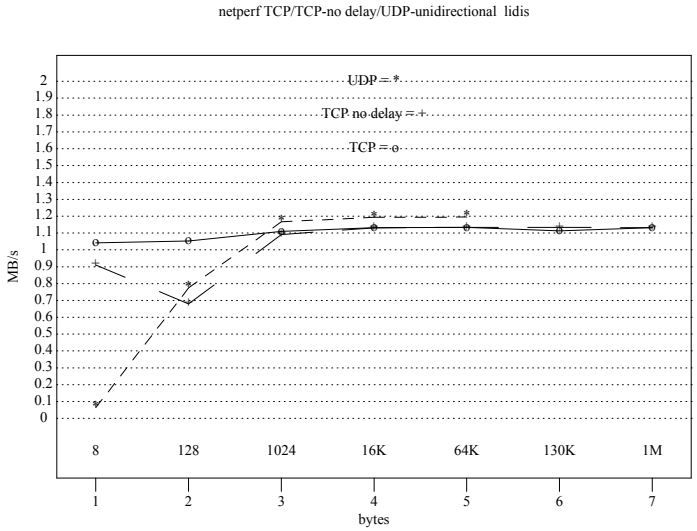

Figura 6.3: TCP/UDP (netperf) sobre red heterogénea del LIDI 10Mb/s

Las conclusiones que se obtienen a partir de 6.5 y 6.6 son que para la red de 10Mb/s se tiene un buen rendimiento a bajo nivel para las comunicaciones y a partir de los 16K bytes se alcanza un máximo, aunque algo alejado del pico físico. Los resultados para TCP y UDP son parecidos y el *nagle algorithm* no mejora las cosas. Para la red de 100Mb/s el rendimiento no es malo pero está en proporción, aún más lejos del pico físico. El máximo se alcanza también para 16 K bytes. Si se mira a 6.3 y 6.4 se ve que para la red de 10Mb/s el ancho banda alcanzable es cercano al óptimo, cuestión que no sucede en la red de 100Mb/s para TCP (Esto se debe al switching hdx con pruebas bidireccionales de *Streamming*).

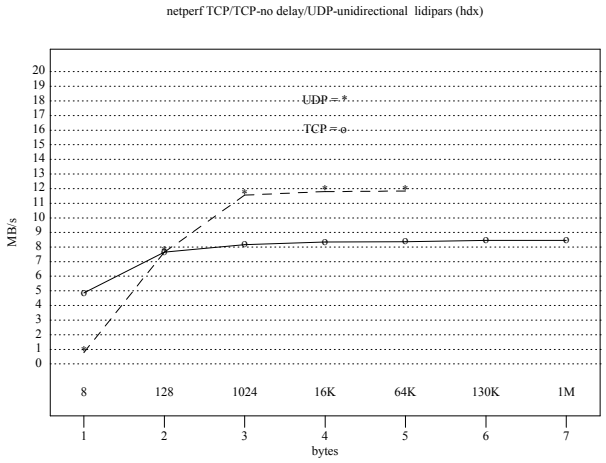

Figura 6.4: TCP/UDP (netperf) sobre cluster homogéneo - 100Mb/s (hdx)

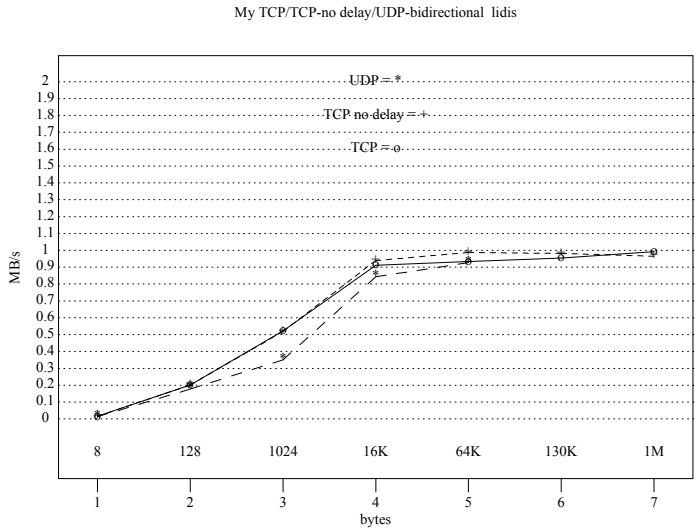

Figura 6.5: TCP/UDP (programas de tests propios) sobre red heterogénea del LIDI - 10Mb/s

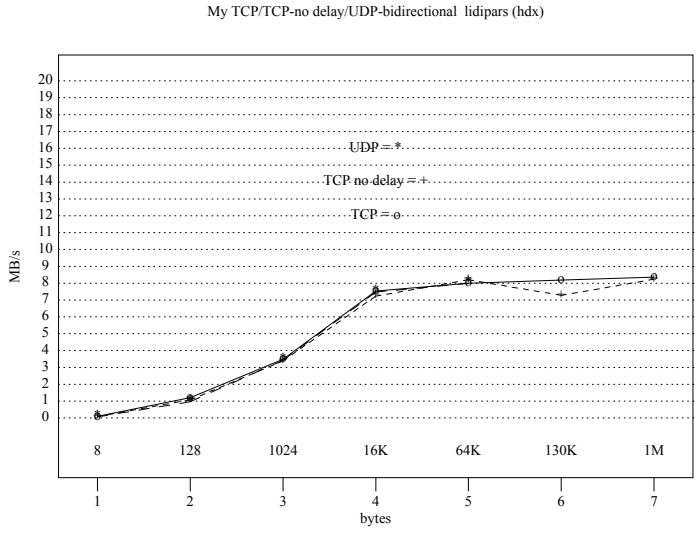

Figura 6.6: TCP/UDP (programas de tests propios) sobre cluster homogéneo - 100Mb/s

6.2.3 Resultados para PVM y MPI

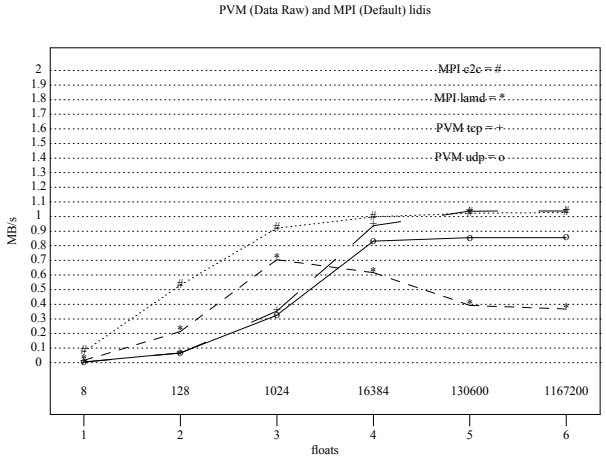

Figura 6.7: PVM/MPI sobre red heterogénea del LIDI - 10Mb/s

Usando PVM y MPI se midió el rendimiento con las primitivas básicas de comunicaciones que ofrecen ambos *middlewares*. En los resultados se destaca que LAM/MPI en modo directo (c2c) es superior a PVM, aunque a partir de los 16KFloats comienzan a tener un rendimiento similar. El límite superior esta dado por el desempeño de TCP/IP al nivel del kernel (bajo nivel). Sobre la red de 10Mb/s LAM/MPI usando el modo *daemon-to-daemon* el rendimiento es muy pobre para muchos datos, cuestión que no sucede sobre la red de 100Mb/s.

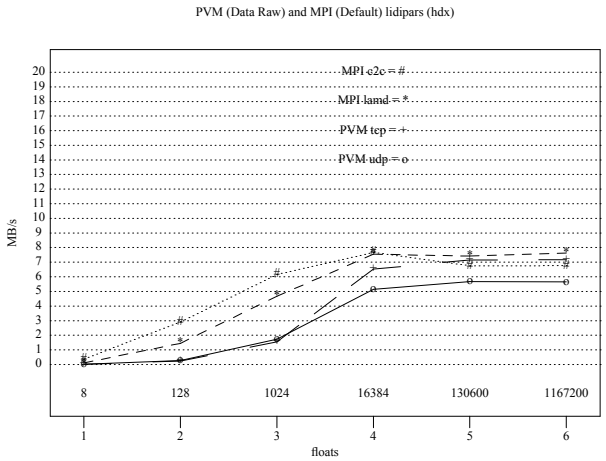

Figura 6.8: PVM/MPI sobre cluster homogéneo - 100Mb/s

Capítulo 7

Resultados de las Pruebas I

7.1 Introducción

Para mostrar y analizar las pruebas de las comunicaciones colectivas en PVM y LAM/MPI se ha decidido presentarlas en dos capítulos por razones de practicidad y para no generar secciones extremadamente largas. En este capítulo (*Resultados de las Pruebas I*) se ve el *barrier*, el *broadcast*, y al final, se analiza el *multicast* que solo es provisto por PVM [1]. En el próximo (*Resultados de las Pruebas II*) se ve el *gather*, *scatter* y *reduce*.
Un punto a recordar es que la implementación de MPI elegida es LAM/MPI, en el capítulo cuando se haga referencia a MPI debe leerse como la implementación usada: LAM/MPI.

7.2 Resultados Obtenidos

7.2.1 Barrier

Los gráficos que se presentan a continuación tienen sobre el eje X las barras agrupadas de a pares para la misma cantidad de hosts, la primera corresponde al ruteo default y la segunda a *PvmRouteDirect*.
Comenzando con el *barrier* que es el benchmark más simple. Los resultados en la fig. 7.1 muestrán que en la red del LIDI, PVM se comporta de la forma esperada, a medida que se adicionan hosts el tiempo requerido para la operación crece. Se advierte que con ruteo *PvmRouteDirect* es levemente mejor, y que al agregar máquinas con hardware algo anticuado (y por lo tanto más lento) el rendimiento se ve degradado en una proporción mayor, esto sucede con *lidi47*.
En la fig. 7.3 los resultados para una red de caracterísiticas similares son un poco mejor aunque no se alejan demasiado de lo encontrado en la red del LIDI.
Los resultados obtenidos en el cluster mostrados en la fig. 7.2 son similares a los anteriores salvo claro, que los tiempos son de un orden de magnitud más chicos. Se encuentra un valor bastante anómalo con 5 hosts: sucede que el

[1] En el manual de MPI explicítamente dice [MPI]: MPI no soporta una función de multicast, ... Tal función es fácil de implementar directamente si el root envía directamente a cada receptor.

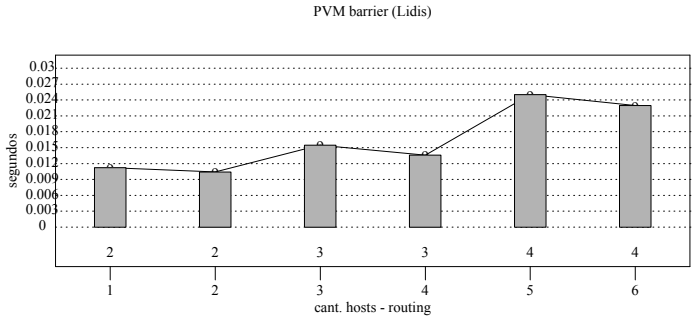

Figura 7.1: PVM barrier sobre red heterogénea del LIDI 10Mb/s

ruteo *PvmRouteDirect* es más lento que el default y los tiempos obtenidos en esta instancia son mayores que ejecutando con 6 y 7 hosts.

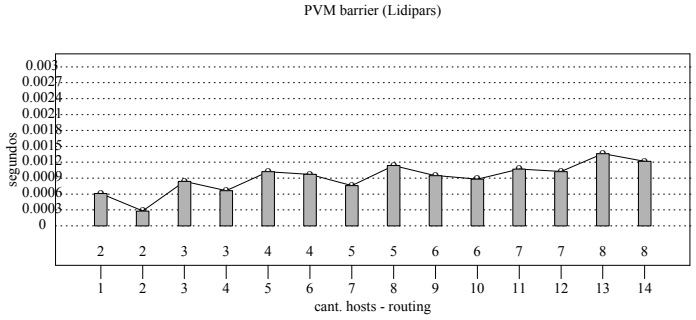

Figura 7.2: PVM barrier sobre cluster homogéneo 100Mb/s

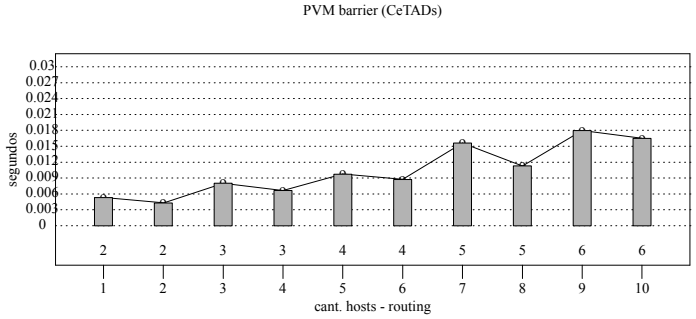

Figura 7.3: PVM barrier sobre red heterogénea del CeTAD 10Mb/s

Con LAM/MPI sobre el eje X se tienen las barras de a pares, de forma similar que se hizo con PVM para la misma cantidad de hosts, la primera corresponde al ruteo default que es *daemon-to-daemon* y la segunda a *route direct* o ruteo directo entre procesos.
Los resultados parecen ser bastante mejores. Los tiempos para la red de 10Mb/s mostrados en la fig. 7.4 son de 2 a 3 veces menores que los de PVM si se los compara con la fig. 7.2. También se encuentra el aumento fuera de lo esperado para *lidi47* usando ruteo default. Para la red de 100Mb/s los resultados con ruteo directo superan holgadamente a los de PVM, en cambio los de ruteo *daemon to daemon* o default son similares. En la fig. 7.5 se resalta la diferencia entre un ruteo y el otro. La ventaja que logra LAM/MPI sobre PVM sobre 100Mb/s se debe a que basa su algoritmo para 5 o más procesos en un árbol el cuál aprovecha la capacidad de switching de la red.

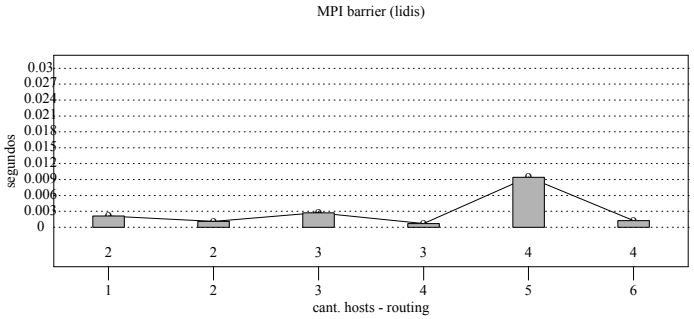

Figura 7.4: MPI barrier sobre red heterogénea del LIDI 10Mb/s

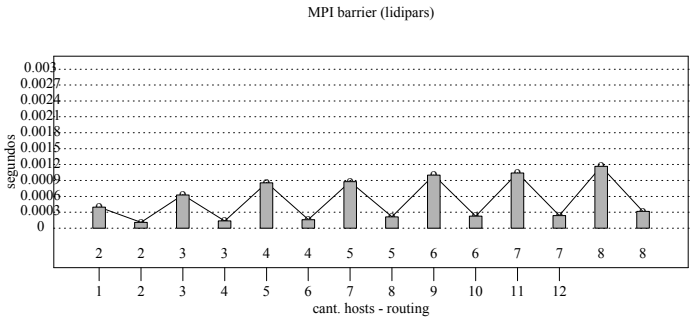

Figura 7.5: MPI barrier sobre cluster homogéneo 100Mb/s

7.2.2 Broadcast

El broadcast es el benchmark más importante para este estudio. Se analiza primero el start-up y luego el ancho de banda. Las gráficas comparativas son sólo del ancho de banda.

Broadcast en PVM

Es necesario antes de hacer el análisis de los resultados obtenidos con PVM aclarar que éstos fueron bastante incomprencibles y, se necesitó hacer gran cantidad de tests hasta alcanzar una configuración que se vea lo menos afectada por efectos laterales (side-effects), como son las pruebas que se realizaron antes. Los problemas encontrados se acentuaron principalmente en la red de 10Mb/s del LIDI.
Los problemas encontrados sobre esta red fueron:

- Si se hacen pruebas de *broadcast* con pocos datos (8 a 128 Floats) y luego se hacen pruebas con más datos (16K, 130K o 1M Floats) sin reinicializar a la PVM los tiempos obtenidos en estas últimas son tres veces superiores de lo esperado, por ejemplo para 1MFloat se esperaba aprox. 10 seg. y los tiempos obtenidos están entre 30 y 33 seg.

- Otro problema que se encontró fue cuando se realizó el *multicast* previo al *broadcast*, se obtuvo también tiempos mucho más altos de lo esperado.

- Otro efecto lateral menos relevante es el host donde ejecuta el *pvmgs* , esto puede ser solucionado usando grupos estáticos o encontrando en que host debe correr el *pvmgs* para no tener problemas y forzar a que PVM lo "arranque" allí.

El start-up (tabla 7.2) en la red de 10Mb/s del LIDI es bastante alto, sus valores están entre 0.002 y 0.003 segundos (para ambos ruteos). Debido a que la cantidad de hosts de la red es chica y que los tiempos son demasiado altos el aumento de máquinas no impacta en los tiempos. Cabe destacar que la codificación para el start-up es insignificante, esto es de esperarse debido a que los datos usados en las pruebas son los mínimos. En la red del CeTAD los resultados son similares salvo que se advierte que el ruteo directo es un orden de magnitud mejor que el default, situación que en el caso anterior no sucedía. Entre 1 y 3 hosts destinos no se nota el incremento de los tiempos y sus valores son 0.005 para ruteo default y 0.0003 para *PvmRouteDirect*. Se nota un aumento del start-up a partir de los 4 hosts destinos. En la red de 100Mb/s los tiempos son parejos desde 1 hasta 7 hosts destinos.

Network	Hosts	Default Routing	Direct Routing
LIDI (10Mb/s)	1..3	0.0030	0.0020
CeTAD (10Mb/s)	1..3	0.0050	0.0003
CeTAD (10Mb/s)	4..5	0.0080	0.0010
LIDIpars (100Mb/s)	1..7	0.0007	0.00030

Tabla 7.1: Start-up del *broadcast* en PVM

En el análisis del ancho de banda es de esperarse que la curva sea casi monótonamente creciente y asintótica bajo un paralela al eje X, esto con determinadas configuraciones (Codificación, ruteo y cantidad de hosts) no sucede. Observando primero los resultados en la red de 10Mb/s del LIDI, para un receptor del *broadcast* usando codificación XDR[XDR] (tanto en ruteo directo como default), figs. 7.7 y 7.6 se encuentra que la curva del ancho de banda tiene un máximo local en los 1024 floats y un mínimo local en los 130600 floats. Esto es bastante extraño ya que debería ser similar a una comunicación punto a punto. Si no se usa codificación el ancho de banda con un receptor es como una punto a punto mostrando una curva creciente y asintótica (fig. 7.8).

Para dos receptores (fig. 7.9) el comportamiento es similar pero el ancho de banda alcanzado se ve reducido casi a la mitad, esto es debido a que PVM resuelve los *broadcast* con varias comunicaciones punto a punto. Otra característica para destacar es que los mínimos y máximos locales comienzan a aparecer en comunicaciones sin codificación aunque se dan de forma menos acentuada.

Para tres receptores el ancho de banda se reduce aún más y las variaciones de los máximos y mínimos se conservan aunque la del mínimo se desplaza a los 16KFloat. El no uso de codificación aumenta el ancho de banda y se observa comparando en las figuras 7.10 y 7.11 las tres últimas barras, pero solo es útil sobre plataformas con igual representación de datos.

En general el ancho de banda que se debería ganar con el ruteo directo según las recomendaciones de PVM con respecto al default es escaso.

En los gráficos el ruteo directo se especifica con una **t** (de TCP) y el default con una **u** (de UDP). La codificación usando XDR se nota con **d** (default) y sin codificación con **p** (in place). Muchos gráficos se omiten para acortar el capítulo, por ejemplo los que tienen codificación raw (cruda, sin codificar) no se ponen porque son similares a los **p** (in place). La diferencia entre unos y otros es el la carencia de buffers en los últimos.

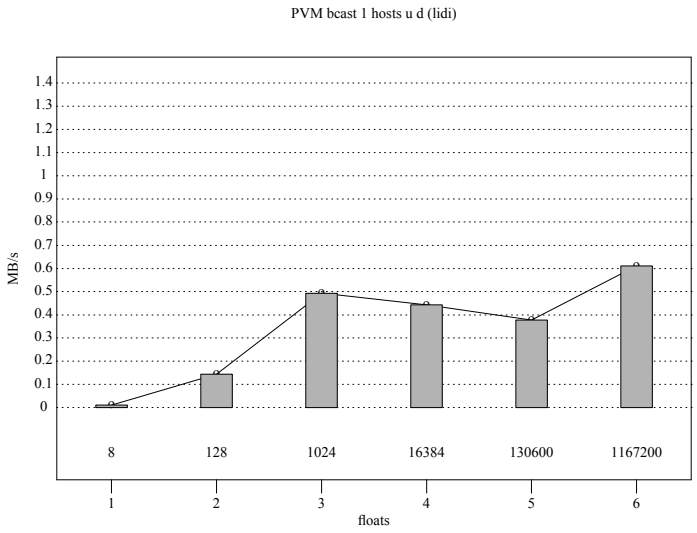

Figura 7.6: PVM *broadcast* sobre red heterogénea del LIDI 10Mb/s - 1 receptor - *daemon-to-daemon* - XDR

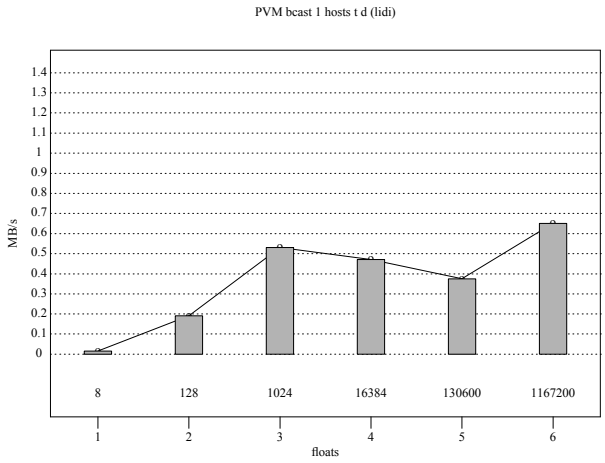

Figura 7.7: PVM *broadcast* sobre red heterogénea del LIDI 10Mb/s - 1 receptor - directo - XDR

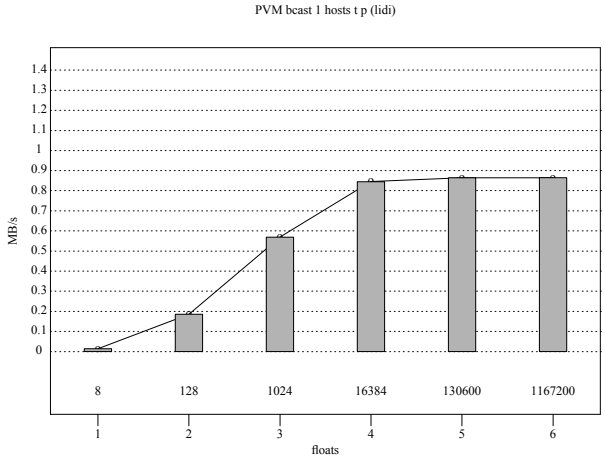

Figura 7.8: PVM *broadcast* sobre red heterogénea del LIDI 10Mb/s - 1 receptor - directo - In Place

Para corroborar los resultados de los picos altos y bajos obtenidos en el ancho de banda en la red del LIDI y descartar sospechas de que sean causa de algún problema en la red se llevaron las pruebas al CeTAD donde los resultados fueron similares. Se obtienen los máximos y los mínimos usando XDR para un receptor como se ve en la fig. 7.12 y luego éstos aparecen también sin usar codificación para más receptores. Estos resultados se muestrán en las figuras 7.15 y 7.17. Si se comparan estos gráficos con los gráficos 7.14 y 7.16 donde se usa codificación se advierte que también el no uso de codificación mejora el ancho de banda para cantidades de datos significantes (a partir de 16KFloats). El uso de ruteo directo no mejora el rendimiento.

