Tesis 4

CASA ABIERTA AL TIEMPO

UNIVERSIDAD AUTONOMA METROPOLITANA

UNIDAD: Iztapalapa

DIVISION: CBI

CARRERA: Licenciatura en Computación.

MATERIA: Proyecto Terminal

TITULO: Apoyo a la Construcción de Fuentes Bibliograficas Sobre la Programación Paralela.

FECHA: 29 de Marzo de 1999

ALUMNO: Laura María Blanco Nieves

MATRICULA: 89223125

ASESOR: Graciela Roman Alonso.

FIRMA DEL ASESOR:

UNIVERSIDAD AUTONOMA METROPOLITANA

IZTAPALAPA

CASA ABIERTA AL TIEMPO

PROYECTO DE INVESTIGACION I

TITULO:

APOYO A LA CONSTRUCCION DE FUENTES BIBLIOGRAFICAS SOBRE LA PROGRAMACION PARALELA. (PRIMERA PARTE)

INDICE

OBJETIVOS:

1. Desarrollar un panorama general de la programación concurrente, así como algunas de sus aplicaciones.

2. Mostrar algunas de las posibles aplicaciones de la programación concurrente, así como los posibles problemas a los que se pueden presentar.

3. Mencionar los lenguajes de programación usados para la programación concurrente.

4. Dar algunos ejemplos de aplicación y su posible solución mediante la programación concurrente.

5. Presentar un panorama general de la programación en paralelo, así como algunas de sus aplicaciones.

6. Mostrar algunos de los mejores algoritmos para resolver la programación en paralelo.

7. Mencionar algunas de las diferencias existentes entre la programación paralela y concurrente.

8. Dar algunas de las aplicaciones de la programación paralela, y los posibles problemas a los que nos podemos enfrentar.

9. Explicar algunos de los lenguajes de programación usados para crear programas en paralelo, y las cualidades para usar uno u otro.

El presente trabajo pretende dar un panorama general de lo que es la programación concurrente y la programación paralela, así como algunos de los lenguajes de programación usados para cumplir dichos propósitos. Esperando que el presente trabajo sirva como apoyo para futuras investigaciones en el área de la programación en paralelo y la programación concurrente.

1. ESTUDIO DE LA PROGRAMACION CONCURRENTE

1.1 INTRODUCCION

Sabemos que las computadoras modernas pueden realizar varias cosas al mismo tiempo, es decir, pueden ejecutar a la vez un programa de usuario, puede leer de un disco e imprimir en una terminal o impresora. En los sistemas de multiprogramación, la CPU también alterna de programa en programa, ejecutando cada uno de ellos por decenas o cientos de milisegundos, lo que da una apariencia de paralelismo. A veces se habla de seudoparalelismo para indicar este rápido intercambio de los programas en la CPU, para distinguirlo del paralelismo real del hardware, donde se hacen cálculos en la CPU a la vez que operan uno o más dispositivos de E/S. Como resulta difícil mantener un registro de las distintas actividades paralelas, los diseñadores de sistemas operativos han desarrollado un modelo que facilita el uso del paralelismo.

En este modelo, todo el software ejecutable de la computadora, inclusive el sistema operativo, se organiza en varios procesos secuenciales, o en forma breve de procesos. Un proceso es tan solo un programa en ejecución, lo que incluye las valores activos del contador, registros y variables del programa. De manera conceptual, cada proceso tiene su propia CPU virtual. Por supuesto, la realidad es que la verdadera CPU alterna entre los procesos; pero para comprender el sistema, es mucho más fácil pensar en un conjunto de procesos en ejecución (seudo) paralela, que pensar en llevar un registro de la forma en que alterna la CPU de programa en programa. Esta rápida alternancia se llama multiprogramación.

La diferencia entre un proceso y un programa es sútil, pero también crucial. La idea clave es que un proceso es una actividad de cierto tipo. Tiene un programa, entrada, salida y estado. Un solo procesador puede ser compartido entre varios procesos, con cierto algoritmo de planificación, que se utiliza para determinar cuando detener el trabajo en un proceso y dar servicio a otro distinto.

Jerarquías de procesos

Los sistemas operativos que soportan el concepto de proceso deben ofrecer cierta forma de crear todos los procesos necesarios. En los sistemas demasiado sencillos, o en los sistemas diseñados para la ejecución de un solo programa, es posible que todos los procesos que podrían ser necesarios en algún momento pueden estar presentes durante la inicialización del sistema. Sin embargo en la mayoría de los sistemas, es necesaria una forma de crear y destruir procesos cuando se requiera durante la operación.

En UNIX, los procesos se crean mediante la llamada al sistema FORK, el cual crea una copia idéntica del proceso que hace la llamada. Después de la llamada FORK, el padre sigue su ejecución, en paralelo con el hijo. El padre puede dar lugar entonces a más hijos, de

profundidad arbitraria. En MS-DOS, existe una llamada al sistema que carga un archivo binario dado en la memoria y lo ejecuta como un proceso hijo. En contraste con UNIX, en MS-DOS esta llamada suspende al padre hasta que el hijo ha finalizado su ejecución, de forma que el hijo y el padre no se ejecuten en paralelo.

Después de ver que es y como se manejan los procesos en general, procederemos a analizar los procesos concurrentes que es el tema de estudio.

1.2 MANEJO DE PROCESOS CONCURRENTES

Se dice que dos procesos son concurrentes (o simultáneos) si están activos (o listos) simultáneamente (aunque esto no implique que su ejecución física sea simultánea). Podemos representar esta situación mediante la figura 1 que muestra los períodos de tiempo en que los procesos están activos.

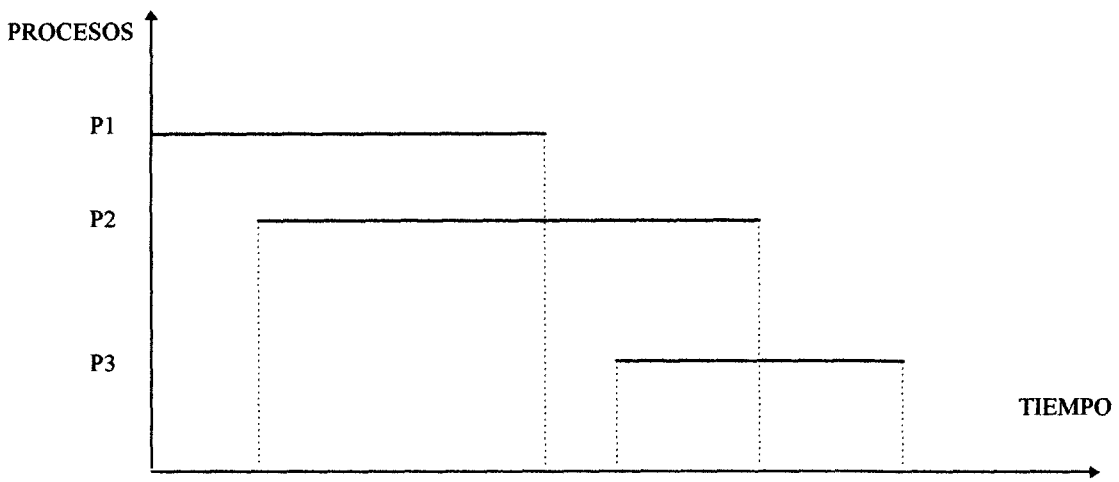


Figura1. Representaciòn de procesos concurrentes.

En el esquema anterior puede verse que los pares de procesos P1 y P2, y P2 y P3 son concurrentes, pero que P1 y P3 no lo son.

En general encontramos concurrencia de procesos siempre que hay multiprogramación. Ejemplos de procesos concurrentes pueden ser los asociados con trabajos de distintos usuarios que corren simultáneamente o procesos del sistema operativo como los descritos en la introducción.

Los procesos concurrentes pueden ser independientes si no hay ningún tipo de interacción entre ellos, o dependientes, si las hay. En el primer caso no presentan problemas de coordinación entre ellos (salvo posibles interferencias indeseadas por errores de protección

o fallas, o la disminución natural en la velocidad de ejecución de los distintos procesos por la presencia de muchos de ellos en el sistema). Como ejemplo típico de procesos concurrentes independientes podemos citar los asociados con dos usuarios distintos en un sistema clásico de multiprogramación.

Los procesos que presentas problemas son entonces los concurrentes dependientes, pues deben crearse mecanismos de comunicación y coordinación entre ellos. Existiendo así la necesidad de la comunicación de procesos; de preferencia, en una forma estructurada sin utilizar interrupciones. En lo siguiente analizaremos algunos de los aspectos relativos a la comunicación entre procesos(IPC).

1.2.1 MANEJO DE LA EXCLUSION MUTUA

El problema de la exclusión mutua se debe a que hay ciertos recursos (llamados "recursos críticos") que deben ser usados por un solo proceso a la vez, como en el caso de la lectora y la impresora, los bloques de control (BCP y tablas), las bases de datos de actualización, etc.

Así que en esta parte nos dedicaremos a examinar varios métodos para lograr la exclusión mutua, de tal forma que si un proceso está ocupado con la actualización de la memoria compartida en su región crítica, ningún otro proceso entre a su región crítica y provoque problemas.

Desactivación de interrupciones

La solución más simple es hacer que cada proceso desactivara todas sus interrupciones justo antes de entrar a la región crítica y los activara de nuevo una vez que saliera de ella. Así, una vez que un proceso ha desactivado las interrupciones, puede examinar y actualizar la memoria compartida, sin temer la intervención de otros procesos.

Este punto de vista es, en general, poco atractivo, puesto que no es correcto que los procesos del usuario tengan el poder de desactivar las interrupciones.

La conclusión es que la Desactivación de interrupciones es a veces una técnica sútil dentro del núcleo, pero no es adecuada como mecanismo general de exclusión mutua para los procesos de usuario.

Variables de Cerradura

Un segundo intento, es analizar una solución en software. Consideremos primero el caso de una sola variable de cerradura. Si se desea que un proceso entre a su región crítica, primero se hace una prueba de cerradura. Si ésta es 0, el proceso cambia el valor a 1 y entra a la región crítica. Si ésta ya vale 1, el proceso sólo se espera hasta que obtiene el valor 0 de nuevo. Así, un 0 indica que ningún proceso se encuentra en la región crítica y un 1 indica que cierto proceso está en su región crítica.

Por desgracia esta idea tiene un error fatal. Suponga que un proceso lee la cerradura y ve que tiene un valor 0. Antes de que pueda modificar su valor por 1, se planifica otro proceso,

se ejecuta o se establece el valor de la cerradura como 1. Cuando el primer proceso se vuelve a ejecutar, también establece el valor de la cerradura en 1; por ello, los dos procesos estarían es sus regiones críticas al mismo tiempo.

Alternancia estricta

Otro punto de vista del problema de exclusión mutua se muestra en la figura 2.

Este fragmento de programa está escrito en C.

```
while (TRUE)                          While(TRUE)
{                                     {
  while (turn != 0) /* wait */          while (turn != 1)  /* wait */
  critical_section();                   critical_section();
  turn = 1;                             turn = 0;
  noncritical_section();                noncritica_section();
}                                     }
```

(a) (b)

Figura 2. Una propuesta de solución al problema de la sección critica.

En la figura 2, la variable entera turn, con valor inicial 0, mantiene un registro del turno para entrar a la sección critica y examina o actualiza la memoria compartida. En principio, el proceso 0 analiza turn, determina que su valor es 0 y entra a su región critica. El proceso 1 también determina que es 0, por lo que espera en un ciclo, preguntando en forma continua el valor de turn para ver si toma el valor 1. La prueba continua de una variable, en espera de que aparezca cierto valor se llama espera ocupada. La espera ocupada sólo debe utilizarse cuando existe una esperanza razonable de que la espera será corta. Los turnos no son una buena idea cuando uno de los procesos es mucho más lento que el otro. De hecho, esta solución requiere que los dos procesos alternen en forma estricta su entrada a sus regiones críticas respectivas. Ninguno debe entrar dos veces seguidas. Aunque este algoritmo elimina todos los conflictos, no es un candidato serio como solución.

Solución de Peterson

El algoritmo de Peterson se muestra en la figura 3. Este algoritmo consta de 2 procedimientos escritos en ANSI C, lo que indica que hay que proporcionar prototipos de función para todas las funciones definidas y utilizadas. Es convencional la agrupación de dichos prototipos en archivos de encabezado. Antes de utilizar las variables compartidas (es decir, antes de entrar a su región crítica), cada proceso llama a enter_region con su propio número de procesos, 0 o 1, como parámetro. Esta llamada provoca una espera, en caso necesario, hasta que pueda entrar. Después de terminar con las variables compartidas, el proceso llama a leave_region para indicar que ha terminado y permitir la entrada de otro proceso, si así lo desea.

```
#include "prototypes.h"
#define FALSE  0
#define TRUE   1
#define N      2                        /* número de procesos */

int turn;                               /* ¿ de quién es el turno ?*/
int interested [N]                      /* todos los valores iniciales son 0 (FALSE)*/

void enter_region(int process)          /* proceso: quién entra (0 o 1)*/
{
 int other;                             /* número de los otros procesos */

 other = 1 process                      /* el opuesto del proceso */
 interested[process] = TRUE;            /*muestra que usted está interesado */
 turn = process;                        /* establece bandera */
 while (turn == process && interested[other] == TRUE)          /*enunciado null */
 ;
}

void leave_region (int process)         /* proceso: quien sale (0 o 1) */
{
  interested[process] = TRUE;           /* indica salida de la región crítica */
 }
```

Figura 3. Solución de Peterson para lograr la exclusión mutua.

Veamos como funciona esta solución. En principio, ningún proceso está en su región crítica. Ahora, el proceso 0 llama a enter_region. Indica su interés por determinar el elemento de su arreglo y hace turn=0. Puesto que el proceso 1 no está interesado, enter_region regresa en forma inmediata. Si el proceso 1 llama ahora a enter_region, esperará hasta que interested[0]=FALSE, evento que solo ocurre cuando el proceso 0 llama a leave_region.

Consideremos ahora el caso en que ambos procesos llaman a enter_region de forma simultánea. Ambos almacenarán su número de proceso en turn. Sólo cuenta la última operación; la primera se pierde. Supongamos que el proceso 1 almacena el número en el último lugar, por lo que turn=1.Cuandos ambos procesos lleguen al enunciado while, el proceso 0 se ejecuta 0 veces y entra a su región crítica. El proceso 1 hace un ciclo y no entra a su región crítica.

La instrucción TSL

Analizaremos ahora una propuesta que requiere poca ayuda del hardware. Muchas computadoras, en particular aquellas diseñadas teniendo en mente varios procesadores, tienen una instrucción TEST AND SET LOCK (TSL) que funciona como sigue. Lee el contenido de una palabra de memoria en un registro para después almacenar un valor distinto de 0 en esa dirección de memoria. Las operaciones de lectura y almacenamiento de la palabra tienen la garantía de ser indivisibles: ninguno de los demás procesadores puede tener acceso a la palabra hasta terminar la instrucción la CPU que ejecuta la instrucción TSL cierra el bus de memoria para prohibir a las demás CPU el acceso a la memoria hasta terminar. Para utilizar la instrucción TSL se emplea una variable compartida, flag la cual coordina el acceso a la memoria compartida. Cuando flag=0, cualquier proceso puede darle el valor de 1 mediante la instrucción TSL y después leer o escribir en la memoria compartida. Al teminar el proceso vuelve a ser flag=0 mediante una instrucción normal move .

¿Como puede utilizarse esa instrucción para evitar que dos procesos entren en forma simultánea a sus regiones críticas? La solución aparece en la figura 4. En elle se muestra una subrutina de 4 instrucciones en un lenguaje ensamblador ficticio (pero típico). La primera instrucción copia el valor anterior del flag en un registro y después hace flag=1. Después, el valor anterior se compara con 0. Si es distinto de 0, la cerradura ya estaba establecida, por lo que el programa regresa al principio y realiza de nuevo la prueba. Tarde o temprano, valdrá 0 (cuando salga de su región crítica) el proceso que se encuentra en ese momento en dicha región) y la subrutina regresa con la cerradura establecida. La eliminación de la cerradura es sencilla: el programa solo tiene que almacenar un 0 en flag. No se requieren instrucciones especiales.

Ahora se dispone de una solución directa al problema de la sección crítica. Antes de entrar a su sección crítica, un proceso llama a enter_region, que hace una espera ocupada hasta que se elimina la cerradura para entonces retomar la cerradura y regresar. Al salir de la sección critica el proceso llama a leave_region, que almacena 0 en flag. Como con todas las soluciones basadas en las regiones críticas, los procesos deben llamar a enter_region y leave_region en los instantes correctos para que el método funcione. Si un proceso hace algún engaño, la exclusión mutua faltara.

```
enter_region:
    tsl register, flag      copia flag al registro y hace flag=1
    cmp register,#0         ¿flag=0?
    jnz enter_region;       si era distinta de cero, la cerradura estaba
                            establecida, por lo que hay que hacer un ciclo.
    ret                     regresa a quien hizo la llamada; entra a la
                            región critica.
```

```
leave_región:
    mov flag,#0          almacena un cero en flag
    ret                  regresa a quien hizo la llamada
```

Figura 4. Establecimiento y eliminación de cerraduras mediante TSL.

1.3 DORMIR Y DESPERTAR

Tanto la solución de Peterson como la solución mediante TSL son correctas, pero ambas tienen el inconveniente de la necesidad de la espera ocupada. En esencia, lo que hacen estas soluciones es lo siguiente: cuando un proceso desea entrar a su sección crítica, verifica si esta permitida la entrada. Si no, el procesador se queda esperando hasta obtener el permiso.

Este punto de vista no sólo desperdicia el tiempo de la CPU, sino que también tiene efectos inesperados. Consideramos a una computadora con dos procesos H con máxima prioridad y L, con menor prioridad. Las reglas de planificación son tales que H se ejecuta siempre que está el estado listo. En cierto momento, con L en su región crítica, H está listo para su ejecución (por ejemplo, termina una operación de E/S).

H comienza una espera ocupada, pero como L nunca es planificada cuando H está en ejecución, L nunca tiene oportunidad de salir de su región critica, por lo que H hace un ciclo infinito. Está situación se denomina a veces como el problema de inversión de prioridad. Analicemos ahora ciertas primitivas de comunicación entre procesos que bloquean a la CPU en vez de desperdiciar su tiempo cuando no se les permite entrara sus secciones críticas. Las mas sencillas son dormir(SLEEP) y despertar(WAKEUP).SLEEP es una llamada al sistema que provoca el bloqueo de quien hizo la llamada, es decir, que se ha suspendido hasta que otro proceso lo despierte. La llamada WAKEUP tiene un parámetro, el proceso para despertar. Otra alternativa es que tanto SLEEP como WAKEUP tengan un parámetro, una dirección de memoria utilizada para que concuerden ambas llamadas.

El problema del Productor y el Consumidor

Como el ejemplo del uso de estas primitivas, consideremos el problema del productor y el consumidor (también conocido como problema de almacén limitado). Dos procesos comparten un almacén (buffer) de tamaño fijo. Uno de ellos, el productor, coloca información en el almacén (buffer), mientras que el otro el consumidor, la obtiene de él. El problema surge cuando el productor desea colocar un nuevo elemento en el almacén(buffer), pero este está totalmente ocupado. La solución para el productor es irse a dormir, para ser despertado cuando consumidor ha eliminado uno o más elementos. En forma análoga, si el consumidor desea eliminar un elemento del almacén (buffer) y ve que

este esta vacío, se va a dormir hasta que el productor coloca algo en el almacén (buffer) y despierta.

El código de productor y consumidor aparece en la figura 5.

```
#include "prototypes.h"

#define N 100                       /* número de espacios en el almacén (buffer) */
int count=0;                        /* número de elementos en el almacén (buffer) */


void producer(void)
{
  int item;

    while (TRUE)                    /* se repite eternamente */
  {
    produce_item(&item);            /* se genera el siguiente elemento */
    if (count == N)  sleep();       /* si el almacén (buffer) está lleno, va a dormir */
    enter_item(item);               /* colocar elemento en el almacén (buffer) */
    count = count + 1;              /* incrementa el contador de elementos en el almacén */
                                    /* (buffer) */
     if (count == 1) wakeup(consumer);      /*¿Estaba vacío el almacén (buffer) */
    }
  }


void consumer (void)
{
  int item;

  while (TRUE)                      /* se repite para siempre */
  {
    if (count == 0)  sleep();       /* si el almacén (buffer) está vacío, va a dormir */
    remove_item(&item);             /* retira el elemento del almacén (buffer) */
    count = count - 1;              /* decrementa el contador de elementos en el almacén */
                                    / * (buffer) */
     if (count == N) wakeup(producer);      /*¿Estaba lleno el almacén (buffer) */
     consume_item(item);            /* imprime elemento */
    }
  }
```

Figura 5. El problema del productor y el consumidor con una condición de competencia fatal.

Para poder expresar las llamadas al sistema tales como SLEEP Y WAKEUP en C, las exhibiremos como llamadas a rutinas de una biblioteca. No son parte de la biblioteca usual de C, pero podrían estar disponibles en cualquier sistema que contara con dichas llamadas al sistema. Los procedimientos enter_item y remove_item que no se muestran, controlan las operaciones de colocación y eliminación de elementos en el almacén (buffer).

La esencia del problema es que un despertar enviado a un proceso que todavía no este durmiendo se pierde. Si esto no ocurriese, todo iría bien. Un arreglo rápido consiste en modificar las reglas con el fin de añadir un bit de espera de despertar en este marco. Cuando se intente despertar a un proceso ya despierto, el bit se activa. Mas tarde, cuando el proceso intente ir a dormir, si el bit esta activo, será desactivado, aunque el proceso siga despierto. Este bit de espera es como una alcancía de las señales para despertar.

Aunque el bit de espera funcione en este sencillo ejemplo, es fácil construir ejemplos con 3 o más procesos en los que un solo bit no es suficiente. Podría hacer otro parche y añadir un segundo bit de espera o tal vez 8 0 32 de ellos, pero en principio el problema sigue ahí.

1.4 SEMAFOROS

Está era la situación cuando, en 1965, E.W Dijkstra(1965) sugirió el uso de una variable entera para contar el numero de despertares almacenados para su uso posterior.

En su propuesta se presento un nuevo tipo de variable, llamada: semáforo. Un semáforo no puede tener valor cero, lo que indica que no existen despertares almacenados; o bien algún valor positivo si están pendientes uno o más despertares. Dijkstra propuso 2 operaciones DOWN y UP (generalizaciones de SLEEP y WAKEUP, respectivamente). La operación DOWN verifica si el valor de un semáforo es mayor que 0. En ese caso, decrementa el valor (es decir, utiliza un despertar almacenado) y continua. Si el valor es 0, el proceso se va a dormir. La verificación y modificación del valor, así como la posibilidad de irse a dormir se realiza en conjunto, como una sola e indivisible acción atómica. Se garantiza que al iniciar una operación con un semáforo, ningún otro proceso puede tener acceso al semáforo hasta que la operación termine o se bloquee. Esta atomicidad es absolutamente esencial para resolver los problemas de sincronización y evitar condiciones de competencia.

La operación UP incrementa el valor del semáforo correspondiente. Si uno a más procesos dormían en ese semáforo y no podían completar una operación DOWN anterior, el sistema elige alguno de ellos(por ejemplo, en forma aleatoria) y se le permite terminar DOWN. Así, después de un UP en un semáforo con procesos durmiendo, el semáforo seguirá con valor 0, pero habrá un menor número de procesos durmiendo. La operación de incremento del semáforo y despertar un proceso también es indivisible. Ningún proceso llega a bloquear mediante un UP, así como ningún proceso establecía un bloqueo con WAKEUP del modelo anterior.

En forma colateral, en el artículo original de Dijkstra. El utilizó los nombres P y V para DOWN y UP, respectivamente(la figura 6, muestra los algoritmos para P y V); pero como estos nombres son difíciles de recordar para los que no hablamos holandés(o solo tiene un significado marginal para aquellos que si lo hablan), utilizaremos en vez de ellos los nombres UP y DOWN. Estos se presentaron por primera vez en algol 68.

```
PROCEDIMIENTO P (SEM)
 Entero.SEM ← Entero.SEM - 1
 Si (Entero.SEM < 0) ENTONCES
   Proceso p ⇒ Cola.SEM
   Bloquee proceso p
 FINSI
FIN-P

PROCEDIMIENTO V(SEM)
  Entero.SEM ← Entero.SEM + 1
  Si (Entero.SEM <= 0) ENTONCES
     q⇐ Cola.SEM
      Active proceso q
  FINSI
FIN V
```

Figura 6. Algoritmo de Dijkstra para P y V (Sustituidos después por UP y DOWN).

En los algoritmos anteriores p y q se refieren a los procesos que ejecutan la primitiva.

Solución del Problema del productor y el consumidor mediante semáforos.

Los semáforos resuelven el problema del despertar perdido, como lo muestra la figura 7. Es esencial que sean implantados en forma indivisible. La forma normal consiste en implantar UP y DOWN como llamadas al sistema, haciendo que el sistema operativo desactive por un momento todos los interruptores mientras hace una prueba del semáforo. Está solución utiliza tres semáforos, uno de los cuales es ***full***, que cuenta el número de entradas ocupadas; otro llamado ***empty*** para el conteo de las entradas vacías y otro llamado ***mutex*** que sirve para garantizar que el productor y el consumidor no tienen acceso simultáneo al almacén (buffer). El valor inicial de ***full*** es 0, el de ***empty*** es igual al número de entradas en el almacén (buffer) y el de ***mutex*** es 1. Los semáforos que se inicializan con un valor de 1 y que son utilizados por dos o más procesos para garantizar que sólo uno de ellos pueda entrar a su región crítica se llaman **semáforos binarios.** Si cada proceso realiza un DOWN justo antes de entrar a su región crítica y un UP antes de salir de ella, queda garantizada la exclusión mutua.

En el ejemplo de la figura 7, realmente utilizamos los semáforos en dos formas. Esta diferencia es importante, lo suficiente como para hacerlo explícito. El semáforo ***mutex*** se

usa para la exclusión mutua. Está diseñado para garantizar que sólo un proceso a la vez lee o escribe en el almacén (buffer) y las variables asociadas. Esta exclusión mutua es necesaria para evitar el caos.

```
#include "prototypes.h"

#define N  100                  /* Número de entradas en el almacén (buffer) */

typedef  int  semaphore;        /* Los semáforos son un tipo particular de int */

semaphore  mutex = 1;           /* Controla el acceso a la región crítica */
semaphore  empty = N;           /* Cuenta las entradas vacías del almacén (buffer)*/
semaphore  full = 0;            /* Cuenta los espacios ocupados en el almacén (buffer)*/


void producer (void)
{
   int item;

   while (TRUE)                 /* TRUE  es la constante 1 */
   {
      produce _item(&item);     /* genera algo por colocar en el almacén (buffer) */
      down(&empty);             /* decrementa el contador de espacios vacíos */
      down(&mutex);             /* entra a la región crítica */
      enter_item(item);         /* coloca un nuevo elemento en el almacén (buffer)*/
      up(&mutex);               /* sale de la región crítica */
      up(&full);                /* incrementa el contador de entradas ocupadas */
    }
}


void consumer (void)
{
   int item;

   while (TRUE)                 /* ciclo infinito */
   {
     down(&full);               /* decrementa el contador de entradas ocupadas */
     down(&mutex);              /* entra a la región crítica */

     remove_item(item);             /* toma un elemento del almacén (buffer)*/
```

```
        up(&mutex);                 /* sale de la región crítica */
        up(&empty);                 /* incrementa el contador de entradas vacías */
        consume_item(item);         /* hace algo con el elemento */
    }
}
```

Figura 7. El problema del productor y el consumidor, con uso de semáforos.

Otro de los usos de los semáforos es la **sincronización.** Los semáforos *full* y *empty* son necesarios para garantizar la ocurrencia o no de ciertas secuencias de eventos. En este caso, garantizan que el productor detiene su ejecución cuando el buffer est. completo, mientras que el consumidor se detiene cuando el buffer est. vacío. Este uso es distinto de la exclusión mutua.

1.5 CONTADORES DE EVENTOS.

La solución al problema del productor y consumidor mediante semáforos se basa en la exclusión mutua para evitar las condiciones de competencia. También es posible programar la solución sin que se requiera la exclusión mutua. En esta sección describiremos uno de esos métodos. Utiliza un tipo particular de variable, llamada **contador de eventos** (Redd y Kanodia,1979).

Se definen tres operaciones obre un contador de eventos E:

1. *Read* (E): Regresa el valor actual de E.
2. *Advance* (E): Incrementa en forma atómica el valor de E en 1.
3. *Await* (e, v): Espera hasta que E tenga el valor de v o mayor.

El problema del productor y el consumidor no utiliza READ, pero es necesario para otros problemas de sincronización.

Observe que los contadores de eventos sólo aumentan, nunca disminuyen. Siempre inician en 0. La figura 8 muestra el problema del productor y el consumidor, una vez más, pero utilizando ahora los contadores de eventos.

```
#include "prototypes.h"

#define N 100                       /* número de entradas en el almacén (buffer)*/

typedef int event_counter;          /* event_counter son un tipo especial de int */

event_counter in = 0;               /* cuenta los elementos insertados en el almacén (buffer) */
event_counter out = 0;              /* cuenta los elementos retirados del almacén (buffer) */
```

```
void producer (void)
{
  int item, sequence = 0;
  while ( TRUE )                /* ciclo infinito */
  {
    produce _item (&item);      /* genera algo que colocar en el almacén (buffer) */
    sequence = sequence +1;     /* cuanta los elementos producidos hasta este momento */
    await (out , sequence N);   /* espera a que haya lugar en el almacén */
    enter_item (item);          /* pone el elemento en la entrada  (sequence 1)%N */
    advance(&in);               /* deja que el consumidor conozca algún otro elemento */
  }
}


void consumer (void)
{
 int item, sequence = 0;
  while ( TRUE )                /* ciclo infinito */
  {
    int item, sequence = 0;
    while ( TRUE )              /* ciclo infinito */
  {
    sequence = sequence +1;     /* # de elementos a eliminar del almacén (buffer) */
    await (in , sequence );     /* espera a que este presente el elemento del espacio */
    remove_item (&item);        /* retira el elemento de la entrada  (sequence 1)%N */
    advance (&out);             /* deja que el productor se entere de que el elemento ha */
                                /* sido retirado */
    consume_item (item);        /* hace algo con el elemento */
  }
}
```

Figura 8. El problema del productor y el consumidor, con el uso de contadores de evento.

1.6 MONITORES.

¿ Parece fácil la comunicación entre procesos con los semáforos y los contadores de eventos? Observe en la figura 7 los DOWN que aparecen antes de introducir o retirar elementos del almacén (buffer). Suponga que ambos DOWN del código del productor se invirtieran, de forma que *mutex* se decremente antes y no después de *empty*. Si el almacén (buffer) estuviera completamente ocupado, el productor se bloquearía y *mutex* tomaría el valor 0. En consecuencia, la vez siguiente en que el consumidor intentara el acceso al almacén (buffer), ejecutaría un DOWN en *mutex* = 0 y también se bloquearía. Ambos procesos se bloquearían para siempre y no se podría seguir trabajando. Esta desafortunada situación se conoce como **bloqueo (deadlock).**

Señalamos este problema para mostrar el cuidado que se debe tener al utilizar semáforos. Un pequeño error y todo se vendría abajo. Es como la programación el lenguaje ensamblador, pero es peor, puesto que los errores son condiciones de competencia, bloqueos y otras formas de comportamiento impredecible e irreproducible.

Para facilitar la escritura de programas correctos, Hoare (1974) y Brinch Hansen (1975) propusieron una primitiva de sincronización de alto nivel, llamada **monitor.** Sus propuestas tenían ligeras diferencias, las cuales señalaremos a continuación. Un monitor es un mecanismo que sirve para manejar procesos concurrentes y para representar distintas condiciones de sincronización y exclusión mutua. Está compuesto por un conjunto de datos, un conjunto de procedimientos para manejarlos y ciertas variables de condición, que permiten representar diferentes condiciones de sincronización. La siguiente figura muestra los distintos componentes de un monitor.

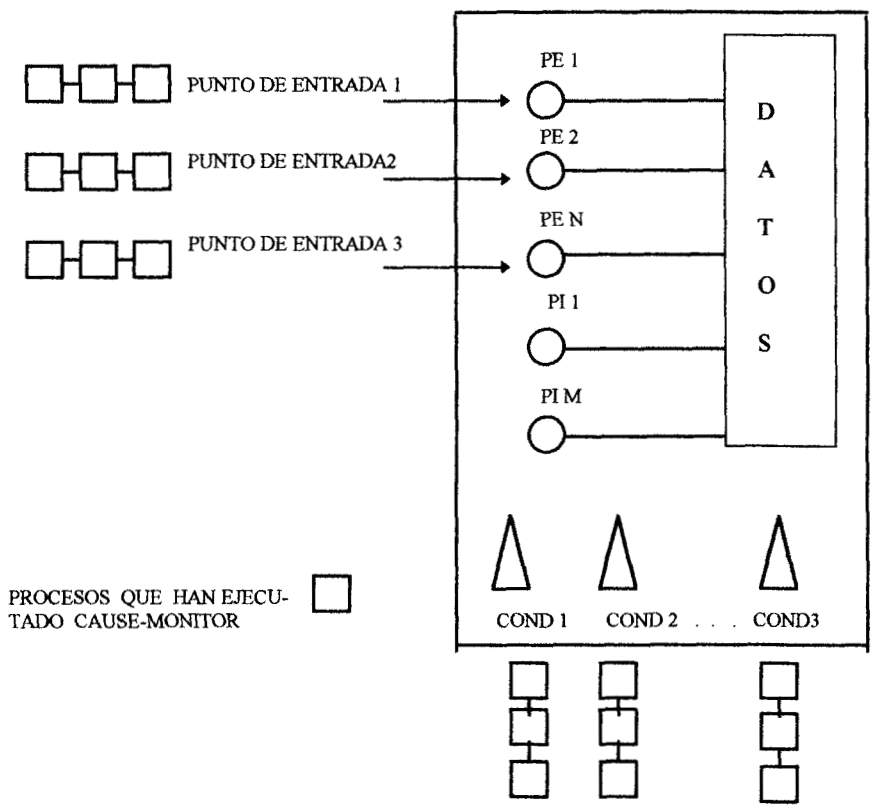


Figura 9. Representación de la estructura de un monitor.

El acceso a los datos sólo puede hacerse a través de ciertos procedimientos predefinidos del monitor (que podemos asimilar a puntos de entrada). Esto garantiza que se haga debidamente.

Los procedimientos pueden ser internos o externos. Los primeros son locales y no se puede tener acceso a ellos desde afuera. Los segundos representan los distintos puntos de entrada del monitor. Por definición, los procedimientos se ejecutan en exclusión mutua. Asociada a cada punto de entrada hay una cola de procesos que quieren tener acceso al monitor a través de éste (y que no pueden entrar por la condición de exclusión mutua).

Los monitores tiene una propiedad importante que los hace útiles para conseguir la exclusión mutua: sólo uno de los procesos puede estar activo en un monitor en cada momento. Los monitores son construcción del lenguaje de programación, por lo que el compilador sabe que son especiales y puede controlar las llamadas a los procesos del monitor en forma distinta a las llamadas de los demás procedimientos. Por lo general, cuando un proceso llama a un procedimiento de monitor, las primeras instrucciones de éste verificàn si hay otro proceso activo dentro del monitor. En caso afirmativo, el proceso que hace la llamada será suspendido hasta que el otro proceso salga del monitor. Si no hay otro proceso que esté utilizando el monitor, el que hace la llamada podrá entrar.

Aunque ya hemos visto que los monitores son una forma sencilla de lograr la exclusiòn mutua, no son suficientes. También se requiere de una vía para bloquear los procesos cuando éstos no procedan. En el problema del productor y el consumidor, basta colocar todas las pruebas de buffer vacìo o buffer completamente ocupado en los procedimientos del monitor, pero ¿ cómo se bloquearía el productor si sabe que el buffer está completamente ocupado?

La solución reside en la introducción de **variables de condición,** junto con dos operaciones sobre ellas, WAIT y SIGNAL. Si un procedimiento de monitor descubre que no puede continuar (por ejemplo, cuando el productor encuentra el almacén (buffer) completamente ocupado), ejecuta un WAIT en alguna variable de condición, como por ejemplo, *full*. Esto hace que el proceso que hizo la llamada se bloquee. También permite que otro proceso que tenía prohibida la entrada al monitor pueda hacerlo ahora.

Las variables de condición no son contadores. No acumulan señales para su uso posterior, como lo hacen los semáforos. Así, si una variable de condición queda señalada sin que nadie la espere, la señal se pierde. WAIT debe aparecer antes de SIGNAL. Está regla hace más sencilla la implantación. En la práctica, esto no representa un problema, puesto que es fácil mantener un registro del estado de cada proceso con las variables, en caso necesario. Un proceso que podría ejecutar un SIGNAL puede ver si esta operación es innecesaria, analizando las variables.

En la figura 10 se muestra un esqueleto en semi-Pascal del problema del productor y el consumidor con monitores.

```
Monitor ProcedureConsumer
  condition full, empty;
  integer count;
  procedure enter;
  begin
    if count = N the wait (full);
    enter_item;
    count : = count +1;
    if count = 1 the signal (empty);
  end;

procedure remove;
begin
    if count = 0 the wait (empty);
    remove_item;
    count : = count -1;
    if count = N - 1 the signal (empty);
 end;
count :=0;
end monitor;

procedure producer;
begin
    while true do
    begin
     produce_item;
     ProducerConsumer.enter;
    end
end;

procedure consumer;
begin
    while true do
     begin
      ProducerConsumer.remove;
      consume_item;
    end
end:
```

Figura 10. Un bosquejo completo del problema del productor y consumidor con monitores. El almacèn (buffer) tiene N entradas.

Uno de los problema de los monitores y de los semáforos, es que se diseñaron para resolver el problema de la exclusión mutua en una o varias CPU que tienen acceso a una memoria común. Si los semáforos (o incluso los contadores de eventos) se colocan en la memoria compartida y se protegen con instrucciones TSL, se puede evitar la competencia. Cuando se pasa al caso de un sistema distribuido con varias CPU, cada una con su memoria privada, unidas mediante una red de área local, estas primitivas ya no se pueden aplicar. La conclusión es que los semáforos también son de un nivel muy bajo y que los monitores no se pueden utilizar más que en pocos lenguajes de programación. Además, ninguna de estas primitivas sirve para el intercambio de información entre computadoras. Se necesita algo más.

1.7 TRANSFERENCIA DE MENSAJES

Ese "algo más" es la **tarnsferencia de mensajes.** Este método de comunicación entre procesos utiliza dos primitivas, SEND y RECEIVE , las cuales, al igual que los semáforos y a diferencia de los monitores, son llamadas al sistema en vez de comandos del lenguaje. Como tales, se pueden colocar con facilidad en procedimientos de biblioteca, como por ejemplo,

```
send (destination, &message);
y
receive (source, &message);
```

El primer procedimiento envìa un mensaje a un destino dado y el segundo recibe un mensaje desde cierto origen (o desde cualquier origen, si el receptor no le importa este punto). Si no hay mensajes disponibles, el receptor se puede bloquear hasta que llegue alguno.

Aspectos del diseño de sistemas con transferencia de mensajes

Los sistemas con transferencia de mensajes pueden tener grandes retos y aspectos de diseño que no surgen en los casos de semáforos o monitores, en particular si los procesos en comunicación se encuentran en máquinas distintas unidades mediante una red. Por ejemplo, los mensajes se pueden perder en la red. Como protección contra la perdida de mensajes, el emisor y el receptor pueden convenir en que tan pronto se reciba el mensaje, el receptor envìe un mensaje especial de **confirmación.** Si el emisor no ha recibido una confirmación después de cierto tiempo, retransmite el mensaje.

Si el número de máquinas es muy grande y no existe una autoridad central que establezca los nombres de las máquinas, podrìa ocurrir que dos máquinas tuvieran el mismo nombre. La posibilidad de conflictos se puede reducir en forma considerable al agrupar las máquinas en **dominios** y dirigirse a los procesos como *proceso@máquina.dominio.* En este esquema,

no hay problema si dos máquinas tiene el mismo nombre, siempre que estén en dominios diferentes. Por su puesto, el nombre de los dominios también debe ser único.

La **autentificaciòn** también es otro aspecto de los sistemas con mensajes:¿ Cómo puede indicar el cliente si se comunica con su verdadero servidor y no con un impostor? ¿Cómo puede determinar el servidor cuál cliente solicitó un archivo dado? En este punto, puede ser de utilidad el ciframiento de mensajes con una clave que sólo conozcan los usuarios autorizados.

El problema del productor y el consumidor con transferencia de mensajes.

Veamos como se puede resolver el problema del productor y el consumidor mediante la transferencia de mensajes, sin memoria compartida. Una solución aparece en la figura 11.

Existen muchas variantes para la transferencia de mensajes. Para los principiantes, veamos cómo establecer las direcciones de los mensajes. Una forma consiste en asignar a cada proceso una dirección única y que las direcciones de los mensajes están ligadas a los procesos. Otra alternativa es inventar una nueva esrustura de datos, llamada **buzón.** Un buzón es un lugar donde almacenar un número determinado de mensajes, los cuales se especifican al crear el buzón. Al utilizar los buzones, los parámetros de dirección en las llamadas SEND y RECEIVE son buzones, no procesos. Cuando un proceso intente enviar un mensaje a un buzón completamente ocupado, se suspende hasta que se elimina un mensaje de dicho buzón.

```
#include "prototypes.h "

#define N 100                       /* número de espacios en el almacén */
#define MSZINE 4                    /* tamaño del mensaje */
typedef int message [MSZINE];

void producer (void)
{
  int item;
  message m;                        /* almacén (buffer) del mensajes */

  while (TRUE)
  {
    produce_item (&item);           /* genera algo que colocar en el almacén (buffer) */
    receive (consumer, &m);         /* espera que llegue alguno vacío */
    build_message (&m, item);       /* construye un mensaje para enviarlo */
    send (consumer, &m);            /* envía al elemento al consumidor */
  }
}
```

```
void consumer (void)
{
  int item, i;
  message m;                          /* almacén (buffer) del mensajes */

  for (i = 0; i < N; i++)  send(producer, &m);          /* envía N vacíos */
  while (TRUE)
  {
     receive (producer, &m);          /* obtiene el mensaje que contiene al elemento */
     extract_item (&m, &item);        /* retira el elemento del mensaje */
     send (producer, &m);             /* envía una respuesta vacía de regreso */
     consumer_item (item);            /* hace algo con el elemento */
   }
}
```

Figura 11. El problema del productor y el consumidor con N mensajes.

El caso opuesto al uso de buzones es la eliminación del almacenamiento. Desde este punto de vista, si se ejecuta SEND antes de RECEIVE, el proceso emisor se bloquea hasta la ejecución de RECEIVE, momento en que el mensaje se pueda copiar en forma directa del emisor al receptor sin almacenamiento intermedio. En forma análoga, si se ejecuta primero RECEIVE, el receptor se bloquea hasta que se ejecute un SEND. Esta estrategia se conoce como **rendezvous**. Es más fácil de implantar que el esquema de mensajes almacenados; pero es menos flexible, puesto que el emisor y receptor se ven forzados a ejecutarse al mismo paso.

1.8 CONDUCTOS (PIPES)

En algunos sitemas (por ejemplo Unix) existe el concepto de conducto que constituye un mecanismo muy elegante de intercambio de información.

Un conducto es un archivo en el cual los procesos pueden escribir información, que otros procesos pueden leer posteriormente. Ademàs, puede desempeñar funciones de sincronización.

En el caso del sistema Unix existe la primitiva

Pipe (da)

que le permite al usuario crear un conducto, y que le devuelve en la variable de dos descriptores para leer y escribir en el conducto. A partir de ese momento, el usuario puede realizar sobre el mismo las operaciones clásicas sobre archivos (leer y escribir).

El tipo de conducto creado mediante la primitiva anterior no tiene un nombre dentro del sistema (sólo un descriptor); por esta razón no puede ser utilizado por procesos de diferente familia (que no son descendientes directos, y por consiguiente no heredan los descriptores) del proceso que lo creó. Para resolver este problema se crearon los conductos con nombre, que tienen un comportamiento similar a los anteriores, pero tienen un nombre dentro del sistema y pueden por lo tanto ser utilizados por todos los procesos de éste. Los conductos con nombre se crean por medio de la primitiva corriente para creación de archivos especiales, especificando que se trata de un archivo tipo conducto.

Las primitivas de lectura y escritura sobre conductos tienen asociadas operaciones de sincronización (si un proceso lee de un conducto en el que no hay información, se bloquea y es despertado cuando otro proceso escriba), lo cual facilita la comunicación entre procesos. En el caso de los conductos sin nombre otra ventaja adicional es que son archivos temporales que desaparecen automáticamente cuando se destruyen todos los procesos que tienen descriptores a ellos (lo cual no ocurriría si se utilizaran archivos corrientes, en cuyo caso el usuario tendría que explícitamente destruirlos).

El siguiente trozo de programa ilustra cómo se pueden utilizar conductos en Unix para comunicar procesos:

```
Pipe (da)
zona ← "Hola"
REPITA INDEFINIDAMENTE
  Write (da [1], zona, 5)
  Read (da [0], zona, 5)
FIN-REPITA
```

El programa anterior crea un conducto y luego se envìa a sì mismo el mensaje "Hola", escribiendo y leyendo del conducto creado. En el arreglo da, de dos elementos, el primero (da [0]) se refiere al descriptor de lectura, y el segundo (da [1]) al de escritura.

Aunque en el ejemplo anterior hay un solo proceso comunicándose consigo mismo, puede servir para ilustrar la utilización del mecanismo de conductos en Unix.

1.9 EQUIVALENCIA DE PRIMITIVAS

Reed y Kanodia 819799 describieron otro método de comunicación entre procesos llamado de **secuenciadores.** Campbell y Habermann (1974) analizaron el método de **expresiones de trayectorias**. Atkinson y Hewitt (1979)introdujeron los **serializadores**. Aunque la lista de métodos distintos no es infinita, ciertamente es larga, además, todo el tiempo se sueña con alguna de ellas. Por fortuna, los limites de espacio nos evitarán analizar todas ellas. Además, varios de los esquemas propuestos son similares a otros.

En las secciones anteriores hemos estudiado cuatro primitivas distintas para la comunicación entre procesos. Atravesé de los años, cada uno ha acumulado simpatizantes, los cuales afirman que el mejor método es el suyo. La verdad es que todos los métodos son en esencia equivalentes desde el punto de vista de la semántica de ellos para construir los demás.

Mostraremos ahora la equivalencia esencial de los semáforos, monitores y mensajes. Esto no sólo es interesante en sí, sino que también da una idea y ayuda a comprender la forma de funcionamiento de las primitivas, así como su implantación. La falta de espacio nos evita el tener que trabajar también con los contadores de eventos, pero se puede obtener el punto de vista general a partir de los demás ejemplos.

Uso de semáforos para la implantación de monitores y mensajes

Analicemos en primer lugar el problema de la construcción de monitores y mensajes por medio de los semáforos. Si el sistema operativo proporciona como característica básica los semáforos, cualquier creador de un compilador puede implantar con facilidad los monitores en su lenguaje de la manera siguiente. Primero se construye una pequeña colección de procedimientos (que se emplean durante el tiempo de ejecución) para el control de los monitores, la cual se coloca en la biblioteca; estos procedimientos se enlistan en la figura 12. Así, cada vez que se genere un código que utilice monitores, se hacen las llamadas al procedimiento al tiempo de ejecución, para llevar a cabo la función necesaria.

A cada monitor se le asocia un semáforo adicional, *mutex,* con valor inicial 1, con el cual se controla la entrada al monitor, además de un semáforo adicional, con valor inicial 0, por cada variable de condición. Cuando un proceso entra un monitor, el compilador entra una llamada al procedimiento *enter_monitor* al tiempo de ejecución, lo que ejecuta un DOWN en el *mutex* asociado al monitor al que se entra. Si el monitor está en uso en ese momento, el proceso se bloqueará.

Aunque pareciera lógico que el código para salir de un monitor sólo realiza un UP en *mutex*, esta solución sencilla no siempre funciona. Cuando el proceso no ha dado una señal a otros procesos, entonces puede, de hecho, hacer un UP en *mutex* y salir del monitor. Este caso se muestra en *leave_normally* de la figura 12.

```
#include "prototypes.h"

  typedef int  semaphore;
  semaphore mutex = 1;              /* para controlar el acceso al monitor */

  void enter_monitor (void)         /* código para ejecutar a la entrada del monitor */
  {
    down (mutex);                   /* solo uno a la vez adentro, por favor */
  }
```

```
void leave_normal (void)          /* sale del monitor sin señalar */
{
  up (mutex);                     /* permite la entrada de otros procesos */
}


void leave_with signal (c)        /* señala a c y sale del monitor */
semaphore c;                      /* la variable de condición por señalar */
{
  up (c);                         /* libera a un proceso que espera a c */
}

void wait (c)                     /* se va a dormir en espera de una condición */
semaphore c;                      /* cual es dicha condición */
{
  up (mutex);                     /* permite la entrega de otros procesos */
  down (c);                       /* se va a dormir en espera de una condición */
}
```

Figura 12. Procedimientos de biblioteca para la implantación de monitores por medio de semáforos

La compilación proviene de las variables de condición. El WAIT sobre una variable de condición c se lleva a cabo como una secuencia de dos operaciones con semáforos. Primero se hace un UP sobre *mutex*, con lo que se permite que otros procesos entren al monitor. Luego se hace un DOWN en c para bloquear la variable de condición.

Conviene recordar que se debe realizar SIGNAL, como última operación antes de salir del monitor. Esta regla es necesaria para combinar la señalización y la salida en el procedimiento *leave_with_signal* de la biblioteca. Lo único que hace es un UP en la variable de condición. Entonces sale del monitor sin liberar *mutex*. El truco consiste en que, como *mutex*=0, ningún otro proceso puede entrar al monitor en ese momento. Sin embargo, el proceso recién despertado está activo en el monitor y hará un DOWN en *mutex* cuando salga de él. De esta forma, el proceso de señalización transfiere de manera eficaz la exclusión mutua al proceso señalado, con lo que se garantiza que ningún competido intervenga.

Analicemos ahora la forma de implantar la transferencia la de mensajes mediante semáforos. A cada proceso se le asocia un semáforo, con valor inicial 0, por medio del cual se hará un bloqueo cuando un SEND o RECEIVE deba esperar para terminar. Se utilizará un área compartida del almacén (buffer) para conservar los buzones, cada uno de los cuales contendrá un arreglo de entradas para mensajes. Las entradas de cada buzón se encadenan como una lista ligada, de forma que los mensajes se entreguen en el orden en que se reciben. Cada buzón tiene variables enteras indican el numero de espacios ocupados y

vacíos. Por último, cada buzón contiene además el inicio de dos colas, una para los procesos que no se pueden enviar al buzón y otra cola para los procesos que no pueden recibir información del buzón. Estas colas solo necesitan proporcionar los números de proceso de los procesos en espera de forma que se pueda ejecutar un UP en el semáforo respectivo. Todo el almacén (buffer) compartió queda protegido mediante un semáforo binario, *mutex*, para garantizar que solo un proceso a la vez pueda inspeccionar o actualizar las estructuras de datos compartidas. El almacén (buffer) se muestra en la figura 13.

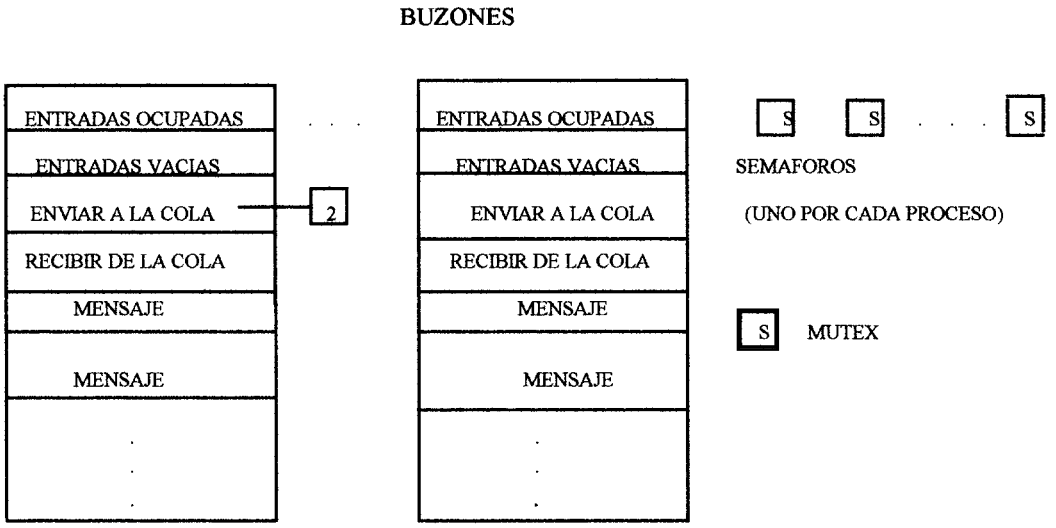


Figura 13. El almacén (buffer) compartido por la implantación de la transferencia de mensajes por medio de semáforos.

Cuando se ejecuta en SEND o RECEIVE en un buzón con al menos una entrada vacía o ocupada, respectivamente, la operación inserta o elimina un mensaje, actualiza los contenedores y los enlaces y realiza una salida normal. El uso de *mutex* al principió y final de la región crítica garantiza que solo un proceso a la vez puede utilizar los contadores y apuntadores, con el fin de evitar las condiciones de competencia.

Cuando se realiza un RECEIVE en un buzón vacío, el proceso que intenta recibir un mensaje entra a la cola de recepción para el buzón, lleva a cabo un UP en *mutex* y un DOWN en su propio semáforo, con lo que se pone a dormir. Más tarde, cuando despierta, hace un DOWN en *mutex*, al igual que en el caso del uso de semáforos para la construcción de monitores.

Cuando se realiza un SEND, si existe espacio en el buzón de destino, el mensaje se coloca ahí y el emisor verifica si la cola receptora de ese buzón tiene un proceso en espera. En tal

caso, el primero de ellos se elimina de la cola y el emisor hace un UP en su semáforo. El emisor sale entonces de la región crítica y el recién despertado receptor puede continuar. Sus respectivos DOWN y UP en *mutex* se cancelan (sin importar el orden en que se llevaron a cabo) y no ocurren problemas, siempre que, al igual que en el caso de los monitores, un proceso que despierte a otro realice WAKEUP como su última acción antes de salir de la región crítica.

Cuando un SEND no puede terminar su labor debido a un buzón completamente ocupado, el emisor se forma en la cola del buzón de destino, realiza un UP en *mutex* y un DOWN en su propio semáforo. Más tarde, cuando un receptor elimina un mensaje del buzón completamente ocupado y observa que alguien está formado en espera de poder enviar algo al buzón, se despierta al emisor.

Uso de monitores para implantar semáforos y mensajes

La implantación de semáforos y mensajes por medio de monitores sigue casi el mismo patrón que el descrito con anterioridad, aunque es más sencillo, porque los monitores son una construcción de un nivel superior al de los semáforos. Para la implantación de los semáforos necesitamos un contador y una lista ligada para cada uno de los semáforos por implantar, además de una variable de condición para cada proceso.

Al realizar un DOWN, el proceso que hace la llamada verifica, dentro del monitor, si el contador de ese semáforo es mayor que 0. En ese caso el contador se decrementa y el proceso sólo sale del monitor. Si el contador es 0, el proceso que hace la llamada añade su propio número de proceso a la lista ligada de ese semáforo y realiza un WATT en su variable de condición.

Cuando se lleva a cabo un UP en un semáforo, el proceso al que se llama incrementa su contador (dentro del monitor, por supuesto) y verifica si la lista ligada de ese semáforo contiene algún dato. En este caso, el proceso elimina uno de ellos y lleva a cabo un SIGNAL en la variable de condición de ese proceso. Observe que el proceso no está obligado a elegir el primer proceso de la lista ligada. En una implantación más sofisticada, cada proceso podría establecer su prioridad en la lista junto con su numero, de forma que se despertara primero el proceso de mayor prioridad.

La implantación de mensajes por medio de monitores es en esencia la misma que en el caso de los semáforos, excepto que en vez de un semáforo por proceso se tiene una variable de condición por cada proceso. Las estructuras de los buzones son las mismas para ambas implantaciones.