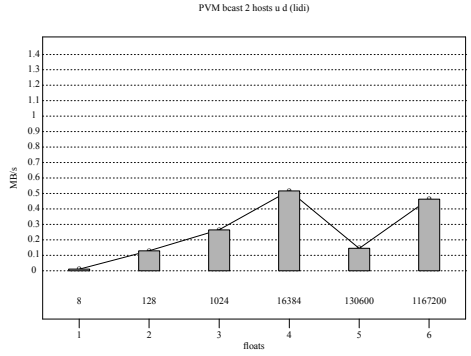

Figura 7.9: PVM *broadcast* sobre red heterogénea del LIDI 10Mb/s - 2 receptores - daemon to daemon - XDR

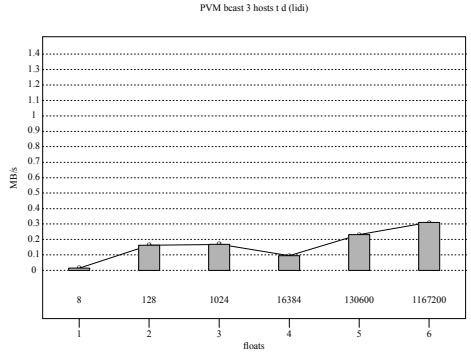

Figura 7.10: PVM *broadcast* sobre red heterogénea del LIDI 10Mb/s - 3 receptores - directo - XDR

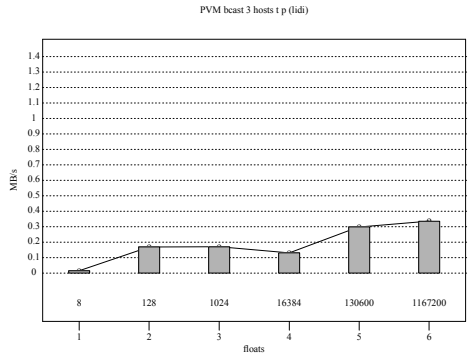

Figura 7.11: PVM *broadcast* sobre red heterogénea del LIDI 10Mb/s - 3 receptores - directo - In Place

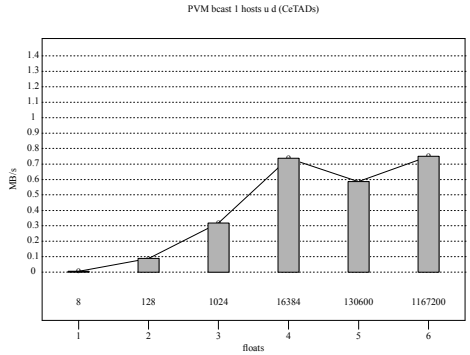

Figura 7.12: PVM *broadcast* sobre red heterogénea del CeTAD 10Mb/s - 1 receptor - *daemon-to-daemon* - XDR

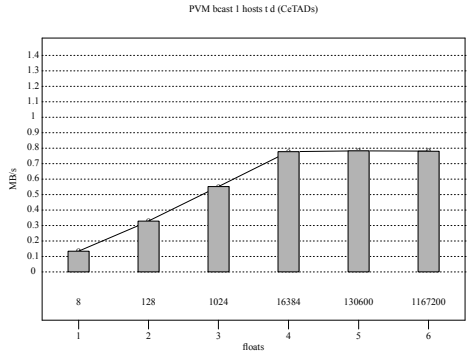

Figura 7.13: PVM *broadcast* sobre red heterogénea del CeTAD 10Mb/s - 1 receptor - directo - XDR

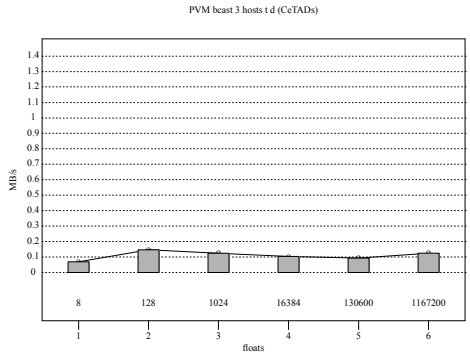

Figura 7.14: PVM *broadcast* sobre red heterogénea del CeTAD 10Mb/s - 3 receptores - directo - XDR

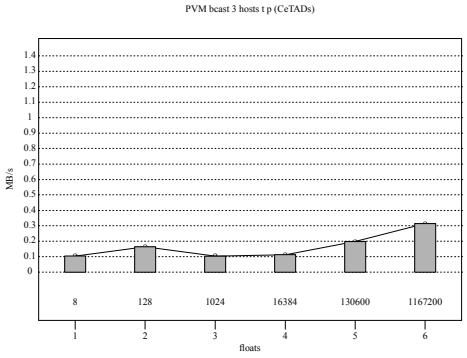

Figura 7.15: PVM *broadcast* sobre red heterogénea del CeTAD 10Mb/s - 3 receptores - directo - In Place

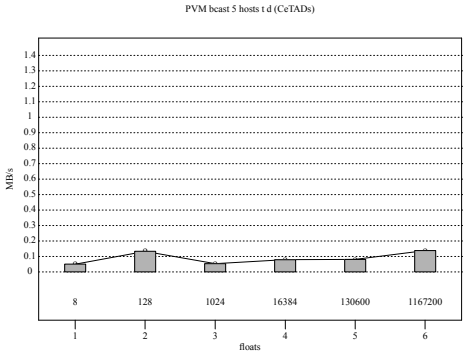

Figura 7.16: PVM *broadcast* sobre red heterogénea del CeTAD 10Mb/s - 5 receptores - directo - XDR

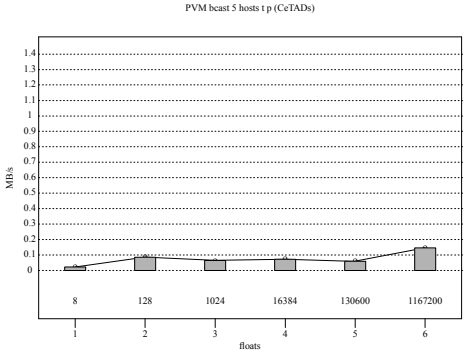

Figura 7.17: PVM *broadcast* sobre red heterogénea del CeTAD 10Mb/s - 5 receptores - directo - In Place

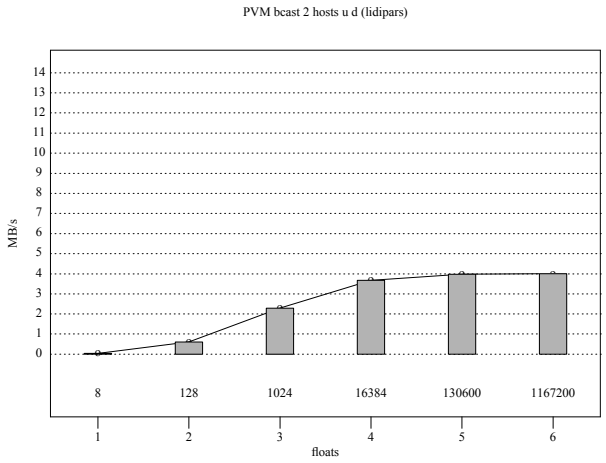

Figura 7.18: PVM *broadcast* sobre red homogénea del LIDI 100Mb/s (lidipars) - 2 receptores - *daemon-to-daemon* - XDR

Las pruebas llevadas a la red de 100Mb/s del LIDI (las lidipars) terminan por confirmar que el rendimiento del *broadcast* de PVM es bastante pobre. Los resultados son casi una extrapolación de 10Mb/s a 100Mb/s. Las diferencias que se encuentran aparte del orden de magnitud de los valores son que los picos en la curva no aparecen hasta los 3 receptores (con 2 receptores y XDR en la figura 7.18 no se ven). Además el uso de codificación deja de ser significante, y para pocos datos el ruteo cobra un poco más de importancia resultado mostrado en los gráficos 7.19, 7.20 y 7.21.

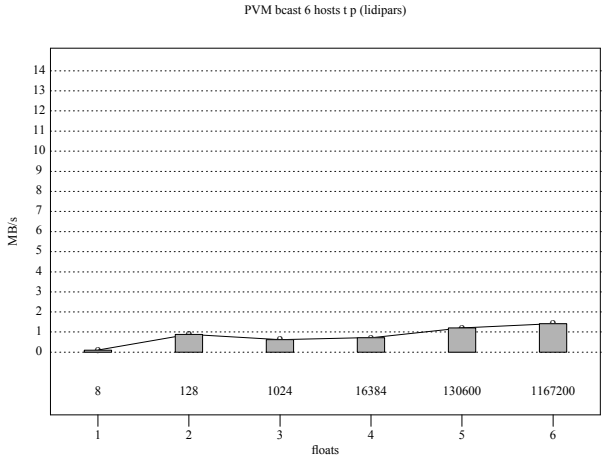

Figura 7.19: PVM *broadcast* sobre red homogénea del LIDI 100Mb/s (lidipars) - 6 receptores - directo - In Place

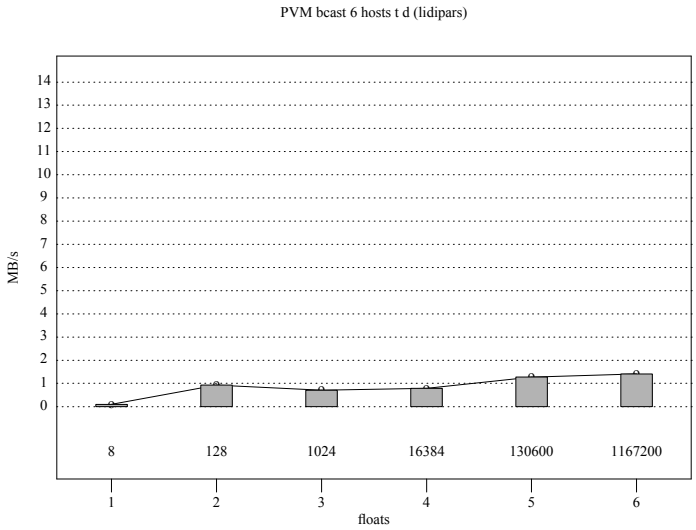

Figura 7.20: PVM *broadcast* sobre red homogénea del LIDI 100Mb/s (lidipars) - 6 receptores - directo - XDR

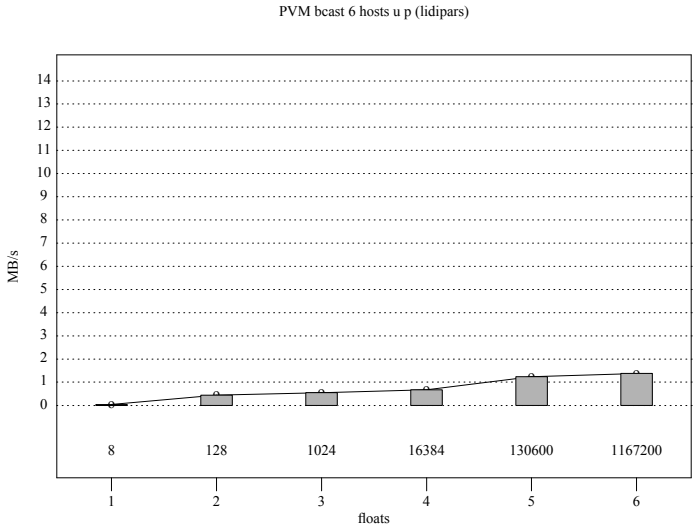

Figura 7.21: PVM *broadcast* sobre red homogénea del LIDI 100Mb/s (lidipars) - 6 receptores - *daemon-to-daemon* - In Place

Broadcast en MPI

Los resultados obtenidos con las pruebas con el *broadcast* de MPI son mejores que los de PVM. El start-up (tabla 7.2) en la red del 10Mb/s del LIDI para ruteo *daemon-to-daemon* es alto como el de PVM, sus valores están entre 0.001 y 0.005 segundos, pero para ruteo directo está un orden de magnitud por debajo mejorando significativamente los tiempos, sus valores van ente 0.0001 a 0.0005 segundos. La codificación para el start-up es insignificante igual que sucedía con PVM. En la red de 100Mb/s los tiempos mejoran el orden de magnitud esperado, salvo el start-up para más de dos hosts con ruteo directo que encuentra su límite en los 0.0001 segundos.

Network	Hosts	Default Routing	Direct Routing
LIDI (10Mb/s)	1..2	0.001	0.0001..0.0002
LIDI (10Mb/s)	3	0.005	0.0005
LIDIpars (100Mb/s)	1..2	0.0002	0.00004
LIDIpars (100Mb/s)	3..7	0.0004	0.00010

Tabla 7.2: Start-up del broadcast en LAM/MPI

Para la red de 10Mb/s el ancho de banda con ruteo directo, o como se lo denomina en las páginas del manual del LAM/MPI *client-to-client* (c2c), se comporta como es de esperarse, curva monótonamente creciente y asintótica (gráficos 7.23, 7.25 y 7.26), algo que no sucede con ruteo *daemon-to-daemon* (lamd) como se muestra en las figuras 7.22 y 7.24, observándose en este último un máximo entre los 1024Floats y 16KFloats. Esto no es de extrañar para el *broadcast* , pues lo mismo sucedió con las pruebas punto a punto (ver capítulo de *Pruebas Punto a Punto*).

Comparando la figura 7.25 con 7.26 se observa que el uso o no de la codificación casi no afecta el rendimiento, esto puede ser debido a que LAM/MPI detecta que todas los hosts involucrados tienen la misma representación de datos y por lo tanto no codifica.

El ancho de banda decrece de manera inversamente proporcional aumentese la cantidad de hosts, esto se debe a que el algoritmo que se usa es implementado mediante comunicaciones punto a punto hasta 3 receptores (Ver capítulo *Análisis de Grupos y Comunicaciones colectivas en MPI*), si hubiera más hosts se resolvería con un árbol pero el efecto para este caso sería el mismo ya que la red está compuesta por un hub que solo deja acceder a uno por vez al medio para transmitir (Bus lógico)[802.3].

Siguiendo la nomenclatura usada con PVM el ruteo directo se especifica con una **t** y el *daemon-to-daemon* o default con una **u** La codificación se nota con **d** (default) y sin codificación con **w** (raw). No se usó la opción de In Place disponible a partir de la versión de MPI 2.0. Es necesario aclarar que para MPI el proceso que envía es también receptor del mensaje, en las leyendas de los gráficos este se omite por lo que todas las recepciones son externas al host de origen.

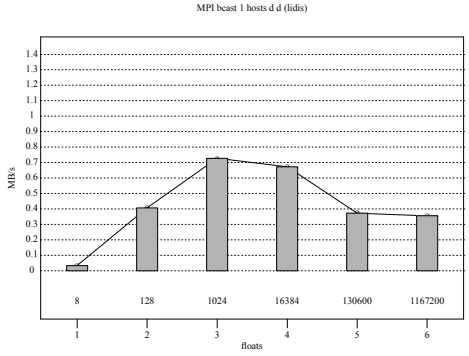

Figura 7.22: MPI *broadcast* sobre red heterogénea del LIDI 10Mb/s - 1 receptor - *daemon-to-daemon* - Codificación

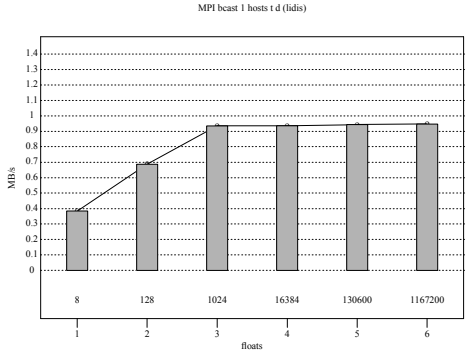

Figura 7.23: MPI *broadcast* sobre red heterogénea del LIDI 10Mb/s - 1 receptor - directo - Codificación

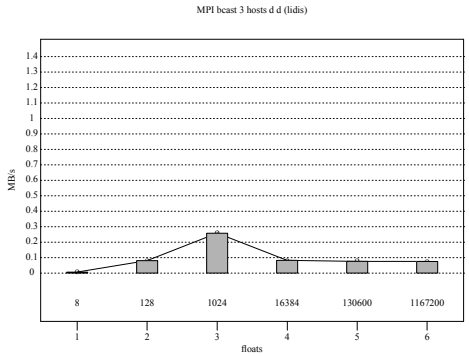

Figura 7.24: MPI bcast sobre red heterogénea del LIDI 10Mb/s - 3 receptor - *daemon-to-daemon* - Codificación

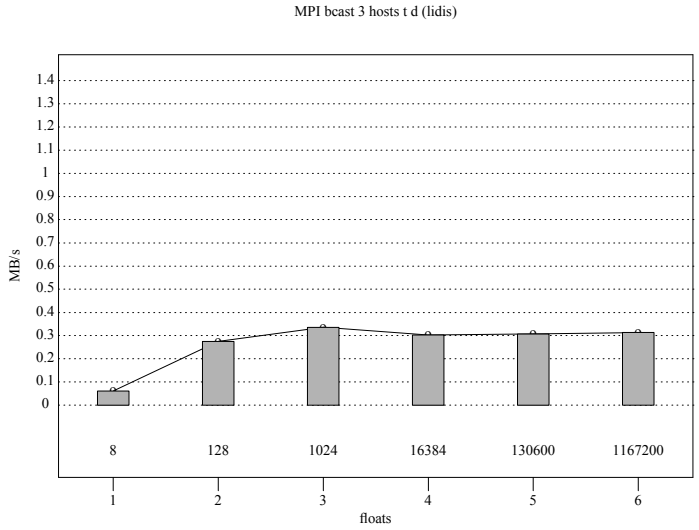

Figura 7.25: MPI *broadcast* sobre red heterogénea del LIDI 10Mb/s - 3 receptores - directo - Codificación

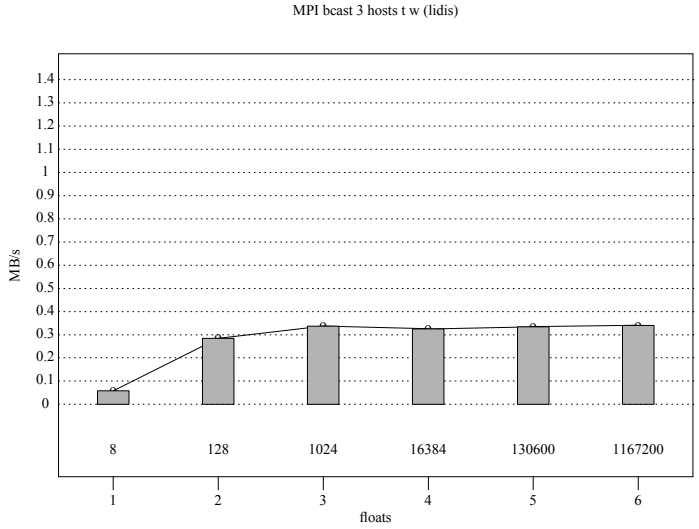

Figura 7.26: MPI *broadcast* sobre red heterogénea del LIDI 10Mb/s - 3 receptores - directo - Raw

En la red de 100Mb/s parece suceder a la inversa que con la de 10Mb/s, el ancho de banda con *client-to-client* (c2c), se comporta de forma no esperada con máximos entre los 1024 floats y los 16KFloats (figs. 7.28, 7.32 y 7.34) y con ruteo *daemon-to-daemon* (lamd) crece hasta los mismos máximos y luego se mantiene constante (figs. 7.27 7.29, 7.31). y 7.33). Con lamd el ancho de banda alcanzado es algo superior y los resultados obtenidos para dos tareas (figs. 7.27 y 7.28) son similares a los alcanzados con las pruebas punto a punto. Con la codificación sucede lo mismo que en la red de 10Mb/s.
Los valores decrecen de manera inversamente proporcional aumentase la cantidad de hosts hasta 3, pues se resuelve linealmente. Con más hosts se resuelven con un árbol (Ver capítulo *Análisis de Grupos y Comunicaciones colectivas en MPI*) que a diferencia de la red de 10Mb/s explota la capacidad de switching [SWITCH] de la red. Esto hace que se mantenga igual mientras no se aumente la cantidad de niveles del *spanning tree* y este factor para 5, 6 y 7 hosts se mantiene a un valor de 3 dejandose solo percibir el cambio con 4 hosts [2]. Si se comparan los gráficos 7.34 y 7.32 o 7.33 con 7.31 se comprueba esta caracterísitca.

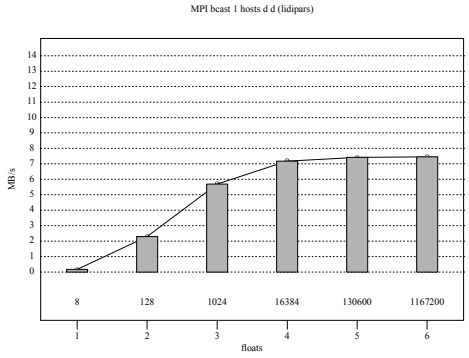

Figura 7.27: MPI *broadcast* sobre red homogénea de 100Mb/s - 1 receptor - *daemon-to-daemon* - Codificación

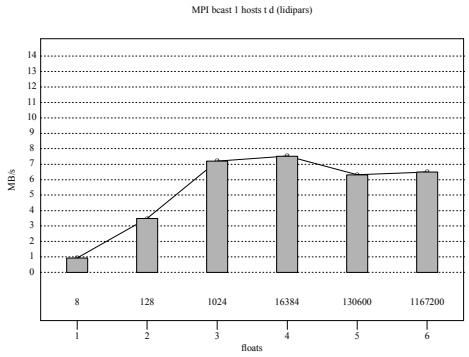

Figura 7.28: MPI *broadcast* sobre red homogénea de 100Mb/s - 1 receptor - directo - Codificación

[2]la cantidad de niveles para 4 hosts es de 2

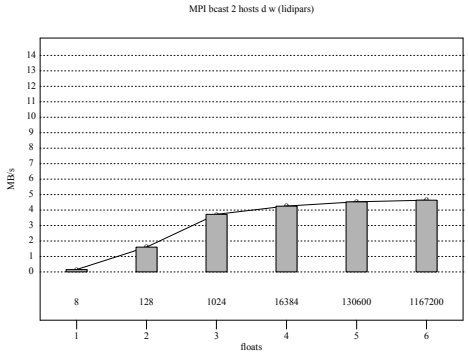

Figura 7.29: MPI *broadcast* sobre red homogénea de 100Mb/s - 2 receptores - *daemon-to-daemon* - Raw

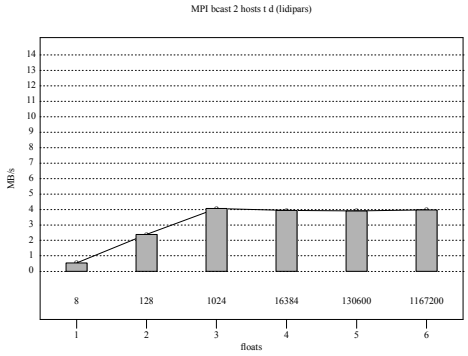

Figura 7.30: MPI *broadcast* sobre red homogénea de 100Mb/s - 2 receptores - directo - Codificación

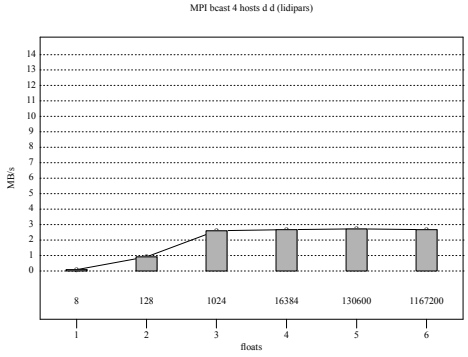

Figura 7.31: MPI *broadcast* sobre red homogénea de 100Mb/s - 4 receptores - *daemon-to-daemon* - Codificación

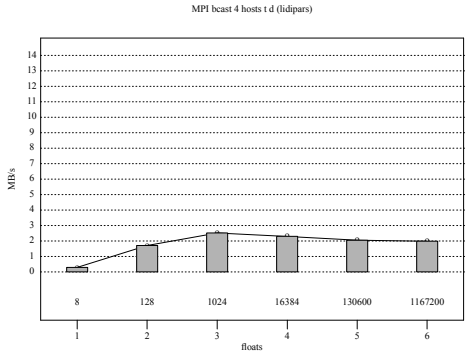

Figura 7.32: MPI bcast sobre red homogénea de 100Mb/s - 4 receptores - direct - Codificación

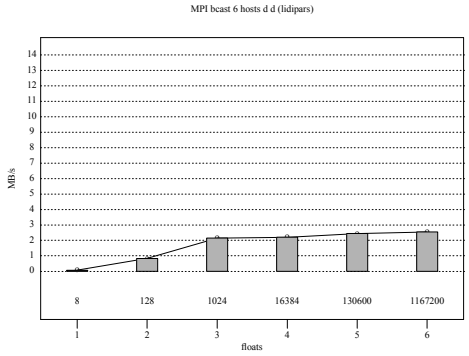

Figura 7.33: MPI *broadcast* sobre red homogénea de 100Mb/s - 6 receptores - *daemon-to-daemon* - Codificación

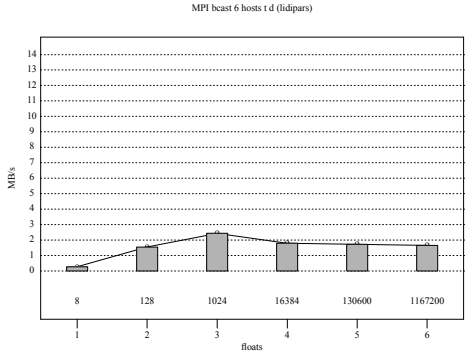

Figura 7.34: MPI *broadcast* sobre red homogénea de 100Mb/s - 6 receptores - directo - Codificación

7.2.3 Multicast

El *multicast* es un *broadcast* selectivo, donde los receptores son un subconjunto del total de receptores potenciales. El benchmark es solo realizado sobre PVM púes MPI no lo define.

Multicast en PVM

El start-up (tabla 7.3) en la red de 10Mb/s del LIDI es bastante alto como ocurría con el *broadcast*, aunque a diferencia de éste el impacto de la adhesión de nuevos hosts a la operación se hace notar. Sus valores están entre 0.0015 y 0.0900 segundos (para ambos ruteos).

Receptores	Default y Direct Routing
1	0.0018
2	0.0030
3	0.0900

Tabla 7.3: Resultados del start-up del PVM *multicast* sobre red de 10Mb/s del LIDI

En la red del CeTAD (tabla 7.4) el incremento de los tiempos también marca el incremento de hosts en las pruebas. A medida que se agregan equipos, los tiempo suben. Del mismo modo pasa en el cluster homogéneo (fig. 7.5).

Receptores	Default y Direct Routing
1	0.003
2..3	0.025
4..5	0.045

Tabla 7.4: Resultados del start-up del PVM *multicast* sobre red de 10Mb/s del CeTAD

En la red de 100Mb/s los tiempos son crecientes desde 1 hasta 7 hosts destinos con valores y aceleración proporcional a la velocidad de la red.

Receptores	Default y Direct Routing
1	0.0004
2	0.0040
3	0.0060
4	0.0080
5	0.0100
6	0.0140
7	0.0170

Tabla 7.5: Resultados del start-up del PVM *multicast* sobre cluster de 100Mb/s

Pare el ancho de banda en las redes de 10MB/s (figs. 7.35, 7.36, 7.37 y 7.38) se comporta de la forma esperada. Es creciente hasta encontrar un máximo a

los 16KFloats y luego, a partir de ese punto se mantiene casi constante, aunque se puede apreciar en ocasiones que decae un poco. Para un host receptor (fig. 7.35 y 7.36) el rendimiento alcanzado es el de una punto a punto y a medida que se agregan nuevos receptores su rendimiento cae proporcionalmente. El ruteo y la codificación son poco significantes, solo se advierten cambios para un solo receptor.

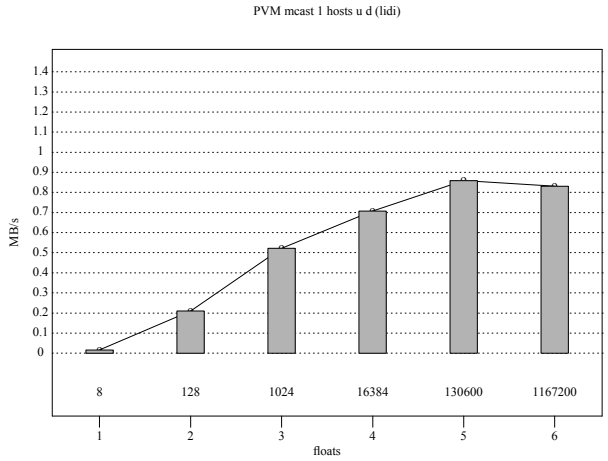

Figura 7.35: PVM *multicast* sobre red heterogénea del LIDI 10Mb/s - 1 receptor - *daemon-to-daemon* - XDR

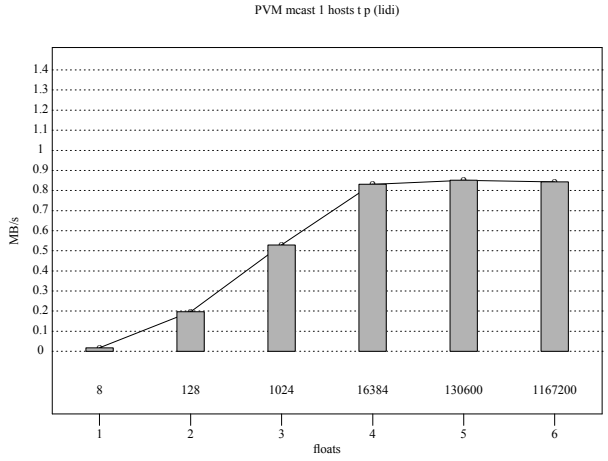

Figura 7.36: PVM mcast sobre red heterogénea del LIDI 10Mb/s - 1 receptor - direct - In Place

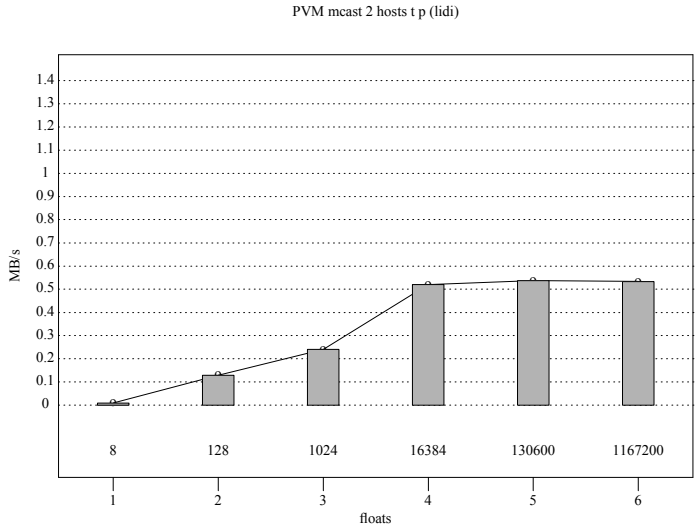

Figura 7.37: PVM *multicast* sobre red heterogénea del LIDI 10Mb/s - 2 receptores - directo - In Place

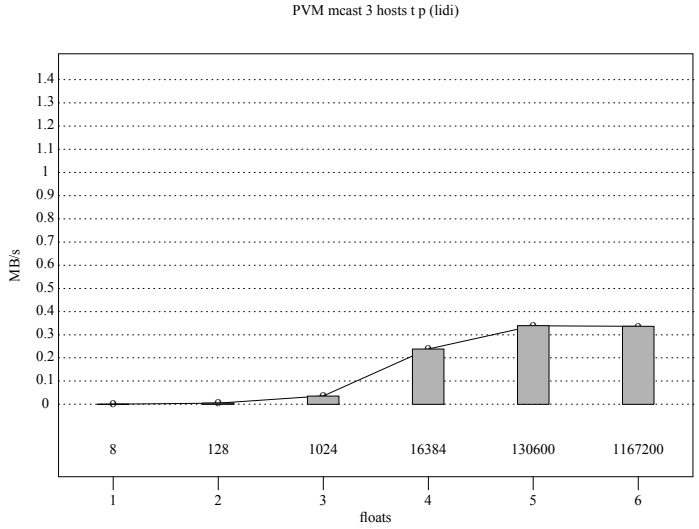

Figura 7.38: PVM *multicast* sobre red heterogénea del LIDI 10Mb/s - 3 receptores - directo - In Place

En el cluster de 100Mb/s los resultados son similares (figs. 7.39, 7.40, 7.41 y 7.42) aunque con algunas diferencias. Por ejemplo el ancho de banda alcanzado para un receptor (fig. 7.40) es un poco menor que el logrado con las pruebas punto a punto. Además para estas mismas pruebas, con un receptor, usando XDR y UDP luego de los 16KFloats se observa que el rendimiento decrece (fig. 7.39).