Uso de los mensajes para la implantación de semáforos y monitores

Si se dispone de un sistema de mensajes, se pueden implantar los semáforos y los monitores mediante un pequeño truco. Este consiste en introducir un nuevo proceso, el

proceso de sincronización. Veamos primero la forma en que se puede utilizar este proceso para implantar los semáforos. El proceso de sincronización mantiene, para cada semáforo, un contador y una lista ligada de procesos en espere. Para llevar a cabo un UP o un DOWN, un proceso llama al correspondiente procedimiento de biblioteca, *up* o *down,* el cual envía al proceso de sincronización un mensaje que incluye la operación deseada y el semáforo por utilizar. El procedimiento de biblioteca recibe entonces un RECEIVE para obtener la réplica del proceso de sincronización.

Cuando llega el mensaje, el proceso de sincronización verifica el contador para ver si la operación pedida se puede llevar a cabo. Los UP siempre se pueden llevar a cabo, pero los DOWN se bloquean si el valor del semáforo es 0. Si la operación es permitida, el proceso de sincronización envía de regreso un mensaje vació, con lo que elimina el bloqueo del proceso que hizo la llamada. Si, por el contrario, la operación es un DOWN y el semáforo es un 0, el proceso de sincronización forma al proceso que hizo la llamada en la cola y no forma una replica. Las condiciones de competencia se evitan en este caso debido a que el proceso de sincronización sólo trabaja con una solicitud a la vez.

Los monitores se pueden emplear por medio de mensajes con el mismo truco. Ya hemos mostrado la forma en que se implantan los monitores por medio de semáforos. Ahora mostramos la forma de implantar los semáforos por medio de mensajes. Al combinar ambas formas, se obtienen los monitores por medio de mensajes. Una forma de lograr esto es haciendo que el compilador implante los procedimientos del monitor mediante llamadas a los procedimientos de biblioteca *up* y *down* para *mutex* y semáforos para cada proceso, como se describió al principio. Estos procedimientos serán implantados entonces mediante el envío de mensajes al proceso de sincronización. También son posibles otras implantaciones.

2. PROBLEMAS CLASICOS DE LA COMUNICACIÒN ENTRE PROCESOS.

En la siguiente parte revisaremos algunos de los problemas más clásicos de la comunicación entre procesos, los cuales ya han sido analizados y discutidos ampliamente.

2.1 El problema de la cena de los filósofos

En 1965, Dijkstra planteó y resolvió un problema de sincronización llamado el **problema de la cena de los filósofos.** Desde entonces, todas las personas que idean cierta primitiva de sincronización intentan demostrar lo nuevo de la maravillosa primitiva al mostrar su elegancia para resolver el problema de la cena de los filósofos. El problema se puede enunciar de la manera siguiente. Cinco filósofos se sientan a la mesa. Cada uno tiene un plato de espaghetti. El espaghetti es tan escurridizo, que un filósofo necesita 2 tenedores para comerlo. Entre cada dos platos hay un tenedor. En la siguiente figura se muestra la mesa.

La vida de un filósofo consta de períodos alternados de comer y pensar.(Esto es un abstracción, incluso para los filósofos, pero las demás actividades son irrelevantes en este caso). Cuando un filósofo tiene hambre, intenta obtener un tenedor para su mano izquierda y otro para su mano derecha, alzando uno a la vez y en cualquier orden. Si lograra obtener los dos tenedores, come un rato y después deja los tenedores y continúa pensando. La pregunta clave es: ¿puede el lector escribir un programa para cada filósofo que lleve a cabo lo que se supone debería y nunca se detenga?(Se ha hecho la observación de que el requisito de dos tenedores es un tanto artificial; tal vez se debería cambiar la comida italiana por la china; es decir, el espaghetti por el arroz y los tenedores por los palillos chinos).

La figura 14 muestra la solución obvia. El procedimiento *take_fork* espera hasta que el tenedor especificado está disponible y lo toma. Por desgracia, la solución obvia es incorrecta. Supongamos que los cinco filósofos toman sus tenedores izquierdos en forma simultánea. Ninguno de ellos podría tomar su tenedor derecho, con lo que ocurriría un bloqueo.

```
#include "prototypes.h"

#define N=5                     /* número de filósofos */

void philosopher (int i)        /* cuál filósofo (desde 0 hasta N-1) */
{
  while(TRUE)  {
    think ( );                  /* el filósofo está pensando */
    take_fork(i);               /* toma el tenedor izquierdo */
    take_fork ((i+1) % N);      /* toma el tenedor derecho: % es el operador módulo N */
    eat( );                     /* mmm.... Espaghetti ! */
    put_fork(i);                /* coloca de regreso el tenedor izquierdo de regreso en la */
                                /*  mesa */
    put_fork((i+1) % N);        /* coloca el tenedor derecho de regreso en la mesa */
   }
}
```

Figura 14. Una no solución al problema de la cena de los filósofos.

Podríamos modificar el programa de forma que después de tomar el tenedor izquierdo, el programa verificara si el tenedor derecho está disponible. Si no, el filósofo deje el izquierdo espere cierto tiempo y vuelva a repetir todo el proceso. Esta propuesta también falla aunque por razones distintas. Con un poco de mala suerte, todos los filósofos podrían comenzar el algoritmo en forma simultánea, por lo que recogerían sus tenedores izquierdos, verían que los derechos no están disponibles, dejarían sus tenedores izquierdos, esperarían, volverían a recoger sus tenedores izquierdos en forma simultánea etc, eternamente. Una situación como ésta, en la que todos los programas se mantendrían en ejecución por tiempo indefinido pero sin hacer progresos se llama **inanición.** (Se llama inanición aunque el problema no se desarrolle en un restaurante italiano o chino).

El lector podría pensar: "si los filósofos esperan tan solo un cierto tiempo arbitrario, en vez del mismo tiempo, después de que no pudieran recoger el tenedor derecho, la probabilidad de que todo terminara bloqueado, incluso una hora sería muy pequeña". Esta observación es correcta, pero en ciertas aplicaciones uno preferiría una solución que siempre funcionara y que no fallara a una serie poco probable de números aleatorios. (Piensa en el control de seguridad de una planta nuclear.)

Una mejora a la figura 14 que no tiene bloqueos e inanición es la protección de semáforo binario. Antes de empezar a tomar los tenedores, un filósofo haría un DOWN en *mutex*. Después de volver a colocar los tenedores, haría un UP en *mutex*. Desde el punto de vista teórico, esta solución es adecuada. Desde el punto de vista práctico, tiene un error de desempeño; en cada instante sólo existiría un filósofo comiendo. Si se dispone de cinco tenedores, deberíamos poder permitir que dos filósofos comieran al mismo tiempo.

La solución que aparece en la figura 15 es correcta y permite el máximo paralelismo para un número arbitrario de filósofos. Utiliza un arreglo, *state*, para llevar un registro de la actividad de un filósofo: si está comiendo, pensando o hambriento (intenta obtener los tenedores). Un filósofo puede comer sólo si los vecinos no están comiendo. Los vecinos del i-èsimo filósofo se definen en los macros LEFT y RIGHT. En otras palabras, si i=2, entonces LEFT = 1 y RIGHT = 3.

```
#include "prototypes.h"

#define N                5      /* número de filósofos */
#define LEFT     (i+1)%N        /* número del vecino izquierdo de i */
#define RIGHT    (i+1)%N        /* número del vecino derecho de i */
#define THINKING     0          /* el filósofo está pensando */
#define HUNGRY       1          /* el filósofo intenta conseguir los tenedores */
#define EATING       2          /* el filósofo está comiendo */

typedef  int  semaphore;        /* los semáforos son un caso particular de int */
int state[N];                   /*arreglo para llevar un registro del estado de cada quien */

semaphore  mutex = 1;           /* exclusión mutua para las regiones críticas */
semaphore  s[N];                /* un semáforo por filósofo */


void philosopher (int i)        /* de cuál filósofo  se trata (desde 0 hasta N-1) */
{
  while(TRUE)  {                /* se repite por siempre */
    think ( );                  /* el filósofo está pensando */
    take_forks(i);              /* obtiene dos tenedores o se bloquea */
```

```
      eat( );                    /* mmm... espaghetti ! */
      put_forks(i);              /* coloca ambos tenedores en la mesa */
    }
}

void take_forks (int i)          /*  i: de cuál filósofo se trata (desde 0 hasta N-1) */
{
  down (&mutex);                 /* entra en la región crítica */
  state [i] = HUNGRY ;           /* registra el hecho de que el filósofo i tiene hambre */
   test (i );                    /* intenta tomar 2 tenedores */
  up (&mutex);                   /* sale de la región crítica */
  down (&s[i]);                  /* se bloque si no consiguió tenedores */
}

void put_forks (int i)           /* i: de cuál filósofo se trata (desde 0 hasta N-1) */
{
  down (&mutex);                 /* entra a la región crítica */
  state [i] = THINKING;          /* el filósofo ha terminado de comer */
  test (LEFT);                   /* ve si el vecino izquierdo puede comer ahora */
  test (RIGHT);                  /* ve si el vecino derecho puede comer ahora */
  up (&mutex);                   /* sale de la región crítica */
}

void test (int i)                /* i: de cuál filósofo se trata (desde 0 hasta N-1) */
{
   if (state[i] = HUNGRY  && state[LEFT] != EATING && state [RIGHT] != EATING )
   {
      state [i] == EATING;
      up (&a[i]);
    }
}
```

Figura 15. Una solución al problema de la cena de los filósofos.

El programa utiliza un arreglo de semáforos, uno por cada filósofo, de manera que los filósofos hambrientos pueden bloquearse si los tenedores necesarios están ocupados. Observe que cada proceso ejecuta el procedimiento philosofher como código principal, pero que los demás procedimientos, *take_forks*, *put_forks* y *test* son procedimientos ordinarios y no procesos separados.

2.2 El problema de los lectores y escritores

El problema de la cena de los filósofos es útil para modelar procesos que están en competencia por el acceso exclusivo a un número limitado de recursos, como las unidades de cinta u otros dispositivos de E/S. Otro famoso problema es el de los lectores y los escritores (Courtois *et al.*,1971), el cual modela el acceso a una base de datos. Imaginemos una enorme base de datos, como por ejemplo, un sistema de reservaciones en una línea aérea, con muchos procesos en competencia, que intentan leer y escribir en ella. Se puede aceptar que varios procesos lean la base de datos al mismo tiempo, pero si uno de los procesos está escribiendo(es decir, modificando) la base de datos, ninguno de los demàs procesos debería tener acceso a ésta, ni siquiera los lectores. La pregunta es ¿cómo programaría usted los lectores y escritores? Una solución se muestra en la figura 16.

En esta solución, el primer lector que obtiene el acceso a la base de datos lleva a cabo un DOWN en el semáforo *db*. Los siguientes lectores sólo incrementan un contador, *rc*. Al salir los lectores, éstos decrementan el contador y el último en salir hace un UP en el semáforo, lo cual permite entrar a un escritor bloqueado, si es que existe.

Una hipótesis implícita en esta solución es que los lectores tienen prioridad sobre los escritores. Si surge un escritor mientras varios lectores se encuentran en la base de datos, el escritor debe esperar. Pero si aparecen nuevos lectores, de forma que exista almenos un lector en la base de datos, el escritor deberá esperar hasta que no haya màs lectores interesados en la base de datos.

```
#include "prototypes.h"

typedef int semaphore;                  /* use su imaginación */
semaphore mutex = 1;                    /* controla el acceso a 'rc' */
semaphore db = 1;                       /*  controla el acceso a la base de datos */
int rc = 0;                             /* # de procesos que leen o desean leer */

void reader (void)
{
  while (TRUE)                          /* se repite por siempre */
  {
    down (&mutex);                      /* obtiene el acceso exclusivo a 'rc' */
    rc = (rc+1);                        /* un lector más ahora */
    if (rc == 1 ) down(&db);            /* si éste es el primer lector */
    up (&mutex);                        /* libera el acceso exclusivo a 'rc' */
    read_data_base( );                  /* acceso a los datos */
    downnn(&mutex);                     /* obtiene el acceso exclusivo a 'rc' */
    rc = rc - 1;                        /* un lector menos ahora */
    if (rc == 0) up (&db);              /* si éste es el último lector */
```

```
    up (&mutex);                    /* libera el acceso exclusivo a 'rc' */
    use_data_read ( );              /* sección no crítica */
    }
}

void writer (void)
{
  while (TRUE )                     /* se repite para siempre */
  {
    think_up_data ( );              /* sección no crítica */
    down (&db);                     /* obtiene el acceso exclusivo */
    write_data_base ( );            /* actualiza los datos */
    up (&db);                       /* libera el acceso exclusivo */
  }
}
```

Figura 16. Una solución al problema de los lectores y escritores.

2.3 El problema del barbero dormilón

Otro de los problemas clásicos de la comunicación entre procesos ocurre en una peluquería. La peluquería tiene un barbero, una silla de peluquero y *n* sillas para que se sienten los clientes en espera, si es que los hay. Si no hay clientes presentes, el barbero se sienta en su silla de peluquero y se duerme. Cuando llega un cliente, éste debe despertar al barbero dormilón. Si llegan mas clientes mientras el barbero corta el cabello de un cliente, ellos se sientan (si hay sillas desocupadas) o salen de la peluquería (si todas las sillas están ocupadas). El problema consiste en programar al barbero y los clientes sin entrar en condiciones de competencia.

```
#include "prototypes.h"

#define CHAIRS 5                    /* el número de sillas para los clientes que esperàn */
typedef int semaphore;              /* use su imaginación */
semaphore customers = 0;            /* # de clientes que esperán el servicio */
semaphore barbers = 0;              /*  # de barberos que esperan clientes */
semaphore mutex  = 1;               /* para la exclusión mutua */
int waiting = 0;          /* los clientes están esperando (no se les está cortando el pelo) */
```

```
void Barber (void)
{
  while (TRUE)
  {
    down (customers);          /* se va a dormir si el número de clientes es 0 */
    down (mutex);              /* procura el acceso a 'waiting' */
    waiting = waiting - 1;     /* decrementa el contador de clientes de espera */
    up (barbers);              /* un barbero está listo para cortar el pelo */
    up (mutex);                /* libera a 'waiting' */
    cut_hair( );               /* corta el pelo (fuera de la región crítica) */
  }
}

void customer (void)
{
  down (mutex);                /* entra a la región crítica */
  if (waiting < CHAIRS )       /* se va si no existen sillas desocupadas */
  {
    waiting = waiting + 1;     /* incrementa el contador de clientes de espera */
    up (customers);            /* despierta al barbero si es necesario */
    up (mutex);                /* libera el acceso a 'waiting' */
    down (barbers);            /* se va a dormir si # de barberos desocupados es 0 */
    get_haircut ( );           /* se sienta y se le da servicio */
  } else{
        up (mutex);            /* la peluquería está completamente llena; no espere */
  }
}
```

Figura 17. Una solución al problema del barbero dormilón.

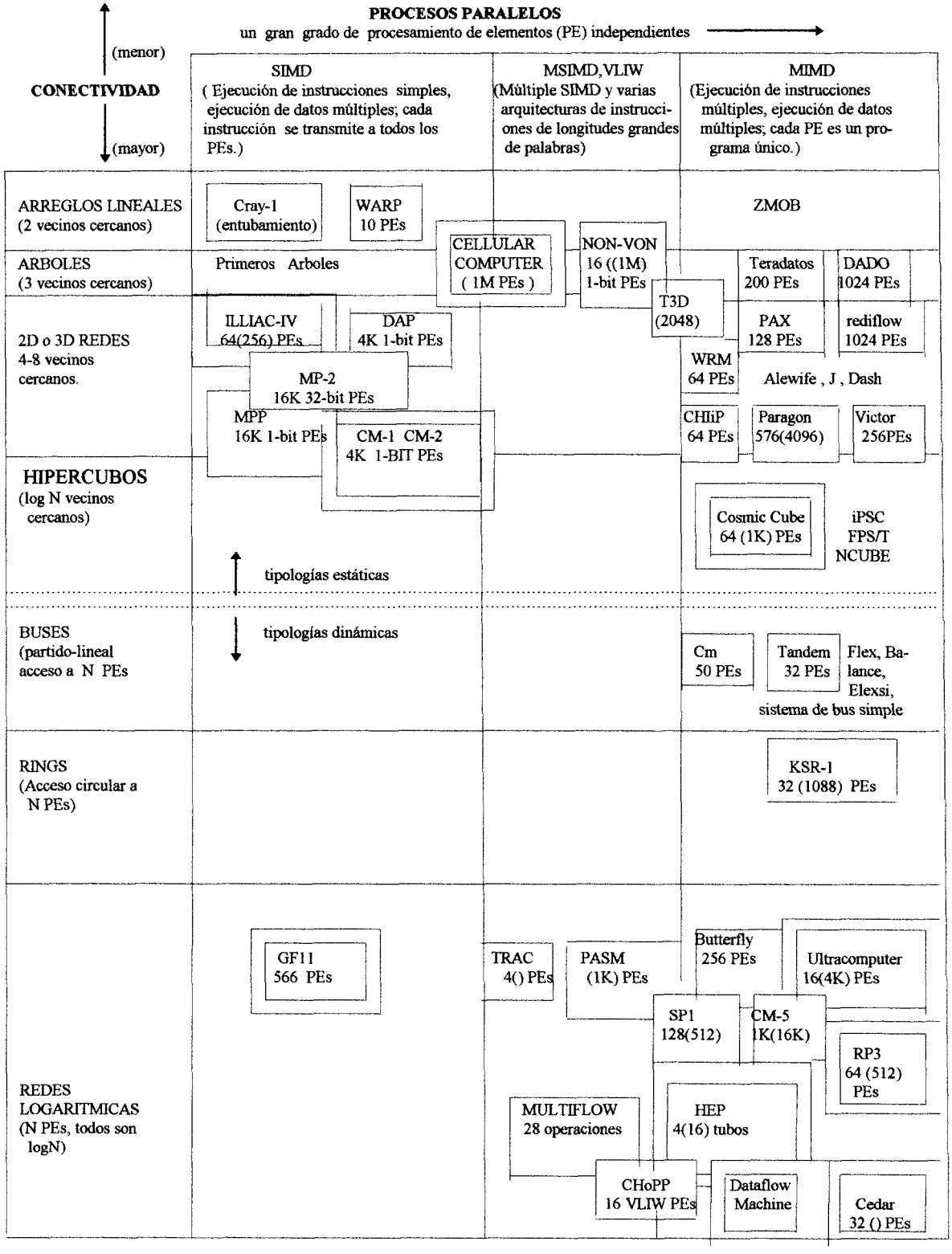
Nuestra solución utiliza tres semáforos; *customers,* que cuenta el número de clientes en espera (sin incluir al cliente en la silla de peluquero, que se está esperando), *barbers* el número de barberos inactivos y que esperan clientes (0 o 1) y *mutex*, que se utiliza para la exclusión mutua. También necesitamos una variable *waiting*, que también cuenta el número de clientes que esperan. Esto es en esencia una copia de *customers* . La razón para el uso de *waiting* es que no hay forma de leer el valor activo de un semáforo; en esta solución, un cliente que entra a la peluquería debe contar el número de clientes que esperan. Si es menor que el número de sillas, él se queda; en caso contrario, se va.

Nuestra solución se muestra en la figura 17. Cuando el barbero abre su negocio por la mañana, ejecuta el procedimiento *Barber*, lo que establece un bloqueo en el semáforo *customers* hasta que alguien llega; después se va a dormir.

Cuando llega el primer cliente, èl ejecuta *Customer*, que inicia procurando que *mutex* entre a su región crítica. Si otro cliente llega poco tiempo después, el segundo no podrá hacer nada, hasta que el otro haya liberado a *mutex*. El cliente verifica entonces si el número de clientes que esperan es menor que el número de sillas. Si esto no ocurre, libera *mutex* y sale sin su corte de pelo.

Si existe una silla disponible, el cliente incrementa la variable entera *waiting*. Luego realiza un UP en el semáforo *customers*, con lo que despierta al barbero. En este momento, tanto el cliente como el barbero están despiertos. Cuando el cliente libera a *mutex*, el barbero lo retiene, ordena algunas cosas e inicia el corte de pelo.

Al terminar el corte, el cliente sale del procedimiento y deja la peluquerìa. A diferencia de nuestros ejemplos anteriores, no existe un ciclo para el cliente, puesto que el corte de pelo es una operación idempotente. Sin embargo, sì existe un ciclo para el barbero, ya que intenta continuar con el siguiente cliente. Si existe otro cliente presente, el barbero hace otro corte de pelo. Si no, el barbero se va a dormir.


PROCESOS PARALELOS
un gran grado de procesamiento de elementos (PE) independientes
(menor)
CONECTIVIDAD
(mayor)
SIMD
(Ejecución de instrucciones simples, ejecución de datos múltiples; cada instrucción se transmite a todos los PEs.)
MSIMD,VLIW
(Múltiple SIMD y varias arquitecturas de instrucciones de longitudes grandes de palabras)
MIMD
(Ejecución de instrucciones múltiples, ejecución de datos múltiples; cada PE es un programa único.)
ARREGLOS LINEALES
(2 vecinos cercanos)
Cray-1
(entubamiento)
WARP
10 PEs
ZMOB
ARBOLES
(3 vecinos cercanos)
Primeros Arboles
CELLULAR COMPUTER
(1M PEs)
NON-VON
16 ((1M)
1-bit PEs
Teradatos
200 PEs
DADO
1024 PEs
2D o 3D REDES
4-8 vecinos
cercanos.
ILLIAC-IV
64(256) PEs
DAP
4K 1-bit PEs
T3D
(2048)
PAX
128 PEs
rediflow
1024 PEs
MP-2
16K 32-bit PEs
WRM
64 PEs
Alewife , J , Dash
MPP
16K 1-bit PEs
CM-1 CM-2
4K 1-BIT PEs
CHIiP
64 PEs
Paragon
576(4096)
Victor
256PEs
HIPERCUBOS
(log N vecinos
cercanos)
Cosmic Cube
64 (1K) PEs
iPSC
FPS/T
NCUBE
tipologías estáticas
tipologías dinámicas
BUSES
(partido-lineal
acceso a N PEs
Cm
50 PEs
Tandem
32 PEs
Flex, Balance, Elexsi,
sistema de bus simple
RINGS
(Acceso circular a
N PEs)
KSR-1
32 (1088) PEs
REDES
LOGARITMICAS
(N PEs, todos son
logN)
GF11
566 PEs
TRAC
4() PEs
PASM
(1K) PEs
Butterfly
256 PEs
Ultracomputer
16(4K) PEs
SP1
128(512)
CM-5
1K(16K)
RP3
64 (512)
PEs
MULTIFLOW
28 operaciones
HEP
4(16) tubos
CHoPP
16 VLIW PEs
Dataflow
Machine
Cedar
32 () PEs

CROSSBARS (acceso privado a N PEs)		Cray YMP	YSE/EVE 32K PEs	C.mmP 16 PEs	Alliant 8 PEs

Figura 18. Proyectos de Arquitectura Paralela agrupados según su independencia y su interconectividad con el procesamiento de elementos (PEs). Las cajas con doble borde denotan proyectos que son tratados como casos de estudio detallados, las cajas sencillas denotan menos descripción de detalles, y los nombres sin cajas dan un breve resumen. El nombre nominal de PEs es dado entre paréntesis, precedido por el número de accesos.

3. PROGRAMACION PARALELA

3.1 INTRODUCCION

Actualmente la investigación de computación en paralelo indica que podemos aplicar directamente nuestros conocimientos de computación concurrente a computación en paralelo porque la eficiencia de los algoritmos concurrentes no necesariamente nos llevan a algoritmos en paralelo eficientes. Además, allí existen varios problemas peculiares en el paralelismo que deberían ser vencidos en un orden de ejecución eficiente los cálculos en paralelo. La discusión de estos problemas y como poder resolverlos en un caso particular.

El estudio de la computación en paralelo es revelar un número de características principales que señalamos en el tema para no salirnos de la investigación del área. En este documento, discutiremos las 4 características que creemos más importantes en la computación en paralelo que se diferencian de el área sobre la computación concurrente. En general indicaremos cuales son los conocimientos de la computación concurrente que no se llevan sobre la computación en paralelo. Citaremos ejemplos de problemas de computación para saber cual es la solución màs eficiente, la concurrente o la paralela y las diferencias entre ellas. Preparémonos para la discusión en la siguiente sección, donde daremos una breve descripción sobre el desarrollo de la computación en paralelo.

Analizaremos los avances recientes en los sistemas de computación en paralelo, observamos que no solo la arquitectura de computación en paralelo tiene la única salida. Cuando hablamos de la computación en paralelo deberíamos identificarla como un arreglo computacional en la clase ILLIAC-IV, un conducto, un bit-corto, un multiproceso, etc. Flynn, cuidadosamente categorizo los sistemas computacionales dentro de 4 distintos tipos, 2 de los cuales son los que nos interesan. La primera instrucción lleva datos múltiples que corren en sistemas (SIMD) que tienen únicamente un flujo de instrucciones en ejecución en un tiempo, pero cada instrucción puede afectar muchos datos diferentes. Estos sistemas operan esencialmente un vector de datos simultáneamente en lugar de hacerlo en puntos

individuales. Arreglos de procesos, conductos de procesos, asociación de procesos, bit-corto de procesos son de estos tipos.

El segundo tipo de sistemas de computación en paralelo Flynn lo llama instrucción múltiple que lleva datos múltiples que corren en el sistema (MIMD). Este tipo de sistemas es equivalente a un sistema de procesos interconectados convencionalmente, en el sentido de que en un tiempo dado cada proceso en el sistema ejecuta una instrucción, y la instrucción opera en un dato. El sistema es como un soporte de todas las operaciones paralelas en que el proceso individual puede ser ejecutado simultáneamente, programas completos son corridos en particiones dentro de los pequeños programas en serie que corren uno de los procesos individuales. Los programas en varios de los procesos se comunican con cada uno de los demás durante la ejecución compartiendo información y cooperando de manera distinta en la solución del problema.

La discusión está fuertemente orientada a los sistemas MIMD. Esto no quiere decir que los sistemas SIMD son superiores a los sistemas MIMD.

Màs bien el problema que enfrentan los sistemas MIMD parecen más difíciles de resolver que los que enfrentan los sistemas MIMD, así que para fabricarlos hay que tener en cuenta la simplificación de los materiales para la mejor construcción de los sistemas SIMD. Los sistemas MIMD tienden a ser construidos con mucho cuidado y teniendo únicamente 2, 4 o en raras ocasiones 8 procesos, en tanto los sistemas SIMD tienen un paralelismo efectivo muy pequeño que puede ser de 32, 128 y hasta 258 operaciones. Es posible que la investigación en computación paralela continuara a ser orientada a sistemas SIMD, hasta la computación paralela en la máquina tal y como la comprendemos. Así los mejores sistemas paralelos MIMD son posiblemente desarrollados más lentamente y tendrán una madurez tardía. Y en un futuro cercano los sistemas MIMD probablemente serán limitados a sistemas con relativamente pocos procesos.

Flynn`s caracterizo a los procesos paralelos particularmente relevantes porque un algoritmo paralelo es esencialmente igualmente substituible en todo sistema para una clase dada. que es, un buen algoritmo para una ILLIAC IV presentando un buen algoritmo para conductos como son CDC STAR pero ambos sitemas de computo son mucho màs orientados a procesos en vector. Además, la sustituibilidad de los algoritmos para computadora SIMD son esencialmente incorregibles o quizás negativamente correlacionados y substituibles por sistemas MIMD.

En un estudio de sistemas de computo, fueron tomadas más medidas y eficiencia en la evaluación de los algoritmos y tambien en los sistemas de computo que son investigados. Una medida apropiada para especificar problemas es la relación sobre la velocidad definida como:

$$\text{relación sobre la velocidad} = \frac{\text{tiempo computacional de programas concurrentes}}{\text{tiempo computacional en los programas en paralelo}}$$

Realizándose una comparación justa, siempre comparando el mejor algoritmo concurrente para la computadora contra el mejor algoritmo paralelo, cuando son igualados los 2 algoritmos con arquitecturas totalmente diferentes.

Después de dar una breve introducciòn de lo que es la programación en paralelo, continuaremos dando una definición de lo que es la programación en paralelo, así como cuales son las mejores arquitecturas, formas de comunicaciòn de los procesos, los lenguajes usados para este propósito, así como algunos otros temas que se consideraron importantes incluir en esta parte.

3.2 DEFINICION Y PREGUNTAS ACERCA DE LOS PROCESOS PARALELOS

La siguiente definición describe mucho mejor los procesos paralelos:

Es una gran colecciòn de elementos procesados que pueden comunicarse y cooperar en la solución de problemas grandes rápidamente.

Unicamente podemos incluir otro importante factor como es la seguridad, que facilita la programación.

Figura 19. Generalización de procesos paralelos mostraremos dos caminos de arranque para el procesamiento de elementos (P) y módulos de memoria (M). La configuración izquierda 0"Dancehall" es usada principalmente para designar modelos de computación con memoria compartida en el que todas las P tienen igual acceso a todas las M. La configuración derecha "boudoir" es usada principalmente para designar el paso de mensajes en el que cada procesamiento de elementos tiene su propia memoria y se comunican por intercambio de mensajes, pero ahora es frecuente usar la designación de una buena memoria compartida. La memoria compartida tiene un mecanismo de comunicación mucho más fuerte, pero un mayor costo de implementación.

Nos permitimos introducir ahora la definición dada para procesos paralelos. Decimos que es una colección de elementos procesados que pueden comunicarse y cooperar en la solución de grandes problemas rápidamente. Dos de las implementaciones que podemos

concebir para esta definición son descritas en la figura 19. Sin embargo se pueden dedicar pocos minutos a esta definición con mucha lógica cortando el camino mostrado que se da exactamente en la lista de preguntas fundamentales. La información, estimula al lector ha hacer una pausa, para formar una lista de preguntas en su memoria mientras leemos el resto del texto. Algunas de las claves son dadas en las preguntas por las diferentes partes de su definición que son las siguientes:

1. Una colección de elementos procesados.....

   - ¿ Cuantos son muchos ?
   - ¿ Es poderoso cada uno de ellos ?
   - ¿ Qué operaciones ejecutan ?
   - ¿ Qué tecnología usan ?
   - ¿ Requieren de mucha memoria ?
   - ¿ Necesitan un disco secundario de almacenamiento, y como ejecutan los accesos de entradas y salidas ?

2. ... Qué pueden comunicarse ....

   - ¿ Como se comunican ?
   - ¿ Qué protocolos usan ?
   - ¿ Qué interconexión de red necesitan ?

3. ... y cooperar ...

   - ¿ Como se realiza el esfuerzo de sincronización ?
   - ¿ Qué tan largo sería el proceso que realiza ?
   - ¿ Sería autónomo cada proceso ?
   - ¿ Como operarían los sistemas en el esfuerzo de coordinación?

4. .... en la solución de problemas rápidamente .

   - ¿ Qué problemas son sensibles a los procesos paralelos ?
   - ¿ Qué modelo computacional es usado ?
   - ¿ Qué algoritmos se usaran ?
   - ¿ Se esperaría que fueran rápidos ?
   - ¿ Qué programas se ven en la máquina ?
     - ¿ Qué lenguajes de programación y software estarían disponibles ?
     - ¿ Como se descompone el problema ?
     - ¿ Como es la ejecución concurrente específicamente?

Todas las preguntas que se hicieron son interesantes y de un alto grado. Las respuestas son dadas en forma independiente para cada una. Si únicamente dividimos el mundo dentro de la salida de la arquitectura y dentro de la salida del software, el punto 1 en la lista quedaría por encima como un punto aparte en las preguntas relacionadas primeramente con la arquitectura y la tecnología, mientras el punto 4 está más relacionado con el software. Además, los puntos 2 y 3 caen fuera del área, y las áreas interactuan entre ellas. ¿La computación designa la conducción examinando los problemas para darles solución, o bien examinando la solución para entender el problema? La pregunta es para ambos. Esto esta bien ilustrado por la descripción simple del diagrama en la figura 19, como mostramos la clase de elementos para sistemas computacionales son relativos.

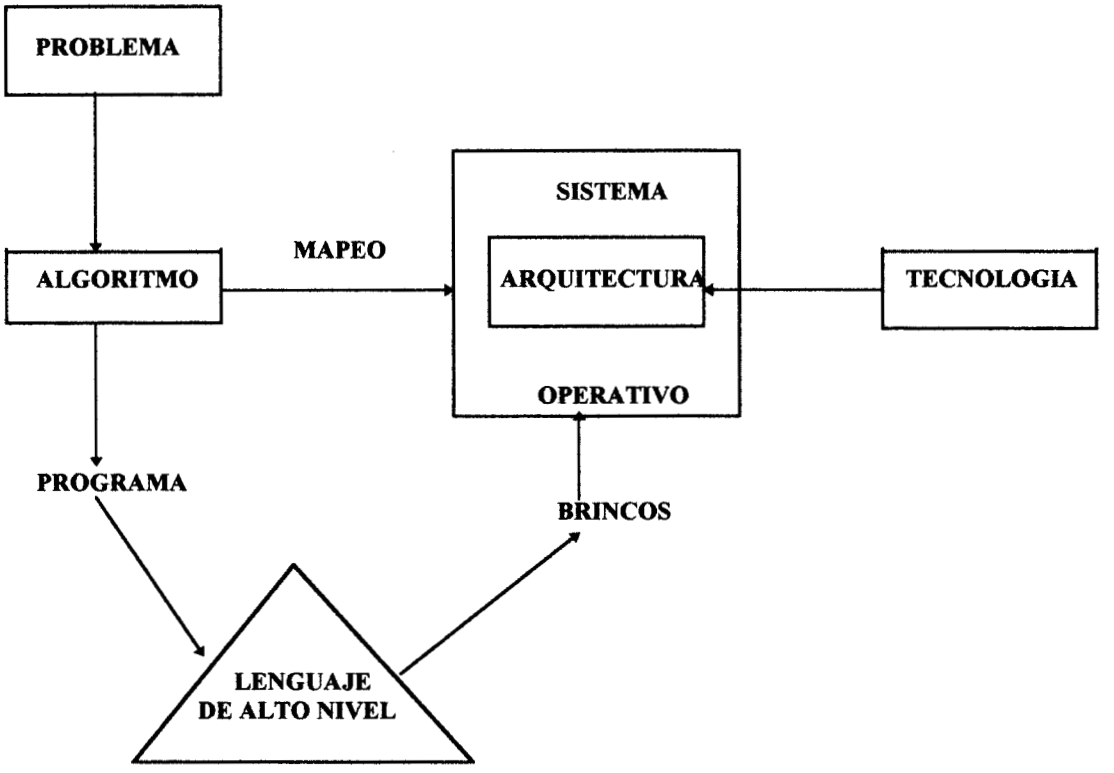


Figura 19. Elementos clave en un sistema computacional y sus interrelaciones.

3.3 SOLUCION A LAS PREGUNTAS

Discutiremos brevemente las respuestas que salieron de las preguntas arriba mencionadas, particularmente como se representan por algunos de los proyectos mostrados en la figura 17. La discusión siguiente se da en el orden de como se hicieron las preguntas.

Procesamiento de elementos: los números fuerza y naturaleza.

Observando la figura 18 nos muestra que los procesos paralelos son designados en decenas de grupos dentro de 3 grupos: vemos algunos de los procesos elementales, vemos que tienen un orden de mil, y una aproximación al millón. Estas sentencias no podrían ser tomadas en sentido estricto: en muy pocas decenas como 4 u 8 y en muchas como 32, en el orden de los miles se cubren como mínimo de 256 a 4096, y estas son designadas como la caída entre procesos largos. Sin embargo, esta tendencia, es la razón de diferentes filosofías cuando se selecciona un proceso de tamaño elemental.

Primero todas son designadas por su parecido a la Cray Y-MP C90 y IBM ES/9000, cuando se necesita muy poca fuerza en los procesos. Los representantes de designar el parecido lo marcan la velocidad de la computadora (normalmente una máquina con vector de conducto) y la combinación de ellos es mayor en la práctica.

Otra designación son las "decenas de PEs (elementos procesados)" categoría que tiene un bus base en la máquina que recalca un costo/cambio. El Hardware es construido un poco atrasado pero pequeño, poca-fuerza, y fresco como los transistores NMOS o CMOS. Aunque el bus es económico y todos los siguientes PEs se comunican directamente, se fija el límite del ancho de banda y el número de PEs.

Las computadoras con cientos de PEs son muy usadas y toman ventaja de la disponibilidad de los microprocesadores o de otras máquinas con cálculos VLSI. El ancho de banda requerido por el gran número de procesos es obtenidos usando las ventajas de una estación de trabajo.

En suma, los PEs son usados por la computación paralela e incluyendo los datos en un único almacén de la computadora, los microprocesadores comerciales, y las pequeñas computadoras con grupos de chips VLSI.

¿ Como se comunican los procesos ?

La interconexión de redes (estaciones de trabajo) permiten la comunicación entre PEs y la representación común de salidas entre la fuerza de comunicación y el costo. En un o de los extremos del espectro de la red, nos permite completar la conectividad con un mínimo de retraso entre los procesos pero implica un gran costo en la implementación. En el otro extremo hay tres tipos de redes en que los PEs son conectados únicamente a 3 vecinos cercanos. El intermediario común de salida, tal como en la red y la multiorganización de la red, son mostrados por el borde izquierdo en la figura 18.

El modo de comunicación en la red tambien puede ser especificado. En el intercambio de circuitos, una conexión es establecida paraque dure poco hasta que el mensaje sea transmitido. Un ejemplo clásico es el sistema telefónico en donde los circuitos se establecen cuando se hace una llamada y se desestablecen cuando esta termina. En una red de

intercambio de paquetes, el mensaje es dividido dentro del paquete, y cada uno contiene una dirección de destino.

Los mecanismos de comunicación de redes y protocolos pueden ser usados para construir un sistema de comunicación de memoria compartida o de paso de mensajes.

4. SOFTWARE PARALELO

En esta parte examinaremos la programación en paralelo y veremos algunas de sus aplicaciones, así como el uso de los sistemas paralelos. ¿Como obtenemos la ejecución de un programa en paralelo? ¿que en lenguajes de programación son válidos para este propocito? En otras palabras ¿que modelos computacionales pueden ser usados para este propósito?.

En la "Construcción esencial de la Ejecución en paralelo" discutiremos una operación extra que es necesaria para una buena ejecución de un programa en paralelo - la capacidad para definirlo, y la habilidad para particionar y distribuir los datos - tambièn daremos la construcción de algunos ejemplos usados en la transformación de operaciones.

4.1 LENGUAJES IMPERATIVOS Y DECLARATIVOS.

La clasificación de lenguajes puede ser declarativa o imperativa, en esta parte se dividen en un mismo camino. Los lenguajes imperativos son comunes. Los lenguajes declarativos pueden tener muchas salidas o caminos. Esto depende de las características del lenguaje declarativo realmente, compensando el incremento de su paralelismo en gran contraste con la inversión de los programas imperativos y las técnicas de programación. Está comparación resulta justa, así como la unión de ellos.

Los lenguajes imperativos son sentencias de comandos. Los comandos actualizan las variables que tienen almacenadas, un programa imperativo se supone que comienza haciendo intercambios conforme avanza el programa.

Durante un periodo de 40 años el punto central eran los lenguajes imperativos (Fortran, Cobol, C, Adda, etc.), y la compilación para el modelo computacional de Von Newmann, así que no es sorprendente que estos lenguajes se hayan escogido por los programadores profesionales, cuya comparación hecha posteriormente por todos, fue la escritura eficiente de los programas en la computadora por ellos, naturalmente la representación de un programa imperativo. La llegada de las computadoras en paralelo trajo con ello la llegada de extensiones en paralelo a los lenguajes imperativos, y compiladores paralelos para la ejecución automática del paralelismo en los lenguajes imperativos seriales. Esta aproximación tiene algunos sucesos, y posteriormente una rápida revisión de los lenguajes

imperativos no paralelos, haremos una descripción entre los programación imperativa y la programación en paralelo más adelante. Pero veremos que quienes creen que la computación en paralelo es necesaria ahora, en los lenguajes imperativos liberan al programador del modelo de Von Newmann, pero en los programas la liberación de los datos es un nivel completamente detallado, y deja que se declare lo necesario separadamente de lo que se obtiene. Los lenguajes declarativos son distintos de los lenguajes imperativos, y son designados como el centro principal en expresiones que un programador propone y es màs bien como se supone en la computadora que puede ser transportados desde afuera. Los lenguajes de programación funcionales y de programación lógica son ambos declarados en categorías, y estas se discutirán posteriormente.

4.2 TRES ESCENARIOS PARA EL PROGRAMADOR

El uso de la computación en paralelo ha hecho ya una gran investigación en la existencia de programas y técnicas de programación, y naturalmente las semejanzas y ventajas que ha tomado para estar en ese sitio. Ahora nos interesa la programación en paralelo para este caso. Pero diremos que una buena ejecución de un programa en paralelo depende de la habilidad con que se define, de las entradas y salidas, de la coordinación de la ejecución en paralelo para las substraes hechas en paralelo, y la probabilidad de dividir y distribuir los datos en el programa. Quienes salen y como? Existen 3 escenarios actualmente, con un rango de representación cliente - salida entre el trabajo requerido por el programador y la ejecución obtenida.

Paralelizaciòn automática de "Dusty Decks"

En el escenario "Dusty Decks" o "Herencia de Código", un compilador paralelo acepta el programa original y produce un programa paralelo. El escenario de la paralelización automática demanda una mínima forma de uso, pero la perfección de la ejecución es frecuentemente modesta, y como los compiladores no son siempre válidos, especialmente si el objetivo del programador es un sistema paralelo con memoria distribuida.

Extensión paralela para lenguajes imperativos

El segundo escenario requiere más trabajo para su uso, para evitar el silencio es necesario aprender a completar el nuevo lenguaje o rectificar completamente el programa serial original. El uso de sobre estudiar la partición en la ejecución y los datos del programa original, en el acarreo de salida usando, un determinado paralelismo construido apropiadamente para el objetivo del sistema. Son ejemplos Express, PVM y linda para el paso de mensajes, y otra determinación es la construcción de memoria compartida. Este es un juego de herramientas para la construcción de un paralelismo explícito y explícitamente las variables del lenguaje paralelo son mucho más validas que los compiladores de paralelización automática, y el grado de paralelismo obtenido puede ser porque el programador conoce que el programa es realmente difícil de hacer. Sin embargo, está ligera diferencia influye para que los datos del programa original sean escritos para un claro

modelo de ejecución secuencia y únicamente tendrìa que preguntar acerca de la potabilidad en el código de resultado.

Lenguajes no imperativos

En este tercer escenario, el usuario decide a aprender un nuevo lenguaje con menos rastros que el secuencia del modelo computacional de Von Newmann, y reescribir el programa en un nuevo lenguaje. Está es un área de mayor actividad en la investigación de lenguajes, pero este escenario aún no tiene mucha práctica. Notablemente, el énfasis no es mucho en un lenguaje paralelo como en un lenguaje de alto nivel que no asume un modelo especifico de ejecución como en estos, si es paralelo o serial. Es un lenguaje parecido a Occam, con esta adaptación de mecanismos para el paso de mensajes, pero es mucho más ligero en lenguajes declarativos, cuando son designados a expresiones que les hacen falta para hacerlo.

4.3 REVISION DEL MEJOR LENGUAJE SERIAL

Esto es natural para programadores que usan lenguajes seriales y se preguntan como sustituir el lenguaje para una computadora en paralelo, tambièn muchos lenguajes paralelos y medios de programación son poco desarrollados o bien son extendidos a lenguajes seriales. Entender bien la versión serial lleva a entender mejor la versión en paralelo. Además, entender las diferencias entre lenguajes seriales, ayuda a entender algunas de las diferencias entre los lenguajes paralelos. Por esta razòn comenzamos con una breve revisión de los lenguajes seriales.

4.3.1 Resumen Histórico

FORTRAN es un viejo lenguaje; está influencia es extendida y puede ser vista uniformemente en el camino con estructuras de programas paralelos. Como el primer lenguaje de alto nivel, introduce características semejantes en expresiones simbólicas y subprogramas con parámetros, Fortran tenìa que demostrar como lenguaje si podrìa generar un código eficiente. Está designación pone considerablemente mayor énfasis en la eficiencia que en la sintaxis, y el lenguaje soporta un número de características adaptadas por los objetivos computacionales. Los tipos de arreglos de datos son soportados, pero en versiones anteriores a Fortran90, el lenguaje primitivo puede únicamente distribuir con arreglo de elementos, no con el total de arreglos. Decimos que el único poder que tiene Fortran es que es "Fácilmente compilable". Este modo de eficiencia en el lenguaje es escogido por la ciencia computacional en los problemas de ingeniería que son una de las divisiones de gran fuerza para el procesamiento en paralelo, pero esta eficiencia en máquinas de programación serial también marca como los programas difíciles se pueden rehacer con la computación paralela.

ALGOL fue designado como el que más nos concierne para la definición, ya que la maquina es independiente de la sintaxis e inspira más subsecuentemente el trabajo teórico en lenguajes de programación. Algol-60 introduce el concepto de bloques estructurados,

variables, procedimientos, pudiendo ser declarados en cualquier parte que el programa lo requiera, resultando un programa con más estructura jerárquica que Fortran.

COBOL es el lenguaje más usado en el mundo. Es un programa altamente estructurado y son específicamente equipados para varias aplicaciones, se pone un gran énfasis en el movimiento de datos y presentación. La razón de revivir una pequeña atención en la discusión de procesamiento en paralelo es que Cobol tiene más aplicaciones hoy en día que son limitadas no por la rapidez computacional pero si por la rapidez de las entradas y salidas, cuando es más fácil remediarlo con procesamiento paralelo.

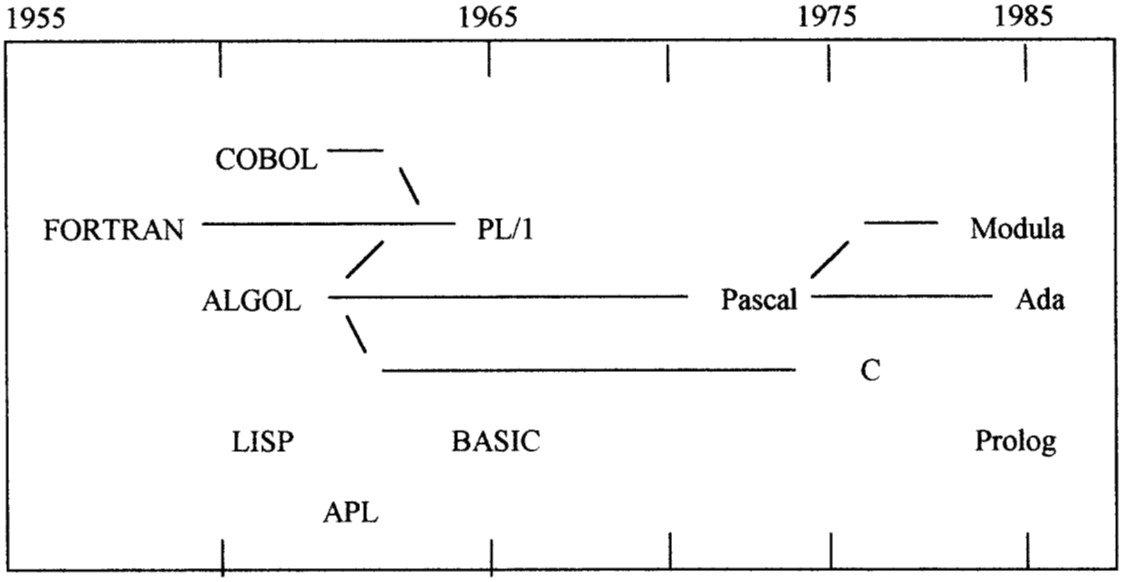


Figura. Un áspero árbol genealógico de el mejor lenguaje de programación concurrente

LISP mientras Fortran fue un mecanismo orientado hacia problemas científicos e ingeniería donde los datos son representados por arreglos regularmente grandes, LISP fue designado para problemas de inteligencia artificial y otras áreas con estructuras de datos irregulares que son representadas como listas.

APL es un mecanismo que maneja arreglos, pero es un lenguaje de más alto nivel que Fortran en el que las características tienen un gran potencial, que actúa en el arreglo completo, no únicamente en un arreglo de elementos en u tiempo.

PASCAL fue usado originalmente por maestros programadores y fue influenciando a todos los lenguajes, incluyendo Ada. Pascal concurrente es una extensión paralela que usa memoria compartida y es caracteristica de *procesos y monitores* para describir y coordinar la concurrencia.

C es un lenguaje comúnmente usado por programadores de sistemas. C habilita el acceso al hardware aguardando a quitar y ganar sobre Pascal en la escritura de operaciones de sistemas (ya que UNIX se escribió en C). Otra diferencia es que C es un programa interactivo porque es más flexible en el tratamiento de entradas y salidas. Con la popularidad de UNIX en multiprocesos, varias variables paralelas de C pueden aparecer para usarlas en aplicaciones.

4.4 VARIABLES COMPARTIDAS Y SUS EFECTOS

La clave estándar en un lenguaje de programación son las diferencias con otros mecanismos en función de una variable en el lenguaje. En matemáticas una variable es un símbolo usado para representar un determinado o posible valor. Además, un punto en donde la variable es introducida, y no conocemos su valor, esos valores son fijos. Si x^2=4, entonces x es siempre igual a 2 o -2. En contraste esta situaciòn con la siguiente asignación

$$x = x + 1$$

es usado en un lenguaje de programación imperativo semejante a Fortran, donde el valor de x es cambiado como un resultado de la ejecución de la asignación. Los lenguajes declarativos y funcionales son semejantes retiene el matemático Haskell en el sentido de las variables.

Los lenguajes para las computadoras de Von Newmann son en general divididos dentro de los tipos imperativos y declarativos. En un estilo de programación funcional, en noción a variables es muy similar en el sentido de las matemáticas. La aplicación de una función a una variable puede producir una nueva variable y un nuevo valor pero nunca cambios en el viejo valor. En la programación lógica, también una variable adquiere un valor, directamente del programa.

Existe una buena mejoría de programación para la computadora de Von Newmann y no se hace con lenguajes funcionales, pero si con estilo y con un lenguaje que es llamado imperativo, pero el programa resultante es bastante bueno. En un lenguaje de programación imperativo, el mundo de las variables es muy distinto en este sentido: llegar a cantidades con valores que pueden ser obtenidos de una localidad especifica de memoria. Esta información en la localidad de memoria puede ser leída y modificada por otras partes en el programa. Esa modificación del concepto de variables compartidas, y los efectos laterales de estos mecanismos tan poderosos reflejan el acceso global a memoria que es la caracteristica clave del modelo de Von Newmann, este provee un fuerte mecanismo para compartir estructuras de datos entre varias partes de el programa. Es posiblemente innecesario escribir la liberación de los efectos laterales en programas como Fortran, pero este solo usa una parte de la memoria total.

La reescritura de la nueva tabla de variables en un lenguaje imperativo puede llevarnos también a otra agradable sorpresa en el contenido de la localidad de memoria: por ejemplo, *aliasing*, o más explícitamente una clase de mecanismo como es la sentencia COMMON y la sentencia EQUIVALENT, en el protocolo de paso de parámetros en un lenguaje similar a

Fortran. Este mecanismo admite una ejecución más eficiente de programas imperativos, pero también marca un riguroso paralelismo para un programa, un punto que es usado para discutir en favor de los lenguajes funcionales.

4.5 LENGUAJES IMPERATIVOS PARALELOS Y EXTENCIONES

En la parte anterior , se describió el desarrollo de 3 escenarios para programar computadoras en paralelo - paralelismo automático, extensiones a los últimos lenguajes y nuevos lenguajes. Las personas han escrito programas para máquinas especificas en paralelo, pero ninguno de los tres escenarios ofrece un ambiente de programación para crear programas portables - no ofrece un camino práctico para escribir programas que tengan una buena rapidez en una variedad de hardware. Desde entonces, se ha logrado un considerable progreso para formar un escenario central. Varias extensiones en Fortran común y C se podrían aceptar y proporcionar un juego de herramientas para la construcción de un paralelismo estándar que admite correr el programa en una variedad de sistemas paralelos. Estas extensiones incluyen Fortran-90, Express, PVM y Linda. A continuaciòn describiremos estos sistemas.

FORTRAN-90

La idea básica detrás del paralelismo de datos en Fortran-90 nos permite expresiones en que colocamos un arreglo de nombres para todos los elementos procesados concurrentemente, recordando APL. Por ejemplo, si X es un arreglo unidimensional de 50 elementos, pero

```
DO  I=1,50
    X(I) = 2 * X(I)
END DO
```

y X = 2 * X
para cada doble elemento de X. Puede ser imaginada y descrita una buena sección de arreglos. Por ejemplo

X (1 : 25)

refiriéndonos a los primeros 25 elementos de X y

X (1 : 50 : 2)

refiriéndonos a las entradas con índices impares. Con un arreglo multidimensional surgirían posibilidades más interesantes. Por ejemplo dado un arreglo Y de 100 * 100

Y (1 : 50 , 1 : 50)

damos un cuarto más a la izquierda, y

Y (1: 100 : 2 , :)

damos un número impar de filas.

Como en APL, varios operadores y funciones intrínsecas pueden ser usadas, así

X = SIN(X)

colocando los 100 senos. Así como la reducción intrinseca parecida a SUM, cuando calculamos la suma de todos los elementos en un arreglo, y MAXVA, cuando calculamos el máximo.

Los saltos son fácilmente usados en la secciòn de operaciones del arreglo:

X (1: 40) = X (11: 50)

especificando a la izquierda el salto de 10. Girarlo puede ser llevado a cabo por la instrucción intrinseca CSHIFT. Saltar y rotar en un arreglo multidimensional puede también ser válido. Es como el caso, con diferentes elementos en el arreglo son almacenados localmente en diferentes procesos, en estos saltos y rotaciones son actualmente la comunicación de los operadores.

Multilisp.

Otra modificación que existe en los lenguajes (LISP) es descrita en "Halstead's Multilisp y paper" como tambien da una buena discución de algunos de los resultados de los lenguajes paralelos. La idea bàsica detrás de Multilisp es la evaluación de expresiones en paralelo, es la principal construcción usada para su creaciòn al igual que la sincronizaciòn como una tarea en el futuro.

El lenguaje usa un modelo de memoria compartida, los efectos laterales, y explicitamente la construcción en paralelo. Esto explica y arca la comparación con otros lenguajes.

La memoria compartida puede ser un importante filtro para la explotación de un mediano y un fino paralelismo porque la alternativa de el copiado explícito de datos compartidos que se requerirìan en una máquina de memoria no compartida desalentando la escritura de programas con interacciones frecuentes. Estos efectos se suman a las grandes expresiones de los lenguajes de programación y son importantes para arreglos, bases de datos y otras operaciones que son más directas en términos de los objetos con los que intercambian información, por otra parte, la presencia de estos efectos marca un paralelismo tosco y duro de explicar a menos que construyamos un paralelismo explicito, sus usos e indiquemos el limite de sus tareas. Multilisp es hecho con PCALL, un trabajo conjunto y una combinación de llamadas a procedimientos.

Un papel más reciente describe un compilador basado en la implementaciòn de Multilisp para un multiprocesador (memoria compartida) Encore Multimax, presentando los cambios resultantes para 12 procesos, y marca la comparaciòn con otros paralelismos de LISP.

Pascal Concurrente

Pascal Concurrente es un lenguaje paralelo de memoria extendida que extiende el pascal secuencial con la ayuda de la programación concurrente en las llamadas a *procesos y monitores*. El intento original fue para programación estructurada en los sistemas

operativos de las computadoras, estos se inclinaban a hacer un poco de énfasis en dos actividades paralelas.

Un proceso es definido como una estructura de datos privada y un programa secuencial operando en èl, constituyendo una parte del programa principal que puede ser ejecutado simultáneamente con otro proceso. Un proceso no puede operar en un dato privado de otros procesos, pero ellos pueden compartir ciertas estructuras de datos como son entradas y salidas de buffer. Un monitor define una estructura de datos compartida y todas las operaciones que los procesos puedan cambiar en el. Un monitor también trabaja y sincroniza procesos concurrentes, transmitiendo los datos entre ellos, y controlando el orden en que compiten los procesos usando una distribución física de lo ocurrido.