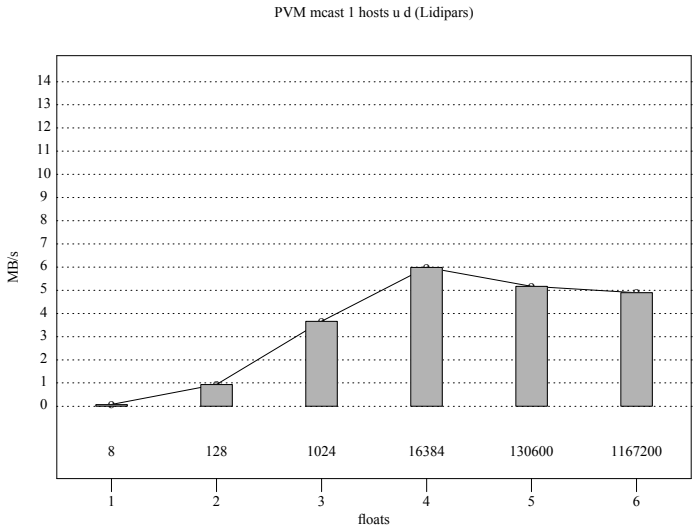

Figura 7.39: PVM *multicast* sobre cluster de 100Mb/s - 1 receptor - *daemon-to-daemon* - XDR

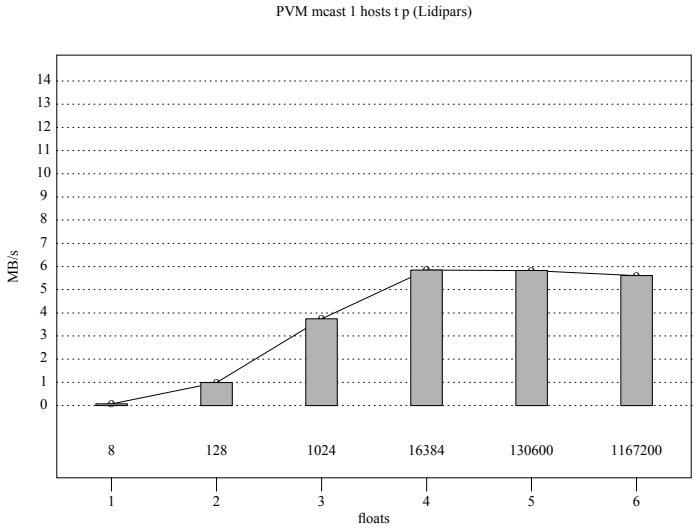

Figura 7.40: PVM *multicast* sobre cluster de 100Mb/s - 1 receptor - directo - In Place

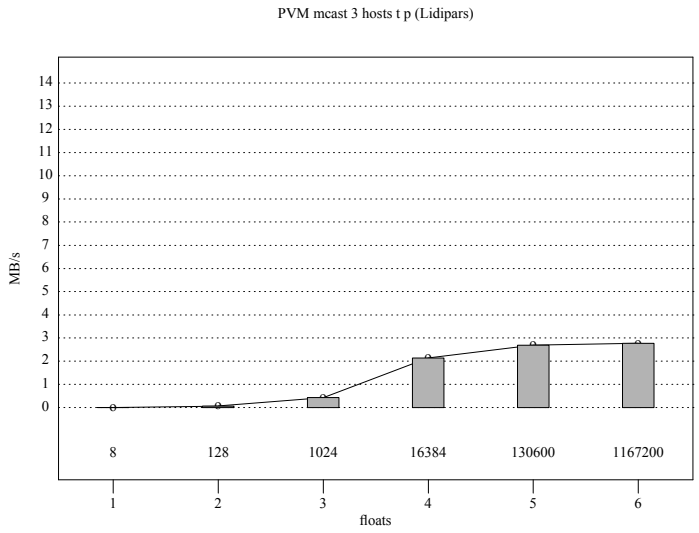

Figura 7.41: PVM *multicast* sobre cluster de 100Mb/s - 3 receptores - directo - In Place

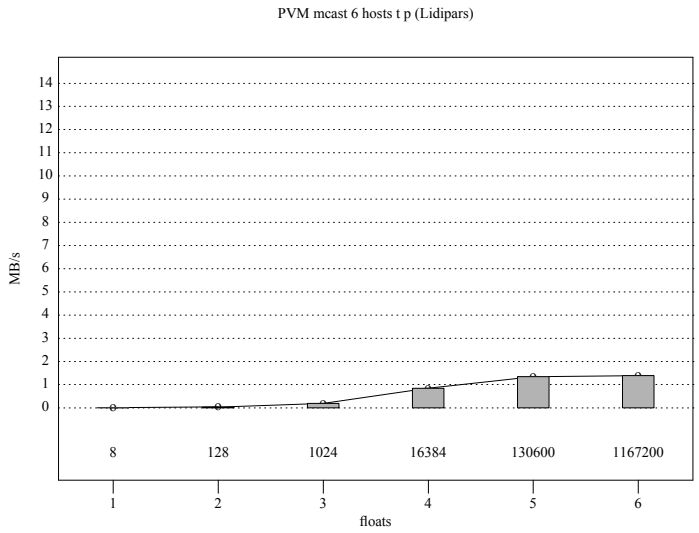

Figura 7.42: PVM *multicast* sobre cluster de 100Mb/s - 6 receptores - directo - In Place

7.2.4 Comparaciones

Para terminar el capítulo se presentan los gráficos comparativos entre las diferentes implemenataciones de *broadcast*:

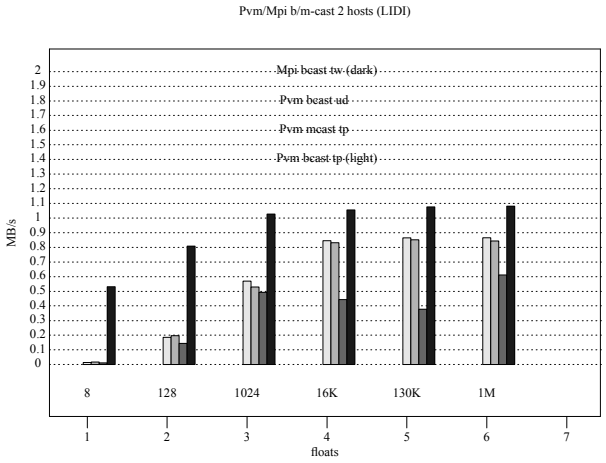

Figura 7.43: Comparación de *broadcast* en red de 10Mb/s - 1 receptor externo

En las figuras 7.43 y 7.44 se compara para la red de 10Mb/s la mejor configuración para *broadcast* en LAM/MPI (t,w) -t=direct, u=daemon; w=raw, p=in place,d=default- la mejor para PVM (t,p), la peor para PVM (u,d) y el mejor *multicast* de PVM (t,p). Las conclusiones que se pueden obtener es que LAM/MPI posee un rendimieno superior a PVM, que las implementaciones estan basadas en puntos a puntos y que el ancho de banda final alcanzado tiende a ser simlar a medida que se mandan más datos.

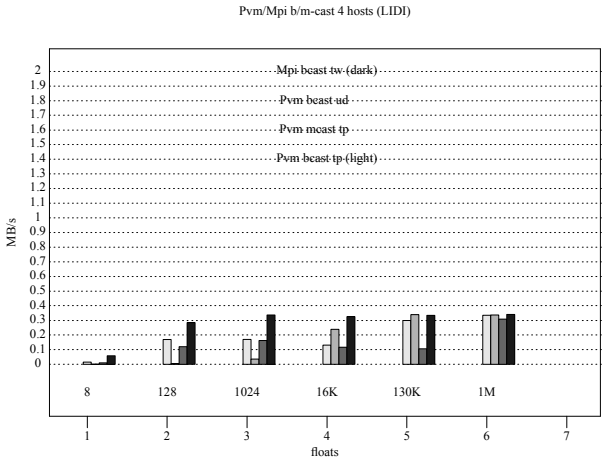

Figura 7.44: Comparación de *broadcast* en red de 10Mb/s - 3 receptores externos

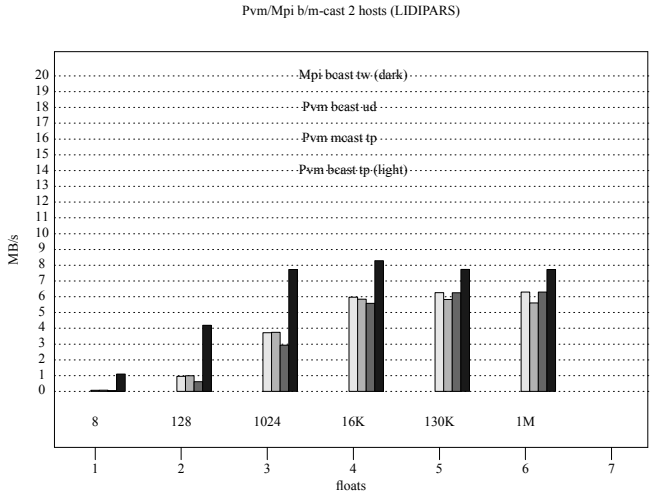

Figura 7.45: Comparación de *broadcast* en red de 100Mb/s - 1 receptor externo

En las figuras 7.45 y 7.46 se compara las mismas configuraciones mostradas anteriomente pero sobre el cluster homogéneo de 100Mb/s. Las conclusiones que se obtienen son similares a las anteriores. Es necesario aclarar que sobre la red de 100Mb/s la mejor implementación para LAM/MPI no se muestra que es la obtenida usando ruteo *daemon-to-daemon*. Esta configuración posee un start-up mucho más alto, pero para mucho datos logra un rendimiento de 1MB/s sobre el alcanzado por el ruteo directo.

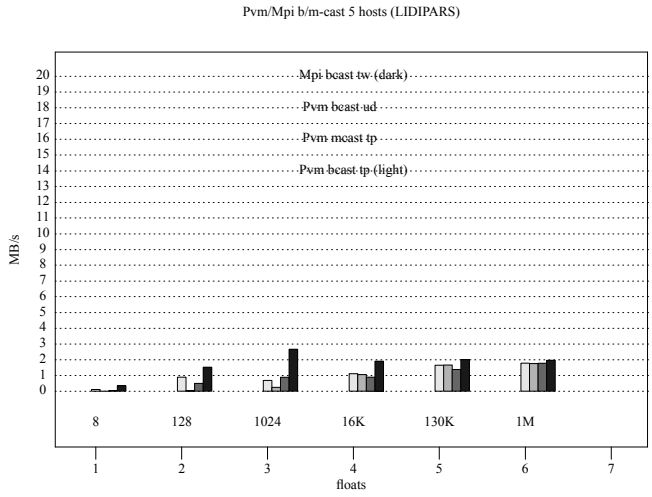

Figura 7.46: Comparación de *broadcast* en red de 100Mb/s - 4 receptores externos

Bibliografía

[XDR] Sun Microsystems, Inc. XDR: External Data Representation Standard. RFC 1014, Sun Microsystems, Inc. 1987.

[802.3] Institute of Electrical and Electronics Engineers, Local Area Network - *CSMA/CD Access Method and Physical Layer Specifications* ANSI/IEEE 802.3 - 1985, IEEE Computer Society.

[MPI] MPI: The Complete Reference - Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra - The MIT Press, Cambridge, Massachusetts - 1996.

[SWITCH] Anixter Technology White Paper Ethernet Switiching - Anixter Inc., 4711 Golf Road Skokie, IL 60076 - http://www.anixter.com.

Capítulo 8

Resultados de las Pruebas II

8.1 Introducción

Los resultados que se muestran en este capítulo son los obtenidos con las pruebas del *gather*, *scatter* y *reduce*. Se anticipa que los valores resultantes muestran que sobre un mismo *middleware* las variaciones entre las diferentes operaciones son pocas y tienden a tener un start-up y ancho de banda similares.

Para todas las pruebas se analiza primero el start-up luego el ancho de banda. Las gráficas son de ancho de banda y tiempo. Para muchos gráficos se ve que el valor alcanzado es mayor que el físico teórico (1.25MB/s o 12.5MB/s). Esto se debe a que al calcularse se toma en cuenta la transferencia de datos al mismo proceso que envía, operación que es mucho menos costosa debido a que no sea hacen los accesos a la red.

8.2 Resultados Obtenidos

8.2.1 Scatter

Scatter en PVM

El start-up en la red de 10Mb/s del LIDI (primera parte de la tabla 8.1) es alto y sus valores van creciendo a medida que se agregan hosts a la operación. Lo mismo sucede sobre el cluster conectado a 100Mb/s (segunda parte de la tabla 8.1). Los resultados muestran que para la red de 100Mb/s el start-up usando ruteo directo crece de forma mucho más acelerada que el ruteo normal, característica que lo convierte en poco escalable y por lo tanto no apto para redes de estaciones de trabajo cuando se desea transmitir relativamente poca información.

Network	Hosts (R)	Default Routing	Direct Routing
LIDI (10Mb/s)	1	0.020	0.015
LIDI	2	0.030	0.021
LIDI	3	0.040	0.043
LIDIpars (100Mb/s)	1	0.0012	0.0018
LIDIpars	2	0.0014	0.0024
LIDIpars	3	0.0016	0.0042
LIDIpars	4	0.0019	0.0064
LIDIpars	5	0.0024	0.0089
LIDIpars	6	0.0026	0.0100
LIDIpars	7	0.0030	0.0130

Tabla 8.1: Start-up del *scatter* con PVM

Con respecto al ancho de banda, el máximo valor logrado con las dos formas de ruteo es el mismo, la diferencia se encuentra en la forma hasta alcanzarlo. Para el ruteo directo sobre la red de 10Mb/s la aceleración lograda es mejor, esto se deduce de comparar los gráficos 8.2 y 8.3. El rendimiento se va degradando conforme se agreguen nuevos hosts, esto se debe al aumento del costo del start-up (En la fig. 8.1 el máximo esta en 1.4MB/s y en 8.2 es de 1.14MB/s). Un hecho que permanece escondido en estos gráficos es que PVM resuelve el *scatter*, el *gather* y el *reduce* de forma lineal haciendo que los tiempos crezcan con la misma forma según se adicionan hosts. En las gráficas de tiempo 8.4, 8.6 y 8.5 se muestra este hecho.

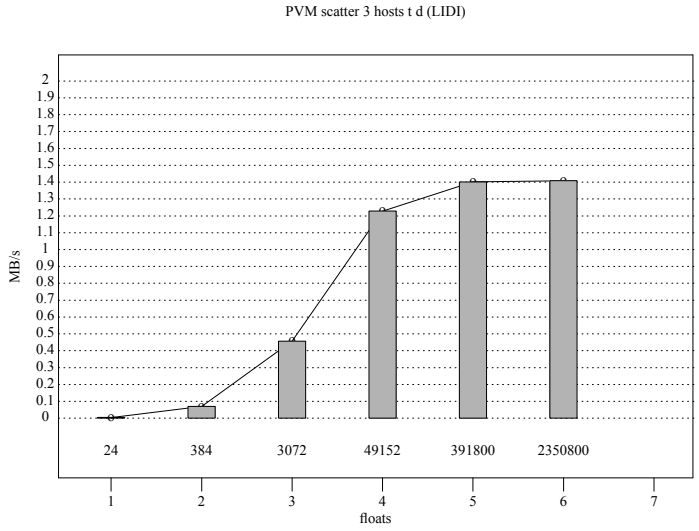

Figura 8.1: PVM *scatter* sobre red heterogénea del LIDI 10Mb/s - 2 receptores externos - directo - XDR

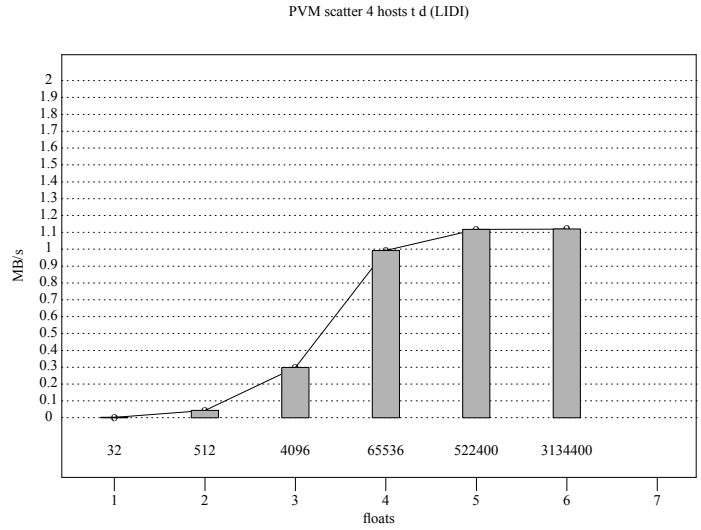

Figura 8.2: PVM *scatter* sobre red heterogénea del LIDI 10Mb/s - 3 receptores externos - directo - XDR

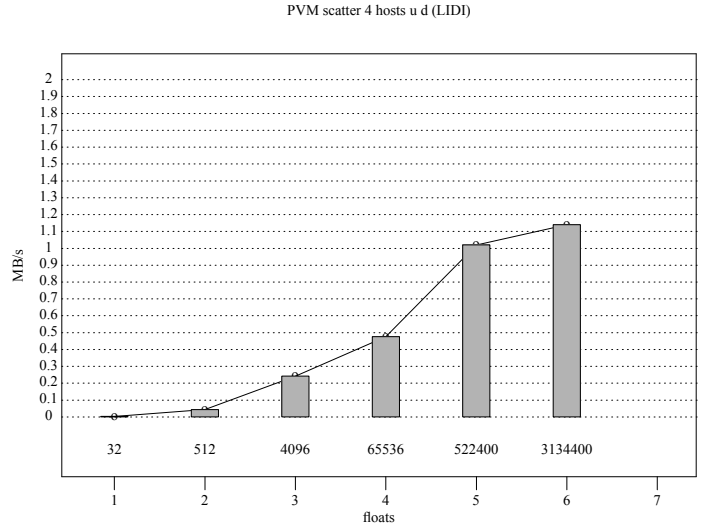

Figura 8.3: PVM *scatter* sobre red heterogénea del LIDI 10Mb/s - 3 receptores externos - *daemon-to-daemon* - XDR

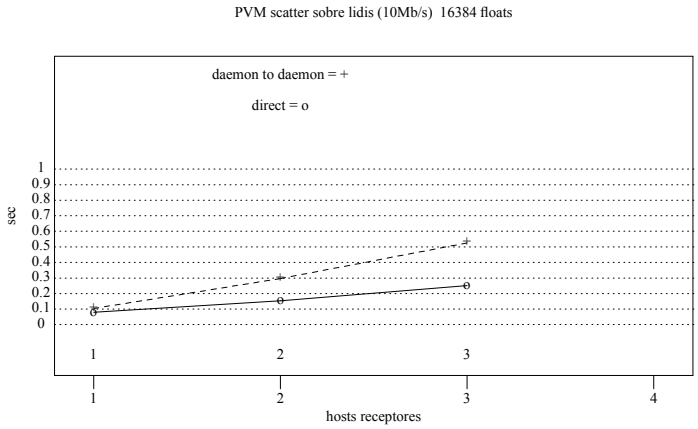

Figura 8.4: Tiempo del PVM *scatter* sobre red heterogénea del LIDI 10Mb/s - porción de 16KFloats para receptores externos

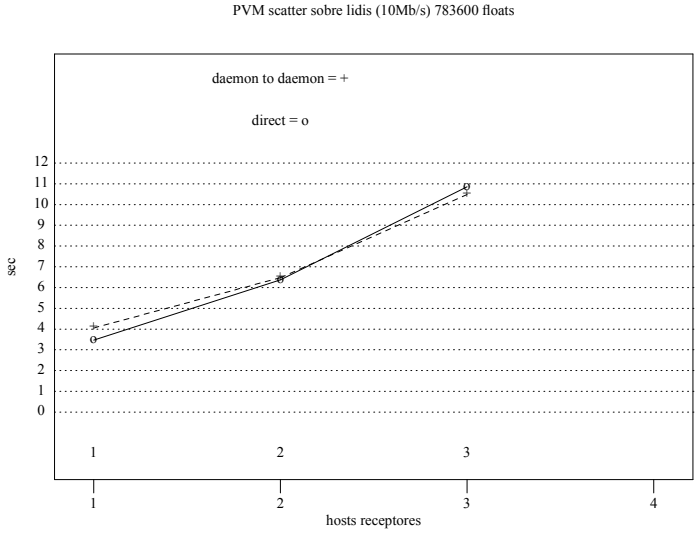

Figura 8.5: Tiempo del PVM *scatter* sobre red heterogénea del LIDI 10Mb/s - porción de 700KFloats para receptores externos

Las pruebas llevadas a la red de 100Mb/s (las lidipars) muestran que el máximo ancho de banda logrado usando ruteo default se mantiene estable cercano a los 9 MB/s sin importar la cantidad de hosts que participen en la operación (figs. 8.8 y 8.10), sin embargo se ve afectado al agregar hosts en la aceleración con la cual alcanza su máximo.
Para ruteo directo (figs. 8.7 y 8.9) los resultados son algo inexplicables, alcanzan su máximo para porciones de 16KFloats (48KFloats en total) y luego decrecen levemente, en cambio para ruteo default son monótonamente crecientes. Se observa que utilizando ruteo directo el máximo valor alcanzado se reduce al agregar hosts, por ejemplo en 8.7 es un poco más que 8.5MB/s y en 8.9 es de 7.5MB/s
En 8.6 parece suceder que el ruteo directo mantiene el tiempo más estable, pero esto no es así pues luego para porciones de 130KFloats y 700KFloats es nuevamente superado dando para 8 hosts los valores de la tabla 8.2.

Tamaño de porción	Conf. t d	Conf. u d
130600 Floats	0.570421 sec.	0.459743 sec.
783600 Floats	3.413393 sec.	2.732121 sec.

Tabla 8.2: Tiempos para PVM *scatter* con 8 hosts

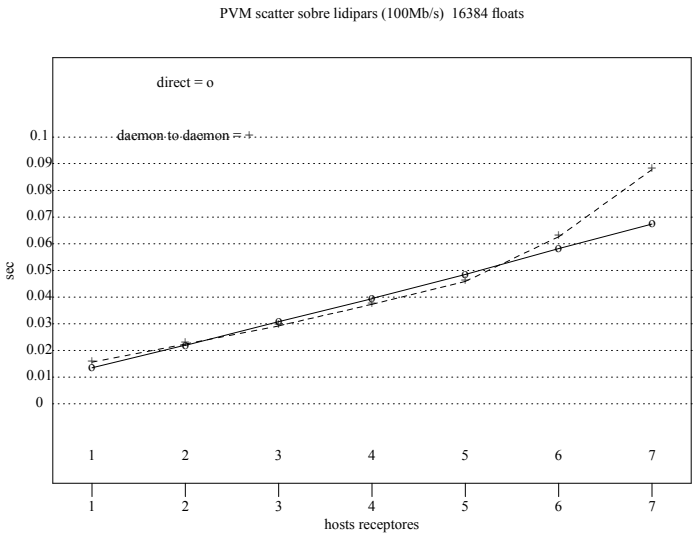

Figura 8.6: Tiempo del PVM *scatter* sobre cluster de 100Mb/s - porción de 16KFloats para receptores externos

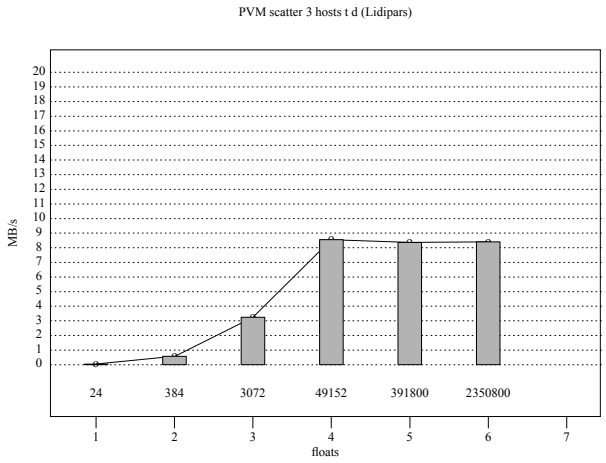

Figura 8.7: PVM *scatter* sobre cluster de 100Mb/s - 2 receptores externos - directo - XDR

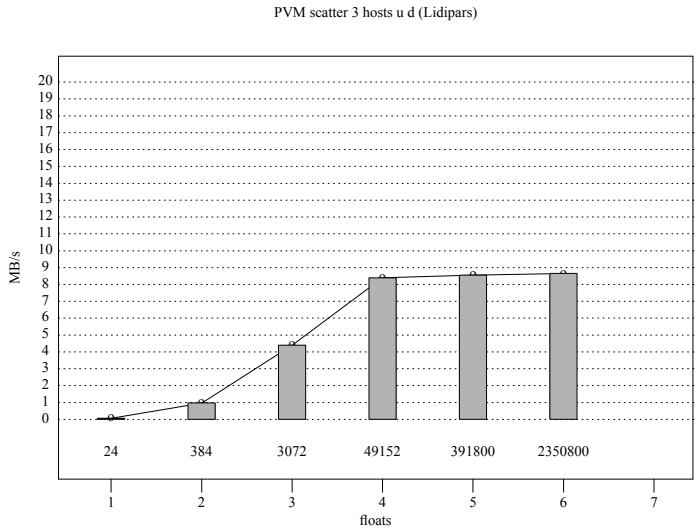

Figura 8.8: PVM *scatter* sobre cluster de 100Mb/s - 2 receptores externos - *daemon-to-daemon* - XDR

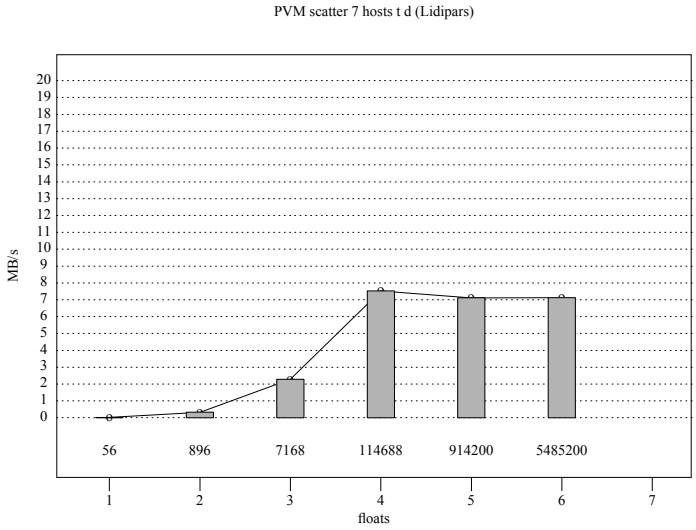

Figura 8.9: PVM *scatter* sobre cluster de 100Mb/s - 6 receptores - directo - XDR

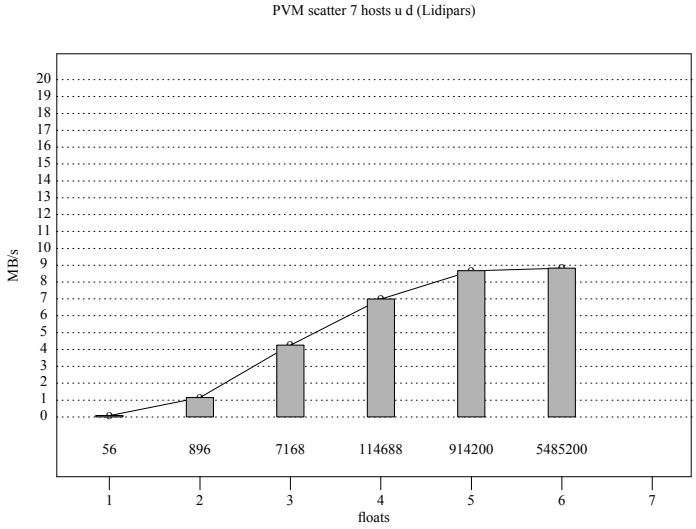

Figura 8.10: PVM *scatter* sobre cluster de 100Mb/s - 6 receptores externos - *daemon-to-daemon* - XDR

Scatter en MPI

El start-up en la red de 10Mb/s del LIDI (parte superior de la tabla 8.3) es 100 o más veces mejor que el de PVM (tabla 8.1) y para el ruteo directo tiene valores de un orden de magnitud más chicos que los obtenidos con ruteo normal, siendo entonces 1000 veces mejor que los de PVM. Para el cluster de 100Mb/s los resultados para el ruteo directo son realmente muy buenos. Para el ruteo normal son un orden de magnitud mejor que los de PVM. Sus valores van creciendo a medida que se agregan hosts a la operación, pero para el ruteo normal parecen estancarse a partir de 5 receptores (Ver algoritmo en el capítulo *Análisis de Grupos y Comunicaciones colectivas en MPI*).