CSP y Occam

La comunicaciòn secuencial de procesos no es realmente entendida como un lenguaje completo, pero una buena propuesta para colocar las primitivas en un procesamiento paralelo es usando paso de mensajes en una buena comunicación de memoria compartida; en otras palabras, la comunicación de procesos por medio de la entrada con sentencias que se declaran como variables globales. El control secuencial es por medio de comandos protegidos como un mecanismo en una sentencia ejecutable que es precedida por una prueba que puede ser satisfecha sin que la sentencia se ejecute. Las 5 primitivas consisten n lo siguiente:

1. Un comando paralelo basado en un comienzo a medias para una ejecución especifica de procesos.
2. Un comando mandado, denotado por (!), que fue implementado como una llamada de entrada que parte del mecanismo rendenvoz de Ada (Figura). La ejecución se detiene si el proceso no recibe ninguna lectura.
3. Un comando recibido, denotado por (?), que se implemento como la aceptación de una llamada de entrada parte del mecanismo rendenvoz de Ada. La ejecución se detiene si el mensaje no es enviado.
4. Una alternativa, son comandos que especifiquen la ejecución de exactamente una de ellas que constituyen comandos protegidos.
5. Una repetición, son comandos que especifiquen cuantas iteraciones son posibles en ellos que constituyen comandos alternativos.

Halstead argumenta que la ausencia de memoria compartida en CSP causa un pequeño paralelismo que se pierde. Además no todos coinciden, èl demanda que la asociación de cada dominio de datos quede protegida con un simple paso secuencial de ejecución y el estilo no uniforme de el acceso a los datos (uno para el dato local en el proceso y otro, a saber en la transmisión del mensaje, para el acceso entre procesos) desanima el uso de un número largo de procesos, que parecen ser consistentes con la situación de Ada. CSP tambien tiene algunas deficiencias en la implantación de tipos abstractos de datos, que es, como en la encapsulaciòn y el control al acceso del objeto en el programa. Por otra parte, CSP simplifica la ayuda en la prevención de corrección de programas.

La idea de paralelismo con paso de mensajes en CSP puede ser incorporada en Occam, un lenguaje implementado en "Inmos Transputer". El transporte es de 32 bits, semejante al microprocesador RISC con comunicación extra que capacita, designa y especifica la implementaciòn de CSP en el modelo de paso de mensajes en la computación en paralelo.

Todo programa Occam es construido con 3 procesos primitivos - asignación, entradas y salidas - y actúan sobre variables, canales y relojes. El reloj suple el tiempo, las variables obtienen valores de asignaciones de entrada, y de los valores de la comunicación de los procesos. Las diferencias de CSP, con los procesos paralelos Occam es que estos se comunican indirectamente a través de canales, unidos por un solo camino punto a punto. Estos canales no necesitan un buffer, además los datos son transferidos únicamente cuando ambos son preparados para leer y escribir (sincronización de el paso de mensajes). La simplicidad de Occam facilita la escritura de programas paralelos con paso de mensajes, los requerimientos en los numerosos procesos pueden ser almacenados en localidades fijas a tiempo de compilación e implementando una eficiencia en la transportaciòn.

4.6 EXPRESS, PVM Y LINDA.

A continuación describiremos tres ambientes de programación muy usados, y las herramientas para escribir programas paralelos que corren en hardware MIMD con memoria distribuida. Express, PVM y Linda proveen la construcción que sigue un programa cambiando las tres funciones clave: su definición, inicio y paro, y la coordinaciòn de la ejecución en paralelo.

Estos tres sistemas pueden ser implementados en varias arquitecturas paralelas, se demanda particularmente como los posibles caminos para obtener ejecución en paralelo en estacione de trabajo con conexión tipo LAN.

Express

El paquete de software Express de ParaSoft Corporatión proporciona un medio de programación para sistemas paralelos MIMD con memoria distribuida, incluyendo hypercubos y grupos de estaciones de trabajo. Express es un descendiente de CrOS (sistema operativo cristalino). Express ofrece librerías en los programas de aplicación que incluyen alrededor de 160 rutinas que pueden ser llamadas desde C o Fortran. Estas rutinas representan un medio de programación para crear programas paralelos con la funcionalidad básica que se les necesita - la habilidad de la comunicación, datos compartidos, lectura de archivos, ambiente gráfico, la obtenciòn de un debugged, y el análisis de los cambios - en otras palabras, las tres claves que habilitan las necesidades para la ejecución en paralelo que se mencionaron al inicio de el punto 4.5, además la partición de datos, las salidas útiles de el programa para el usuario. Las rutinas son resumidas a continuación.

Rutinas de librería de Express

Comunication:22	**Process Contral:16**	**Synch:11**	**I/O:14**	**Debug:5**
snd/rcv{a/sync}	load node(s)	semaphore		
broadcast/combine	alloc/dealloc			
(vectors,too)	start main/node			
Graphics:36	**Performance:24**	**Compile:7**	**Map:8**	**Util:22**
parallel creation	analysis		physical	
of graphics images	monitor		to	
(Plotix)			topology	

PVM

PVM (Parallel Virtual Machine) es un paquete se software para utilzar en una red heterogénea de computadoras como un simple recurso computacional. PVM consiste de dos partes: Un proceso llamado demonio, pvmd, que únicamente puede ser usado si es instalado en la máquina, para la comunicación entre procesos y sincronización de los mismos. PVM es designado para proveer un medio de paso de mensajes para aplicaciones con relativo acoplamiento, y gran paralelismo.

Antes de correr una aplicación en PVM, el usuario invoca archivos pvmd dentro de su estación, y el primer turno es para el pvmd que se encuentra hasta arriba de la lista de la computadora con los nombres de los archivos. Esta colecciòn de pvmd es definida en la configuraciòn (PVM) para el usuario. El usuario puede correr ahora programas de aplicación PVM. La segunda parte de el paquete PVM son las librerías, y son alrededor de 20 rutinas de interfase, resumidas a continuación:

Rutinas de librerías PVM

Communication:9	**Process Control:8**	**Synchronization:3**
snd/rcv	enroll/leave	barrier
put/get	initiate/terminate	waituntil
probe	status/whoami	readi signal

Los programas de aplicación son muy fáciles con las librerías de aplicación en PVM. El Manager/ Worker o host/node son modelos de programación paralelo iguales en PVM, pero se pueden usar tambien otros modelos, sin embargo los procesos PVM pueden inicializar procesos en otras máquinas. Usando procesos de comunicación el demonio pvmd, los empaqueta y manda al pvmd local, que determina la máquina donde se correrán y transmitirá el paquete a la maquina que tiene el pvmd, y que los repartirá dentro de los procesos remoto. Allí se suministraran tambien para convertir los datos a XDR (external data standard), además PVM es designado para medios heterogéneos. Como podemos esperar, transparencia y heterogeneidad son costos que someten y atrasan la comunicación. En PVM son estados representativos de trabajo en las LAN con un orden de 10 milisegundos, pero el mismo con llamadas a procedimientos remoto en el mismo ambiente (IP).

Además, PVM sirve para liberar el código en curso, con aumentos atractivos y tratos amplios (los documentos PVM y el código pueden ser obtenidos enviando un correo electrónico a netlibc@ornl.gov). En general PVM incluye asistencias para particionar algoritmos y registrarlos, el soporte de compilaciòn automática para diferentes arquitecturas, una interfaz gráfica para el usuario, un debugging y un trazo que facilita la ejecución.

Linda

El modelo linda de programación paralela incrementa alguna operaciones de el lenguaje serial base (como Fortran o C) que soporta la notación de tuplas residentes en un espacio reservado para tuplas. Linda puede ser visto como una aproximación modular de el paralelismo en la que introduce notación que es independiente de el lenguaje base, Linda sirve para expresar un buen paralelismo, específicamente, para la creación y coordinación de procesos. Carriero y Gelernter argumentan que hacer en la computación un lenguaje ortogonal (ejemplo, Fortran o C) en la coordinación, es apropiado escoger el lenguaje (ejemplo, Linda) para el lenguaje paralelo designado. Existen 6 operaciones.

Los 6 contructores de Linda

in and **inp**, que *mueve* una tupla al espacio para tuplas dentro de un proceso.

rd and **rdp**, que *copia* una tupla al espacio para tuplas dentro de un proceso.

out, que copia una tupla a un proceso fuera del espacio para tuplas.

eval, que ejecuta la tupla dentro del espacio para tuplas.

El "Tuplescope" es un útil programa de herramientas que presenta un entorno gráfico en donde presenta las operaciones hechas en el espacio de tuplas durante la ejecución, como otro aspecto de un programa en Linda.

Además Linda puede implementar primordialmente una memoria distribuida en la computadora y en redes con estaciones de trabajo, el modelo de espacio de tuplas ofrecido por Linda es como memoria compartida en el que hay múltiples tuplas residentes (una tupla es una serie de tipos de archivos, por ejemplo ("XY", 1, 3.4) es una tupla).

La simplicidad de el uso de el espacio de tuplas es un soporte paralelo vìa la entrada y salida de operaciones. El comando **out** ("task1", 3, x) esta compuesto por una tupla de 3 componentes dentro de el espacio reservado para tuplas. La primer componente es la cadena "task1", el segundo es el entero 3, y el tercero es el valor de la variable x. El comando **in**("task1", ?2, 1) remueve una simple tupla de la máquina de el espacio que le fue designado para tuplas y lo asigna en el segundo componente 2, un parámetro formal de la operación **in**, que es indicado por el símbolo ? que aparece. Una tupla corresponde a la operación **in** si el mismo número de componentes, la correspondencia de las componentes es del mismo tipo, y la correspondencia no-formal de las componentes son del mismo valor. Si varias tuplas corresponden, una es seleccionada como no deterministica; si no es seleccionada en la operación **in** el bloque puede ser sustituido por una tupla que puede ser removida.

La operación **rd** es similar a **in** excepto que la selección de la tupla se reinicia en el espacio reservado para la tupla.

Así las tuplas en Linda unifican el concepto de creación, sincronización y comunicación. Carriero y Gelernter refieren esto como la generación de comunicación: Cuando un proceso quiere comunicarse, se genera una tupla.

5. CLASIFICACIÒN DE SISTEMAS OPERATIVOS PARALELOS.

En esta parte trataremos lo concerniente con los caminos en que un sistema operativo reporta la presencia de múltiples procesos. Asì como acontinuaciòn trataremos, sobre algunos de los sistemas operativos para computadora, en especial de 3 de ellos.

1) DISTINTAS SUPERVISIONES.

El acercamiento simple de una supervisión para tratar con procesos múltiples es usar "distintas supervisiones" en que cada proceso tiene su propio sistema operativo (una copia de el) y tiene sus propios archivos y mecanismos de entrada/salida. Este acercamiento efectivamente trata a cada proceso en un múltiprocesamiento como un sistema

independiente. Efectivamente, el sistema operativo, o màs precisamente cada copia, puede ser tratada si es corrida y controlada en un uniproceso. Ademàs muchas nuevas pequeñas estructuras son necesarias al soportar el aspecto de multiprocesos en el hardware. No necesariamente, los acercamientos de distintas supervisiones tienen varios defectos. Para uno, desde el proceso son controlados independientemente, estos no soportan para múltiprocesamientos dentro de un solo trabajo. En resumen, además la migración de procesos es generalmente soportada, ya que el balance de la carga puede volverse un problema. En resumen, estos no son verdaderos sistemas paralelos.

2) MAESTRO-ESCLAVO.

Este simple sistema de implementación da soporte al múltiprocesamiento dentro dentro de un solo trabajo que es una organización maestro - esclavo en que un proceso se predefine y es declarado el maestro y es permitida la ejecución en el sistema operativo. El otro proceso, denotado esclavo, puede ejecutar únicamente programas de usuario.

En la pràctica un sistema maestro esclavo permite al esclavo cambiar fácilmente alguna funciones de paralelización del sistema operativo. Por ejemplo, un reloj es a menudo asociado con cada proceso, y mas generalmente con las interrupciones, con el reloj puede ser localizado el campo. Los esclavos tambien admiten un servicio de supervisión de llamadas que simplifican la duda de el estado de el sistema, es decir, el tiempo da trabajo, la prioridad de los procesos concurrentes, o la identidad que admite el usuario.

Diferenciándolo de "distintas supervisiones", un sistema maestro - esclavo permite un verdadero paralelismo en un solo trabajo, pero únicamente para procesos de usuario. El mismo sistema operativo es esencialmente serial con todos pero la mayoría de las funciones triviales son ejecutadas en el único proceso maestro. Para un modesto número de procesos y una pesada carga de trabajo computacional, paralelizando el uso de el trabajo puede ser modificado y el sistema maestro - esclavo puede tener la ventaja de simplificar los demás ámbitos del sistema simétrico descrito adelante. Naturalmente un solo maestro puede volverse un límite de velocidad: Si un trabajo requiere el 5% de esta ejecución en el sistema operativo, cuando el aumento es limitado a 20 , independientemente de el número de CPU`s.

El Maestro es también un solo punto de fallo. Ese sería un fallo, y el sistema entero correría los pasos. Además la tolerancia de fallas del sistema semejante a Tandem (en fila) adopta una organización simétrica.

3) SIMETRICO.

En una organización simétrica los recursos del Hardware (es decir, los mecanismos de entrada salida y una memoria central) son unidas así como están disponibles a todos los procesos. El sistema operativo es tambien simétrico, en este puede ser ejecutado un proceso, que en principio admite funciones del sistema operativo y son paralizadas en algunas formas como programas de aplicación, además, lo ideal es llevar a cabo una rápida ejecución igual al número de procesos no ejecutados y muy frecuentemente no es

precisamente aprovechado, como apuntando hacia afuera, libre de ejecución paralela, en particular una libre actualización de datos compartidos, puede llevar a resultados erróneos. La necesidad de organizar el sistema, asegura su atomicidad en las operaciones importantes que pueden tener el efecto de moverse a un embotellamiento serial , de ser un simple proceso maestro a ser una estructura de datos compartidos, esto es, de el hardware a el software.

Por ejemplo, en una simple configuración simétrica, la llamada a un maestro flotante, la entrada al sistema operativo es revisada como una gran sección crítica; esto es, puede ser ejecutada por un único proceso en un tiempo. Esta organización puede ser pensada como un sistema maestro en el que el maestro no es predesignado pero "flota" de proceso en proceso como es requerido. Como en un sistema puro maestro - esclavo, este no es tan veloz para ejecutarlo en un sistema operativo, pero ninguno de ellos es asociado a un solo punto de ruptura.

Medidas menos extremas pueden ser empleadas en la cuestión para la atomicidad, el sistema operativo puede ser dividido dentro de un número de componentes haciendo pequeñas interacciones. A un que cada componente es únicamente ejecutada serialmente, la ejecución paralela entre componentes es soportada. Vemos que cada uno de estos acercamientos, difieren en el número y tamaño de sus componentes, esto es, en la granularización de el paralelismo. Así como algunas veces es posible, quizás con la asistencia de Hardware y un mecanismo de estructura de datos puede ser actualizado concurrentemente y aun queda consistente.

5.1 HISTORIA DE LOS SISTEMAS OPERATIVOS PARALELOS.

Como indicamos , es primeramente importante un sistema operativo para un buen sistema paralelo que es capas de soportar aplicaciones paralelas en que procesos múltiples cooperan cerca de un solo problema. Además, iniciamos por mencionar sistemas operativos para varias computadoras paralelas primitivas, aunque exactamente esas máquinas tienen un bajo nivel de paralelismo.

Quizás sorpresivamente, un multiprocesador comercial es el Burroughs D825, que estuvo disponible en 1960. Este era con memoria compartida designado sobre 4 procesos (idénticos) y sobre datos del sistema operativo llamando a la operación automática y a la lista de programas. Aunque es posible correr aplicaciones paralelas en los multiprocesos que fueron introducidos durante los 60`s, este modo de ejecución se hizo rutinario. En su lugar, otro multiproceso fue proyectado con un costo efectivo alternativo a múltiples uniprocesos. De manera semejante la memoria central, los mecanismos de entrada / salida se unen en el D825; AOSP permite procesos según sus requerimientos y mecanismos de salida como sean necesarios. Todavía podemos categorizar a AOSP a partir de su empleo en una organización maestra flotante, en cada único proceso se puede ejecutar el sistema operativo, pero únicamente un proceso a la vez.

Otro multiproceso introducido en esta década incluyen las series Burroughs B5000/B6000 (2 procesos en 1966), el GE - 645 (4 procesos en 1965), la UNIVAC 1108 (3 procesos en 1965), y el sistema IBM 360 modelo 67 (2 procesos en 1966). En resumen el verdadero multiproceso es justamente mencionado, varios de los primeros sistemas mencionados emplean un proceso especial para dedicarlo a funciones. IBM tiene canales de datos que son usados para entrada / salida, y el CDC 6600 emplea "procesos periféricos" que corren en el sistema operativo. El BCC - 500 contiene aproximadamente 5 procesos de comparación, 2 dedicados a programas de usuario, uno a dirección, uno a memoria, uno a terminales de entrada / salida y uno a programas.

Al inicio de los 70's DEC introduce el PDP - 10 modelo KL - 10, con dos procesos de memoria compartida en la computadora que pueden ser configurados con un maestro - esclavo o bien como un sistema simétrico. El sistema operativo TOPS - 10 soporta ambas configuraciones. En la configuración maestro - esclavo, todos los periféricos se ligan al proceso maestro, y a un solo servicio más los requerimientos del sistema operativo. Además, el esclavo puede admitir su propio programa, en particular podría accesar a la lectura de la lista. El aaceso compartido a esta estructura de datos fue serializada, la exclusión mutua inicia la vía forzando un semáforo. En el modelado simétrico ambos procesos se pueden ejecutar concurrentemente (TOPS - 10), que ellos proporcionan no hacen referencia a la misma estructura de datos. Además muchas de estas estructuras, ya fueron revisadas con anterioridad.

El sistema IBM/370 modelos 158 MP y 168 MP, son semejantes a el PDP - 10s, con 2 procesos de memoria compartida y con dos configuraciones. Diferente la designación la de el DEC, una de estas configuraciones separa esencialmente el proceso. Esta configuración, llamada modo uniprocesos por razones ovias, da a cada PE su propia memoria principal, mecanismos de entrada salida, y copia el sistema operativo OS/VS2 (Operating System Virtul System 2, que es envuelto dentro de MVS). El modo de uniproceso de OS/VS2 es un ejemplo del aprovechamiento beneficio - supervisión. El modelo de multiprocesos es más interesante para usted y puede ser visto como una parte entre el maestro flotante AOSP y el simétrico TOPS - 10 . Parecido al último, VS2 es dividido dentro de sus componentes en cada ejecución serial. Aunque la granularidad es tosca; estos son únicamente 13 componentes y cada uno contiene varias estructuras de datos relacionadas. Obteniendo un solo semáforo tope asociado con una componente menos costosa que adquiriendo todas las estructuras de datos encerradas ceparadamente. La desventaja es que el paralelismo es posiblemente menor. Llegar a un punto neutro previene, donde cada proceso requiere tener un tope por el otro, una convención fue establecida requiriendo se obtiene en un orden fijo. Tambien introduce con esta nueva máquina 2 nuevas instrucciones de coordinación, comparación y cambio, con pruebas atómicas y posibles modificaciones a localidades de memoria, señalando procesos, y un uso del que será recuperado del fracaso de un proceso.

5.2 ALTOS NIVELES DE PARALELISMO.

La primera razón del bajo nivel de paralelismo en estos primeros sistemas es que los procesos individuales son costosos. En 1970, el desarrollo de minicomputadoras, especialmente DEC's PDP - 11, considerablemente es una barrera mínima y ensamblando (uniendo) un largo número de procesos llegaría a ser factible. Dos grandes (PDP - 11) proyectos desarrollados durante los 70's son C.mmp y Cm, ambos hechos en Carnegie - Mellon University. Durante este mismo periodo, el PLURIBUS multiprocesador de propósito especial fue costruido por Bolt, Beranek y Neuman (BBN) usando procesos Lockheed Sue. El CMU fue designado al host para experimentos importantes en sistemas operativos paralelos, a saber Hydra, Medusa y StarOS, que son descritos más adelante.

La explosión de microprocesadores en los 80's fue otro pequeño costo de el hardware asociado con el ensamble en la multiplicidad de los procesos. Efectivamente, un presupuesto de un millón de dólares puede comprar cientos de estaciones de trabajo, cada una contiene un mayor procesamiento de elementos en el uniproceso que se discutió más arriba. Además es verdad que las más recientes investigaciones en sistemas operativos se remarca el soporte para multiprocesos, la mayoría de estos trabajos tiene como objetivo un sistema operativo distribuido, en que el proceso ensambla de forma aproximada el acoplamiento para grandes redes independientemente de la computadora, más bien el acoplamiento es ajustado al sistema paralelo. En particular, directamente el sistema operativo soporta grandes programas de aplicación paralela, en general, no proporciona sistema operativo para esta red. Esto no quiere decir que el acoplamiento del sistema puede ser ignorado.

HYDRA

C.mmp fue una computadora de memoria compartida (MIMD) con 16 procesos conectados vía un crossbar a 16 módulos de memoria. Este sistema operativo fue llamado Hydra, anteriormente en la mitología bestia de múltisaltos. Parecido a Tops - 10, Hydra permite múltiples procesos al ejecutar en el sistema operativo funciones simultáneamente, con tal de que ellos no accesen a la misma estructura de datos. La exclusión mutua necesita ser provista por varios medios, lo mas interesante inicia con un Kernel llave que no consume memoria de banda ancha. Si el recurso no es valido, el proceso ejecuta esencialmente la llave del Kernel cuando pierde la espera para una interrupción indicando que la estructura de datos podría ser accesada exclusivamente. El resultado en el uso de llaves primitivas será prometedor. Un estudio muestra que aunque un proceso pueda pasar sobre el 60% de su tiempo ejecutando el código del Kernel, menos el 1% de el tiempo que pasa esperando para las llaves. Dos razones para esta baja espera de tiempo será de el 90% de las llaves requeridas siendo otorgadas inmediatamente y el tiempo promedio que pasa en una sección crítica es únicamente alrededor de 300 microsegundos.

Diferente al sistema anterior mencionado, Hydra fue (en parte) una posibilidad basada en sistemas orientados a objetos: Un objeto individual, y una capacidad consistente en un identificador infalsificable de un objeto de acuerdo con un grupo de buenos accesos

indicando la acción que el portador de la capacidad puede cambiar en el objeto. Posiblemente la acción incluye llamadas (a procedimientos) y lecturas (a archivos).

Hydra fue escrito como un Kernel que contiene una colección de mecanismos básicos que más de la facilidad proporcionada generalmente por un sistema operativo y es escrita como un programa de usuario, así permite definir varios niveles de usuario de una facilidad a coexistir.

MEDUSA Y StarOS

Cm* contiene 50 procesos más de memoria agrupados dentro de 5 grupos de 10 módulos cada uno. Un significado consecuente de esta jerarquía es que el tiempo de acceso a memoria no es uniforme; la relación para localizar: Intragrupos: Intergrupos es 1:3.9. Dos sistemas operativos son desarrollados: Medusa y StarOS. Parecido a Hydra, ambos tienen la capacidad de soportar orientación de objetos, pero estos son pasado de Hydra por introducir fuertes tareas, que son, una colección de tareas cooperantes para resolver un solo problema. Ambos sistemas intentan un CoSheduler con los componentes de la fuerza de tareas, esto significa que todas correrían concurrentemente (en procesos separados, naturalmente). Medusa y StartOS se implementan como fuertes tareas; comunicación de interprocesos vía paso de mensajes. StarOs defiere de medusa por ocultar más detalles en un hardware subyacente (oculto).

EMBOS

El sistema operativo Embos para el sistema Elxsi 6400, en sistema base - bus con 10 - procesos con memoria compartida, este mismo organiza en una manera estricta paso de mensajes: cada recurso es manejado por una tarea del sistema operativo, y estas tareas compartidas no son datos. Embos hace, como, que soporta ambos paso de mensajes y memoria compartida para procesos de usuario. Este último fue incluido últimamente en el ciclo de desarrollo; se penso que varios de los diseños originales no haría falta usarlos, pero estos impulsarían a un cambio de memoria por su primer cliente.

PERSPECTIVAS Y PASOS A SEGUIR

Las perspectivas de este proyecto son crear una fuente bibliográfica tanto en la programación concurrente como en la programación paralela (en español) para poder ampliarlas a un más en un futuro, así como ampliar el conocimiento sobre el lenguaje de programación PVM que nos permite crear programas paralelos eficientes, es decir aprender a manejar las librerías con las que cuenta dicho lenguaje para crear nuevas bibliotecas de ejercicios, que nos servirán para aplicaciones futuras, que nos permitirán crear programas paralelos tan grandes como se requieran.

Para poder llevar a cabo lo anterior, será necesario obtener información para obtener un panorama general de lo que es la programación concurrente y la programación paralela. Así como el sistema operativo (en este caso será UNIX) y el lenguaje paralelo (PVM) que se debe utilizara para crear programas paralelos, y familiarizarnos con ellos, es decir aprender a manejar los recursos que nos proporciona tanto el sistema Operativo como el lenguaje de programación para nuestros propósitos. Después se empezaran a crear las nuevas bibliotecas de ejercicios (paralelos) escritos en el lenguaje paralelo correspondiente. Finalmente se creara una guía para poder ejecutar los programas paralelos que fueron creados con anterioridad.

CONCLUSIONES:

Después de la investigación hecha, podemos decir que la programación concurrente tiene un gran campo de aplicaciones, así como muy buenos lenguajes para llevar a cabo un buen programa concurrente, pero desafortunadamente el crecimiento tecnológico (en lo que respecta no solo al hardware , sino tambien del software) han hecho que este tipo de programación llegue a ser cambiada en un futuro no muy lejano por la programación en paralelo, que tiene sus bases en la programación concurrente (es decir, podemos aplicar nuestros conocimientos en programaciòn concurrente para crear algoritmos paralelos, pero esto no quiere decir que el algoritmo paralelo sea eficiente, así que habrá que buscar la manera de mejorarlo), pero con más aplicaciones que facilitan el uso de sus lenguajes.

Con esto no se quiere decir que la programación concurrente no sea buena, al contrario es tan buena que gracias ha ella fue posible pensar en crear una programación paralela, que precisamente funda algunas de sus bases en la programación concurrente, pero a la cual se le han agregado un sin fin de características que la han hecho una herramienta muy poderosa. Pero claro no todos los problemas de programación se resuelven con la programación paralela y no todos con la programación concurrente, es por esto que antes de tomar una decisión sobre que tipo de programación usar es necesario analizar el problema, para así poder dar una mejor posible solución a el.

Por lo tanto podemos ver que tanto la programación concurrente como la programación paralela son dos herramientas muy poderosas en el medio de la programación bajo sistemas distribuidos, y que ambas tienen ventajas y desventajas que hay que saber aprovechar y desechar en su momento.

BIBLIOGRAFIA:

1. Complexity of Sequential and Parallel Numericall Algorithms
   J . F Traub
   Academic Press
   New York and London 1973.
   Biblioteca nacional de C.U
   colocaciòn 519.4
   SYM.C

2. Highly Parallel Computing
   George S. Almasi / Allan Gottlieb
   The Benjamin / Cummings Publishing Company, Inc.
   Segunda ediciòn.
   Biblioteca de la UAMI
   colocaciòn: QA16.642
   A4.6
   1994.

3. Sistemas Operativos Modernos
   Andrew S. Tanenbaum
   Prentice Hall Hispanoamericana,S;A
   Biblioteca de UPIICSA
   colocaciòn: 005.4
   T164.5
   EJ.12

4. Sistemas Operativos
   Francisco Rueda
   McGraw - Hill
   Biblioteca de UPIICSA
   colocaciòn: 005.43
   R917.5
   EJ.1

UNIVERSIDAD AUTONOMA METROPOLITANA

IZTAPALAPA

CASA ABIERTA AL TIEMPO

PROYECTO DE INVESTIGACION II

TITULO:

APOYO A LA CONSTRUCCION DE FUENTES BIBLIOGRAFICAS SOBRE LA PROGRAMACION PARALELA. (SEGUNDA PARTE)

INDICE

OBJETIVOS:

1. Presentar una introducción sobre PVM , es decir, que elementos lo conforman y que tan importantes son, así como el desempeño de éste. Para tener una ligera idea de como interactuan cada una de las partes que conforman PVM.

2. Citar los comandos usados por la consola y el hostfile de PVM, así como una descripción de su uso.

3. Describir brevemente al demonio de PVM (*pvmd*), la Tabla de Hosts y la biblioteca de programación.

4. Desarrollar un pequeño manual de usuario que indique la forma de trabajar con el lenguaje de programación PVM, es decir, la manera de crear, compilar y ejecutar un programa escrito en este lenguaje.

5. Crear algunos programas escritos en este lenguaje.

6. Dar una breve descripción de la función que realizan los programas.

7. Anexar un manual con las funciones de biblioteca de PVM.

1. INTRODUCCION

PVM

PVM es un paquete de software (conjunto de librerías) para crear y ejecutar aplicaciones concurrentes o paralelas, basado en el modelo de **paso de mensajes**. Funciona sobre un conjunto **heterogéneo** de ordenadores con sistema operativo **UNIX** conectados a una o más redes, y que permite diferentes arquitecturas de ordenadores así como redes de varios tipos (Ethernet, FDDI, etc.). Los programas de aplicación son escritos en **C** y son un acceso proporcionado a PVM con el uso de llamadas a rutinas de biblioteca de PVM para las funciones tales como lanzamiento, transmisión y recepción de mensajes, y sincronización de procesos vía barreras o rendenvous. Las aplicaciones pueden controlar opcionalmente la localización de la ejecución de los componentes específicos de la aplicación. El sistema transparente de PVM maneja el encaminamiento del mensaje, la conversión de datos para las configuraciones incompatibles, y otras tareas que sean necesarias para la operación en un ambiente heterogéneo de la red.

PVM es determinado eficaz para las aplicaciones heterogéneas que explotan fuerzas específicas de máquinas individuales en una red. El sistema de PVM se ha utilizado para las aplicaciones tales como simulaciones moleculares de la dinámica, estudios de la superconectividad, problemas computacionales como fractales distribuidos, algoritmos de matrices, la ordenación de arreglos, la solución a una suma de N números, por mencionar algunos, y en el salón de clase como la base para enseñar computación simultánea.

Bajo PVM un usuario define una colección de computadoras con un solo procesador para simular una gran memoria distribuida; proporciona las funciones para iniciar tareas en una máquina virtual (El término Máquina Virtual es usado para identificar a una computadora lógica de memoria distribuida) y permite a éstas sincronizarse y comunicarse.

Una tarea se define en PVM como una unidad de cómputo, análoga a un proceso UNIX. Las aplicaciones en **C** pueden ser paralelizadas utilizando un código en común de paso de mensajes. Múltiples tareas de aplicación pueden cooperar para resolver un problema en paralelo pasando y recibiendo mensajes.

PVM proporciona funciones para explotar mejor la arquitectura de cada computadora en la solución de problemas, maneja todas las conversiones de datos que pueden ser requeridas si dos computadoras usan diferentes representaciones de punto flotante ó enteros. Además, permite a la máquina virtual ser interconectada por una variedad de redes diferentes.

El sistema PVM está compuesto por dos partes:

La primera parte del sistema la conforma un demonio (son procesos que siempre están ejecutándose) llamado *pvmd3*, algunas veces abreviado como *pvmd*.

* El proceso UNIX es el encargado de controlar el funcionamiento de los procesos de usuario en la aplicación PVM y de coordinar las aplicaciones.
* El demonio reside y se ejecuta en cada las máquina configurada en la máquina virtual paralela.
* Cada demonio mantiene una tabla de configuración e información de los procesos relativos a su máquina virtual.
* Los procesos de usuario se comunican unos con otros a través de los demonios.
  Primero se comunican con el demonio local vía la librería de funciones de interface.
  Luego el demonio local manda/recibe mensajes de/a demonios de hosts remotos.
* Cada máquina debe tener su propia versión de **pvm** instalada y construida de acuerdo con su arquitectura y además accesible a **$PVM_ROOT.**

Antes de correr una aplicación en PVM, el usuario invoca archivos *pvmd* dentro de su estación, y el primer turno es para el *pvmd* que se encuentra hasta arriba de la lista de la computadora con los nombres de los archivos. Esta colección de *pvmd* es definida en la configuraciòn (PVM) para el usuario. El usuario puede correr ahora programas de aplicación PVM.

La segunda parte de el paquete PVM son las librerías de rutinas de interface (*libpvm.a, libfpvm.a & liggpvm.a*) y son alrededor de 20 rutinas de interfase.

* ***libpvm.a*** - Librería en lenguaje C de rutinas de interface: Son simples llamadas a funciones que el programador puede incluir en la aplicación paralela. Proporciona la capacidad de:
  ⇒ iniciar y terminar procesos.
  ⇒ pack, send and receive mensajes.
  ⇒ sincronizar mediante barreras.
  ⇒ conocer y dinámicamente cambiar la configuración de la máquina virtual paralela.

* las rutinas de librería no se conectan directamente con otros procesos, si no que mandan y reciben la configuración de la máquina virtual paralela.
* ***libgpvm.a*** - Se requiere para usar grupos dinámicos.
* ***libfpvm.a*** - Para programas en Fortran.

Los programas de aplicación deben ser ligados con esta biblioteca para poder utilizar PVM.

Los programas de aplicación son muy fáciles con las librerías de aplicación en PVM. El Manager/ Worker o host/node son modelos de programación paralelo iguales en PVM, pero se pueden usar tambien otros modelos, sin embargo los procesos PVM pueden inicializar procesos en otras máquinas. Usando procesos de comunicación el demonio *pvmd*, los empaqueta y manda al *pvmd* local, que determina la máquina donde se correrán y transmitirá el paquete a la maquina que tiene el *pvmd*, y que los repartirá dentro de los procesos remoto. Allí se suministraran tambien para convertir los datos a XDR (external data standard), además PVM es designado para medios heterogéneos. Como podemos esperar, transparencia y heterogeneidad son costos que someten y atrasan la comunicación. En PVM son estados representativos de trabajo en las LAN con un orden de 10 milisegundos, pero el mismo con llamadas a procedimientos remoto en el mismo ambiente (IP).

Además, PVM sirve para liberar el código en curso, con aumentos atractivos y tratos amplios (los documentos PVM y el código pueden ser obtenidos enviando un correo electrónico a netlibc@ornl.gov). En general PVM incluye asistencias para particionar algoritmos y registrarlos, el soporte de compilaciòn automática para diferentes arquitecturas, una interfaz gráfica para el usuario, un debugging y un trazo que facilita la ejecución.

LA CONSOLA DE PVM

La consola de PVM llamada *pvm*, es la única tarea PVM, que permite al usuario iniciar, preguntar y modificar el estado de la máquina virtual iterativamente. La consola puede iniciar y detener su ejecución múltiples veces en cualquiera de los *host* que conforman la máquina virtual, sin afectar a PVM o cualquier aplicación que pueda estar ejecutándose.

Cuando inicia la consola *pvm*, comprueba si PVM ya se está ejecutando, y si no automáticamente ejecuta *pvmd* en este host, pasando a *pvmd* las opciones de la línea de comandos y el archivo *hostfile*. De esta forma, PVM no necesita estar ejecutándose para que se pueda inicializar la consola.

Una vez inicializada la consola, imprime el prompt :

pvm >

y acepta comandos desde la entrada estándar (desde el teclado).

Los comandos disponibles en la consola son:

add Seguido por uno o más nombres de *host*, agrega estos a la máquina virtual.

alias Define o lista los alias de comandos.

conf Lista la configuración de la máquina virtual, incluyendo nombre del *host*, identificador de la tarea *pvmd* y tipo de arquitectura.

delete Seguido por uno o más nombres de *hosts*, los elimina de la máquina virtual. Los procesos PVM ejecutándose en el *hosts* eliminados son perdidos.

echo Repite argumentos.

halt Termina todos los procesos PVM incluyendo la consola y los demonios, deteniendo la ejecución de PVM.

help Puede ser usado para obtener información acerca de cualquiera de los comandos de la consola. La opción *help* seguida por un comando de la consola *pvm*, lista todas las opciones y banderas disponibles para éste comando.

id Imprime en la consola el ID de la tarea.

jobs Lista los procesos que se ejecutan en la máquina virtual.

kill Termina cualquier proceso PVM .

mstat Muestra el estado de los *host* especificados.

ps -a Lista todos los procesos que se encuentran actualmente en la máquina virtual, sus direcciones, sus ID´s de tareas, los ID´s de sus tareas padre.

pstat Muestra el estado de un sólo proceso PVM .

quit Sale de la consola dejando a los demonios y tareas PVM en ejecución.

reset Elimina todos los procesos PVM excepto la consola, reinicia todas las tablas internas y colas de mensajes PVM . Loes demonios son dejados en un estado estacionario.

setenv Despliega o coloca variables de ambiente.

sig Seguido por un número de señal y *tid*, envía la señal a la tarea con el *tid* correspondiente.

spawn Inicializa una aplicación PVM . Las opciones incluyen:

-*count* Número de tareas, por default 1.
-(*host*) Se produce en un *host*, por default cualquiera.
-(PVM_ARCH) se produce en un *hosts* de tipo PVM_ARCH.
-? Habilita depuración.
-> redirecciona la salida a una tarea a la consola.
->*file* Redirecciona la salida de una tarea a un archivo especificado en *file.*
->>*file* Redirecciona la salida de un atarea anexándola a un archivo especificado en *file.*

version Imprime la version de *libpvm* que es usada.

unalias Deja de definir un alias de comando.

La consola lee el archivo $HOME/*.pvmrc* antes de leer comandos desde el tty (tipo de terminal), así puede hacer cosas como:

```
alias h help
alias j jobs
$ imprime mi id
echo nuevo shell pvm
id
```

Los dos métodos más comunes para ejecutar PVM son: iniciar la consola *pvm* y agregar hosts manualmente (*pvm* también acepta un argumento de nombre *host* opcional), ó iniciar *pvmd* con un *hostfile.*

EL ARCHIVO HOSTFILE

El archivo *hostfile* define la configuración inicial de los *hosts*, que **PVM** combina para crear la máquina virtual. Este archivo también contiene información acerca de los *hosts* que el usuario desee agregar posteriormente a la configuración.

Cada usuario de PVM podrá tener su propio *hostfile*, el cual describe su propia máquina virtual personal.

El archivo *hostfile*, en su forma más sencilla, es solo una lista de nombres de *hosts*, un nombre por línea. Las líneas en blanco son ignoradas, y las líneas que comienzan con un # son líneas de comentario.

Estos símbolos (#) permiten al usuario comentar su *hostfile* y también proporcionar una forma rápida de modificar la configuración inicial.

Un ejemplo de una configuración, se ve a continuación:

 # mi primer configuración
tlalloc4
tlalloc2
xanum.uam.mx
alpha.cs.cinvestav.mx
xhara2.uam.mx
hermes.cs.uh.edu

Las máquinas que pertenecen a la misma red local, no es necesario especificar su dirección completa.

Existen varias opciones que pueden ser especificadas en cada línea después del nombre del *host* .Las opciones son separadas por un espacio en blanco.

Opciones:

lo = userid Permite al usuario especificar un login alternativo para este *host*; de otra manera, se usará el login inicial de la máquina.

so = pw Causará que PVM pida al usuario accese a este *host* con su *password*. Esto es útil en el caso donde el usuario tenga un UID (número entero que identifica un usuario y es asignado por el root) y un *password* distinto en el sistema remoto. PVM usa *rsh* por default para inicializar demonios *pvmd* remotos, pero cuando *pw* es especificado, PVM usará *rexec()* en vez de *ssh*.

dx = location_of_pvmd Permitirá al usuario especificar una dirección distinta a la default de *pvmd* para este *host*. Este es útil si alguien va a usar su propia copia personal de *pvmd*.

ep = paths_to_user_executables Esta opción permite al usuario especificar una serie de rutas de acceso para buscar sus archivos ejecutables. Múltiples rutas son separadas por dos puntos. Si ep = no es especificado, entonces PVM busca las tareas de aplicaciones en $HOME/pvm/bin/PVM_ARCH.

sp = value Especifica la velocidad relativa del *host* comparada con otro en la configuración. El rango de posibles valores es 1 a 1'000,000 con 1 000 como default.

bx = location_of_debugger Especifica cual es el script (archivo que contiene un conjunto de instrucciones que son ejecutadas por el shell) de depuración que se invocará en este *host* si la depuración es requerida en la rutina *spawn*. Por default el depurador está en *pvm/lib/debugger*.

wd = working_directory Especifica un directorio de trabajo en el cual todas las tareas producidas en este *host* se ejecutarán. El default es $HOME.

so = ms Especifica que el usuario iniciará manualmente un *pvmd* esclavo en este *host*. Es útil si los servicios de red *rsh* y *rexec()* están deshabilitados, pero existe conectividad IP.

Si el usuario desea colocar cualquiera de las opciones citadas como default para un conjunto de *hosts*, entonces puede colocar estas opciones en una línea con un * al principio. Las opciones default tendrán efecto para todos los *hosts* siguientes hasta que haya otra línea default.

Los *host* que el usuario no quiere agregar en la configuración inicial, pero que serán utilizados posteriormente, pueden ser especificados en el *hostfile* poniendo al principio de estas líneas un **&**.

Un ejemplo del desplegado de *hostfile* y más de estas opciones son mostradas a continuación:

```
#Las líneas de comentarios son iniciadas con #
#Las líneas en blanco son ignoradas
      gstws
      ipsc  dx = /usr/geist/pvm/lib/I860/pvm
      ibmi.scri.fsu.edu  lo = gst  so = pw


#Las opciones default son colocadas con * para los siguientes host
    * ep = $sun/problem1:  ~/nla/mathlib
     sparky
    #azure.epm.ornl.gov
    midnight.epm.ornl.gov


#reemplaza las direcciones default con los nuevos valores para los hosts
  * lo = gageist   so = pw    ep =  problem1
    thud.cs.utk.edu
```

```
   speedy.cs.utk.edu

#Las máquinas por agragar posteriormente, son especificadas con &
#estas sólo necesitan listarse si se requieren.
  & sun4  ep = problem1
  &castor  dr = /usr/local/bin/pvmd
  &dasher.cs.utk.edu  lo = gageist
  &elvis  dr = ~/pvm/lib/SUN4/pvmd
```

LIMITACIONES DE RECURSOS

Las limitaciones de recursos impuestas por el Sistema Operativo y el Hardware son las mismas para las aplicaciones PVM. Siempre que sea posible, PVM evita colocar límites explícitos, en vez de esto regresa un mensaje de error cuando los recursos se han terminado. Naturalmente, la competencia de usuarios en el mismo *host* y en la red afectan algunos límites dinámicamente.

EL DEMONIO PVM

El número de tareas que cada *pvmd* puede manejar está limitado por dos factores: el número de procesos permitidos a cada usuario por el Sistema Operativo, y el número de archivos descriptores en el *pvmd.*

El *pvmd* puede llegar a un embotellamiento si todas las tareas tratan de comunicarse unas con otras, a través de él.

El *pvmd* usa asignación de memoria dinámica para almacenar paquetes, en la ruta entre tareas, hasta que la tarea receptora acepta los paquetes estos son acumulados por el *pvmd* en una cola de *Primeras Entradas, Primeras Salidas* (First Input First Output, FIFO). El control de flujo no es impuesto por el *pvmd*, éste almacenará todos los paquetes dados a él, hasta que no pueda obtener más memoria.

Una vez que el *pvmd* es ejecutado en cada *host* de la máquina virtual y los *pvmds* son configurados para trabajar juntos. Los *pvmds* de un usuario no interactúan con los de otros. El *pvmd* fue diseñado para ejecutarse bajo un UID no privilegiado que ayude al usuario a reducir los riesgos de seguridad y para minimizar el impacto de un usuario PVM con los demás.

El demonio *pvmd* sirve como ruteador y controlador de mensajes. Esto proporciona un punto de contacto entre cada uno de los *host*, así como autentificación, control de procesos y detección de fallas. Los demonios inactivos, ocasionalmente mandan un mensaje a los demás demonios, para verificar su alcance; cuando un *pvmd* no responde es marcado como deteriorado. Los *pvmds* son más resistentes que los componentes de una aplicación y continúan ejecutándose si el programa es interrumpido, facilitando la depuración.

El primer *pvmd* (iniciado manualmente) es llamado el *pvmd* maestro, mientras los otros (iniciados por el maestro) son llamados *pvmds* esclavos. En la mayoría de las operaciones, todos los *pvmds* son considerados iguales. Únicamente el maestro puede iniciar nuevos *pvmds* esclavos y adicionarlos a la configuración de la máquina virtual. De la misma forma, sólo el maestro puede eliminar esclavos de la máquina virtual. Si el *pvmd* maestro pierde contacto con un esclavo, éste marca al esclavo como dañado y lo suprime de la configuración. Si un esclavo *pvmd* pierde contacto con el maestro, el esclavo por sí mismo se dará de baja. Este algoritmo asegura que la máquina virtual no llegue a ser particionada y continué ejecutándose como dos máquinas virtuales.

Las estructuras de datos más importantes son las Tablas de *hosts* y las tablas de tareas, las cuales describen la configuración de la máquina virtual y la localización de las tareas ejecutándose bajo *pvmd.* Además de ellas están las colas de mensajes y paquetes, y contextos de espera para almacenar información del estado de las multitareas en el *pvmd.*

Cuando se inicia el *pvmd* se configura así mismo, ya sea como un maestro o como un esclavo, dependiendo de sus argumentos en la línea de comandos. Al momento de que *pvmd* crea y liga sockets para comunicarse con las tareas y con otros *pvmds*, abre un archivo de errores e inicializa las tablas. Para un *pvmd* maestro, la configuración puede incluir lectura del hostfile y determinación de parámetros default, tales como el nombre del host (*hostname*). Un *pvmd* esclavo obtiene sus parámetros de la línea de comandos y envía una línea de datos de regreso al proceso inicial para incluirse en la tabla de *hosts.*

Después de configurarse así mismo el *pvmd* entra en un ciclo en la función *work().* El corazón del ciclo *work()* es una llamada a *select()* que prueba todos los recursos de entrada para el *pvmd* (tareas locales y la red). Los paquetes que llegan son recibidos y enrutados a sus destinos. Los mensajes direccionados a *pvmd* son redireccionados a las funciones *loclentry(),netentry() ó schedentry().*

¿POR QUÉ USAMOS PVM?

Una de las razones mas importantes para usar PVM, es la fácil implantación y uso. PVM no requiere la instalación de privilegios especiales. Cualquier persona con un login válido en el anfitrión puede utilizarlo, además, solo una persona en la organización necesita la instalación de PVM para que todos en esta lo puedan usar.

PVM utiliza dos variables ambientales al inicializar y correr. Cada usuario de PVM necesita instalar estas dos variables para poder utilizar PVM. La primer variable es PVM_ROOT, la cual se instala en el directorio donde se encuentra el PVM. La segunda variable es PVM_ARCH, la cual le indica a PVM la arquitectura de este anfitrión. La fuente de PVM viene con directorios y "makefiles" para la mayoría de las arquitecturas.

2. MANUAL DE PVM
(Pasos a seguir para crear, compilar y ejecutar un programa en PVM)

Para programar en PVM hay que tener en cuenta :

- Arquitectura de las maquinas que componen mi gran maquina virtual. Esto es para poder aprovechar las técnicas de optimización propias de cada maquina.
- Velocidad de las componentes de la gran maquina virtual, por ejemplo una estación de trabajo vs una supercomputadora, en este caso hay que tratar que la supercomputadora no tenga que esperar mucho tiempo a recibir el siguiente dato para poder continuar.
- Evitar correr un programa a horas pico, es decir que una estación de trabajo este muy cargada o bien que la red este muy lenta.

Problemas Comunes al Inicializar PVM

Existen algunos problemas bastante comunes al iniciar PVM. aquí mencionaremos las mas fáciles de identificar. En caso de que exista un problema PVM advertirá al usuario del error encontrado. En la mayoría de los casos simples el error se debe a la ausencia de los archivos PVM_ROOT y PVM ARCH.

[t80040000]Can't start pvmd

Si aparece este mensaje se debe revisar que el archivo .rhost en el anfitrión remoto contenga el nombre de del anfitrión desde el que se esta accesando la máquina virtual. Otras razones para este mensaje pueden ser que PVM no se encuentre instalado en algún anfitrión.

[t80040000]Login incorrect

Este mensaje indica que el anfitrión remoto no tiene una cuenta con el nombre de usuario que se esta utilizando.

Después de la recomendación anterior sobre PVM , procederemos a elaborar el manual de usuario, que nos indicara la forma de crear, compilar y ejecutar un programa en éste lenguaje.

Antes de compilar y correr programas realizados para operar en PVM comenzaremos por decir que estando dentro del directorio de PVM ,(en el cuál se encontrarán los directorios **bin** (dentro de este directorio se encuentra otro llamado **SUN4SOL2** que es donde se encontraran los archivos ejecutables creados) y **ejemplos** (donde guardaremos nuestros programas que creamos)) , debemos estar seguros de que podemos inicializar PVM para después configurar la máquina virtual. Lo hacemos de la siguiente forma:

```
 $

/* Ahora tecleamos lo siguiente, para inicializar el demonio */
  $pvm
  pvm>


/*Checamos que máquinas esta dada de alta en el host */
  pvm>conf
        1hosts, 1 data format

        HOST      DTID      ARCH       SPEED
        uam1      40000     SUN4SOL2   1000


/* Después damos de alta las demás máquinas que se desea adicionar, como sigue: */
  pvm> add  uam3
        1 successful          Host     DTID
                              uam3     c0000

  pvm> add  uam4
        1 successful          Host     DTID
                              uam4     100000
/* y así sucesivamente con cada una de las máquinas que se quieran adicionar. */


/* Para ver si las  máquinas se adicionaron bien usamos el siguiente  comando que */
/* se utiliza para revisar la manera en que la máquina virtual  esta  funcionando */
/* en ese preciso momento. */
  pvm> conf
        3hosts, 1 data format

        HOST      DTID      ARCH       SPEED
        uam1      40000     SUN4SOL2   1000
        uam3      c0000     SUN4SOL2   1000
        uam4      100000    SUN4SOL2   1000


/*En caso de querer borrar un host de la máquina virtual */
/*usamos el siguiente comando */

  pvm>delete uam1
```

```
/*Para ver que tareas se están ejecutando en la máquina virtual, hacemos */
/* lo siguiente */

  pvm>pvmps -a
```

Cabe mencionar que basta dar de alta los host en el demonio de PVM una sola vez en una sola máquina, ya que si intentamos dar de alta el mismo host en otra máquina, esto no podrá ser posible, esto es:

```
/* Si estamos en uam3 y adicionamos, ya sea la uam1 ó la uam4 ocurrirá lo siguiente: */
  pvm> add uam1
        0 successful
                        HOST      DTID
                        uam1      Duplicate host
/* lo que nos indica que el host ya fue dado de alta anteriormente en alguna otra máquina */
```

Por lo anterior, cabe mencionar que cuando comencemos a trabajar en una máquina, hay que revisar primero que máquinas fueron dadas de alta con el comando **conf.**

En este momento podemos usar cualquiera de los siguientes comandos según sea nuestro propósito.

```
/* Con el siguiente comando nos salimos de la consola dejando los demonios y */
/* tareas PVM en ejecución. Este es el comando que usaremos a la hora en que */
/* comencemos a trabajar con nuestros programas */

  pvm>quit
        pvmd still running.
  $


/* Con el siguiente comando se terminan todos los procesos PVM incluyendo la */
/* consola y los demonios, deteniendo la ejecución de PVM. Este comando lo */
/* usaremos cuando terminemos de trabajar con PVM */

  pvm>halt
  $
```

3. FORMA DE CREAR Y COMPILAR LOS PROGRAMAS.

Antes que nada aclararemos que el programa se debe teclear en un editor de textos con formato ASCII, y en nuestro caso el editor que se utilizo es **Vi** porque (pero se puede utilizar otro llamado **pico,** que son los que utiliza UNIX) . Después de editar nuestro programa, lo guardaremos en el direcctorio ejemplos. El siguiente paso es compilarlo, para lograr esto es necesario que dentro del directorio ejemplos se encuentre el archivo **Makefile.aimk** en el cual se encuentran las rutas de los archivos de las librerias, el nombre del compilador a usar y el directorio donde se guardaran los ejecutables, pero sobre todo la declaración del programa que creamos. Para esto mostraremos el archivo **Makefile.aimk** y la parte donde se insertara la declaración de el programa a compilar.

```
#
# Makefile.aimk for PVM example programs.
#
# Set PVM_ROOT to the path where PVM includes and libraries are installed.
# Set PVM_ARCH to your architecture type (SUN4, HP9K, RS6K, SGI, etc.)
# Set ARCHLIB to any special libs needed on PVM_ARCH (-lrpc, -lsocket, etc.)
# otherwise leave ARCHLIB blank
#
# PVM_ARCH and ARCHLIB are set for you if you use "$PVM_ROOT/lib/aimk"
# instead of "make".
#
# aimk also creates a $PVM_ARCH directory below this one and will cd to it
# before invoking make - this allows building in parallel on different arches.
#

SDIR  =     ..
#BDIR=      $(HOME)/pvm3/bin
BDIR  =     $(SDIR)/../bin
#Archivo donde se guardaran los archivos ejecutables
XDIR  =     $(BDIR)/$(PVM_ARCH)

OPTIONS     =     -O
CFLAGS      =     $(OPTIONS) -I../../include $(ARCHCFLAGS)

LIBS  =     -lpvm3 $(ARCHLIB)
GLIBS=      -lgpvm3

#F77   =     f77
```

```
FORT = `case "$(FC)@$(F77)" in *@) echo $(FC) ;; @*) echo $(F77) ;; *) echo f77;;
esac`
FFLAGS      =       -g $(ARCHFFLAGS)
FLIBS =     -lfpvm3
LFLAGS      =       $(LOPT) -L../../lib/$(PVM_ARCH)

/* Se declara el nombre de el (los) programa(s) que se va a compilar según el lenguaje */
/*  que se utilizo para crearlo, ya sea C o Fortran. */
CPROGS      =       hello hello_other
FPROGS      =       fgexample fmaster1 fslave1 fspmd hitc hitc_slave testall

/* Programas por default */
default:    hello hello_other

all: c-all f-all

c-all:  $(CPROGS)

f-all:  $(FPROGS)

clean:
        rm -f *.o $(CPROGS) $(FPROGS)

$(XDIR):
        - mkdir $(BDIR)
        - mkdir $(XDIR)



/* Esta es la parte y la forma en la que se deberán declarar todos los programas que /*
/* creemos, para  que después puedan ser compilados */

hello: $(SDIR)/hello.c $(XDIR)
        $(CC) $(CFLAGS) -o $@ $(SDIR)/hello.c $(LFLAGS) $(LIBS)
        mv $@ $(XDIR)

hello_other: $(SDIR)/hello_other.c $(XDIR)
        $(CC) $(CFLAGS) -o $@ $(SDIR)/hello_other.c $(LFLAGS) $(LIBS)
        mv $@ $(XDIR)
```

Una vez hecho lo anterior procederemos a compilar el o los programas creados.

Si el programa creado consta de varios módulos separados, estos se podrán ligar desde la línea de comandos con la siguiente instrucción:

```
$ aimk hello hello_other
making in SUN4SOL2/ for SUN4SOL2 gcc -o -I ../../include - DSYSVSIGNAL -
DNOWAIT3 - DNOUNIXDOM - DRSHCOMMAND = \ "/usr/bin/rsh\" -o hello ../hello.c
-L ../../lib/SUN4SOL2 -lpvm3 -lnsl - lsocket mv hello ../../bin/SUN4SOL2
$

/* Si el programa fue compilado correctamente, únicamente aparecerá el prompt, en caso */
/* contrario, ira acompañado con algún mensaje de error */
```

Si algún archivo sufre alguna modificación y este esta ligado con otro u otros archivos, bastara con compilar únicamente el archivo modificado, ya que el **Makefile.aimk** se encargara de ligarlos automáticamente.

```
/* Si solo modificamos el archivo hello.c, bastara con compilar solo este archivo */
/* y no abra necesidad de volver a compilar el archivo con el que va ligado, ejemplo: */

$aimk hello
$
```

Cuando aparezca el prompt nos moveremos al directorio **bin** y dentro de este al directorio **SUN4SOL2** en el cual se encuentran los archivos ejecutables.