Network	Hosts	Lamd Routing	c2c Routing
LIDI (10Mb/s)	1	0.0010	0.0001
LIDI	2	0.0010	0.0003
LIDI	3	0.0060	0.0005
LIDIpars (100Mb/s)	1	0.00016	0.00003
LIDIpars	2	0.00020	0.00005
LIDIpars	3	0.00035	0.00007
LIDIpars	4	0.00057	0.00009
LIDIpars	5	0.00080	0.00010
LIDIpars	6	0.00100	0.00011
LIDIpars	7	0.00100	0.00020

Tabla 8.3: Start-up del *scatter* con MPI

Para describir el ancho de banda en LAM/MPI se analiza primero los resultados sobre la red de 10Mb/s.
Como se vio en las pruebas punto a punto y en las colectivas anteriores, el rendimiento para el ruteo normal o *daemon-to-daemon* es bueno para pocos datos (hasta 1024-16KFloats), lo mismo sucede con el *scatter*, si se observa la figura 8.13 se ve que tiene un desempeño bueno al iniciar pero luego cae abruptamente. Para ruteo directo crece de manera monótona y según se agreguen hosts decrece un poco como se confirma a partir de las figuras 8.11 y 8.12. La aceleración también se va degradando a medida se agreguen hosts. Los valores obtenidos son altos, y mejores si se los compara con PVM. En la figura 8.14 se comparan los tiempos de las dos formas de ruteo de MPI y se destaca que el ruteo default o *daemon-to-daemon* a medida que se agregan hosts sobre la red de 10Mb/s incrementa los tiempos de forma más que lineal.

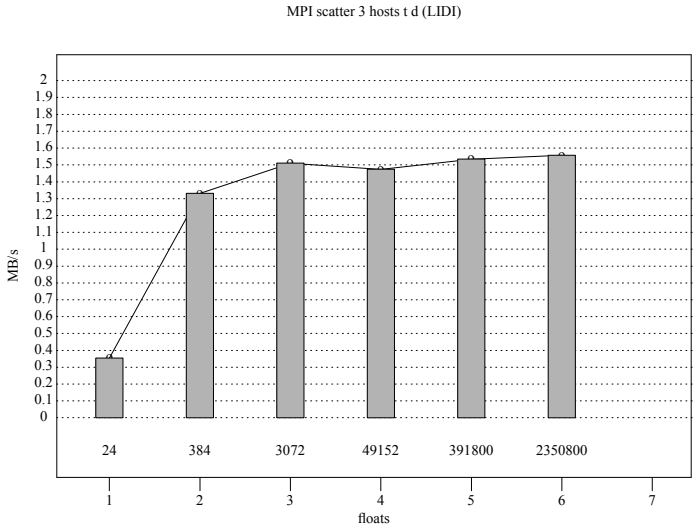

Figura 8.11: MPI *scatter* sobre red heterogénea del LIDI 10Mb/s - 2 receptores externos - directo - Default

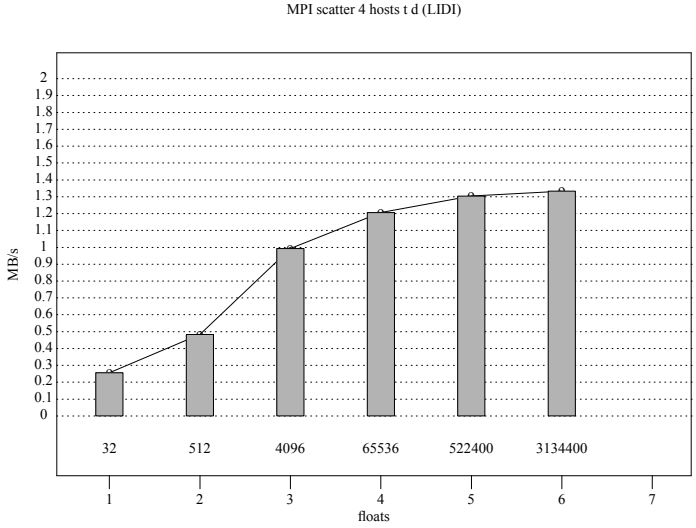

Figura 8.12: MPI *scatter* sobre red heterogénea del LIDI 10Mb/s - 3 receptores externos - directo - Default

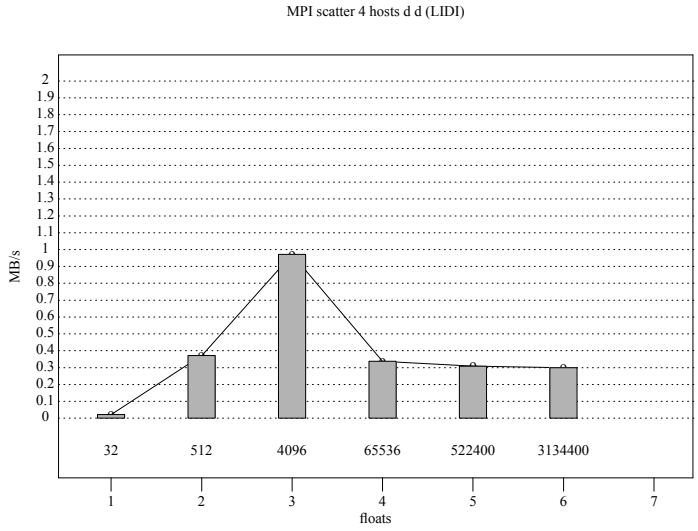

Figura 8.13: MPI *scatter* sobre red heterogénea del LIDI 10Mb/s - 3 receptores - *daemon-to-daemon* - Default

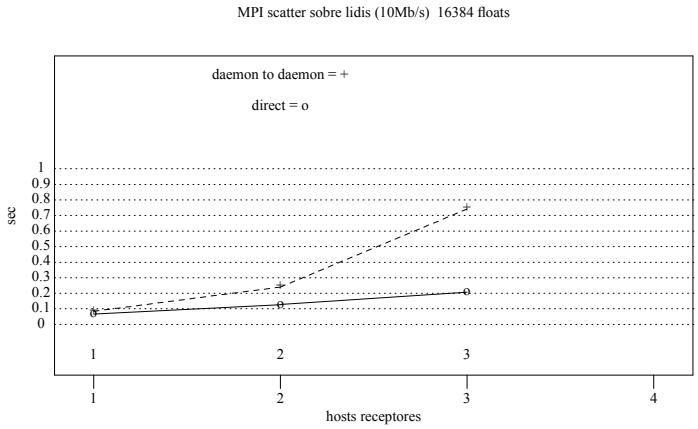

Figura 8.14: Tiempo del MPI *scatter* sobre red heterogénea del LIDI 10Mb/s - porción de 16KFloats para receptores externos

El máximo ancho de banda sobre el cluster de 100Mb/s (fig. 8.15) para la misma cantidad de hosts esta entre 1 y 4 MB/s por encima del alcanzado con PVM. Otro resultado a favor de MPI es que la aceleración hasta alcanzar el pico es más rápida, dentro de MPI la aceleración del ruteo directo es mejor hecho que puede verse comparando las figuras 8.16 y 8.18 y de aquí también se observa que, a diferencia de lo que sucedía a 10Mb/s, el ruteo default supera al directo en el máximo alcanzado. El no uso de codificación toma importancia para muchos datos haciendo que casi se mantenga el máximo alcanzado (fig. 8.17) (Si se usa codificación, a partir de porciones de 1024 Floats -7168 Floats en total para 7 hosts- el ancho de banda empeora). Una característica obtenida sobre la red de 10Mb/s que se conserva es que a medida que se agregan hosts a la operación el ancho de banda se va degradando, esto se debe a que el start-up para más hosts es más costoso en términos de tiempo.
Como se vio en el capítulo *Análisis de Grupos y Comunicaciones colectivas en MPI*, el algoritmo usado para más de 4 procesos es en base a un árbol por lo que los tiempos deberían ser logarítmicos debido a que se esta en presencia de una red *switcheada*, cuestión que no se aprecia en el gráfico 8.20 y parece aparecer en 8.19. El mejor algoritmo con respecto a PVM puede verse en los valores del ancho de banda, los cuales son realmente buenos.

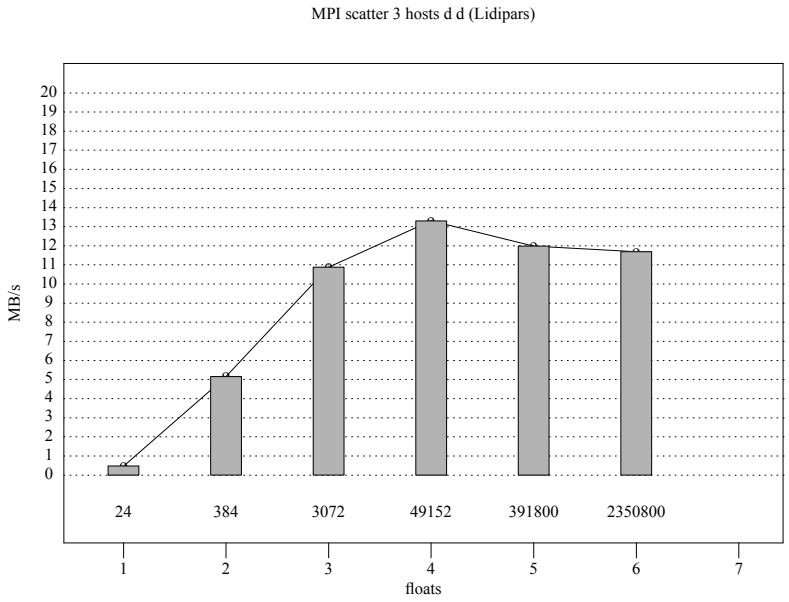

Figura 8.15: MPI *scatter* sobre cluster de 100Mb/s - 2 receptores externos - *daemon-to-daemon* - Default

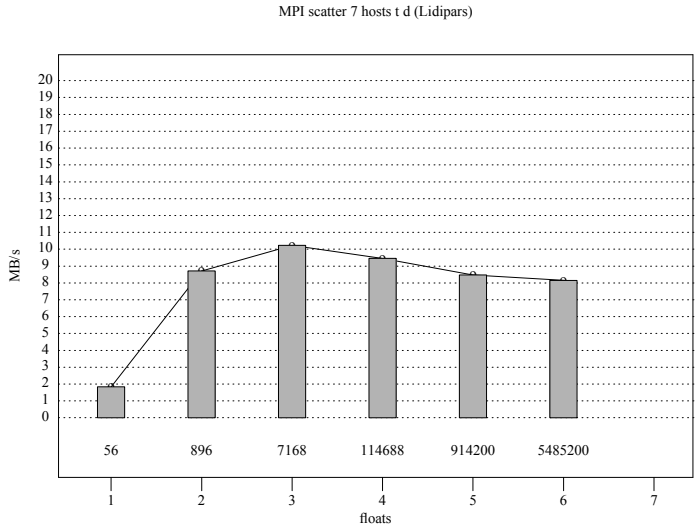

Figura 8.16: MPI *scatter* sobre cluster de 100Mb/s - 6 receptores externos - directo - Default

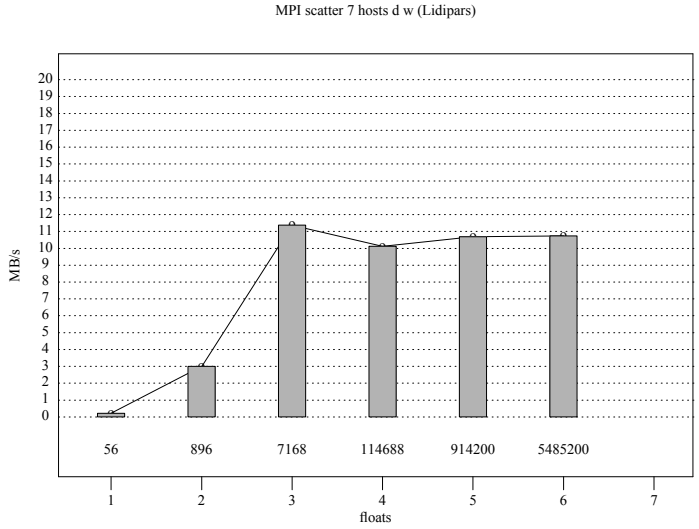

Figura 8.17: MPI *scatter* sobre cluster de 100Mb/s - 6 receptores externos - *daemon-to-daemon* - Raw

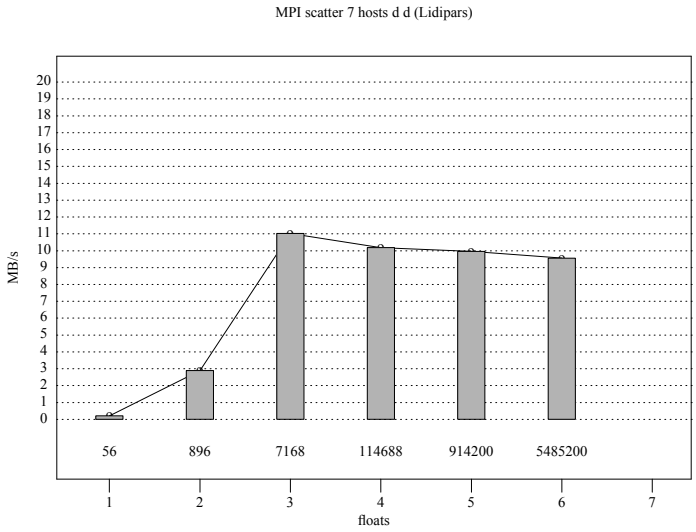

Figura 8.18: MPI *scatter* sobre red heterogénea del LIDIPARS 10Mb/s - 6 receptores - *daemon-to-daemon* - Default

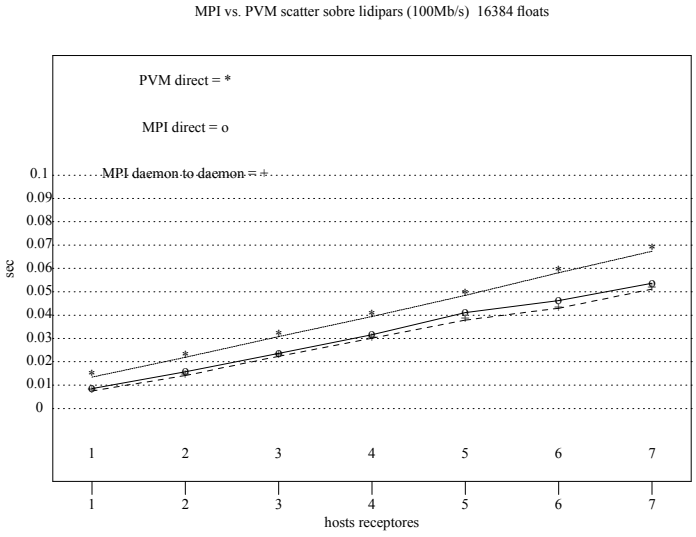

Figura 8.19: Tiempo del MPI vs. PVM *scatter* sobre cluster de 100Mb/s - porción de 16KFloats

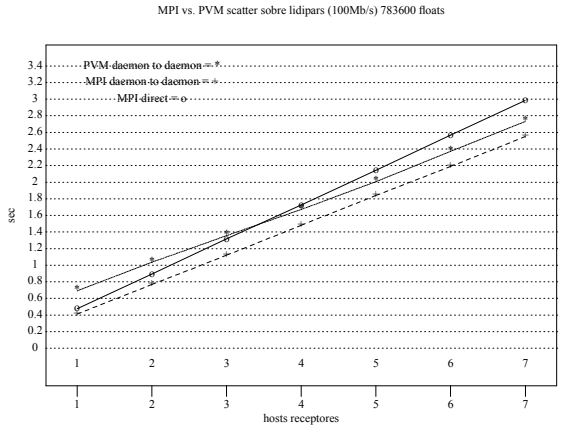

Figura 8.20: Tiempo del MPI vs. PVM *scatter* sobre cluster de 100Mb/s - porción de 700KFloats para receptores externos

8.2.2 Gather

Gather en PVM

El start-up en la red de 10Mb/s para el *gather* de PVM es muy parecido al del *scatter* presentado en la primera parte de la tabla 8.1, la única diferencia que se puede destacar es que: en ocasiones tiene valores una o dos milésimas de segundos más altos. Para el cluster de 100Mb/s los resultados (tabla 8.4) con respecto al ruteo default son similares pero los dados por el ruteo directo son menores y más parejos, si se compara los obtenidos con ruteo directo y con ruteo default para el *gather* se advierte que son muy parecidos.

Network	Hosts	Default Routing	Direct Routing
LIDIpars (100Mb/s)	1	0.0013	0.0013
LIDIpars	2	0.0015	0.0015
LIDIpars	3	0.0019	0.0019
LIDIpars	4	0.0024	0.0024
LIDIpars	5	0.0027	0.0030
LIDIpars	6	0.0030	0.0036
LIDIpars	7	0.0037	0.0037

Tabla 8.4: Start-up del *gather* con PVM

Analizando el ancho de banda de las pruebas a 10Mb/s y comparándolo con los valores descriptos en la sección del *scatter* se arriba a que ambas operaciones tienen un rendimiento similar como se había adelantado. Las diferencias que se encuentran si se compara los gráficos 8.1 con 8.21 son sutiles, solo se advierte que el *scatter* alcanza para porciones de 1024F (3072 Floats en total) y 16KFloats (49152 Floats en total) resultados un poco mejores. Comparando los gráficos para ruteo directo y codificación, 8.2 y 8.22, se observa que las diferencias son aún menores. Las diferencias más grandes se advierten con ruteo default donde se observa que el *gather* tiene una aceleración mejor alcanzando para porciones de 16KFloats (64KFloats en total) el doble de ancho de banda que el *scatter* (figs. 8.3 y 8.23).

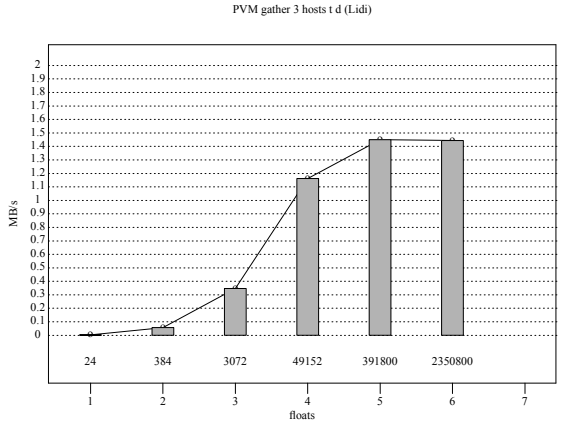

Figura 8.21: PVM *gather* sobre red heterogénea del LIDI 10Mb/s - 2 receptores externos - directo - XDR

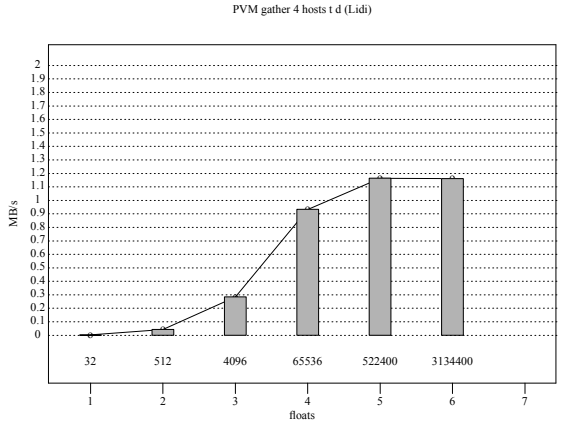

Figura 8.22: PVM *gather* sobre red heterogénea del LIDI 10Mb/s - 3 receptores externos - directo - XDR

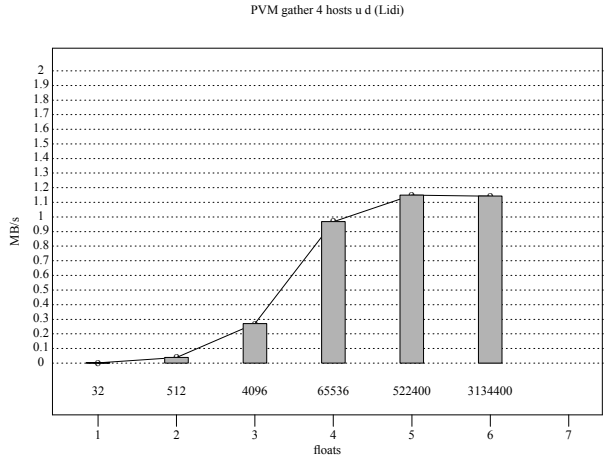

Figura 8.23: PVM *gather* sobre red heterogénea del LIDI 10Mb/s - 3 receptores externos - *daemon-to-daemon* - XDR

Para los 100Mb/s con PVM para el *scatter* y el *gather* se observa que las gráficas son parecidas, si se compara el ruteo directo con 3 receptores, 8.24 con 8.7, se tiene que el *gather* alcanza un ancho de banda un poco mejor, lo mismo sucede para ruteo default. Para 7 receptores se ve que, como sucedía con la red de 10Mb/s para 4 receptores la aceleración del *gather* es mejor (comparar figura 8.25 con 8.10) pero el máximo lo gana el *scatter*.

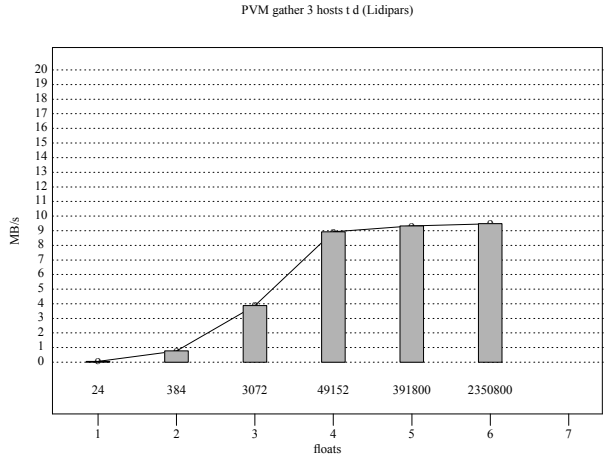

Figura 8.24: PVM *gather* sobre cluster de 100Mb/s - 2 receptores externos - directo - XDR

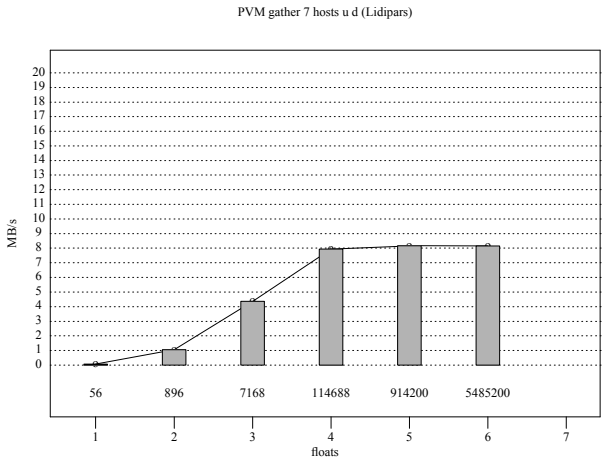

Figura 8.25: PVM *gather* sobre cluster de 100Mb/s - 6 receptores externos - *daemon-to-daemon* - XDR

Gather en MPI

El start-up para el *gather* en LAM/MPI sigue siendo los ordenes de magnitud mejor que el de PVM aunque sus valores son más altos que los del *scatter* sobre el mismo *middleware*. En la tabla 8.5 se muestran los tiempos de *gather* comparados con el *scatter* sobre LAM/MPI.

Network	Hosts	d2d Gather/Scatter	c2c Gather/Scatter
LIDI (10Mb/s)	1	0.0030/0.0010	0.0006/0.0001
LIDI	2	0.0030/0.0010	0.0007/0.0003
LIDI	3	0.0080/0.0060	0.0013/0.0005
LIDIpars (100Mb/s)	1	0.0003/0.00016	0.00012/0.00003
LIDIpars	2	0.0006/0.00020	0.00017/0.00005
LIDIpars	3	0.0008/0.00035	0.00019/0.00007
LIDIpars	4	0.0012/0.00057	0.00020/0.00009
LIDIpars	5	0.0014/0.00080	0.00027/0.00010
LIDIpars	6	0.0017/0.00100	0.00032/0.00011
LIDIpars	7	0.0019/0.00100	0.00035/0.00020

Tabla 8.5: Start-up del *scatter/gather* con MPI

Analizando el ancho de banda del *gather* sobre LAM/MPI se ve que el máximo valor alcanzado es igual al del *scatter* aunque la aceleración con la cual se alcanza es menor, para el *scatter* con 3 hosts ruteo directo y codificación se alcanzaba la cima para porciones de 1024F (49152F en total), en cambio para el *gather* se alcanza en porciones de 130KFloats (391800F en total) (ver fig. 8.26). También se encuentran los problemas del ruteo *daemon-to-daemon* con el cual se obtiene máximos para tamaños de porciones intermedias y luego caen rápidamente como se muestra en la fig. 8.27. Otra característica que se encuentra presente tanto en el *gather* como en el *scatter* es que a medida que se adicionan más hosts el ancho de banda decrece un poco y lo mismo sucede con la aceleración (figs. 8.26 y 8.28).