Ejemplo:

```
$cd ..                  /* para salirnos de el directorio ejemplos */
$cd bin                 /* para movernos al directorio bin */
$cd SUN4SOL2    /* para movernos al directorio SUN4SOL2 */
$ls                     /* desplegara todos los archivos ejecutables */
 hello
 hello_other
```

```
/* Ahora bastara con dar el nombre del archivo para que este pueda ejecutarse */
  $hello

/* Mostrara en pantalla los resultados, y luego aparecerá el prompt */
  $
```

Asta aquí hemos mostrado la forma de crear, compilar y ejecutar un programa para PVM. Ahora anexaremos los programas que se crearón así como una breve descripción de lo que hacen y los resultados de su ejecución.

4. APENDICE DE PROGRAMAS

PROGRAMA No. 1

Ejemplo de Kill

```
/* Ejemplo del uso de Kill . Demuestra como saber si las tareas han finalizado */
/* definiciones y prototipos de la librería PVM */

#include <pvm3.h>

/* Numero máximo de hijos que el programa engendrara */
#define MAXNCHILD  20

/* Etiqueta que usara la tarea para mandar el mensaje de finalización */
#define TASKDIED       11

int
main(int argc, char* argv[])
{
   /* numero de tareas a engendrar, utiliza 3 como default */
   int ntask = 3;
   /* código que regresa de las llamadas PVM */
   int info;
   /* mi identificador de tarea */
   int mytid;
   /* el identificador de tarea de mi padre */
   int myparent;
   /* arreglo de identificadores de tareas de los hijos */
   int child[MAXNCHILD];
   int i, deadtid;
   int tid;
   char *argv[5];

   /* se busca mi identificador de tarea */
   mytid = pvm_mytid();
   /* se verifica que no exista error */
   if (mytid < 0) {
      /* se imprime el error */
      pvm_perror(argv[0]);
/* se termina el programa */
      return -1;
```

```
    }
/* se busca el identificador de tarea de mi padre */
myparent = pvm_parent();

/* finaliza si existe algún error diferente de PvmNoParent */
if ((myparent < 0) && (myparent != PvmNoParent)) {
    pvm_perror(argv[0]);
    pvm_exit();
    return -1;
    }

/* si no tengo padre ... entonces yo soy el padre */
if (myparent == PvmNoParent) {
    /* cuantas tareas voy a engendrar? */
    if (argc == 2) ntask = atoi(argv[1]);
    /* se asegura que ntask es valido */
    if ((ntask < 1) || (ntask > MAXNCHILD)) { pvm_exit(); return 0; }

    /* se engendran las tareas */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDebug, (char*)0, ntask, child);

    /* nos aseguramos que no ocurrió un error */
    if (info != ntask) { pvm_exit(); return -1; }

    /* se imprimen los identificadores de tarea */
    for (i = 0; i < ntask; i++)
        printf("t%x\t", child[i]); putchar ('\n');

    /* se pide la notificación cuando un hijo termina */
    info = pvm_notify(PvmTaskExit, TASKDIED, ntask, child);
    if (info < 0) { pvm_perror("notify"); pvm_exit(); return -1; }

    /* matamos al hijo de en medio */
    info = pvm_kill(child[ntask/2]);
    if (info < 0) { pvm_perror("kill"); pvm_exit(); return -1; }

    /* se espera por una notificación */
    info = pvm_recv(-1, TASKDIED);
    if (info < 0) { pvm_perror("recv"); pvm_exit(); return -1; }
    info = pvm_upkint(&deadtid, 1, 1);
    if (info < 0) pvm_perror("calling pvm_upkint");

    /* debe de ser el hijo de en medio */
    printf("Task t%x has exited. \n", deadtid);
```

```
    printf("Task t%x is middle child. \n", child[ntask/2]);
    pvm_exit();
    return 0;
    }

  /* soy un hijo ... */
  sleep(63);
  pvm_exit();
  return 0;
}
```

PROGRAMA No. 2

Ejemplo de Fork Join

```
/* Ejemplo de Fork Join . Demuestra como engendrar proceso e intercambiar mensajes. */
/* definiciones y prototipos para la librería de PVM */

#include <pvm3.h>

/* Numero máximo de hijos que engendrara este programa */
#define MAXNCHILD 20
/* Etiqueta que usara el menaje de Join */
#define JOINTAG   11

int
main(int argc, char* argv[])
{
  /* numero de tareas que serán engendradas, se utiliza 3 como default */
  int ntask = 3;
  /* código que regresan las llamadas PVM */
  int info;
  /* mi identificador de tarea */
  int mytid;
  /* el identificador de tarea de mi padre */
  int myparent;
  /* un arreglo con los identificadores de tarea de los hijos */
  int child[MAXNCHILD];
  int i, mydata, buf, len, tag, tid;

  /* se busca mi identificador de tarea */
  mytid = pvm_mytid();
```

```
/* se evalúa si hay error */
if (mytid < 0) {
   /* si hay error, se imprime */
   pvm_perror(argv[0]);
   /* fin del programa .. bye! */
   return -1;
   }
/* se busca el identificador de tarea de mi padre */
myparent = pvm_parent();
/* finaliza si encuentra algún error diferente de PvmNoParent */
if ((myparent < 0) && (myparent != PvmNoParent)) {
   pvm_perror(argv[0]);
   pvm_exit();
   return -1;
   }

/* si no tengo un padre ... yo soy el padre */
if (myparent == PvmNoParent) {
   /* cuantas tareas se van a engendrar? */
   if (argc == 2) ntask = atoi(argv[1]);
   /* aseguramos que ntask sea correcto */
   if ((ntask < 1) || (ntask > MAXNCHILD)) { pvm_exit(); return 0; }

   /* engendramos las tareas de los hijos */
   info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0,
      ntask, child);
   /* se imprimen los identificadores de los hijos */
   for (i = 0; i < ntask; i++)
      if (child[i] < 0) /* imprimir el código de error en decimal */
         printf(" %d", child[i]);
      else /* imprimir el identificador en hexadecimal */
         printf("t%x\t", child[i]);
   putchar('\n');
   /* Asegurarnos de que se engendro con éxito */
   if (info == 0) { pvm_exit(); return -1; }
   /* Solo se esperaran respuestas de los hijos que se engendraron correctamente */
   ntask = info;

   for (i = 0; i < ntask; i++) {
      /* recibe un mensaje de cualquier proceso hijo */
      buf = pvm_recv(-1, JOINTAG);
      if (buf < 0) pvm_perror("calling recv");
      info = pvm_bufinfo(buf, &len, &tag, &tid);
```

```
        if (info < 0) pvm_perror("calling pvm_bufinfo");
        info = pvm_upkint(&mydata, 1, 1);
        if (info < 0) pvm_perror("calling pvm_upkint");
        if (mydata != tid) printf("This should not happen! \n");
        printf("Length %d, Tag %d, Tid t%x\n", len, tag, tid);
        }
      pvm_exit();
      return 0;
      }

  /* soy un hijo ... */
  info = pvm_initsend(PvmDataDefault);
  if (info < 0) {
    pvm_perror("calling pvm_initsend"); pvm_exit(); return -1;
    }
  info = pvm_pkint(&mytid, 1, 1);
  if (info < 0) {
    pvm_perror("calling pvm_pkint"); pvm_exit(); return -1;
    }
  info = pvm_send(myparent, JOINTAG);
  if (info < 0) {
    pvm_perror("calling pvm_send"); pvm_exit(); return -1;
    }
  pvm_exit();
  return 0;
}
```

La siguiente figura muestra la salida de correr FORK - JOIN. Nótese que el orden en que se recibieron los mensajes no es deterministico. Dado que el ciclo principal del proceso padre es del tipo FIRST-COME FIRST-SERVE , el orden de las salidas es determinado por el tiempo que toma a los mensajes viajar de las tareas hijo al padre.

```
% forkjoin
t10001c t40149 tc0037
Length 4, Tag 11, Tid t40149
Length 4, Tag 11, Tid tc0037
Length 4, Tag 11, Tid t10001c
% forkjoin 4
t10001e t10001d t4014b tc0038
Length 4, Tag 11, Tid t4014b
Length 4, Tag 11, Tid tc0038
Length 4, Tag 11, Tid t10001d
Length 4, Tag 11, Tid t10001e
```

PROGRAMA No. 3

Paralelización del proceso de cálculo para resolver integrales numéricas definidas.

Ejemplo: Estimación del número pi

La versión serie de este problema aproxima pi calculando pi = integral de 0 a 1 de 4/(1+x*x) dx que a su vez se calcula por la suma desde k=1 a N de los rectángulos de altura 4 / (1+((k-.5)/N)**2) y anchura w=1.0/N. El único parámetro de entrada es N y representa el número de subintervalos (rectángulos) empleados para calcular la integral

Familiarízate con el programa serie y entiende como

- ¿ Cómo calcula pi ?
- ¿ Cómo depende la precisión de N , el número de pasos de aproximación ? (ejecuta con varios valores de Input de 10 a 10000).
- Rutina de inicio : inicializar( )
  El propósito de esta subrutina es generar tareas paralelas, y asegurarse de que cada tarea conoce su ID de tarea, el de su padre, el número total de tareas de la aplicación y los ID de las otras tareas.
- Rutina de solicitud de parámetros de entrada : solicitar ( )
  Una tarea (el maestro) coge la entrada (el número de intervalos de aproximación) del usuario y distribuye este valor al resto de tareas.
- Programa principal : main ( )
  El programa principal se modificará de forma que el trabajo se distribuirá entre las tareas. También debe salir de PVM.
- Rutina de recepción de datos : recoger( )
  El maestro recogerá los resultados de las otras tareas, las combinará, e imprimirá el resultado final.

Este código se paralelizará en los pasos que se describen a continuación.

Paso I : Incluir el proceso en PVM, identificando el primer proceso ejecutado (Maestro)

Ii) Empezaremos incluyendo las instrucciones necesarias para ejecutar el programa serie bajo PVM. Primero el proceso tiene que inscribirse dentro de PVM (pvm_mytid) y

también debe liberar la comunicación(pvm_exit) con el programa residente (daemon) de PVM antes de salir.

Iii) Identificar la primera tarea paralela iniciada(pvm_parent) . Esta tarea será el 'maestro' y actuará de forma diferente que las otras tareas. Nótese que en este punto del tutorial sólo tenemos una tarea, que es la que actuará como maestro .

Paso II : Crear procesos esclavos y redireccionar su salida hacia el maestro

IIi) Desde el maestro, se pregunta al usuario cuántas tareas adicionales deben arrancarse. Entonces el maestro crea las tareas paralelas, y (por ahora) basta con conseguir que todas las tareas hagan el mismo cálculo que en la versión serie.

IIii) El maestro redirecciona la salida ('stdout') de todas las tareas que inicia, de tal forma que todas las salidas de las tareas aparecerán en la ventana donde se ha iniciado el proceso. Si no se hace así, todas las tareas excepto la primera mandarán su salida a /tmp/pvml.<uid>. Sólo el maestro debería permitir la introducción de datos por el usuario.

Paso III : Comunicación con los procesos esclavos: Paso de Mensajes

IIIi) Mandar el primer mensaje entre procesos. El maestro debe compartir el valor de N que obtiene del usuario con el resto de las tareas.

Paso IV : División y Reparto del trabajo

IVi) Prepara el trabajo para dividirlo entre las tareas. Para hacer esto, cada tarea necesitará saber el número de tareas trabajadoras y sus índices en el array de ID's.

IVii) Divide el trabajo entre las tareas usando descomposición cíclica (o entrelazado), donde cada proceso ejecuta la iteración nº (#trabajadores - 1), empezando en su (índice +1). Como el cálculo correcto de pi requiere que se sumen los valores de cada tarea, y esto no pasa hasta el último paso, los valores devueltos por los programas en este paso serán inválidos.

Paso V : Recepción de resultados

Vi) En este momento, cada tarea ha calculado (e imprime) su suma y su error. Ahora, queremos que el maestro recoja las sumas de las otras tareas, calcule la suma total, calcule el error basado en las suma total, e imprima los resultados.

¡¡¡ ENHORABUENA !!!

Ahora tienes un programa paralelo.

Ejemplo de Paralelización usando PVM : pi.c

```
/* Declaración de las librerias a utilizar */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "pvm3.h"

/*Definición de las constantes */

#define f(x) ((double)(4.0/(1.0+x*x)))
#define pi ((double)(4.0*atan(1.0)))
#define MAXTASKS 32
void inicializar (int *pmytid, int *pparent_tid, int *pnworkers, int *index,
        int tids[MAXTASKS]);
int solicitar (int parent_tid, int nworkers, int tids[MAXTASKS]);
void recoger(double sum, int parent_tid, int nworkers);

int main()
{
 /* Este  programa aproxima pi calculando pi = integral de 0 a 1 de  4 / (1+x*x) dx que a su
 *  vez se calcula  por  la suma desde  k = 1 a  N de  los rectángulos de altura
 *  4 / (1+((k-.5)/N)**2).  y anchura w=1.0/N. El único parámetro de entrada es N y
 *  representa el numero de subintervalos (rectángulos) empleados para calcular la integral
 */
 double sum, w;
 int i, N, rc;
 int mytid,                 /*  ID de tarea */
 parent_tid;                /*  ID de tarea de la tarea padre */
 nworkers,                  /* numero de tareas adicionales iniciadas */
 index,                     /* índice de la tarea den el array de ID */
 tids[MAXTASKS];            /* identificadores de las tareas adicionales */

  /*  La rutina inicializar se encarga de crear los procesos  paralelos */
 inicializar(&mytid, &parent_tid, &nworkers, &index, tids);

 /*  La rutina solicitar pide el valor de N y lo propaga a los proceso en la versión paralelo */
```

```
 N = solicitar(parent_tid, nworkers, tids);

 while (N > 0)  {
  w = 1.0/(double)N;
  sum = 0.0;
  for (i = index+1; i <= N; i+=nworkers+1)
   sum = sum + f(((double)i-0.5)*w);
  sum = sum * w;
  /* La rutina recoger recibe e imprime el resultado */
  recoger(sum, parent_tid, nworkers);
  N = solicitar(parent_tid, nworkers, tids);
 }
 /* Liberar enlace con PVM antes de salir del programa principal */
 printf ("tarea %8x saliendo de PVM\n", mytid);
 rc = pvm_exit ();
 return (0);
}

/* ------------------------------------------------------------ */
void inicializar (int *pmytid, int *pparent_tid, int *pnworkers, int *pindex,
          int tids[MAXTASKS])
{
 int mytid, parent_tid, nworkers, index, rc, i;

 /*  Obtiene su ID de tarea con pvm_mytid( ) */
 mytid = pvm_mytid();
 printf ("tarea con ID %8x inscrita en PVM\n", mytid);
 *pmytid = mytid;

 /* Obtiene el identificador ID de la tarea padre.  PvmNoParent indicara la tarea maestro */
 parent_tid = pvm_parent();
 *pparent_tid = parent_tid;

 /* El maestro redirecciona la salida standard de los otros procesos */
 if (parent_tid == PvmNoParent)  {
  rc = pvm_catchout(stdout);

  /* ... y con spawn inicia la ejecución de los procesos */
  printf ("Cuantos procesos esclavos se van a emplear (1-%2d)?\n", MAXTASKS);
  scanf("%d", &nworkers);
 while ((nworkers > MAXTASKS)||(nworkers <= 0))  {
   printf("El numero de esclavos debe estar entre 1 y %2d...",
        MAXTASKS);
   printf("Prueba otro número! \n");
```

```
    scanf("%d", &nworkers);
   }
   *pnworkers = nworkers;
   rc = pvm_spawn("pi", NULL, PvmTaskDefault, NULL, nworkers, tids);
   if (rc != nworkers) {
    printf ("Error en spawn, solo %d\n tareas se han conseguido levantar", rc);
    exit(0);
   }
   printf ("%d procesos esclavos se han levantado \n", rc);

   /* ... asignarle su índice */
   index = 0;
   *pindex = index;


   /* ... y enviar con multicasts el numero de tareas y los ID de los tareas esclavos */
   rc = pvm_initsend(PvmDataDefault);
   rc = pvm_pkint(&nworkers,1,1);
   rc = pvm_pkint(tids,nworkers,1);
   rc = pvm_mcast(tids, nworkers, 222);
  }

  /* Los esclavos reciben y desempaquetan el mensaje enviados con multicast */
  else {
   rc = pvm_recv(parent_tid, 222);
   rc = pvm_upkint(&nworkers, 1, 1);
   *pnworkers = nworkers;
   rc = pvm_upkint(tids, nworkers, 1);
   /* ... y pone su índice en el array de identificadores de tarea */
   for (i = 0; i < nworkers; i++)
    if (mytid == tids[i]) index=i+1;
   *pindex = index;
  }
}

/* ------------------------------------------------------------ */
int solicitar (int parent_tid, int nworkers, int tids[MAXTASKS])
{
 int N, rc;

 /* Solo el maestro pide información de entrada al usuario */
 if (parent_tid == PvmNoParent) {
   printf ("Introduce numero de subintervalos (un valor mayor mejora la aproximación y 0
termina)\n")   scanf("%d",&N);
```

```
   /*  ... y con un multicasts comunica esta entrada al resto de procesos  */
   rc = pvm_initsend(PvmDataDefault);
   rc = pvm_pkint(&N,1,1);
   rc = pvm_mcast(tids, nworkers, 111);
  }

  /*  El proceso esclavo recibe el mensaje del maestro  */
  else  {
   rc = pvm_recv(parent_tid, 111);
   rc = pvm_upkint(&N, 1, 1);
  }
  return (N);
}
/*  ------------------------------------------------------------  */
void recoger(double sum, int parent_tid, int nworkers)
{
  double err, x;
  int i, rc;

  /*  El maestro recibe la suma de cada esclavo y guarda el total  */
  if (parent_tid == PvmNoParent)  {
   for (i=0; i<nworkers; i++)  {
     rc = pvm_recv(-1, 333);
     rc = pvm_upkdouble(&x, 1, 1);
     sum = sum + x;
   }

   /*  ... calcula una estimación del error cometido e imprime el resultado  */
   err = sum - pi;
   printf("pi = ,   error = %7.5f, %10e\n", sum, err);
  }

  /*  Los esclavos mandan su suma al maestro  */
  else  {
   rc = pvm_initsend(PvmDataDefault);
   rc = pvm_pkdouble(&sum, 1, 1);
   rc = pvm_send(parent_tid, 333);
  }
}
```

PROGRAMA No. 4

Programa Hello.c

/* Este programa junto con hello_other se encarga solamente de enviar y regresar el mensaje: hello, world from , así como el número de la maquina que esta contestando el mensaje */

```
#include <stdio.h>
#include "pvm3.h"

main()
{
        int cc, tid;
        char buf[100];

        printf("i'm t%x\n", pvm_mytid());

        cc = pvm_spawn("hello_other", (char**)0, 0, "", 2, &tid);

        if (cc == 1) {
                cc = pvm_recv(-1, -1);
                pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
                pvm_upkstr(buf);
                printf("from t%x: %s\n", tid, buf);

        } else
                printf("can't start hello_other. El valor de regreso es cc=%d\n",cc \n");

        pvm_exit();
        exit(0);
}
```

Programa Hello_Other.c

```
#include "pvm3.h"

main()
{
        int ptid;
        char buf[100];
```

```
        ptid = pvm_parent();
        strcpy(buf, "hello, world from ");
        gethostname(buf + strlen(buf), 64);

        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(ptid, 1);

        pvm_exit();
        exit(0);
}
```

Los resultados de la ejecución del programa son:

Al correrlo en la máquina que estamos trabajando devuelve el numero de tareas que se inicializarón,

```
i'm t40004
can't start hello_other. El valor de regreso es cc=2.
```

y si lo corremos, en alguna otra máquina el resultado es el siguiente:

```
i'm tc0003
from t100005:hello,world from uam4
```

esto es porque la función spawn se encarga de seleccionar el host donde se inicializara el proceso.

PROGRAMA No. 5

```
/* Programa timing.c
*       Does a few communication timing tests on pvm.
*       Uses `timing_slave' to echo messages.
* ------------------------------------------------------------------------
*   If this test is run over machines with differnet data formats
*   Then change 'ENCODING' to PvmDataDefault in timing and timing_slave
* ------------------------------------------------------------------------
*
*       9 Dec 1991  Manchek
* 14 Oct 1992  Geist  - revision to pvm3
*  6 Mar 1994  Geist  - synch tasks and add direct route
*/
```

```
#ifdef HASSTDLIB
#include <stdlib.h>
#endif
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <math.h>
#include "pvm3.h"


#define SLAVENAME "timing_slave"
#define ENCODING  PvmDataRaw

main(argc, argv)
        int argc;
        char **argv;
{
        int mytid;                      /* my task id */
        int stid = 0;                    /* slave task id */
        int reps = 20;                   /* number of samples per test */
        struct timeval tv1, tv2;         /* for timing */
        int dt1, dt2;                    /* time for one iter */
        int at1, at2;                    /* accum. time */
        int numint;                      /* message length */
        int n;
        int i;
        int *iarray = 0;

        /* enroll in pvm */
        if ((mytid = pvm_mytid()) < 0) {
                exit(1);
        }
        printf("i'm t%x\n", mytid);
        /* start up slave task */
        if (pvm_spawn(SLAVENAME, (char**)0, 0, "", 1, &stid) < 0 || stid < 0) {
                fputs("can't initiate slave\n", stderr);
                goto bail;
        }

    /* Wait for slave task to start up */
    pvm_setopt(PvmRoute, PvmRouteDirect);
    pvm_recv( stid, 0 );
```

```
printf("slave is task t%x\n", stid);

/* round-trip timing test */
puts("Doing Round Trip test, minimal message size\n");
at1 = 0;

/* pack buffer */
pvm_initsend(ENCODING);
pvm_pkint(&stid, 1, 1);
puts(" N    uSec");
for (n = 1; n <= reps; n++) {
        gettimeofday(&tv1, (struct timezone*)0);
        if (pvm_send(stid, 1)) {
                fprintf(stderr, "can't send to \"%s\"\n", SLAVENAME);
                goto bail;
        }
        if (pvm_recv(-1, -1) < 0) {
                fprintf(stderr, "recv error%d\n" );
                goto bail;
        }
        gettimeofday(&tv2, (struct timezone*)0);
        dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec;
        printf("%2d %8d\n", n, dt1);
        at1 += dt1;
}
printf("RTT Avg uSec %d\n", at1 / reps);

/* bandwidth test for different message lengths */
puts("\nDoing Bandwidth tests\n");
for (numint = 25; numint < 1000000; numint *= 10) {
        printf("\nMessage size %d\n", numint * 4);
        at1 = at2 = 0;
        iarray = (int*)malloc(numint * sizeof(int));
        puts(" N  Pack uSec  Send uSec");
        for (n = 1; n <= reps; n++) {
                gettimeofday(&tv1, (struct timezone*)0);

                pvm_initsend(ENCODING);
                pvm_pkint(iarray, numint, 1);

                gettimeofday(&tv2, (struct timezone*)0);
                dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000
                        + tv2.tv_usec - tv1.tv_usec;
```

```
                gettimeofday(&tv1, (struct timezone*)0);

                if (pvm_send(stid, 1)) {
                        fprintf(stderr, "can't send to \"%s\"\n", SLAVENAME);
                        goto bail;
                }

                if (pvm_recv(-1, -1) < 0) {
                        fprintf(stderr, "recv error%d\n" );
                        goto bail;
                }
                gettimeofday(&tv2, (struct timezone*)0);
                dt2 = (tv2.tv_sec - tv1.tv_sec) * 1000000
                        + tv2.tv_usec - tv1.tv_usec;

                printf("%2d   %8d   %8d\n", n, dt1, dt2);
                at1 += dt1;
                at2 += dt2;
        }

        if (!(at1 /= reps))
                at1 = 1;
        if (!(at2 /= reps))
                at2 = 1;
        puts("Avg uSec");
        printf("    %8d   %8d\n", at1, at2);
        puts("Avg Byte/uSec");
        printf("    %8f   %8f\n",
                (numint * 4) / (double)at1,
                (numint * 4) / (double)at2);
}

/* we have to do this because the last message might be taking
*  up all the shared memory pages. */
pvm_freebuf(pvm_getsbuf());

puts("\ndone");

bail:
        if (stid > 0)
                pvm_kill(stid);
        pvm_exit();
        exit(1);
}
```

```
/*      Programa timing_slave.c    See timing.c */

#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include "pvm3.h"

#define ENCODING  PvmDataRaw

main(argc, argv)
        int argc;
        char **argv;
{
        int mytid;   /* my task id */
        int dtid;    /* driver task */
        int bufid;
        int n = 0;

        /* enroll in pvm */

        mytid = pvm_mytid();

   /* tell parent I am ready */

   pvm_setopt(PvmRoute, PvmRouteDirect);
   pvm_initsend(ENCODING);
   pvm_send( pvm_parent(), 0 );

        /* pack mytid in buffer */
        pvm_initsend(ENCODING);
        pvm_pkint(&mytid, 1, 1);

        /* our job is just to echo back to the sender when we get a message */
        while (1) {
                bufid = pvm_recv(-1, -1);
                pvm_bufinfo(bufid, (int*)0, (int*)0, &dtid);
                pvm_freebuf(pvm_getrbuf());  /* for shared memory refcount hang */
                pvm_send(dtid, 2);

/*  printf("echo %d\n", ++n);  */
        }
}
```

Este programa realiza una mínima comunicación analizando los tiempos en pvm.
Los resultados de su ejecución son los siguientes:
Al ejecutarlo la primera vez inicializa al esclavo así:

```
i'm t40036
can't initiate slave
```

y al volverlo ejecutar muestra lo siguiente:

```
i'm t40037
slave is task t40038
```

Doing Round Trip Test, minimal message size

N	Usec
1	922
2	821
3	813
4	817
5	817
6	816
7	816
8	815
9	815
10	826
11	814
12	815
13	819
14	819
15	818
16	820
17	816
18	819
19	808
20	825
RTT AvgUsec	825

Doing Bandwidth tests message size 100

N	Pack Usec	SendUsec
1	51	852
2	46	840
3	45	839
4	47	842
5	46	841
6	49	844
7	47	842
8	47	982
9	47	851
10	47	847
11	47	848
12	46	843
13	45	843
14	45	842
15	46	844
16	46	846
17	45	1113
18	44	843
19	45	892
20	45	843
AvgUsec		
	46	864
AvgByte/Usec		
	2.173913	0.115741

Doing Bandwidth tests message size 100

N	Pack Usec	SendUsec
1	79	948
2	53	877
3	54	871
4	54	54049
5	69	898
6	58	864
7	54	865
8	55	868
9	57	863
10	61	867
11	59	863
12	56	860
13	57	863
14	56	901
15	56	871
16	57	864
17	56	863
18	55	863
19	55	865
20	57	869
AvgUsec		
	57	3782
AvgByte/Usec		
	17.543860	0.264410

Doing Bandwidth tests message size 100

N	Pack Usec	SendUsec
1	1054	3119
2	303	2148
3	273	2069
4	249	2135
5	257	2058
6	261	2107
7	271	2077
8	246	160914
9	396	2090
10	287	2111
11	272	2067
12	247	2110
13	263	2051
14	264	2080
15	274	2057
16	247	2116
17	264	2046
18	267	2093
19	277	2077
20	249	2121
AvgUsec		
	311	10082
AvgByte/Usec		
	32.154341	0.991867

Este programa genera 5 corridas diferentes, de las cuales únicamente mostramos 3 y al final aparece:

```
done  /* Termina la ejecución del programa */
$
```