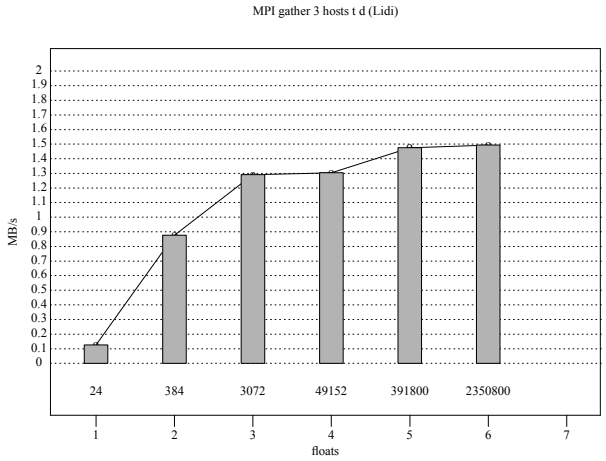

Figura 8.26: MPI *gather* sobre red del LIDI de 10Mb/s - 2 receptores externos - directo - XDR

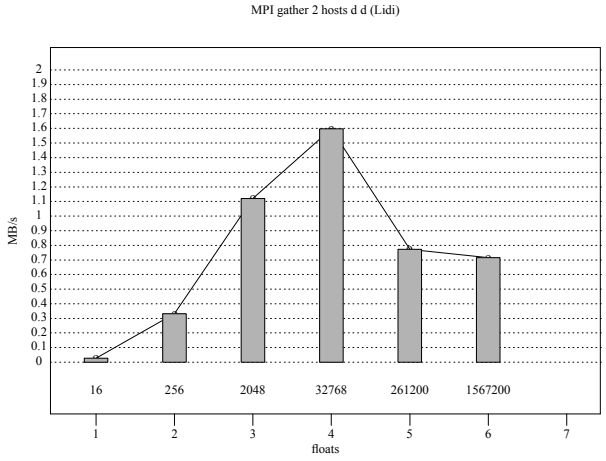

Figura 8.27: MPI *gather* sobre red del LIDI de 10Mb/s - 1 receptor externo - directo - XDR

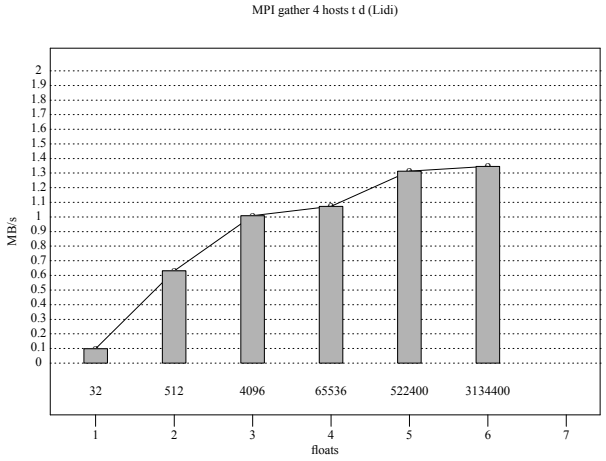

Figura 8.28: MPI *gather* sobre red del LIDI de 10Mb/s - 3 receptores externos - directo - XDR

Para el cluster de 100Mb/s usando LAM/MPI el máximo ancho de banda alcanzado desde el *gather* es menor que el logrado con el *scatter* y la diferencia que existe entre las dos operaciones a medida que se mandan más datos se va reduciendo (figs. 8.30 y 8.31). La única combinación de ruteo y codificación con 3 hosts que supera el máximo físico con el *gather* [1] es la de codificación raw (sin codificar) y ruteo *daemon-to-daemon* (fig. 8.29). Los resultados en general son similares a los del *scatter*.

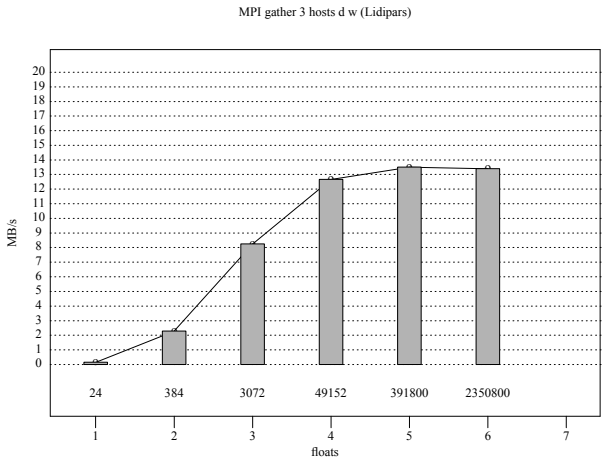

Figura 8.29: MPI *gather* sobre cluster de 100Mb/s - 2 receptores externos - *daemon-to-daemon* - Raw

[1] Recordar que una de las porciones iba dirigida al mismo proceso lo cuál tenía menor costo dando resultados mayores que los físicos

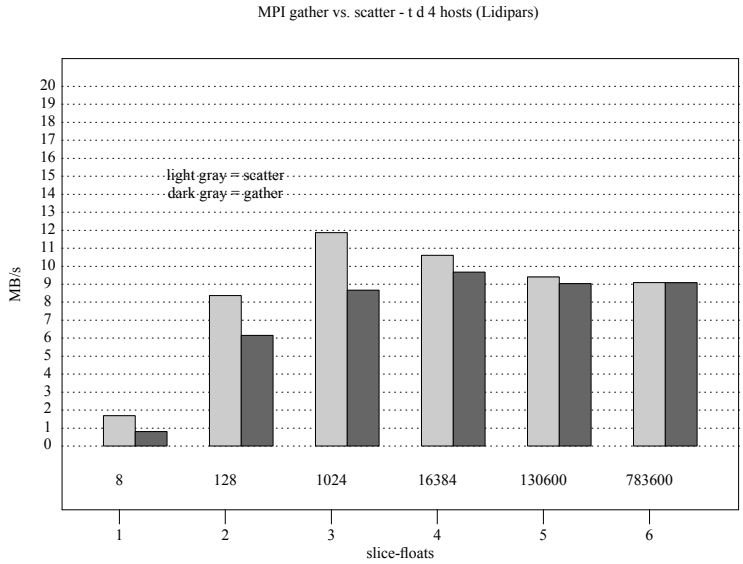

Figura 8.30: MPI *gather* vs. *scatter* sobre cluster de 100Mb/s - 3 receptores externos - directo - XDR

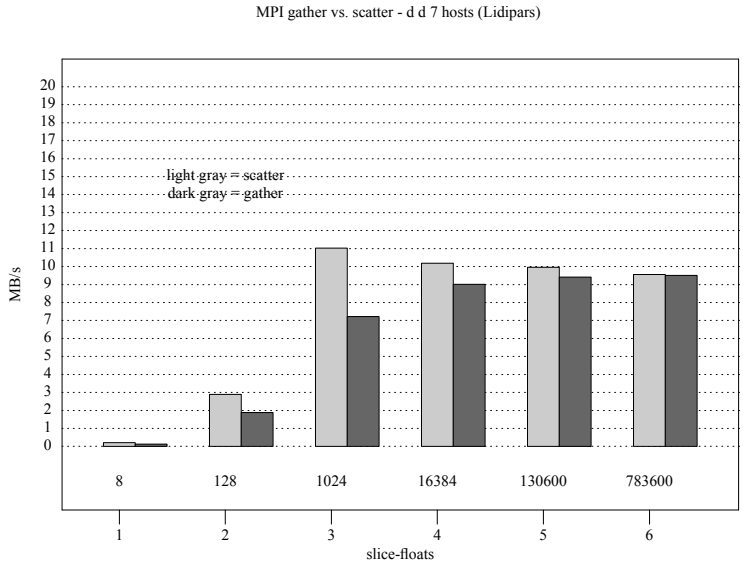

Figura 8.31: MPI *gather* vs. *scatter* sobre cluster de 100Mb/s - 6 receptores externos - *daemon-to-daemon* - XDR

8.2.3 Reduce

Reduce en PVM

Para medir el ancho de banda en el *reduce* se tomó como base la cantidad total de datos reducidos (la suma de todas las porciones). Esta es una medida que sirve para comparar los diferentes algoritmos con una técnica de caja negra sin tener que medir cuanta información realmente es transferida. Se anticipa que como LAM/MPI resuelve esta operación de forma más inteligente se obtendrá que el ancho de banda alcanzado en ocasiones es mayor que el real, esto se debe a que la cantidad de información realmente enviada por la red es menor que el total, debido a que se hacen reducciones en los nodos intermedios del *spanning tree*. A diferencia de esto PVM, para la configuración de un proceso por host, la resuelve con un *gather* y luego reduce todo junto dando un rendimiento bastante más bajo (Ver capítulo *Análisis de Grupos y Comunicaciones colectivas en PVM*). Los resultados obtenidos con el *reduce* usando PVM sobre la red del LIDI de 10Mb/s son muy parecidos a los obtenidos con el *gather* siendo este último un poco superior en el ancho de banda como se ve en las figuras 8.32 y 8.33. Con respecto al start-up también son similares siendo el *gather* un tanto superior.

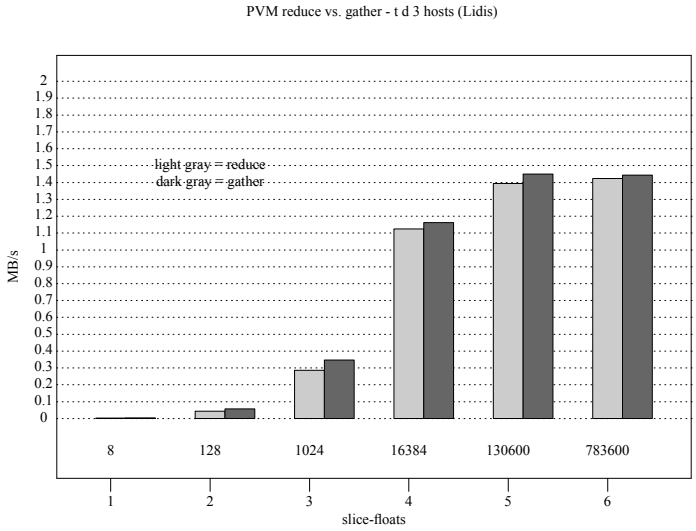

Figura 8.32: PVM *gather* vs. *reduce* sobre red del hetrogénea de 10Mb/s - 2 receptores externos - directo - XDR

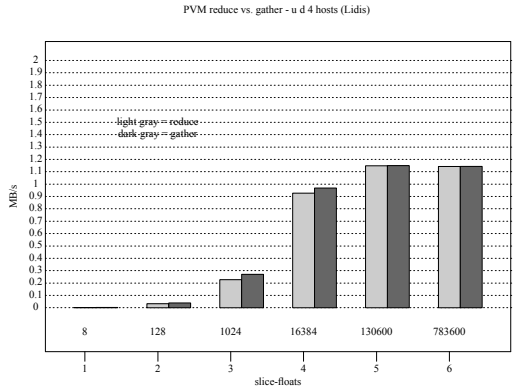

Figura 8.33: PVM *gather* vs. *reduce* sobre red del hetrogénea de 10Mb/s - 3 receptores externos - *daemon-to-daemon* - XDR

Para el cluster de 100Mb/s los resultados son similares a los obtenidos con 10Mb/s si se los compara con el *gather*. Se puede observar una leve superioridad en el ancho de banda del *gather* y una mayor estabilidad en el *reduce* que se observa en la ausencia de picos de máximos locales (figs. 8.35 y 8.34). En el start-up los valores conseguidos por el *reduce* son un poco más altos que los obtenidos con e *gather* pero la diferencia es poco significativa.

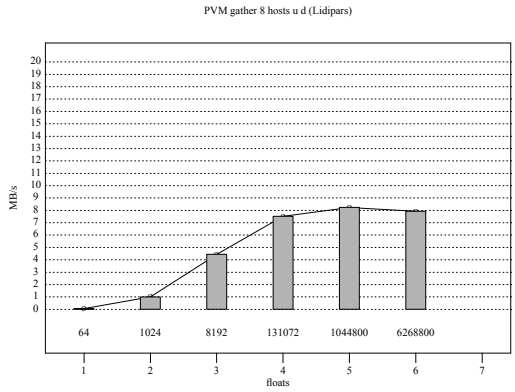

Figura 8.34: PVM *gather* sobre cluster de 100Mb/s - 7 receptores externos - *daemon-to-daemon* - XDR

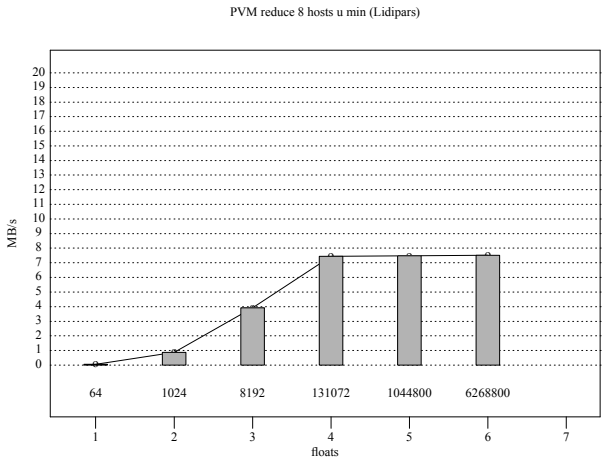

Figura 8.35: PVM *reduce* sobre cluster de 100Mb/s - 7 receptores externos - *daemon-to-daemon* - XDR

Reduce en MPI

Para la red del LIDI a 10Mb/s el start-up es el mismo que el alcanzado con el *gather* para todas las combinaciones de codificación y ruteo. Con respecto al ancho de banda alcanzado también sigue el mismo patrón de comportamiento que el *gather* y sus valores son muy parecidos. En las figs. 8.36 y 8.37 se muestran las similitudes y se puede destacar la diferencia para porciones de 16KFloats sobre la fig. 8.37.

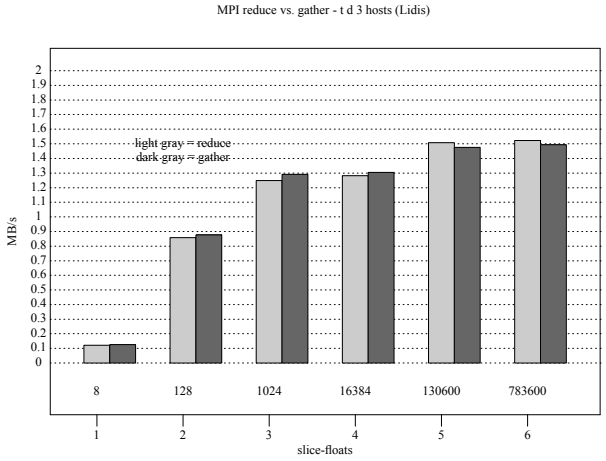

Figura 8.36: MPI *gather* vs. *reduce* sobre red del hetrogénea de 10Mb/s - 2 receptores externos - directo - default

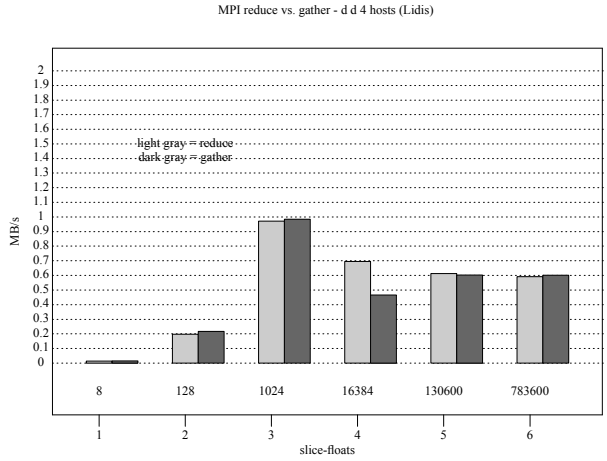

Figura 8.37: MPI *gather* vs. *reduce* sobre red del hetrogénea de 10Mb/s - 3 receptores externos - *daemon-to-daemon* - default

Para el cluster de 100Mb/s los resultados hasta 5 hosts no cambian nada con respecto a lo experimentado con la red de 10Mb/s, siguen teniendo el rendimiento del *gather* tanto con el start-up como con el ancho de banda (figs. 8.39 y 8.30). A partir de 6 hosts inclusive, el ancho de banda alcanzado por el *reduce* comienza a crecer con respecto a los valores alcanzados con menos hosts y supera ampliamente al *gather* como se muestra en la figura 8.38.

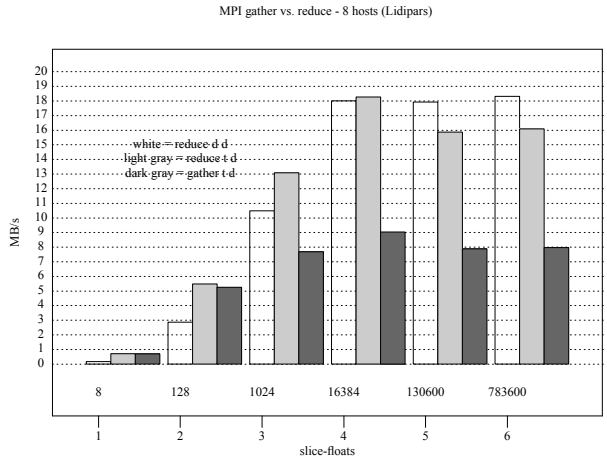

Figura 8.38: MPI *gather* vs. *reduce* sobre cluster homogéneo de 100Mb/s - 7 receptores externos

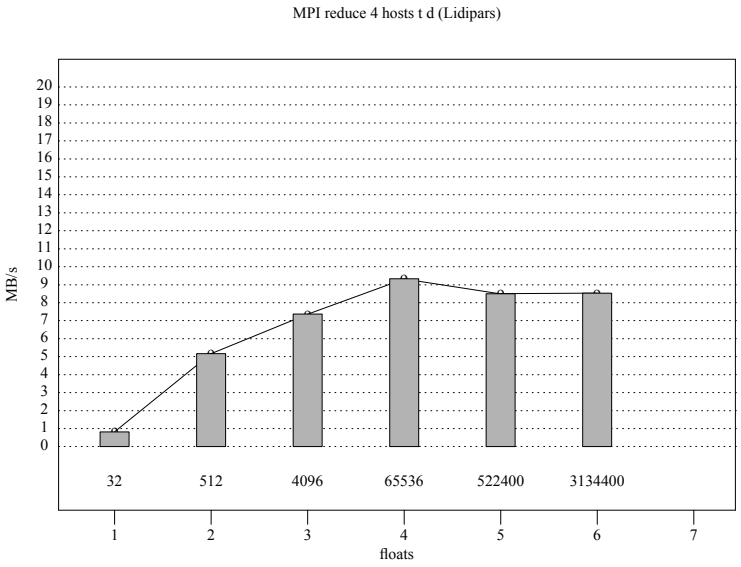

Figura 8.39: MPI *reduce* sobre cluster de 100Mb/s - 3 receptores externos - directo - XDR

Capítulo 9

Propuesta de Implementación

9.1 Introducción

Según los análisis realizados anteriormente se llega a la conclusión que algunas de las operaciones colectivas implementadas sobre los *middlewares* usados no están optimizadas para aprovechar las características de las redes Ethernet. Además de esto se puede apreciar que la sobrecarga (overhead) que adicionan es alta. El primer punto a analizar en este capítulo son las características de las redes de este tipo, luego se hará una comparación entre el modelo más natural para la operación sobre redes Ethernet [METC] y la implementaciones encontradas. Por último se tratarán las formas de disminuir la sobrecarga.

9.2 Características de las redes Ethernet

Este tipo de redes son destinadas a LANs *(Local Area Networks)* y el método de acceso básico que utilizan se conoce como *acceso aleatorio* [SCHW94]. El medio de comunicación usado es un bus lógico compartido por todos los usuarios y la forma de operar es la siguiente:
Si alguien desea enviar información a otro usuario, transmitirá a voluntad por el canal *inundándolo* (en inglés flooding), los datos le llegará a todos pero solo el destino indicado la leerá. Dado que puede suceder que dos o más usuarios decidan transmitir al mismo tiempo, pueden ocurrir *colisiones de paquetes*; por lo tanto debe existir alguna forma de detectarlas y/o evitarlas. El algoritmo usado por Ethernet para solucionar este problema es *CSMA/CD (Carrier-Sense Multiple Access with Collision Detection)* [802.3] y funciona así: todas las estaciones deben escuchar sobre el canal para determinar si hay o no transmisiones en línea, si la estación desea transmitir lo hace cuando censa que el canal esta desocupado (Detección de Portadora o *CS Carrier Sensing*), esto puede suceder desde diferentes estaciones situadas en la misma red (Acceso Múltiple *MA Multiple Access*), y debido a que se pueden dar colisiones, una vez que se comenzó a enviar se debe seguir escuchando sobre el canal para detectar si existe una colisión (Detección de Colisiones *CD Collision Detect*). Si es así se abor-

ta inmediatamente la transmisión. Más tarde volverá a reintentar. Ethernet resuelve los reintentos con un algoritmo conocido como *Back-off*: Una vez detectada la colisión, antes de retransmitir genera un numero aleatorio r dentro de un rango, espera ese tiempo y luego reenvía. Si vuelve a detectar colisión extiende el rango [1] para generar un nuevo valor aleatoria e itera sobre el mismo algoritmo.
De este reducido análisis se pueden enumerar las características principales de estas redes:

- Acceso al medio aleatorio.
- No existe manejo de prioridades.
- Forma de transmisión por *inundación* o *broadcast*.
- Medio transmisión compartido por lo cual solo puede haber una sola estación transmitiendo en un mismo dominio de colisiones [2].
- El receptor de un envío puede ser una estación (*unicast*) varias estaciones (*multicast*) o todas las estaciones (*broadcast*) y solo se genera una transmisión efectiva [3].

9.3 Evaluación de Modelos Encontrados

Las implementaciones de las operaciones colectivas como ya se vio en capítulos anteriores admiten diferentes modelos, que pueden ser o no independientes de la conectividad física y lógica que se tenga en los niveles inferiores.

9.3.1 Broadcast

En el capítulo *Modelos y Métricas* se vio que PVM usa un modelo de *broadcast* lineal haciendo una punto a punto por cada destino y LAM/MPI usa un modelo basado en árbol. Si se piensa en las características que tienen las redes Ethernet se llega a la conclusión que ninguno de los dos modelos es óptimo ni natural pues lo mejor sería aprovechar el *broadcast* soportado por la red. Si se comparan los resultados se ve que sobre las redes de 10Mb/s el comportamiento de PVM es un tanto extraño. Aún así los máximos alcanzados disminuyen conforme se adicionen hosts. Lo mismo sucede con LAM/MPI. Sobre el cluster que posee capacidad de switching el ancho de banda de LAM/MPI es 2 a 2.5 veces superior que el de PVM, aunque esta muy lejano del ofrecido por la red y experimentado con las pruebas punto a punto, 9-12 Mb/s. La propuesta es reimplementar esta operación de forma de hacerla lo más eficiente posible y acercarse al máximo teórico: 12.5Mb/s.

[1] El rango crece en potencias de 2 hasta 2^{10}, si llega alcanzar el valor máximo de rango y no logra transmitir desiste

[2] Se llama dominio de colisiones a un segmento físico de red donde las transmisiones del grupo de estaciones conectados a este son traducidas a *broadcasts*

[3] transmisión efectiva es aquella que llega a su destino

9.3.2 Barrier

El *barrier* en PVM es implementado mediante recepciones y envíos punto a punto hacia el *pvmgs* (Ver capítulo *Análisis de Grupos y Comunicaciones colectivas en PVM*) el cual es el proceso encargado de sincronizar a todos. En LAM/MPI la implementación es basada en árbol para más de 4 participantes y el sincronizador es uno de todos los integrantes de cada grupo, lo cual hace que el procesamiento sea concurrente entre diferentes grupos (de cualquier forma este es un parámetro que no se mide en este trabajo). Si se analizan los mejores tiempos en ambos *middlewares* se tiene que la operación sobre PVM es muy lenta y tiene grandes diferencias con la de LAM/MPI. Estas diferencias son más acentuadas para los datos tomados sobre el cluster de 100Mb/s que tiene capacidad de switching, característica que es aprovechada por el algoritmo en árbol. De acuerdo a los tiempos tomados se llega a la conclusión que el *barrier* de LAM/MPI usando ruteo directo es suficientemente bueno. De cualquier forma se propone implementar la operación por completitud. Si se piensa en un bus común de comunicaciones la forma más natural de hacerlo es la lineal con un sincronizador por grupo. Para esta implementación se debe tener en cuenta la contención por acceder al canal si todos desean sincronizar al mismo tiempo.

9.3.3 Scatter y Gather

Las dos operaciones son implementadas en PVM de forma lineal y en LAM/MPI usando un árbol para más de 4 participantes. Los valores obtenidos sobre el cluster indican que el ancho de banda de LAM/MPI esta entre 1 y 4 MB/s por encima de PVM. Además de tener un start-up mucho más chico y de que los mejores resultados se tienen con ruteo *daemon-to-daemon*. Sobre la red de 10Mb/s LAM/MPI es mejor que PVM con respecto al start-up y al ancho de banda, aunque la diferencia con respecto a este último parámetro no es tan notable. Los mejores resultados sobre la red de 10Mb/s se consigue usando ruteo directo. Para estas operaciones se tienen las siguientes conclusiones: Los valores obtenidos con LAM/MPI son buenos y es probable que no se necesite reimplementarlos pero por una cuestión de completitud y comparación se propone reimplementarlos de la forma más natural usando un modelo lineal.

9.3.4 Reduce

El *reduce* en PVM es implementado casi prácticamente con un *gather* y luego con la aplicación de la operación a todos los datos, si se tiene una tarea por máquina como se mostro en el capítulo *Análisis de Grupos y Comunicaciones colectivas en PVM*). En LAM/MPI el algoritmo esta basado en árbol para más de 4 participantes y los proceso intermedios reciben, reducen y luego envía a los procesos de un nivel superior. Este algoritmo logra un rendimiento óptimo que solo se podría superar construyendo mejores *spanning trees* o disminuyendo la sobrecarga. Como conclusión de esta última operación a analizar se obtiene que la implementación de LAM/MPI es buena si se tiene capacidad de switching pero si se tiene un bus compartido solamente lo más natural es usar un algoritmo lineal para evitar las contenciones al canal por accesos múltiples. Por completitud y comparación se propone una reimplementación basada en envíos lineales y reducción centralizada en un solo proceso.

9.4 Disminución de la Sobrecarga

Algunos de los factores que generan sobrecarga sobre los diferentes *middlewares* son:

- Transferencias de datos mediante procesos intermediarios como lo son los daemons. En ocasiones estos son útiles para facilitar transferencias a varios procesos ejecutando en un mismo host pero en general es más eficiente hacer ruteo directo.
- El manejo de buffers por parte de PVM genera bastante sobrecarga.
- Otros manejos internos y detalles de implementación particulares en ocasiones hacen que el rendimiento sea extraño como ocurría con el *broadcast* de PVM.
- La técnica de codificación XDR hace más lenta las operaciones que impliquen muchos datos y su puede evitar en la mayoría de los casos.

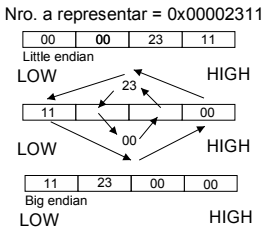

Figura 9.1: Cambio de representación de *little* a *big*

Para solucionar el primero directamente se implementan las colectivas usando comunicaciones directas entre tareas.
Para el segundo y el tercero no se hace uso de buffers ni de mecanismos innecesarios en esta implementación. Se buscará hacer una implementación lo más directa y sencilla posible.
En el último punto será en el cual más énfasis se hará. En general los *middlewares* para resolver problemas de incompatibilidad de representación de datos hacen *marshalling* y *unmarshalling* usando XDR, lo cual significa costos de rendimiento cuando se transmiten muchos datos. Se ha encontrado una sorprendente adhesión al estándar *IEEE 754* [754] para la representación de números en punto flotante, esto se puede aprovechar para reducir los tiempos dedicados a mantener la consistencia de datos. Por lo tanto se puede implementar un método de traducción que implicaría solo transformar los datos en la computadora destino cuando sea necesario. De hecho, el único problema de codificación que se tiene entre diferentes plataformas de estaciones de trabajo es el de representación *little endian* y *big endian* (ver fig. 9.1). En el caso de un *broadcast* o *scatter* el mensaje es enviado sin modificar, y luego cada receptor si tiene diferente representación que el emisario cambiará de representación intercambiando las posiciones de los bytes. En el caso del *gather* cada emisario

cambiará la representación de acuerdo al receptor si es necesario. Este mismo análisis es extensible a otros tipos como enteros, enteros dobles, etc ... Otro punto no tan importante, pero útil para la optimización, es el de la copia de la porción de datos de forma local en el *gather* o en el *scatter*. Esto se puede evitar usando directamente el original que es transmitido.

Bibliografía

[SCHW94] Redes de Telecomunicaciones: Protocolo, Modelado y Análisis - Mischa Schwartz - Addison Wesley Iberoamericana S. A., Wilmington, Delware E.E.U.U. - 1994.

[METC] Ethernet Distributed Packet Switching for Local Computer Networks - R. M. Metcalfe and D. R. Broggs - Comm. ACM, vol. 19, num. 7 - July 1976, pag 395-404.

[802.3] Institute of Electrical and Electronics Engineers, Local Area Network - *CSMA/CD Access Method and Physical Layer Specifications* ANSI/IEEE 802.3 - 1985, IEEE Computer Society.

[754] Institute of Electrical and Electronics Engineers, IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754 - 1984.

Capítulo 10

Desarrollo

10.1 Introducción

La reimplementación de las operaciones colectivas de forma totalmente independiente de los *middlewares* lleva a la necesidad de una serie de servicios y facilidades para lograr que todo funcione de forma armónica. En lugar de implementar todo en un gran núcleo de código donde se mezclan las diferentes funcionalidades se optó por montar la arquitectura de software sobre un modelo en capas. En este capítulo se describen las capas implementadas, su interrelación, y la conexión con el subsistema de red.

10.2 Arquitectura del sistema

Las funcionalidades y componentes esenciales a implementar son:

- Manejo de Grupos
- Traducción de Datos
- Núcleo de la biblioteca de comunicaciones colectivas
- Biblioteca para transferencia de mensajes UDP confiables, *unicast y broadcast*
- *Wrappers* sobre el soporte de comunicaciones utilizado, funciones auxiliares para manejo de temporizadores y estructuras de datos necesarias

El soporte de comunicaciones elegido para montar la reimplementación fue TCP/IP, el *stack* de Internet cuyo espectro de uso son las LANs, MANs y WANs, es el "protocolo de comunicaciones" indiscutible por su uso masivo. La interfaz de programación o API para TCP/IP que se uso fue *sockets BSD*, un estándar para la programación TCP/IP. Los lenguajes de programación utilizados fueron *C y C++*.
Un esquema de la arquitectura completa se muestra en la fig. 10.1. Ahí se resaltan con gris las capas implementadas para este trabajo, las rayadas son interfaces de programación y las blancas capas del subsistema de comunicaciones a nivel de hardware y sistema operativo.

Figura 10.1: Arquitectura del sistema

En las secciones siguientes se describirá cada una de ellas comenzando desde las más bajas implementadas en este trabajo.

10.3 Mensajes UDP Confiables y *Wrapper*

La primera capa es el *Wrapper* sobre los sockets, esta es sencillamente una forma de agrupar y simplificar el conjunto de funciones de los *sockets BSD*. Además define una lista de estados o códigos de error unificados que pueden retornar las funciones. Los mensajes UDP confiables están montados sobre este *Wrapper*.
La necesidad de tener mensajes UDP [UDP768] confiables se debe a que mediante estos se realiza el *broadcast* [BCAST991]. El concepto de confiabilidad se refiere a que si se mandan N cantidad de bytes a M hosts, éstos deben llegar a todos sin error y en el orden correcto, o se debe devolver un mensaje de estado indicando el problema. Se supone que los errores contemplados en las capas del modelo OSI 1,2,3 y 4 por los protocolos son controlados y corregidos por el hardware, el sistema operativo o el *driver*[1], que lo implemente. Las alternativas que se analizaron para implementar el *broadcast* sobre Ethernet fueron 3:

- Implementar un protocolo sobre IP usando directamente el *broadcast* a este nivel usando sockets Raw [Stevens95] en espacio de usuario. O utilizar el mecanismo que ofrezca el sistema e implementar sobre este el control.
- Implementar dentro del stack TCP/IP un *broadcast* confiable usando el código del UDP y ponerlo dentro del kernel junto con los otros protocolos.
- Usar el *broadcast* o el *multicast* de UDP/IP e implementar sobre éste, a nivel de usuario, el control necesario para la confiabilidad.

[1] Se utiliza el término en el sentido amplio de la palabra, y no solamente como controlador de hardware

La elección fue la última alternativa. La primera es buena debido a que tiene una sobrecarga mínima, pues saltea la capa cuatro. Las contras que tienen son que: solo el super-usuario puede ejecutar procesos que realicen esta actividad y que en ocasiones se encuentra una difícil portabilidad, cuestión que es imprescindible para las redes heterogéneas. Otro problema importante que tiene es que todo el control de flujo, el control de error y la multiplexación se debe implementar.
La segunda alternativa tiene como bueno la poca sobrecarga y la reutilización de las estructuras y funciones del kernel. El protocolo estaría implementado a nivel de kernel por lo que el rendimiento sería mejor. Tiene como contra la escasa portabilidad sobre diferentes UNIX y el duro trabajo de tener que reimplementar mucho código para cada plataforma UNIX que exista. Para esta alternativa se encontró el antecedente [Brucks95].
La última alternativa si bien es la que más sobrecarga aporta es la más portable, flexible y sencilla de implementar. Se opto por usar el *broadcast* en lugar del *multicast* por ser más fácil de usar y porque esta soportado en todas las placas de red Ethernet. La ventaja del *multicast* sobre el *broadcast* es que es enrutable y genera menos carga sobre las máqiunas de la red que no participan en la comunicación.

10.3.1 Tipos de Broadcasts

Los tipos de *broadcast* sobre IPv4 son [StevensI98]:

Broadcast directo de red (Net-directed Broadcast): Esta operación usa como dirección IP todos unos en los bits de la parte del host, por ejemplo una dirección de clase "C" sería 192.168.0.255 (netid.255). Los routers deben rutear (forward) estos paquetes pero en general se deshabilita y solo es válido si todas las máquinas están en la misma red.

Broadcast limitado (Limited Broadcast): Esta operación usa la dirección con todos los bits puestos en uno, o sea: 255.255.255.255. Su uso debe ser limitado, como en ocasiones cuando no se tiene una dirección y se desea obtener una como sucede con el protocolo `bootp` [StevensI98].

Broadcast directo a sub red (Subnet-directed Broadcast): Es una generalización del *broadcast directo*. Tiene los bits correspondientes al hosts en uno pero además especifica la sub red. Por ejemplo la dirección de clase "A" 10.0.0.0 con máscara de red 255.255.255.0 tiene como *broadcast* 10.0.0.255 (netid-subnetid.255). Si la máscara fuese 255.255.0.0 el *broadcast* sería 10.0.255.255.

Broadcast directo a todas las sub redes (All-subnets-directed Broadcast): En este caso en la parte del host de la dirección y en la sub red tienen valores unos. Por ejemplo dada la dirección de red de clase "A" 10.0.0.0 con 253 subredes (con máscaras 255.255.0.0 y las subredes son 10.1.0.0, 10.2.0.0, 10.3.0.0, etc ...) el boradcast será 10.255.255.255 pero si la máscara es 255.0.0.0 esta es una dirección de *broadcast* directo.

La direcciones de *broadcast* que se usa son del tipo directo de red y se toman desde la configuración del sistema donde está el equipo que participa en la operación.

10.3.2 Implementación de UDP confiable

La necesidad del UDP se debe a que el *broadcast* es implementado usando datagramas sobre IP. El problema principal que tiene UDP es que no provee confiabilidad *(reliability)*, se envían los datagramas a la capa IP, pero no se garantiza que el destino sea alcanzado. La unidad de transferencia son datagramas que pueden llegar en diferente orden o perderse de forma entera o alguno de sus fragmentos [2] o descartados si el *checksum* indica algún error. Debido a esto, las aplicaciones son las encargadas de controlar que la información llegue de forma correcta, completa y en orden.
Para solucionar este problema se implementó una capa que cumple las funciones de un protocolo a nivel de sesión que se encarga de controlar que lleguen todos los paquetes a todos y en el orden correcto. Se hicieron pruebas de poner el control sobre la misma conexión [3] UDP/IP o sobre otra conexión dedicada. Para usar la misma conexión UDP/IP en el control se necesitaba tener temporizadores, además hacía poco simple al protocolo. Usando una conexión alternativa para el control con UDP/IP seguía requiriendo de temporizadores y no achicaba mucho la complejidad del algoritmo. Se terminó optando por tener una conexión de datos UDP y una de control TCP. Sobre la conexión de control se envían paquetes de *ACK* (confirmación), paquetes de *INIT* (inicio de sesión), paquetes de *NEGOT* (Negociación de parámetros), paquetes de *END* (fin de conexión o sesión), etc.
El protocolo fue llamado RUDP (Reliability over UDP) y consta de 3 etapas o fases:

1. En la primera fase se establece la conexión y se negocian los parámetros.
2. En la segunda se hace el envío o la recepción de los datos. En esta fase se pueden hacer todos los envíos que se desean.
3. Luego que terminó la segunda fase se debe cerrar la conexión para liberar los recursos del sistema. Los buffers alocados por el usuario deben ser liberados por el mismo si no los desea reutilizar.

A continuación se describen las funciones necesarias en cada uno de los pasos para enviar o recibir un *broadcast*.

Envío de un Broadcast

```
C   cc_rudp_establish_conn_as_bcast_send(
                          cc_rudp_bcast_connection_t *conn1,
                          cc_rudp_src_t              src_addr,
                          cc_rudp_dest_t             dest_addr,
                          size_t                     peers);
```

[2] Los datagramas pueden ser fragmentados en el camino de acuerdo al MTU de las redes que atraviesa
[3] Conexión se refiere a un socket abierto

Esta función sirve para establecer los canales lógicos de datos y de control del lado del emisario. El parámetro `conn1` es la conexión *RUDP broadcast.* El parámetro `src_addr` es la dirección desde donde se envía. El parámetro `dest_addr` es un arreglo con las direcciones de todos los receptores. El parámetro `peers` son la cantidad de receptores.

```
C   cc_rudp_negotiate_params_as_bcast_send(
                          cc_rudp_bcast_connection_t *conn1,
                          size_t                     qtty,
                          size_t                     len,
                          size_t                     frag,
                          size_t                     burst);
```

Esta es la siguiente función que debe llamarse. A partir de su invocación se negocian los parámetros de datos de la conexión previamente establecida. Los valores son impuestos por el emisario.
Los parámetros `qtty, len, frag, y burst` son la cantidad de paquetes, la longitud o MPU (Max. Packet Unit), la cantidad de bytes que no completan un paquete y la cantidad de paquetes que conforman una ráfaga. El concepto de ráfaga se explica más adelante.

```
C   cc_rudp_bcast_send4(
              cc_rudp_bcast_connection_t *conn1,
              cc_rudp_unit_t             *buffer,
              size_t                     size,
              size_t                     qtty,
              size_t                     len,
              size_t                     frag,
              size_t                     burst);
```

Mediante esta función se hace efectivo el *broadcast.* El parámetro `buffer` es el arreglo de datos a enviar y `size` la cantidad de unidades en el arreglo.

```
C   cc_rudp_end_conn_as_bcast_send(
              cc_rudp_bcast_connection_t *conn1);
```

Por último mediante esta función se cierra la conexión.

Recepción de un Broadcast

```
C   cc_rudp_establish_conn_as_recv(
              cc_rudp_connection_t *conn2,
              cc_rudp_src_t        src_addr,
              cc_rudp_dest_t       dest_addr,
              size_t               addrlen);
```

Establece los canales del lado de cada receptor que la ejecuta.

```
C   cc_rudp_negotiate_params_as_recv(
              cc_rudp_connection_t *conn2,
              size_t               *qtty,
              size_t               *len,
              size_t               *frag,
              size_t               *burst);
```

Negocia los parámetros. Recibe los valores configurados por el emisario.

```
C   cc_rudp_recv3(cc_rudp_connection_t *conn2,
                  cc_rudp_unit_t       *buffer,
                  size_t               size,
                  size_t               qtty,
                  size_t               len,
                  size_t               frag,
                  size_t               burst);

C   cc_rudp_end_conn_as_recv(cc_rudp_connection_t *conn2);
```

La primera función recibe los datos del *broadcast* y la segunda finaliza la conexión.

La implementación que se hizo es sencilla pero logra tener un rendimiento cercano al pico máximo teórico en las redes que se probó. El funcionamiento es el siguiente:
Una vez que se estableció la conexión y se negociaron los parámetros el total del buffer a enviar se divide lógicamente en porciones llamadas ráfagas. A su vez las ráfagas están divididas en unidades más pequeñas llamadas paquetes que agregándosele la información de control,como número de secuencia, pasan a ser los datagramas UDP (ver fig. 10.2).

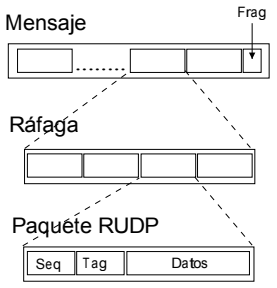

Figura 10.2: Estructura de un mensaje RUDP

Se comienza con el envío de la primera ráfaga, luego la segunda y así hasta la última. El envío de cada ráfaga consiste en mandar un conjunto de datos consecutivos y luego esperar un mensaje *ACK* de control por cada receptor que indique qué paquetes de la ráfaga enviada no fueron recibidos. Los mensajes de *ACK* son vectores de bits que con un uno indican que no se recibió y con un cero que si. Se hace un *OR* con todos los *ACK* y se reenvían los no recibidos. No se pasa a la siguiente ráfaga hasta que todos recibieron todos los paquetes. Existe un límite de error que aborta si luego de N intentos no se logra pasar a la siguiente ráfaga. Las forma de decidir cuándo enviar el paquete de *ACK* se implementó de maneras diferentes, con un temporizador o con un paquete de control *LAST* que obliga a los receptores a enviar la confirmación. El que mejor resultado dio sobre la red de 10Mb/s fue el temporizador y sobre la de 100Mb/s el paquete de *LAST*. Los parámetros de tamaño de paquete y cantidad de paquetes por ráfaga no fueron los mismos para tener resultados óptimos en los dos tipos de redes. Un problema que se encontró es que depende de la plataforma (equipo y sistema operativo) el tamaño máximo de datagrama, por lo cual el resultado más balanceado que se encontró fue usar paquetes chicos.

- Mejor para red de 10Mb/s: `pkt = 10240, burst = 160, timer`
- Mejor para red de 100Mb/s: `pkt = 1024 , burst = 64, last`

10.4 Manejo de Grupos

Al decidir reimplementar las operaciones colectivas de forma totalmente independiente de los *middlewares* se necesita tener alguna forma de coordinar a los participantes de la operación y algún método de intercambiar la meta-información o *meta-data* (datos que sólo tienen significado para la implementación como lo son direcciones IP, números de puertos, etc) entre los procesos participantes. Las alternativas que se evaluaron fueron:

- Implementar un manejo estático de *meta-data* determinado antes de la ejecución.
- Usar los mecanismos provistos por los *middlewares* (PVM, MPI).
- Implementar un mecanismo dinámico y *had-hoc*.

La primera alternativa tiene a favor que es la más sencilla y la que menos sobrecarga genera. La contra que tiene es la poca flexibilidad y que es muy probable que no alcance a dar todo el soporte necesario.
La segunda es buena porque reutiliza lo que ya está implementado pero tiene la contra de que hay que hacer una interfaz común que abstraiga los mecanismos diferentes de cada *middleware*. Además "juega en contra" de la independencia.
La última alternativa es la más costosa en términos de implementación pero la más flexible debido a que el autor será el encargado de analizar los requerimientos e implementarlos de la mejor forma para optimizarlos. La elección fue la última. Lo que se implementa es el manejo de grupos mediante un servidor de grupos de forma similar a la implementación de PVM.

10.4.1 Especificación de Servicios Necesarios

Los servicios que se implementarán serán los necesarios para armar, modificar y desarmar grupos de procesos de forma dinámica. También se debe proveer a las capas de un nivel más abajo la información necesaria para comunicarse con cada una de las tareas. Una especificación en IDL (Interface Definition Language) de algunos de los servicios que se deberían tener puede ser la siguiente:

```
module CC_Groups
{
  // Nombre del grupo
  typedef string cc_gname_t;
  // Entero largo
  typedef long   ulong;
  // Entero corto
  typedef short  ushort;
  // Valor de retorno
  enum cc_callret_t {success, fail};
  /******* Inicio de definici'on de objetos opacos *******/
   struct cc_group_t
  {
    ulong master_ip;
    ulong grp_id;
    // ......
  };
  struct cc_task_t
  {
    ulong task_id;
    // ......
  };
 /******* Fin de definici'on de objetos opacos *******/
 typedef sequence <cc_task_t> cc_glist_t;

  interface cc_grouper
  {
    // Operaci'on que agrega al grupo al invocador,
    // si el grupo no existe lo crea
    cc_callret_t cc_gjoin(in cc_gname_t grpname,out cc_group_t grp);

    // Operaci'on que saca del grupo al invocador
    cc_callret_t cc_gleave(in cc_group_t grp);

    // Operaci'on que da un miembro del grupo
    cc_task_t   cc_gmember(in  cc_group_t grp, in  ulong pos);

    // Operaci'on que testea si existe un grupo
    boolean      cc_gexist(in cc_gname_t grpname);

    // Operaci'on que retorna la cantidad de miembros de un grupo
    ushort       cc_gsize(in  cc_group_t grp);
```

```
    // Operaci'on que hace al grupo est'atico
    cc_callret_t cc_gstatic(in cc_group_t grp);
  };
};
```

10.4.2 Implementación de los Servicios

A partir de esta especificación se realiza la implementación haciendo los cambios que se creyeron necesarios. El lenguaje escogido para implementar el servidor de grupos o *gserver* fue *C++* debido a las facilidades de la OOP (Object Oriented Programming): la reusabilidad (se hace uso de las estructuras de datos ofrecidas por la STL -Standard Template Library- [STL]), el encapsulamiento (la implementación de los grupos y su manejo puede ser cambiada sin que los clientes se enteren), la herencia (en nuestro caso se uso para tener una jerarquía de estados de grupos y derivar su comportamiento de acuerdo a su clase) y la abstracción. Por otro lado se necesitó implementar una biblioteca y una interfaz para comunicarse con el *gserver*, ésta se hizo en *C* para seguir la línea del desarrollo de las operaciones colectivas que se hicieron. La comunicación entre los clientes y el *gserver* se hace mediante TCP y el desarrollo está basado en sockets. La implementación del servidor es bastante sencilla y no se hace hincapié en tolerancia a fallas ni en ofrecer alto grado de concurrencia (Por ejemplo el servicio es brindado por un solo proceso de forma secuencial por lo que si una tarea se está uniendo a un grupo y llega otra para ser atendida deberá esperar que la primera termine). La implementación ofrecida es para brindar un *gserver* que sirva para probar las comunicaciones colectivas desarrolladas y medir sus tiempos.
El resultado final es un programa que debe ejecutarse en una de las máquinas de la red que se utilizará como máquina paralela, de esta forma se inicia un proceso daemon que será el encargado de manejar los grupos. Además de esto se tiene la biblioteca de conexión al *gserver* que deberá linkeditarse junto con el/los ejecutable(s) que conforman las tareas de la aplicación.

10.4.3 Algunos detalles de la Implementación

Los detalles del *gserver* no se mostrarán ni se analizarán en este trabajo porque ese no es el objetivo. Lo que se presenta en esta parte es la interfaz principal de la biblioteca de conexión con el *gserver* y los comentarios sobre las funciones.

```
C      cc_err_cod_t cc_c_createGrp(const char* gname);
```

Esta función crea un grupo con nombre `gname` en el servidor de grupos. Si el grupo existe devuelve un código de error.

```
C      cc_err_cod_t cc_c_basic_joinGrp2(const char*       gname,
                                        cc_gmemhandler_t* hdr);
```

Esta función une al grupo con el nombre `gname` al proceso que la invoca y da como resultado un manejador `hdr`, este es un objeto opaco mediante el cual se mantendrá el estado y los parámetros necesarios para el uso del grupo donde se integró. Si el grupo no existe devuelve un código de error.

```
C      cc_err_cod_t cc_c_exitGrp(cc_peer_gmemhandler_t* hdr);
```

El proceso que llama a esta función abandona el grupo referenciado por el manejador `hdr`. El último en dejarlo destruye el grupo. Si el manejador es inválido retorna un código de error.

```
C     cc_err_cod_t cc_c_getMemberGrp(cc_peer_gmemhandler_t* hdr,
                                     int                member_id,
                                     cc_gmember             *mem);
```

Devuelve en la variable `mem` el miembro identificado por el valor de la variable `member_id` dentro del grupo referenciado por `hdr`. Si el manejador es inválido o `member_id` tiene un valor incorrecto devuelve un código de error.

```
C      bool cc_c_includesGrp(const char* gname);
```

Operación que retorna verdadero o falso si es que existe un grupo con el nombre `gname`.

```
C      cc_err_cod_t cc_c_sizeGrp(cc_peer_gmemhandler_t* hdr)
```

Operación que devuelve la cantidad de miembros de un grupo. Si el manejador es inválido devuelve un código de error.

```
C      cc_err_cod_t cc_c_makeLocalGrp(cc_peer_gmemhandler_t* hdr);
```

Operación que hace estático al grupo mantenido por el *gserver* y hace una copia local del mismo en el manejador del invocador. Si el manejador es inválido o el grupo ya es local devuelve un código de error. Esta implementación se hace para mejorar el rendimiento.

```
cc_err_cod_t cc_c_getIdGrp(cc_peer_gmemhandler_t* hdr)
```

Devuelve la posición asociada en el grupo al manejador `hdr`. Si el manejador es inválido devuelve un error. Esta operación se usa para obtener una identificación única para cada proceso dentro de un grupo.

10.5 Traducción de Datos

Como ya se mencionó en el capítulo anterior: "Análisis de Resultados", el único inconveniente encontrado con respecto a la representación de datos es el tipo de "endianización", *little* o *big*. Para solucionar este problema se implementa una capa que resuelva el problema cambiando el orden de los bytes conforme sea necesario. Esta capa de software ofrece un conjunto de tipos básicos, y operaciones para transformar de una representación a otra. Cuando una tarea envía datos, automáticamente detecta el tipo de representación local y agrega un encabezado que la describe. Los procesos receptores reciben los datos, detectan su representación y la comparan con el encabezado que llego junto con los datos, si son diferentes realiza la transformación y luego entrega los datos a la capa superior, si son iguales los entregan sin hacer cambio alguno. El problema que puede surgir con respecto a este método es que las diferentes plataformas pueden utilizar modelos de datos diferentes[64bits], tanto en 32 como en 64 bits. La capa de representación además ofrece codificación XDR [XDR] para los casos que con la transformación no se pueda resolver el problema de la representación. El mecanismo de elección del algoritmo de codificación debe ser negociado de antemano. Esta cuestión no ha sido tratada en este trabajo. Una posible idea sería chequear el modelo de datos con `sysconf` o en el archivo `unistd.h` (Si es un Unix que cumple con los estándares de "The Open Group, X/Open" [SingleUnix2] [Open Group] y ver si son compatibles con los demás).

Los tipos de datos ofrecidos son:

- Characters
- Bytes
- Short Integers
- Integers
- Pointers (No tienen mucho sentido pero pueden ser útiles para datos opacos)
- Long Integers
- Floats
- Double Floats

10.6 Núcleo de las Operaciones Colectivas

Las operaciones colectivas implementadas, llamadas `cc_operations` son cinco, soportada cada una por las siguientes funciones:

10.6.1 Barrier

```
C      cc_err_cod_t cc_c_barrier(cc_handler_t *handler,
                                 int          root,
                                 size_t       peers);
```

El *barrier* está implementado directamente sobre el *Wrapper* de sockets usando TCP. La operación requiere que el usuario designe un proceso que cumpla el rol de coordinador o proceso `root`, y que se le indique la cantidad de procesos a sincronizar. Además recibe un manejador obtenido mediante una operación de inicialización que se describe más adelante. Esta función opera haciendo una llamada por cada par al coordinador y éste le responde a todos recién cuando todos le han llamado.

10.6.2 Broadcast

```
C      cc_err_cod_t cc_c_recvBcast(cc_handler_t    *handler,
                                   int              root,
                                   size_t           size,
                                   cc_rudp_unit_t  *buffer,
                                   cc_type_t        type);

C       cc_err_cod_t cc_c_sendBcast(cc_handler_t    *handler,
                                   size_t           size,
                                   cc_rudp_unit_t  *buffer,
                                   cc_type_t        type);
```

El *broadcast* está basado en el envío UDP confiable y en lugar de tener una única función se lo divide en dos, una para recibir y una para enviar. Los parámetros requeridos para recibir son el manejador, quién es el proceso que envía (un entero como identificador), qué cantidad de datos se reciben, en qué buffer se ponen y de qué tipo de datos son (los tipos de datos son los mencionados en la sección *Traducción de Datos*). Para la función de envío se requieren los mismos parámetros, salvo, quién es el que envía debido a que está implícito. La cantidad de procesos a recibir el mensaje son todos los que pertenecen al grupo sin contar al proceso `root`, este último ya posee los datos localmente.

10.6.3 Scatter y Gather

```
C       cc_err_cod_t cc_c_scatter(cc_handler_t      *handler,
                                  int               root,
                                  size_t            peers,
                                  size_t            size,
```

```
                                    cc_rudp_unit_t    *buffer,
                                    cc_type_t         type,
                                    cc_distributor_t  dist);

C       cc_err_cod_t cc_c_gather(cc_handler_t      *handler,
                                 int               root,
                                 size_t            peers,
                                 size_t            size,
                                 cc_rudp_unit_t    *buffer,
                                 cc_type_t         type,
                                 cc_distributor_t  dist);
```

El *gather* y el *scatter* están desarrollados directamente sobre el *Wrapper* de sockets usando TCP. Se implementan de forma sencilla haciendo envíos punto a punto desde o hacia el proceso designado `root`. Los parámetros necesarios son el manejador, el identificador del proceso `root`, la cantidad de procesos desde donde recibir o a donde enviar, el buffer, la cantidad de datos, el tipo de los datos y por último una estructura de tipo `cc_distributor_t` que indica como se debe hacer la distribución de las porciones. Las porciones son pedazos de buffer que se numeran desde 0 hasta *N-1*, donde *N* es el valor de `peers`. Los identificadores de los procesos son los enteros que se pueden obtener a partir de la función `cc_c_getIdGrp` explicada en la sección de "Manejo de Grupos". A cada proceso le toca recibir o enviar *size/N* datos.

10.6.4 Reduce

```
C       cc_err_cod_t cc_c_reduce(cc_handler_t      *handler,
                                 int               root,
                                 size_t            peers,
                                 size_t            size,
                                 cc_rudp_unit_t    *buffer,
                                 cc_type_t         type,
                                 cc_distributor_t  dist,
                 void (*func)(cc_type_t,char*,char*,char*,int));
```

El *reduce* se implementa con un *gather* y luego a todos los datos recibidos se le aplica la función `func`. Esta función recibe como parámetros un tipo, dos punteros a valores de este tipo y da como resultado otro valor del mismo tipo y un código de estado. Para las operaciones es conveniente que sean asociativas y conmutativas. Al programador se le ofrecen las siguientes:

```
void cc_sum_func(cc_type_t type, char *x, char *y,  char *result,
                 int status);

void cc_prod_func(cc_type_t type, char *x, char *y,  char *result,
                  int status);

void cc_or_func(cc_type_t type, char *x, char *y,  char *result,
                int status);
```

```
void cc_and_func(cc_type_t type, char *x, char *y,  char *result,
                 int status);

void cc_min_func(cc_type_t type, char *x, char *y,  char *result,
                 int status);

void cc_max_func(cc_type_t type, char *x, char *y,  char *result,
                 int status);
```

La suma, el producto, el *OR* lógico, el *AND* lógico, el mínimo y el máximo. Pero este tiene la flexibilidad de poder implementar otras.

10.7 Detalles de la Interfaz

Además del núcleo de las operaciones colectivas se tiene una operación para inicializar el entorno, otra para finalizar su uso y liberar los recursos. Se agregan a estas las necesarias para la manipulación de estructuras de tipo `cc_distributor_t` comentadas anteriormente.

```
/** Initialize/Finalize **/

cc_err_cod_t cc_c_init(cc_handler_t *h, char *groupname,
                       size_t peers, BOOL as_root);

cc_err_cod_t cc_c_end(cc_handler_t *h, size_t peers);
```

La primera función configura el manejador `h` tomando como entrada un nombre de grupo, la cantidad de pares que van a formar el grupo y una bandera que marca si este proceso va a ser el encargado de crear el grupo. La segunda libera los recursos ocupados por el manejador y el grupo creado en la llamada a `cc_c_init`

```
/** Distributor Handling **/

cc_err_cod_t cc_c_init_distr(cc_distributor_t *dist,
                             size_t len, BOOL asc);

cc_err_cod_t cc_c_set_distr(cc_distributor_t *dist,
                            int target, int slice);

cc_err_cod_t cc_c_free_distr(cc_distributor_t *dist);
```

Estas funciones son las encargadas de manipular los distribuidores usados en el *scatter*, *gather* y *reduce*. La primera inicializa un distribuidor para `len` porciones de forma ascendente o descendente según se indique por la bandera

`asc`. La segunda indica que al proceso identificado con `target` se lo relacione con la porción `slice`. La última libera el distribuidor. Existen también operaciones para consultar el distribuidor pero se cree que no es necesario explicarlas.

10.8 Resultado de la implementación: ECCLib

El resultado de todo este desarrollo es puesto en una biblioteca de soporte para hacer programas paralelos que requieren comunicaciones colectivas usado el paradigma de pasaje de mensajes en redes Ethernet. La operación colectiva que esta optimizada para redes Ethernet es el *broadcast*. La biblioteca es llamada ECCLib (Ethernet Collective Comunications Library). En el siguiente capítulo se presentan los resultados obtenidos usando esta biblioteca.

Bibliografía

[XDR] Sun Microsystems, Inc. XDR: External Data Representation Standard. RFC 1014, Sun Microsystems, Inc. 1987.

[UDP768] User Datagram Protocol. RFC 768. J. Postel. ISI 28 August 1980.

[Brucks95] Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Catalin Rosu,Ray Strong. SPAA '95 Santa Barbara CA USA 1995 ACM 0-89791-717.

[StevensI98] TCP/IP Illustrated, Volume 1, The Protocols. W. Richards Stevens. Addison Wesley Longman Inc. 1. 11th. Printing February 1998.

[64bits] Data Size Neutrality and 64-bit Support. Copyright (c) 1997-1999 The Open Group.

[SingleUnix2] The online version of te Single Unix Specification can be found at URL `http://www.UNIX-systems.org/online`. The Single UNIX Specification, Version 2. Copyright (c) 1997 The Open Group.

[Open Group] Open Group www URL: `http://www.UNIX-systems.org/`.

[Stevens95] TCP/IP Illustrated, Volume 2: The Implementation, W. Stevens, R. Wright. Addison-Wesley, 1995, ISBN 0-201-63354-X.

[BCAST991] Broadcasting Internet Datagrams. RFC 919. Jeffrey Mogul. Computer Science Department Stanford University. October 1984

[STL] Introduction to the Standard Template Library. Copyright (c) 1994 Hewlett-Packard Company. Online at SGI techpub site, URL http://www.sgi.com/tech/stl/

Capítulo 11

Resultados con la Nueva Implementación

11.1 Introducción

En este capítulo se muestran los resultados obtenidos con la reimplementación de las comunicaciones colectivas mediante la biblioteca ECCLib.
La operación más importante a analizar es el *broadcast* y por esta razón se analizará primero. Luego se mostrarán los resultados del *barrier*, *scatter*, *gather* y *reduce*.

11.2 Resultados Obtenidos

11.2.1 Broadcast

Se analiza primero el start-up y luego el ancho de banda.
El start-up (tabla 11.1) con ECCLib en la red de 10Mb/s del LIDI es el más alto de todos. La diferencia con PVM y LAM/MPI con ruteo *daemon-to-daemon* no es tan notoria. En el cluster de 100Mb/s el start-up de ECCLib vuelve a ser el más alto y la relación entre diferencias parece conservarse. Estos valores se deben a que la implementación del protocolo RUDP genera mucha sobrecarga para tener UDP confiable.

Network	Hosts	PVM d	PVM t	MPI d	MPI t	ECCLib
LIDI (10Mb/s)	2..3	0.0030	0.0020	0.0010	0.00010	0.0030
LIDI (10Mb/s)	4	0.0030	0.0020	0.0050	0.00050	0.0100
LIDIpars (100Mb/s)	2..3	0.0007	0.0003	0.0002	0.00004	0.0008
LIDIpars (100Mb/s)	4..8	0.0007	0.0003	0.0004	0.00010	0.0018

Tabla 11.1: Comparación de start-up del *broadcast*

Este tipo de resultados implica que la implementación realizada no sirve para los casos en los cuales se requiere enviar poca información. Para estas

situaciones se probaron *broadcasts* a nivel de usuario con la menor sobrecarga posible. Se utilizaron dos implementaciones:

- *Broadcast* sobre UDP directo sin hacer control de pérdidas.
- Conexiones TCP directas.

La primera alterativa dio como resultado que para paquetes menores de 4KFloats no se perdió ninguno y obtuvo los mejores tiempos. Estos resultados no dejan de ser pruebas estadísticas lo que implica que el protocolo puede fallar, debido a que está basado en un protocolo no confiable, UDP. La otra alternativa, conexiones TCP directas, tiene la ventaja de ser confiable pero no es escalable ya que por cada conexión se necesita un socket, además del envío de los datos punto a punto. Los resultados obtenidos sobre el cluster de 100Mb/s para 1 receptor y 7 receptores son presentados en los gráficos 11.1 y 11.2.

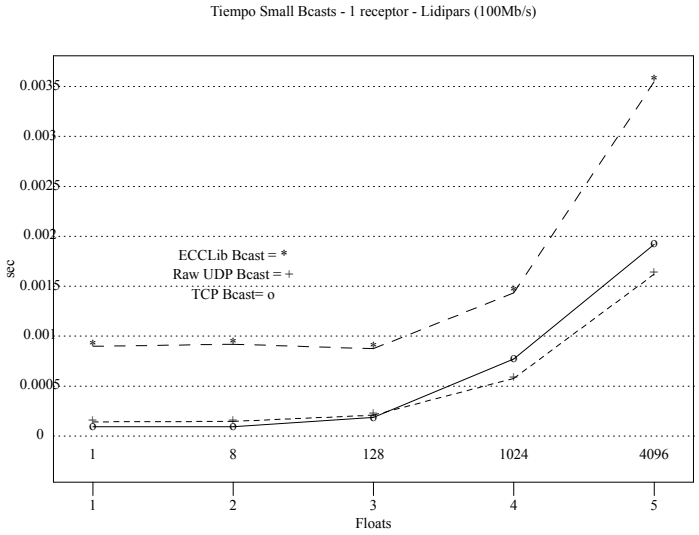

Figura 11.1: Comparación del tiempo de los *broadcasts* implementados - cluster de 100Mb/s - 1 recpetor externo

Se puede ver que los tiempos obtenidos para pocos datos usando directamente conexiones UDP y TCP con un solo receptor son mejores que los resultados con RUDP. En la figura 11.2 se puede ver la poca escalabilidad que comienza a tener el uso de conexiones TCP. Otro resultado interesante que se observa es que RUDP se comporta de forma muy parecida a UDP directo, pero agrega una sobrecarga constante que sitúa los valores un tiempo fijo por encima. Esto deja como conclusiones que para lograr un start-up mínimo es útil usar el *broadcast* UDP, pero a éste se le debe agregar un control mínimo que lo haga confiable. En este trabajo se busca como objetivo optimizar el ancho de banda, un factor más relevante para aplicaciones que necesitan enviar muchos datos, por lo cual los resultados obtenidos se consideran satisfactorios.
Si se analiza el ancho de banda en la red de 10Mb/s se ve que los resultados para muchos datos son altamente favorables al explotar el *broadcast* que tienen

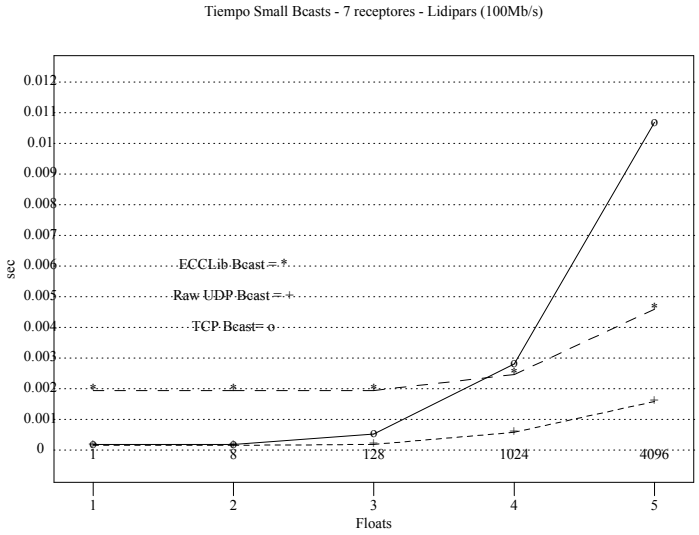

Figura 11.2: Comparación del tiempo de los *broadcasts* implementados - cluster de 100Mb/s - 7 receptores externos

las redes Ethernet. Comparando la gráfica 11.3 con 11.4 se obtiene que son iguales y que no se pierde rendimiento al agregar un host como sucedía con los *middlewares* analizados anteriormente.

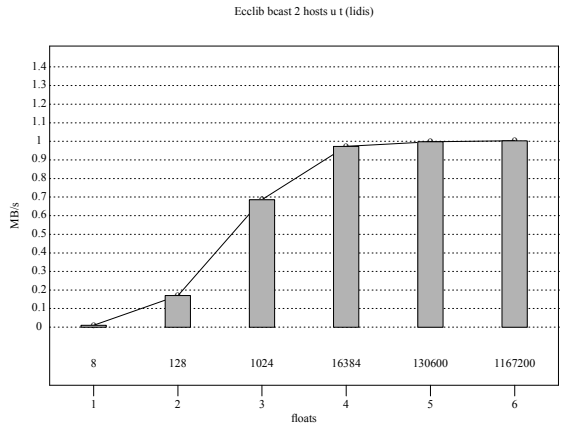

Figura 11.3: *Broadcast* implementado en el trabajo - red del LIDI 10Mb/s - 1 receptor externo

Observando la figura 11.5 se ve que para muchos datos existe un rendimiento de casi el doble de los encontrados en capítulos anteriores. Hay una caída en el rendimiento si se lo compara con 3 hosts (fig. 11.4) debido a que la última máquina que se agrega a la operación es bastante más lenta, tanto en comunicaciones como en procesamiento.

Para la red de 100Mb/s los resultados siguen confirmando que la implemen-

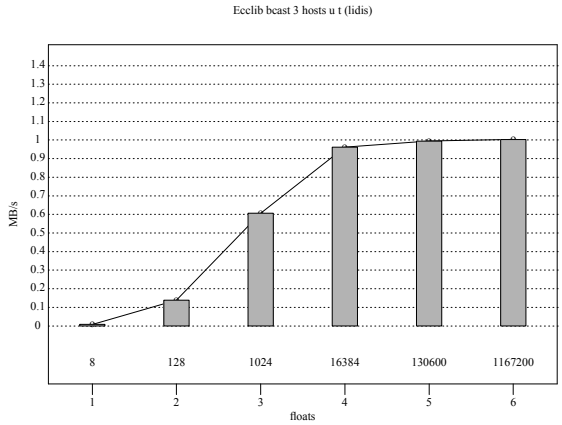

Figura 11.4: *Broadcast* implementado en el trabajo - red del LIDI 10Mb/s - 2 receptores externos

tación de ECCLib logra un rendimiento muy bueno para muchos datos. Se muestra en la figura 11.6 la comparación entre los mejores resultados conseguidos con LAM/MPI y PVM sobre el cluster 100Mb/s y los obtenidos con la implementación del trabajo.

Se ve en la figura 11.6 que ya, para 1024 floats, la implementación de este trabajo está casi a la par de la mejor para pocos datos de LAM/MPI[1], y, a partir de ese punto supera a todas. La diferencia de rendimiento entre la nueva implementación y los *middlewares* analizados aumenta conforme se aumenta la

[1] Recuerdese que LAM/MPI lo resolvía contruyendo un árbol

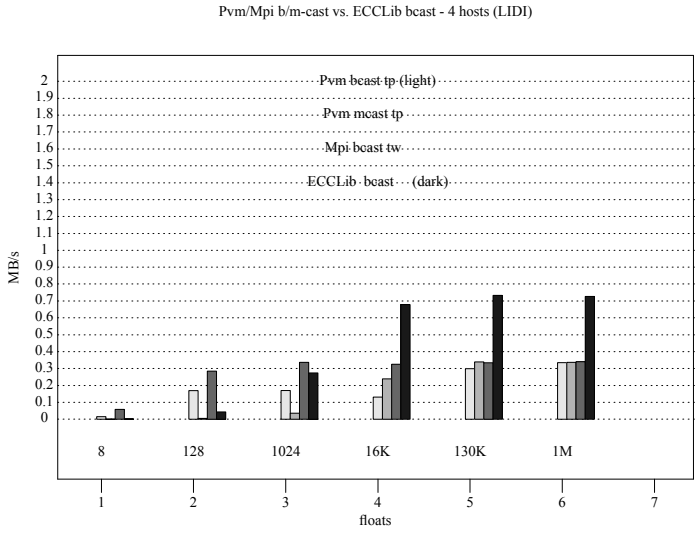

Figura 11.5: *Broadcast* implementado en el trabajo vs. *middlewares* - red del LIDI 10Mb/s - 3 receptores externos

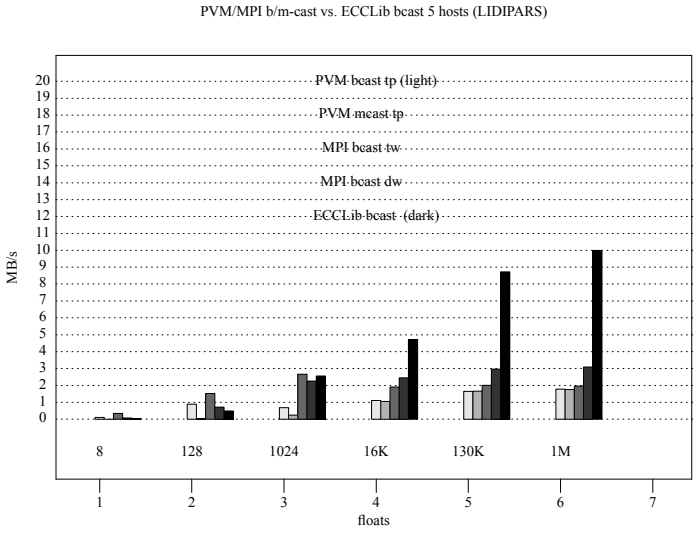

Figura 11.6: *Broadcast* implementado en el trabajo vs. *middlewares* - cluster de 100Mb/s - 4 receptores externos

cantidad de datos enviados. Otro resultado interesante que se tiene también para 100Mb/s es la gran similitud del ancho de banda logrado entre la mínima cantidad de receptores, 1, y la máxima, 7, como se muestra en las figuras 11.7 y 11.8, lo que indica escalabilidad con respecto al ancho de banda, cuestión que no sucede con los otros *middlewares*.

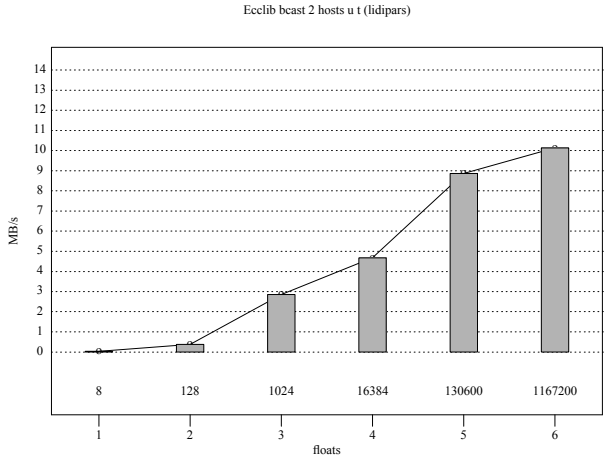

Figura 11.7: *Broadcast* implementado en el trabajo - cluster de 100Mb/s - 1 receptor externo

Como análisis final con respecto al *broadcast* de ECCLib se puede concluir que alcanza un rendimiento muy bueno para muchos datos. El inconveniente principal es el alto start-up que podría solucionarse con una implementación

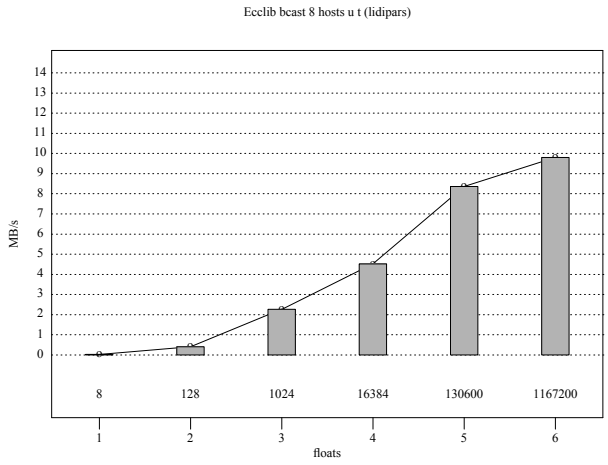

Figura 11.8: *Broadcast* implementado en el trabajo - cluster de 100Mb/s - 7 receptores externos

que, de acuerdo al volumen de datos o a la cantidad de receptores, utilice el algoritmo actual para muchos datos, u otro como los propuestos anteriormente (UDP "crudo" o TCP directo) para pocos datos.

11.2.2 Barrier

En la fig. 11.9 se compara la nueva implementación contra las provistas por PVM y LAM/MPI para 2,3,... y 7 hosts participantes ejecutando sobre el cluster de 100Mb/s. La gráfica omite el tiempo para 8 hosts debido a que los tiempos alcanzados por la nueva implementación son demasiado elevados como para entrar en el espacio asignado. Los valores alcanzados para 8 hosts son mostrados en la tabla 11.2. Como ECCLib no tiene diferentes configuraciones solo se muestra un valor.

Implementación	Peor conf.	Mejor conf.
ECCLib	———	0.005116
LAM/MPI	0.001212	0.000318
PVM	0.001149	0.001121

Tabla 11.2: Comparación de *barrier* en segundos para 8 hosts - red 100Mb/s

Los resultados obtenidos fueron buenos hasta 7 hosts pero para 8 resultaron extremadamente malos. El problema que se encontró fue que para esa cantidad de procesos (cada uno ejecutando en un host diferente) la contención por el intento de mandar la señal de llegada al proceso `root` provocaba *channel contention*. No se tiene identificada la causa concreta, la cual puede ser provocada porque: los algoritmos de *back-off* se extendían hasta esperas realmente significativas, o problemas de switching al producirse las colisiones consecutivas. Los tiempos obtenidos llegaron a ser hasta de 3 segundos y medio como se muestra en la tabla 11.3. Otro punto interesante es que en ocasiones los tiempos que

se lograban eran muy buenos (valor de la primera columna última fila de tabla 11.3).

Configuración	Mínimo	Promedio	Máximo
ECCLib a 100Mb/s (esperas forzadas)	0.001882	0.005116	0.007743
ECCLib a 10Mb/s	0.005986	0.008054	0.014808
ECCLib a 100Mb/s (sin esperas)	0.000579	0.060420	3.500182

Tabla 11.3: *Barrier* en segundos para 8 hosts con ECCLib - red 100Mb/s

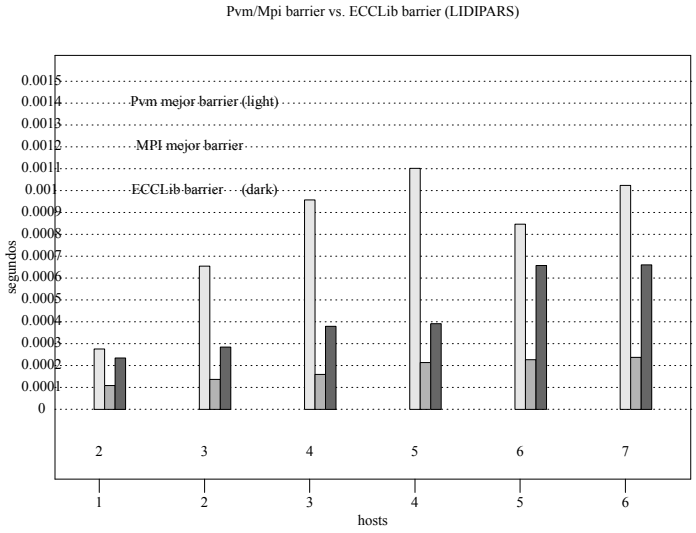

Figura 11.9: Comparación de *barrier* - cluster de 100Mb/s

Para evitar los tiempos extremadamente grandes debido a la contención por acceder al canal se forzó una espera aleatoria antes de enviar la señal de arribo en cada proceso. Para esto se encontró el siguiente problema: en algunos kernels, como por ejemplo el caso de linux-2.2.X sobre el que se ejecutarón varias pruebas, la porción mínima de tiempo es 10 milisegundos (0.0010 segundos) por lo que no se pueden alcanzar tiempos menores haciendo la espera aleatoria con llamadas al sistema como `usleep(3)`. Se resolvió implementar la espera de forma activa con un loop, logrando resultados un poco mejores comparados con los probados con `usleep(3)` pero altos comparados con los obtenidos en LAM/MPI.
En la fig. 11.10 se comparan los tiempos obtenidos sobre la red de 10Mb/s. Los valores obtenidos usando PVM solo se muestran para 2 hosts debido a que para más son demasiado grandes. En la figura se puede ver que el rendimiento alcanzado con ECCLib es el mejor para 2 y 3 hosts pero al incorporar la última máquina se ve superado por LAM/MPI.
Las conclusiones que se obtienen de las mediciones del *barrier* son que para una red con capacidad de switching conviene armar algoritmos que la aprovechen como lo es el basado en árbol de LAM/MPI o como el *butterfly*. Para redes que conforman un solo dominio del colisiones el algoritmo de que todos envían

la señal de arribo a un coordinador es bueno pero necesita algún mecanismo de evitar la contención del canal masiva si son muchos los participantes.

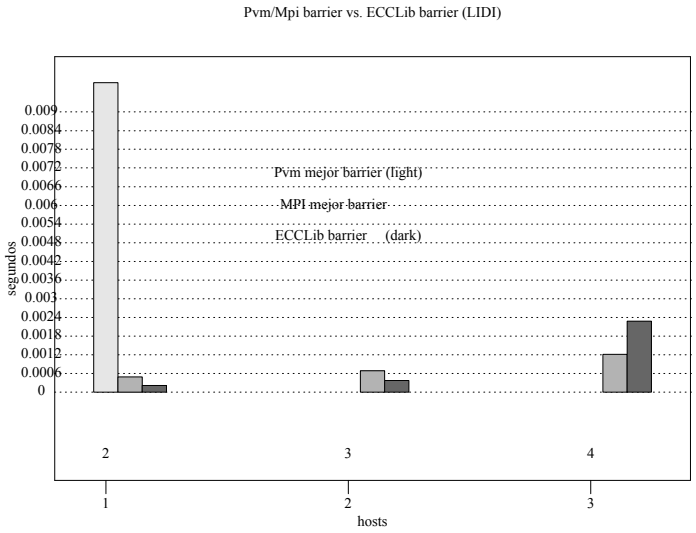

Figura 11.10: Comparación de *barrier* - red heterogénea del LIDI 10Mb/s

11.2.3 Scatter

El start-up (tabla 11.4) en la red de 10Mb/s del LIDI es mejor que el de PVM y casi como el obtenido con ruteo *daemon-to-daemon* en LAM/MPI, pero es un orden de magnitud más alto que el mejor de LAM/MPI con ruteo directo. En la red de 100Mb/s sucede algo similar.

Network	Hosts	PVM d	PVM t	MPI d	MPI t	ECCLib
LIDI (10Mb/s)	1	0.02000	0.01500	0.00100	0.0001	0.00180
LIDI (10Mb/s)	2	0.03000	0.02100	0.00100	0.0003	0.00300
LIDI (10Mb/s)	3	0.04000	0.04300	0.00600	0.0005	0.00700
LIDIpars (100Mb/s)	1..3	0.00140	0.00300	0.00020	0.00005	0.00040
LIDIpars (100Mb/s)	4..5	0.00160	0.00500	0.00060	0.00001	0.00065
LIDIpars (100Mb/s)	6	0.00260	0.01000	0.00100	0.00011	0.00090
LIDIpars (100Mb/s)	7	0.00300	0.01300	0.00100	0.00020	0.00100

Tabla 11.4: Comparación de start-up del *scatter*

Con respecto al ancho de banda se obtiene que la nueva implementación alcanza resultados aceptables figs. 11.11 y 11.12. Se puede observar en la fig. 11.11 que, por lo menos a partir de 16KFloats tiene el mismo rendimiento que la mejor configuración de LAM/MPI para esta red (ruteo directo y no usar codificación lo cual ofrece cierta ventaja sobre PVM). Una cuestión desfavorable que aparece en el gráfico es que para pocos datos tiene un rendimiento pobre. A PVM lo supera para todas las longitudes. En algunos casos los valores del

ancho de banda graficado sobrepasan al máximo de la red, 1.25MB/s, consecuencia de que uno de los receptores es local y el costo en tiempo, del envío hacia ese par es insignificante. Nótese que los mejores resultados sobre la red de 10Mb/s se obtuvieron usando ruteo directo (t), esto se debe a que son pocos los hosts participantes y en todos los casos se usan algoritmos lineales. Otra ventaja que tiene el ruteo directo es que tiene menos start-up lo que lo hace más importante en una red más lenta.
En la fig. 11.12 están los resultados sobre la red de 100Mb/s, acá, PVM con los dos modos de ruteos y a LAM/MPI también. En estas barras se ve cómo el start-up con ruteo directo (t) es inferior al logrado con ruteo default o *daemon-to-daemon* (d en LAM/MPI y u en PVM). El mejor ancho de banda se logra con LAM/MPI. Se puede ver que los resultados alcanzados con la nueva implementación son como una extrapolación de los obtenidos con 10 a 100 Mb/s. En la misma figura se puede ver cómo la configuración de ruteo directo de LAM/MPI es buena para pocos datos pero no tanto para muchos, pasando de forma inversa con el ruteo *daemon-to-daemon*.
Una buena característica de la implementación de este trabajo es que para diferente cantidad de hosts participando se mantienen similares los valores alcanzados del ancho de banda, como se muestra en las figs. 11.13 y 11.14, donde el ancho de banda está medido sin tener en cuenta la porción de datos que queda localmente, por lo cual es un tanto menor que el descripto en las figuras combinadas.

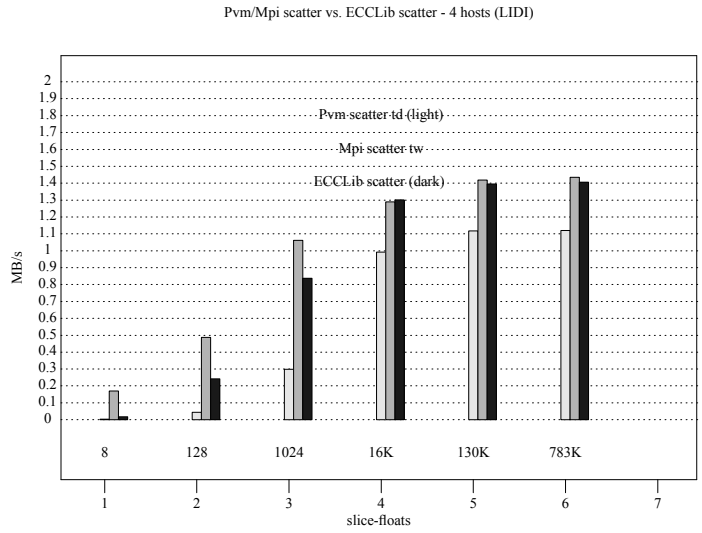

Figura 11.11: *Scatter* implementado en el trabajo vs. *middlewares* - red del LIDI 10Mb/s - 3 receptores externos

Como conclusiones de los resultados del *scatter* se puede decir: que las diferencias entre el ancho de banda de las distintas implementaciones, para muchos datos, parece tender a igualarse pero esto sucede porque no se tienen más máquinas para continuar las pruebas. Con los resultados no queda claro algo que es totalmente cierto, para obtener mejor rendimiento es necesario elaborar algoritmos más inteligentes descartando el código "naive" de los envíos consecutivos, y

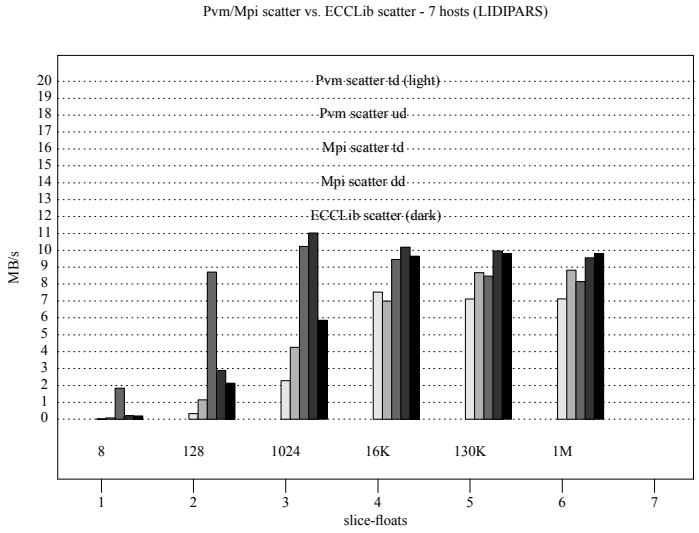

Figura 11.12: *Scatter* implementado en el trabajo vs. *middlewares* - cluster de 100Mb/s - 6 receptores externos

aprovechar la capacidad de switching, aunque según el algoritmo implementado en el trabajo en ocasiones se tiene mejor rendimiento que el algoritmo de árbol. Esto es fácilmente demostrable teóricamente: Con un algoritmo que construye arboles binarios (orden $O(log_2(n))$) y otro que es lineal (orden $O(n)$) y se esta en presencia de una red switcheada, a medida que se bajan niveles en el árbol existe más concurrencia en los envíos relación que se puede demostrar con la ecuación 11.1.

$$\lim_{n \to \infty} \frac{log_2(n)}{n} = 0 \tag{11.1}$$

Lo cual indica que a medida que se aumente la cantidad de hosts la diferencia va a ser cada vez más grande.

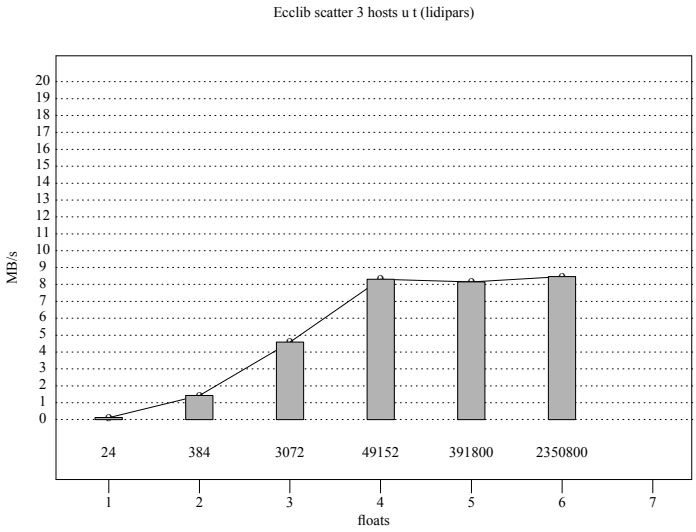

Figura 11.13: *Scatter* implementado en el trabajo - cluster de 100Mb/s - 2 recpetores externos

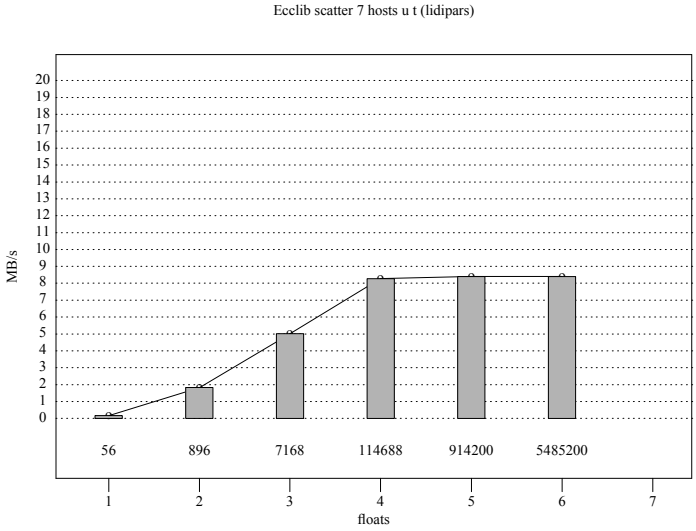

Figura 11.14: *Scatter* implementado en el trabajo - cluster de 100Mb/s - 6 receptores externos

11.2.4 Gather

El análisis para el *gather* es similar que el realizado para el *scatter*. De la misma forma se analiza el start-up y luego el ancho de banda. Para PVM, el start-up alcanzado es el mismo con ambos ruteos, y esto sucede en las dos redes. En este caso solo se tabula el default. Para LAM/MPI se ubica solo el mejor que es el alcanzado con ruteo directo. Los resultados son presentados en la tabla 11.5. En la nueva implementación se encuentra un resultado adverso con 7 hosts además del proceso `root`. De la misma forma que sucedía con el *barrier* se produce *channel contention* dando tiempos muy altos, mayores a un segundo. Para el resto de los valores se observa que los tiempos obtenidos se posicionan más cerca de los mejores tiempos (LAM/MPI) que de los peores (PVM).

Network	Hosts	PVM d	MPI t	ECCLib
LIDI (10Mb/s)	1	0.01500	0.00060	0.00149
LIDI	2	0.02100	0.00070	0.00248
LIDI	3	0.04300	0.00130	0.00669
LIDIpars (100Mb/s)	1	0.00130	0.00012	0.00032
LIDIpars	2	0.00150	0.00017	0.00045
LIDIpars	3	0.00190	0.00019	0.00055
LIDIpars	4	0.00240	0.00020	0.00080
LIDIpars	5	0.00270	0.00027	0.00082
LIDIpars	6	0.00300	0.00032	0.00102
LIDIpars	7	0.00370	0.00035	1.49915

Tabla 11.5: Comparación de start-up del *gather*

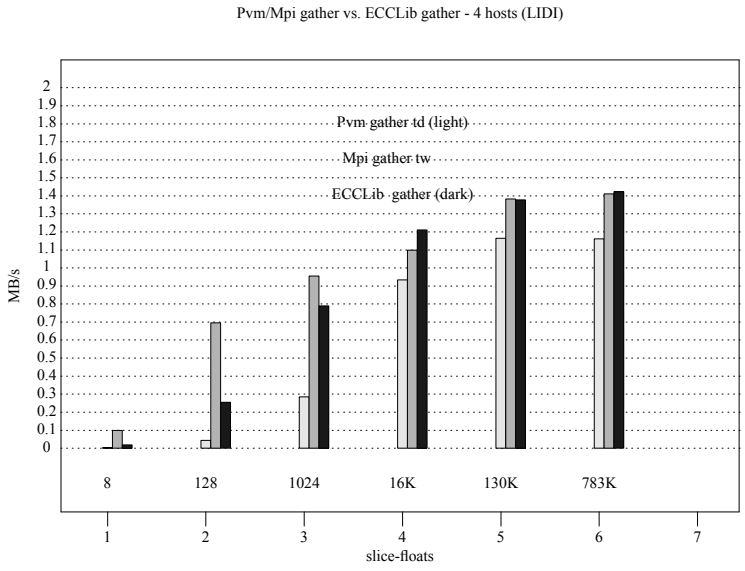

Figura 11.15: *Gather* implementado en el trabajo vs. *middlewares* - red del LIDI 10Mb/s - 3 receptores externos

Para el ancho de banda los valores graficados son los mismos que los de las configuraciones usadas para el *scatter*. Sobre la red heterogénea de 10Mb/s (fig. 11.15) la nueva implementación obtiene los mejores resultados a partir de 16KFloats. Igualmente para cantidades de datos chicas como se observó con el *gather* tiene un desempeño más bien pobre si se la compara con LAM/MPI.

En el cluster de 100Mb/s (fig. 11.16) se tienen resultados parecidos a los obtenidos con el *scatter* (fig. 11.12) aunque se pueden destacar las siguientes diferencias:

- Las diferencias entre las distintas implementaciones son más chicas.
- El máximo valor se alcanza con la máxima cantidad de datos.
- La nueva implementación alcanza un régimen de ancho de banda bueno a partir de 1024KFloats (En el *scatter* para esta cantidad de datos alcanza la mitad del ancho de banda)

La buena característica que se encontró en la nueva implementación del *gather*, es que para diferentes cantidades de hosts participando se mantienen similares los valores alcanzados del ancho de banda, lo mismo que sucedía para el *scatter* (figs. 11.17 y 11.18). Otro punto distinto encontrado es que para pocos datos las diferencias se marcan más. Con más hosts la cantidad de datos transmitidas es mayor por lo que, un valor de ancho de banda alto se alcanza con porciones más chicas.

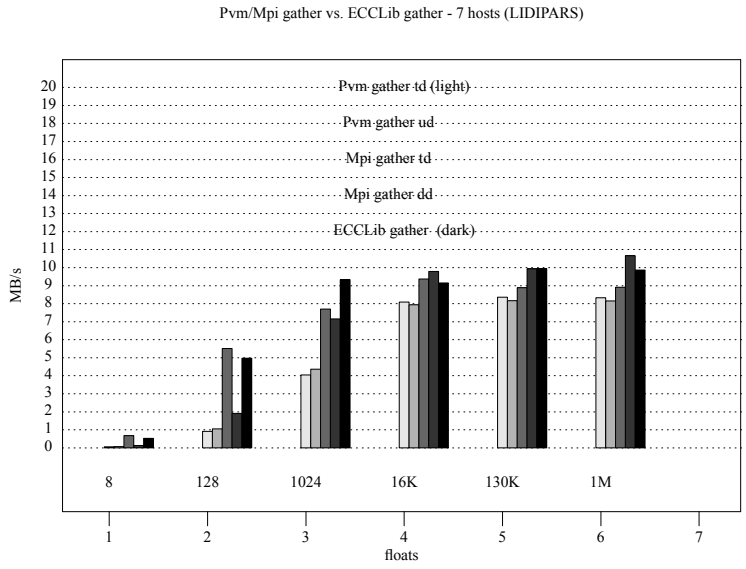

Figura 11.16: *Gather* implementado en el trabajo vs. *middlewares* - cluster de 100Mb/s - 6 receptores externos

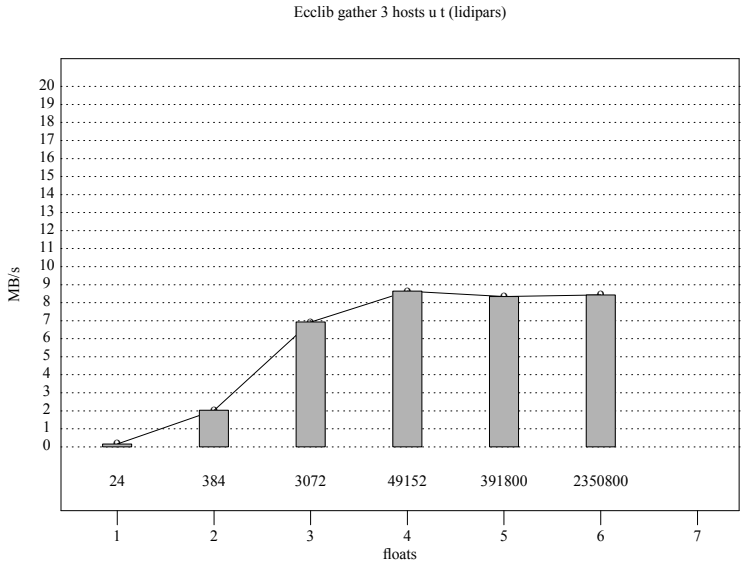

Figura 11.17: *Gather* implementado en el trabajo - cluster de 100Mb/s - 2 receptores externos

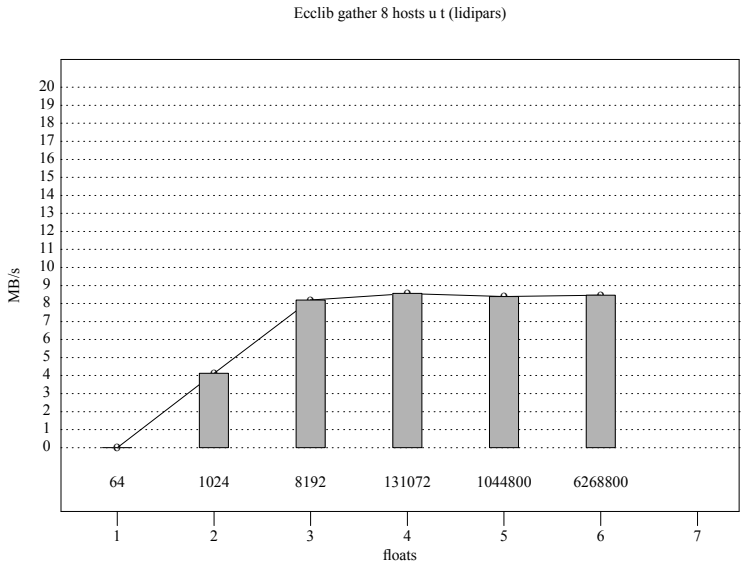

Figura 11.18: *Gather* implementado en el trabajo - cluster de 100Mb/s - 7 receptores externos

11.2.5 Reduce

En la operación *reduce* los valores que se consiguen para el start-up son los mismos que los conseguidos con el *gather* por lo que no se necesita hacer un nuevo análisis. El parámetro que sufre modificaciones es el ancho de banda debido a que se agrega el overhead del procesamiento de la operación. De cualquier forma la relación entre la velocidad de cómputo y la de comunicaciones hace la diferencia poco importante.
Para la red de 10Mb/s (fig. 11.19) el ancho de banda conseguido con la nueva implementación es muy parecido al obtenido en el *gather* (fig. 11.15) hasta 1024KFloats. A partir de este punto es que comienza a tener un peso significante el cómputo por la cantidad de datos (además se hace el cálculo todo junto en un mismo host) y produce que se obtenga una diferencia de hasta 0.2 MB/s. Para LAM/MPI el costo del procesamiento, al realizarse de forma distribuida, parece no tomar tanta relevancia. Otro punto importante a destacar del procesamiento distribuido es que no se transfieren tantos datos y los requerimientos de memoria son menores. Un caso común que ocurre con el algoritmo centralizado es la utilización de *swap* cuando la cantidad de datos es suficientemente grande, lo que enlentece significativamente el tiempo total. Para PVM el costo de las comunicaciones parece absorber el cómputo.
En el cluster de 100Mb/s (fig. 11.20) se encuentra que la implementación del autor otorga un rendimiento malo. Los desarrollos que utiliza la operación de reducción centralizada, como sucede con el ECCLib y PVM alcanzan tan solo la mitad del ancho de banda que logra un algoritmo que aprovecha la capacidad de switching y hace la reducción de forma distribuida. Al estar en presencia de una red más veloz, el tiempo de cómputo toma más relevancia además de que no se aprovecha el switching. Los resultados con LAM/MPI sobrepasan el máximo teórico debido a que la cantidad de datos transmitida es menor por las reducciones intermedias y se hacen comunicaciones solapadas a partir de la red switcheada. Como conclusión de los resultados de la operación *reduce* se menciona que es indispensable tener algoritmos que puedan sacar provecho de la arquitectura subyacente, tal como lo hace LAM/MPI.

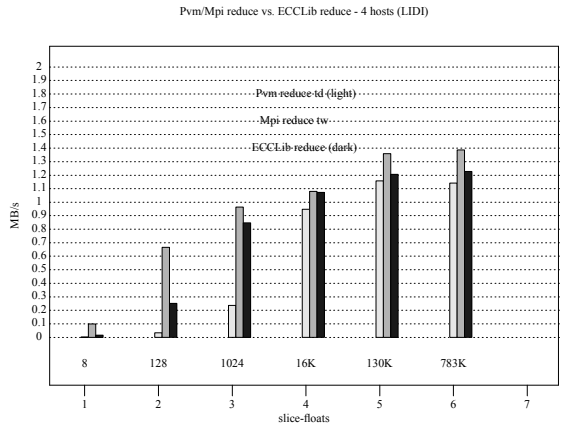

Figura 11.19: *Reduce* implementado en el trabajo vs. *middlewares* - red del LIDI 10Mb/s - 3 receptores externos

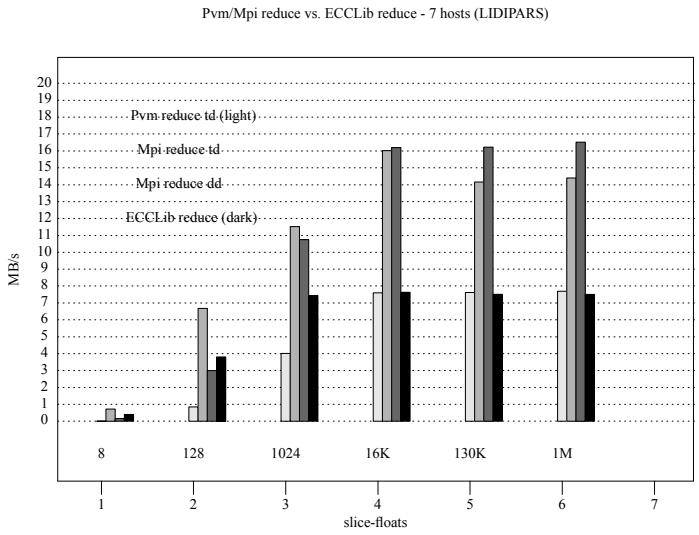

Figura 11.20: *Reduce* implementado en el trabajo vs. *middlewares* - cluster de 100Mb/s - 6 receptores externos

Capítulo 12

Conclusiones

12.1 Conclusiones Principales

- Como se suponía en un principio, ninguno de los *middlewares* analizados sacan provecho del *broadcast* provisto por Ethernet e IP, de aquí que los rendimientos alcanzados con nuestra implementación para porciones de datos considerables son los mejores. La operación de *broadcast* es una colectiva muy importante y si se quiere tener eficiencia se debe sacar el máximo provecho de la red.

- Es importante destacar que al existir diferentes topologías para implementar LANs, se debe tener en cuenta este factor al momento de desarrollar el algoritmo de la colectiva si se desea tener el mejor rendimiento. Esta conclusión se ve respaldada si se comparan los algoritmos lineales y los basados en árbol en una topología switcheada. Sin ir más lejos, Ethernet está basada en un bus lógico que implementa el *broadcast* eficientemente lo cual se debe aprovechar.

- De los dos puntos anteriores se deriva que la forma de tener operaciones rápidas es optimizándolas al hardware. Para tenerlas disponibles desde las aplicaciones se las debe incorporar a los *middlewares* existentes. La mejor forma de hacerlo es dejar éstos intactos y ofrecer un "mini-sub" conjunto de operaciones optimizadas disponibles y compatibles con lo que ya hay, y no modificar los *middlewares*.

- Otra conclusión es que: PVM parece haber cumplido ya su función como *middleware* y hoy se ve superado por nuevas implementaciones basadas en el estándar formal: MPI. Si bien es útil como herramienta didáctica para ingresar en la programación paralela basada en el paradigma de pasaje de mensajes, al momento de la eficiencia se lo debe descartar. En un área que aún parece tener presencia PVM es la del cómputo heterogéneo, campo en el cual, las implementaciones más usadas de MPI no parecen interesarse mucho. Lo que tiende a suceder en general es que los proyectos basados en multicomputadores de estaciones de trabajo se arman a partir de clusters (redes homogéneas) cuestión que facilita mucho más las cosas (desarrollo, balance de carga, administración, etc).

- Una cuestión que se confirma con este trabajo es que las comunicaciones en las redes locales aún tienen un camino que recorrer para competir con supercomputadores tradicionales. Hoy en día hay puesto un gran interés en el tema [Myrinet] [VIA].
- Por último se puede catalogar de resultado importante según los objetivos el hecho de tener un análisis a bajo nivel de como se implementan las operaciones y así poder saber cuales son los factores que influyen en los tiempos de las comunicaciones. Además de tener los resultados de las corridas reales con overhead incluido.

12.2 Resultados Secundarios

Luego de sacar las conclusiones y resultados principales el autor cree importante ver los resultados secundarios, aquellos que se derivados de este trabajo pero que no estaban contemplados dentro de los objetivos cuando fue planteado:

- Un resultado derivado es la metodología propuesta para la medición de las diferentes operaciones colectivas, que como se mencionó antes, no existe un consenso ni un modelo ampliamente aceptado.
- Otro resultado más palpable es una herramienta de medición y análisis automático del rendimiento de la red para aplicaciones paralelas [TinBar02]. Esta herramienta fue necesaria para poder obtener los resultados analizados de las múltiples corridas realizadas. Un aspecto interesante que tiene esta herramienta con respecto a otras ya existentes [DHPC104] es que sirve tanto para medir PVM y MPI, y luego poder comparar las implementaciones. Sirve para medir el ancho de banda real y la latencia (sin overhead) y para saber cual es el "techo" (valor máximo alcanzable), más allá del valor máximo teórico.

12.3 Trabajos a Futuro

Los trabajos a futuro que deja pendiente este documento son varios, sólo se enumeran los que parecen más importantes:

- Uno de los más importantes es el de la utilización de la biblioteca de operaciones colectivas en aplicaciones reales. Esto ya se está poniendo en práctica con trabajos para resolver problemas del álgebra lineal (Se las usa en cálculo de LU y multiplicación de matrices [TinLuq02] [TinDen02]).
- Más allá de que los actuales emprendimientos dejan un poco de lado la computación heterogénea, el autor cree que es un punto importante a explorar por la gran cantidad de cómputo potencial latente que ofrece (Todas las instalaciones de oficina y otras que ya existen con otros fines) [NOW94].
- Hasta ahora se han planteado problemas y algoritmos basados en LANs, pero una cuestión que tiene que ver mucho con el punto anterior es la explotación de redes WANs (como lo es Internet) y como se plantea en muchos proyectos [Grid98] [Grid01].

Bibliografía

[TinBar02] Fernando G. Tinetti, Andres Barbieri. Cómputo Paralelo en Clusters: Herramienta de Evaluación de Rendimiento de las Comunicaciones. CACIC 2002. VIII Congreso Argentino de Ciencias de la Computación. Organizado por la Universidad de Buenos Aires, del 15 al 18 de octubre de 2002.

[DHPC104] Duncan Grove, Paul Coddington. Precise MPI Performance Measurement Using MPIBench. Technical Report DHPC-104. Department of Computer Science, Adelaide Univeristy, Australia.

[Grid98] Ian Foster, Carl Kesselman. Computational Grid. Chapter 2 from The Grid: Bluprint for a Future Computing Infrastructure. Morgan Kaufmann Publishing, 1998.

[Grid01] Ian Foster, Carl Kesselman, Steven Tuecke. The Anatomy of the Grid, Enabling Scalable Virtual Organizations. Intl J. Supercomputer Applications, 2001.

[TinLuq02] Fernando G. Tinetti, Emilio Luque. Parallel Matrix Multiplication on Heterogeneous Networks of Workstations. CACIC 2002. VIII Congreso Argentino de Ciencias de la Computación. Organizado por la Universidad de Buenos Aires, del 15 al 18 de octubre de 2002.

[TinDen02] Fernando G. Tinetti, Mónica Denham. Paralelización de la Factorización de Matrices en Clusters. CACIC 2002. VIII Congreso Argentino de Ciencias de la Computación. Organizado por la Universidad de Buenos Aires, del 15 al 18 de octubre de 2002.

[Myrinet] Site at: `http://www.myri.com/myrinet/`.

[VIA] Philip Buonadonna, Andrew Geweke, David Culler. An Implementation and Analysis of the Virtual Interface Architecture. University of California, Berkeley. Berkeley CA 94720 USA. Site at: `http://www.cs.berkeley.edu/~philipb/via/`.

[NOW94] A Case for NOW (Networks of Workstations) - Thomas E. Anderson, David E. Culler, David A. Patterson, and NOW team. Dec. 9 1994.

Índice General

Índice de Figuras