PROGRAMA No. 6

Master.c

/* Este programa va ligado con el **Slave.c** */
/* Este programa genera un código de error si el número de tareas es menor que el número de procedimientos, es decir, hay problemas al generar los esclavos, y manda un código de error junto con el identificador de tarea que no se pudo generar . */

```
#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "slave1"
```

```
main()
{
  int mytid;                                  /* my task id */
      int tids[32];                           /* slave task ids */
  int n, nproc, numt, i, who, msgtype, nhost, narch;
  float data[100], result[32];
  struct pvmhostinfo *hostp[32];

  /* enroll in pvm */
  mytid = pvm_mytid();

  /* Set number of slaves to start */
  /* Can not do stdin from spawned task */
  if( pvm_parent() == PvmNoParent ){
    puts("How many slave programs (1-32)?");
    scanf("%d", &nproc);
  }
  else{
    pvm_config( &nhost, &narch, hostp );
    nproc = nhost;
    if( nproc > 32 ) nproc = 32 ;
  }

  /* start up slave tasks */
  numt=pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);
  if( numt < nproc ){
    printf("Trouble spawning slaves. Aborting. Error codes are: \n");
    for( i = numt ; i<nproc ; i++ ) {
      printf("TID %d %d\n" , i, tids[i]);
    }
    for( i=0 ; i<numt ; i++ ){
      pvm_kill( tids[i] );
    }
    pvm_exit();
    exit();
  }

  /* Begin User Program */
  n = 100;
  /* initialize_data( data, n ); */
  for( i=0 ; i<n ; i++ ){
    data[i] = 1;
  }
```

```
  /* Broadcast initial data to slave tasks */
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&nproc, 1, 1);
        pvm_pkint(tids, nproc, 1);
        pvm_pkint(&n, 1, 1);
        pvm_pkfloat(data, n, 1);
  pvm_mcast(tids, nproc, 0);

  /* Wait for results from slaves */
  msgtype = 5;
  for( i=0 ; i<nproc ; i++ ){
    pvm_recv( -1, msgtype );
    pvm_upkint( &who, 1, 1 );
    pvm_upkfloat( &result[who], 1, 1 );
    printf("I got %f from %d\n", result[who],who);
  }

  /* Program Finished exit PVM before stopping */
  pvm_exit();
}
```

Programa Slave.c

```
#include <stdio.h>
#include "pvm3.h"

main()
{
  int mytid;      /* my task id */
  int tids[32];   /* task ids   */
        int n, me, i, nproc, master, msgtype;
        float data[100], result;
  float work();

  /* enroll in pvm */
        mytid = pvm_mytid();

  /* Receive data from master */
  msgtype = 0;
  pvm_recv( -1, msgtype );
        pvm_upkint(&nproc, 1, 1);
        pvm_upkint(tids, nproc, 1);
```

```
        pvm_upkint(&n, 1, 1);
        pvm_upkfloat(data, n, 1);

    /* Determine which slave I am (0 -- nproc-1) */
    for( i=0; i<nproc ; i++ )
      if( mytid == tids[i] ){ me = i; break; }

    /* Do calculations with data */
    result = work( me, n, data, tids, nproc );

    /* Send result to master */
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &me, 1, 1 );
    pvm_pkfloat( &result, 1, 1 );
    msgtype = 5;
    master = pvm_parent();
    pvm_send( master, msgtype );

    /* Program finished. Exit PVM before stopping */
    pvm_exit();
}
float
work(me, n, data, tids, nproc )

    /* Simple example: slaves exchange data with left neighbor (wrapping) */
    int me, n, *tids, nproc;
    float *data;
{
    int i, dest;
    float psum = 0.0;
    float sum = 0.0;
    for( i=0 ; i<n ; i++ ){
      sum += me * data[i];
    }

    /* illustrate node-to-node communication */
    pvm_initsend( PvmDataDefault );
    pvm_pkfloat( &sum, 1, 1 );
    dest = me+1;
    if( dest == nproc ) dest = 0;
    pvm_send( tids[dest], 22 );
    pvm_recv( -1, 22 );
    pvm_upkfloat( &psum, 1, 1 );
```

```
    return( sum + psum );
}
```

Los resultados de la ejecución del programa son:

```
How many slave programs (1 - 32)? : 9
Trouble spawning  slaves. Aborting.
Error codes are:
TID  6 - 7
TID  7 - 7
TID  8 - 7
```

Haciendo otra prueba con un valor diferente:

```
How many slave programs (1 - 32)? : 30
Trouble spawning  slaves. Aborting.
Error codes are:
TID  20 - 7
TID  21 - 7
TID  22 - 7
TID  23 - 7
TID  24 - 7
TID  25 - 7
TID  26 - 7
TID  27 - 7
TID  28 - 7
TID  29 - 7
```

PROGRAMA No. 7

```
/*
*   Spmd.c example using PVM 3
*  Este programa registra el numero de llamadas en el grupo, e imprime me=0 si se
*  encuentra en la primera instancia a si como el identificador de la tarea.
*   Tambien ilustra las funciones de grupo.
*/

#define NPROC 8

#include <stdio.h>
#include <sys/types.h>
```

```
#include "pvm3.h"

main()
{
   int mytid;              /* my task id */
   int tids[NPROC];        /* array of task id */
   int me;                 /* my process number */
   int i;

   /* enroll in pvm */
   mytid = pvm_mytid();


   /* Join a group and if I am the first instance */
   /* i.e. me=0 spawn more copies of myself     */

   me = pvm_joingroup( "foo" );
   printf("me = %d mytid = %d\n", me, mytid);

   if( me == 0 )
     pvm_spawn("spmd", (char**)0, 0, "", NPROC-1, &tids[1]);

   /* Wait for everyone to startup before proceeding. */
   pvm_freezegroup("foo", NPROC);
   pvm_barrier( "foo", NPROC );
/*----------------------------------------------------------------------*/

    dowork( me, NPROC );

    /* program finished leave group and exit pvm */
    pvm_lvgroup( "foo" );
    pvm_exit();
    exit(1);
}

/* Simple example passes a token around a ring */

dowork( me, nproc )
    int me;
    int nproc;
{
    int token;
    int src, dest;
    int count  = 1;
```

```
    int stride = 1;
    int msgtag = 4;

    /* Determine neighbors in the ring */
    src = pvm_gettid("foo", me-1);
    dest= pvm_gettid("foo", me+1);
    if( me == 0 )       src = pvm_gettid("foo", NPROC-1);
    if( me == NPROC-1 ) dest = pvm_gettid("foo", 0);

    if( me == 0 )
    {
      token = dest;
      pvm_initsend( PvmDataDefault );
      pvm_pkint( &token, count, stride );
      pvm_send( dest, msgtag );
      pvm_recv( src, msgtag );
      printf("token ring done\n");
    }
    else
    {
      pvm_recv( src, msgtag );
      pvm_upkint( &token, count, stride );
      pvm_initsend( PvmDataDefault );
      pvm_pkint( &token, count, stride );
      pvm_send( dest, msgtag );
    }
}
```

Los resultados de la ejecución son:

```
/* En una máquina */

me = 0    mytid 786465
token ring  done

/* En otra máquina */

me = 0    mytid 262186
token ring  done
```

PROGRAMA No. 8

Programa nntime.c

```
/*  This program collects timing statistics for node-to-node message passing
 * To measure performance of native send/receive, compile as
 *              Paragon:        icc -nx -O -DPGON_NODE nntime.c
 *              iPSC/860:       icc -O -DI860_NODE nntime.c -node
 *  CM5(CMMD):  cc -c -DCM5_NODE nntime.c; cmmd-ld -comp cc -node  nntime.o
 */

#include <stdio.h>

#if !defined(CNVX) && !defined(CNVXN)
#include <memory.h>
#endif

#ifdef CM5_NODE
#include <cm/cmmd.h>
#include <cm/timers.h>
#endif

#ifdef SUNMP
#include <sys/types.h>
#include <sys/processor.h>
#include <sys/procset.h>
#endif

#ifdef PVM
#include "pvm3.h"
#endif

#define NPROC       4
#define SAMPLE      100             /* sample rate */
#define MAXSIZE     100000          /* must be a power of 10 */
#define ENCODE      PvmDataRaw
#define ENCODE2     PvmDataInPlace


#ifdef CM5_NODE

#include <cm/cmmd.h>
#define TIMER_CLEAR             CMMD_node_timer_clear(1)
#define TIMER_START             CMMD_node_timer_start(1)
```

```
#define TIMER_STOP              CMMD_node_timer_stop(1)
#define TIMER_ELAPSED  CMMD_node_timer_elapsed(1)

#endif /*CM5_NODE*/

#if defined(PGON_NODE) || defined(I860_NODE)

double tstart, tstop, dclock();
#define TIMER_CLEAR             (tstart = tstop = 0)
#define TIMER_START             (tstart = dclock())
#define TIMER_STOP              (tstop = dclock())
#define TIMER_ELAPSED  (tstop - tstart)

#endif /*PGON_NODE/I860_NODE*/

#if !defined(PGON_NODE) && !defined(I860_NODE) && !defined(CM5_NODE)

#include <sys/time.h>
struct timeval tv1, tv2;
#define TIMER_CLEAR    (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec =0)
#define TIMER_START    gettimeofday(&tv1, (struct timezone*)0)
#define TIMER_STOP     gettimeofday(&tv2, (struct timezone*)0)
#define TIMER_ELAPSED  (tv2.tv_sec-tv1.tv_sec+(tv2.tv_usec-tv1.tv_usec)*1.E-6)

#endif

int mytid, momtid;

main(argc, argv)
int argc;
char *argv[];
{
        int kidtid;
        int ntask = 0, i;
        char *grpname;
        char **tmp_argv;
#if defined(SP2MPI_NODE)
     int info, ntasks;
     struct pvmtaskinfo *taskp;
#endif /*SP2MPI_NODE*/

#if defined(PVM) && !defined(SP2MPI_NODE)

        if ((mytid = pvm_mytid()) < 0)
```

```
                exit(-1);
        momtid = pvm_parent();
        pvm_setopt(PvmRoute, PvmRouteDirect);

/* --- Create a unique group name from the PID of the parent.
 *  --- The parent will pass the group name to the child through cmd line
*/
        if(argc > 1)
                grpname = argv[1];
        else {
                grpname = (char *) malloc (20);
                sprintf(grpname,"NNTIME_%d", getpid());
        }

        if((ntask = pvm_joingroup(grpname)) < 0) {
                pvm_perror("Error joining nntime group");
                exit(-1);
        }

        switch (ntask) {
        case 0:
                tmp_argv = (char **) malloc (sizeof (char *)*2);
                tmp_argv[0] = grpname;
                tmp_argv[1] = 0;
                pvm_spawn("nntime", tmp_argv, 0, "", 1, &kidtid);
                pingpong(kidtid);
                break;
        case 1:
                pingpong(momtid);                       /* she's mom */
                break;
        default:
                printf("Too many 'nntime' processes running: %d\n", ntask);
        }

        pvm_lvgroup(grpname);
        pvm_exit();

#else /*PVM*/

#if defined(SP2MPI_NODE)
#define PROGRAM "nntime"

        if ((mytid = pvm_mytid()) < 0)
                exit(-1);
```

```
        momtid = pvm_parent();
        pvm_setopt(PvmRoute, PvmRouteDirect);
        info = pvm_tasks(0, &ntasks, &taskp);
        if (!info) {
                for (i=0;i < ntasks; i++) {
                        if ((strcmp(taskp[i].ti_a_out, PROGRAM) == 0) &&
                           (mytid != taskp[i].ti_tid)){
                                pingpong(taskp[i].ti_tid);
                        }
                }
        }
        pvm_exit();

#endif /*SP2MPI_NODE*/

#if defined(PGON_NODE) || defined(I860_NODE)

        if (mytid = mynode())
                pingpong(0);
        else
                pingpong(1);

#endif /*I860_NODE*/

#ifdef CM5_NODE

        CMMD_fset_io_mode(stdout, CMMD_independent);
        if ((mytid = CMMD_self_address()) == 1)
                pingpong(0);
        if ((mytid = CMMD_self_address()) == 0)
                pingpong(1);
        CMMD_all_msgs_wait();

#endif /*CM5_NODE*/

#endif /*PVM*/

        exit(0);
}
```

```
#ifdef PVM

void
validate(data, size)
        double data[];
        int size;
{
        int i;

        for (i = 0; i < size; i++)
                if ((i * i - data[i]) > 0.01) {
                        printf("error: data[%d] = %g \n", i, data[i]);
                        break;
                }

        if (i == size)
                printf("t%x: %d doubles received correctly \n \n \n", mytid, i);
}

#endif

/* exchange messages and measure the transit time */
pingpong(hertid)
        int hertid;
{
        int n, size;
        static double data[MAXSIZE];
        char str[32];
        int t;

        /* test node-to-node send */

        if (mytid > hertid) {

                for (n = 0; n < MAXSIZE; n++)
                        data[n] = n * n;
#ifdef PVM
#ifdef PACK
                pvm_recv(-1, 0);
                pvm_upkstr(str);
                printf("t%x: %s\n", mytid, str);
                pvm_upkdouble(data, MAXSIZE, 1);
                validate(data, MAXSIZE);
                pvm_initsend(ENCODE);
```

```
                sprintf(str, "%d doubles from t%x", MAXSIZE, mytid);
                pvm_pkstr(str);
                pvm_pkdouble(data, MAXSIZE, 1);
                pvm_send(hertid, 0);
#else
                pvm_precv(-1, 0, data, MAXSIZE, PVM_DOUBLE, (int*)0, (int*)0,
(int*)0);
                validate(data, MAXSIZE);
                pvm_psend(hertid, 0, data, MAXSIZE, PVM_DOUBLE);
#endif /*PACK*/

/*              pvm_freebuf(pvm_setsbuf(0));
                pvm_freebuf(pvm_setrbuf(0));
*/
#endif /*PVM*/
/*
#ifdef SUNMP
if (processor_bind(P_PID, P_MYID, 2, NULL) == -1)
        perror("processor_bind");
#endif
*/
                for (size = 0; size <= MAXSIZE; size = 10*(size ? size : 1))
                        for (n = 0; n < SAMPLE; n++) {
#ifdef PVM
#ifdef PACK
                                pvm_recv(-1, 0);
                                pvm_initsend(ENCODE);
                                pvm_pkdouble(data, size, 1);
                                pvm_send(hertid, 0);
#else
                                pvm_precv(-1, -1, data, MAXSIZE, PVM_DOUBLE, (int*)0,
                                        (int*)0, (int*)0);
                                pvm_psend(hertid, size, data, size, PVM_DOUBLE);
#endif /*PACK*/
#else /*PVM*/
#if defined(PGON_NODE) || defined(I860_NODE)
                                crecv(-1, (char *)data, MAXSIZE * sizeof(double));
                                csend(size, data, size * sizeof(double), hertid, 0);
#endif
#ifdef CM5_NODE
                                CMMD_receive_block(CMMD_ANY_NODE,
CMMD_ANY_TAG, (char *)data,
                                        MAXSIZE * sizeof(double));
                                CMMD_send_block(hertid, size, data, size * sizeof(double));
```

```
#endif
#endif /*PVM*/
                        }

        } else {

                for (n = 0; n < MAXSIZE; n++)
                        data[n] = n * n;
#ifdef PVM
#ifdef PACK
                pvm_initsend(ENCODE);
                sprintf(str, "%d doubles from t%x", MAXSIZE, mytid);
                pvm_pkstr(str);
                pvm_pkdouble(data, MAXSIZE, 1);
                pvm_send(hertid, 0);
                pvm_recv(-1, 0);
                pvm_upkstr(str);
                printf("t%x: %s\n", mytid, str);
                pvm_upkdouble(data, MAXSIZE, 1);
#else
                pvm_psend(hertid, 0, data, MAXSIZE, PVM_DOUBLE);
                pvm_precv(-1, 0, data, MAXSIZE, PVM_DOUBLE, (int*)0, (int*)0,
(int*)0);
#endif /*PACK*/
                validate(data, MAXSIZE);
/*
                pvm_freebuf(pvm_setsbuf(0));
                pvm_freebuf(pvm_setrbuf(0));
*/
#endif

                /* do timing measurements */
                puts("Node-to-node Send/Ack \n");
                for (size = 0; size <= MAXSIZE; size = 10*(size ? size : 1)) {

#ifdef TIMER_CLEAR

/*
#ifdef SUNMP
if (processor_bind(P_PID, P_MYID, 0, NULL) == -1)
        perror("processor_bind");
#endif
*/
                        TIMER_CLEAR;
```

```
                    TIMER_START;
                    for (n = 0; n < SAMPLE; n++) {
#ifdef PVM
#ifdef PACK
                            pvm_initsend(ENCODE);
                            pvm_pkdouble(data, size, 1);
                            pvm_send(hertid, 0);
                            pvm_recv(-1, 0);
#else
                            pvm_psend(hertid, size, data, size, PVM_DOUBLE);
                            pvm_precv(-1, -1, data, MAXSIZE, PVM_DOUBLE, (int*)0,
                                    (int*)0, (int*)0);
#endif /*PACK*/
#else /*PVM*/
#if defined(PGON_NODE) || defined(I860_NODE)
                            csend(size, (char *)data, size * sizeof(double), hertid, 0);
                            crecv(-1, data, MAXSIZE * sizeof(double));
#endif
#ifdef CM5_NODE
                            CMMD_send_block(hertid, size, data, size * sizeof(double));
                            CMMD_receive_block(CMMD_ANY_NODE,
CMMD_ANY_TAG, data,
                                    MAXSIZE * sizeof(double));
#endif
#endif /*PVM*/
                    }
                    TIMER_STOP;
                    t = 1000000*TIMER_ELAPSED/SAMPLE;
                    printf("Roundtrip T = %d (us)  (%.4f MB/s)     Data size: %d\n",
                            t, 2.0*8.0*(float)size/(float)t, sizeof(double)*size);
            }
#endif /*TIMER_CLEAR*/

/* sleep(1); */

        }
}

#ifdef PVM

/* measure packet transit time */
time_one(size, dtid)
int size;
int dtid;
```

```
{
        int i;
        static double data[MAXSIZE];
        int t;

#ifdef TIMER_CLEAR

        for (i = 0; i < size; i++)
                data[i] = i * i;

/*
        pvm_initsend(ENCODE2);
        pvm_pkdouble(data, size, 1);
*/

/*
#ifdef SUNMP
if (processor_bind(P_PID, P_MYID, 0, NULL) == -1)
        perror("processor_bind");
#endif
*/

        TIMER_CLEAR;
        TIMER_START;
        for (i = 0; i < SAMPLE; i++) {
                int dummy;

                pvm_psend(dtid, 1, data, size, PVM_DOUBLE);
                pvm_precv(-1, -1, &dummy, 1, PVM_INT, (int*)0, (int*)0, (int*)0);
/*
                pvm_send(dtid, 0);
                pvm_recv(-1, 0);
*/
        }
        TIMER_STOP;
        t = 1000000*TIMER_ELAPSED/SAMPLE;
        printf("Send + ACK T = %d (us)  (%.4f MB/s)     Data size: %d\n",
                t, 8.0*(float)size/(float)t, 8*size);

/* sleep(1); */

#endif /*TIMER_CLEAR*/
}
```

```
#endif /*PVM*/
```

Los resultados de la ejecución son:

```
tc001f: 100000 doubles received correctly
nodo - to - node Send / Ack
Round trip T = 1076 (US)      ( 0.0000 MB / s)
                1365            0.1172
                1416            1.1299
                2297            6.9656
               15620           10.2433
              184491            8.6725


Data Size: 0
          80
          800
          8000
          80000
          800000
```

PROGRAMA No. 9

gexample.c

```
/* Example of some group function and reduction functions in PVM
 *
 * 11 March 1994 - Creation by P. Papadopoulos (phil@msr.epm.ornl.gov)
 *
*/
#include <stdio.h>
#ifdef HASSTDLIB
#include <stdlib.h>
#endif
#include "pvm3.h"
#define min(u, v) ( (u) < (v) ? (u) : (v) )
#define max(u, v) ( (u) > (v) ? (u) : (v) )
#define MATRIXGROUP "matrix"
#define DEFAULT_DIMENSION 100
#define DEFAULT_NPROC     10
#define INITTAG 1000
```

```
#define SUMTAG INITTAG + 1
#define PRODTAG SUMTAG + 1
extern void calcprod();
main()
{
int info, mytid, myinst, gsize, nproc = 0;
int maxmax;
int dimension = 0;
int ninst, error;
int tids[32]; int *subblock, *colsum;
double *colprod;
int blksize, nextra, mysrow, i, j, sumsqr,itemp;
  mytid = pvm_mytid(); /* enroll */
  if( (myinst = pvm_joingroup(MATRIXGROUP)) < 0 )
  {
     pvm_perror( "Could not join group \n" );
     pvm_exit();
     exit( -1 );
  }
  if ( myinst == 0 )
  {
     printf(" This program demonstrates some group and reduction  \n");
     printf(" operations in PVM.  The output displays the \n");
     printf(" the product of the first column of a Toeplitz matrix\n");
     printf(" and the matrix 1-norm. The matrix data is distributed \n");
     printf(" among several processors.  The Toeplitz matrix is \n");
     printf(" symmetric with the first row being the row \n");
     printf(" vector [1 2 ... n].\n");
     printf("\n\n");
     if ( pvm_parent() == PvmNoParent )
     {
       while ( nproc <= 0 || nproc > 32 || dimension <= 0)
       {
          printf( " Input dimension ( >0 ) of matrix: " );
          scanf( "%d", &dimension );
          printf( " Input number of tasks (1-32): " );
          scanf( "%d", &nproc );
       }
     }
     else
     {
        nproc = DEFAULT_NPROC;
        dimension = DEFAULT_DIMENSION;
     }
```

```
        if (nproc > 1)
        {
          ninst = pvm_spawn( "gexample", (char **) 0, 0, "",
                        nproc-1, tids);
          nproc = min (nproc, ninst + 1);
        }
        pvm_initsend( PvmDataDefault );
        pvm_pkint( &nproc, 1, 1 );
        pvm_pkint( &dimension, 1, 1 );
        pvm_mcast( tids, nproc - 1, INITTAG );
    }
    else
    {
        pvm_recv( -1, INITTAG );
        pvm_upkint( &nproc, 1, 1 );
        pvm_upkint( &dimension, 1, 1);
    }
    /* Make the group static. freezegroup will wait until nproc tids have
      joined the group.                */
    info = pvm_freezegroup(MATRIXGROUP, nproc);
/* Map matrix rows to processors --      */
    blksize =  dimension/nproc ;
    nextra =   dimension % nproc;
    if( myinst < nextra )
    {
      mysrow = (blksize + 1) * myinst;
      blksize ++;
    }
    else
      mysrow = (blksize + 1)*(nextra) + blksize*(myinst - nextra);
    subblock = (int *) calloc(blksize * dimension, sizeof(int));
    colsum   = (int *) calloc(dimension, sizeof(int));
    colprod  = (double *) calloc(dimension, sizeof(double));
    if (mysrow >= dimension)    /* too many processors ! */
      blksize = 0;

/* Assign data to this subblock.  The entries below make the entire matrix
   a symmetric Toeplitz matrix (i.e. diagonals are of constant value)  */
    for (i = 0; i < blksize; i++)
      for (j = 0; j < dimension; j++)
        *(subblock + i * dimension + j) = 1 + abs(mysrow + i - j);
```

```
/* Locally compute the sum of each column and put into colsum */
    for (j = 0; j < dimension; j++)
    {
      colsum[j] = 0;
      colprod[j] = 1.0;
    }
    for (i = 0; i < blksize; i ++)
     for(j = 0; j < dimension; j++)
     {
         itemp = *(subblock + j + i * dimension);
         colsum[j] += abs( itemp );
         colprod[j] *= abs( itemp );
     }
/* synchronize the computation and then reduce using pvm_sum. This gives
  a row vector that has the all the columns sums  */
    pvm_barrier(MATRIXGROUP,-1);       /* make sure processes are finished */
    if ( pvm_reduce(PvmSum, colsum, dimension, PVM_INT, SUMTAG,
                MATRIXGROUP,0) < 0  )
       pvm_perror( "pvm_reduce had an error \n") ;
    pvm_reduce(calcprod, colprod ,dimension, PVM_DOUBLE, PRODTAG,
                MATRIXGROUP,0);
    if( myinst == 0)
    {
       maxmax = 0;
       for (j = 0; j < dimension; j++)
         maxmax = max(colsum[j],maxmax);
       printf(" The 1-Norm is %d  \n" , maxmax );
       printf(" ( Should be the sum of the integers from 1 to %d )\n",
                    dimension );
       printf(" The product of column 1 is %g  \n" , colprod[0] );
       printf(" ( Should be %d factorial)\n",
                    dimension);
    }
    info = pvm_barrier(MATRIXGROUP,-1);  /* make sure processes are finished */
    if (info < 0)
       printf("Barrier failed with result code %d\n", info);
    pvm_lvgroup(MATRIXGROUP);
    free(subblock); free(colsum); free(colprod);
    pvm_exit();
}

/*** A User-defined Reduction Function ***/
void
calcprod(datatype, x, y, num, info)
```

```
int *datatype;
double  *x, *y;
int *num, *info;
{
int i;
    for (i = 0 ; i < *num; i++)
      x[i] *= y[i];
}
```

Los resultados de la ejecución son los siguientes:

/*Utilizando los valores 2 y 9 como entrada */

This program demonstrates some group and reduction
operations in PVM. The output displays the
the product of the first column of a Toeplitz matrix
and the matrix 1-norm. The matrix data is distributed
among several processors. The Toeplitz matrix is
symmetric with the first row being the row vector [1 2 ... n].

Input dimension (>0) of matrix: 2
Input number of tasks (1 - 32): 9
The 1-Norm is 3
(Should be the sum of the integers from 1 to 2)
The product of column 1 2
(Should be 2 factorial)

/*Utilizando los valores 8 y 8 como entrada */

This program demonstrates some group and reduction
operations in PVM. The output displays the
the product of the first column of a Toeplitz matrix
and the matrix 1-norm. The matrix data is distributed
among several processors. The Toeplitz matrix is
symmetric with the first row being the row vector [1 2 ... n].

Input dimension (>0) of matrix: 8
Input number of tasks (1 - 32): 8
The 1-Norm is 3
(Should be the sum of the integers from 1 to 18)
The product of column 1 36
(Should be 40329 factorial)

PROGRAMA No. 10

```
/* Programa montar_SAD.
 * Este programa monta el servidor en todas las estaciones de trabajo que se quieren.
*/

#include <stdio.h>
#include "pvm3.h"
#include "global.h"


int MontarServidores(void){
  FILE *temp;
  char *nombre[MAX_NUM_SERVIDORES]={"uam1","uam2","uam3","uam4"};
  int  i, tid_serv, resp, n_servs;
  n_servs=0;
  temp = fopen ("/usr/pvm3/SAD/TablaTids.SAD.tmp","w");
  printf("\n");
  for(i=0;i<=(MAX_NUM_SERVIDORES-1);i++){
    resp = pvm_spawn("servidor",(char**)0,PvmTaskHost,nombre[i],1,&tid_serv);
    if (resp==1){
      fprintf(temp, "%d% s\n", tid_serv, nombre[i]);
      printf("Se levanto exitosamente el servidor en %s ...\n", nombre[i]);
      n_servs ++;
    }else{
      fprintf(temp, "%d% s\n", ERR_CREANDO_PROC, nombre[i]);
      printf("NO fue posible levantar el servidor en %s ...\n", nombre[i]);
    }/*end_if_else*/
  }/*end_for*/
  fclose(temp);
  printf("... se levantaron exitosamente %d servidores de un total de %d. \n\n",
        n_servs, MAX_NUM_SERVIDORES);
  pvm_exit();
  return(n_servs);
}

void main(void){
   MontarServidores();
}
```

Los resultados obtenidos al ejecutar el programa son:

```
Se levanto el servidor en uam1
Se levanto el servidor en uam3
Se levanto el servidor en uam4
No fue posible levantar el servidor en uam2
Se levantaron 3 servidores.
```

PROGRAMA No. 11

```
/* Programa desmontar_SAD.
 * Este programa desmonta el servidor en todas las estaciones de trabajo que se quieren.
*/

#include <stdio.h>
#include "pvm3.h"
#include "global.h"

int DesmontarServidores (void){

  int   mi_tid, servicio = DESMONTAR;
  int   i,tid_serv,resp,n_servs=0;
  char  nom_serv[MAX_TAM_NOM_ARCH];
  FILE  *f;

  mi_tid = pvm_mytid();
  f = fopen("/usr/pvm3/SAD/TablaTids.SAD.tmp","r");
  for(i=0;i<MAX_NUM_SERVIDORES;i++){
    fscanf(f, "%d% s", &tid_serv, nom_serv);
    printf("Desmontando el servidor %s con tid %d...\n", nom_serv, tid_serv);
    if (tid_serv != ERR_CREANDO_PROC){
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&mi_tid,1,1);
        pvm_send(tid_serv,1);

        pvm_initsend(PvmDataDefault);
        pvm_pkint(&servicio,1,1);
        pvm_send(tid_serv,1);

        pvm_recv(tid_serv,1);
        pvm_upkint(&resp,1,1);
```

```
        if (resp == HECHO){
           n_servs ++;
           printf("  servidor desmontado con éxito \n");
        }else
           printf("  error desmontando el servidor \n");
     }
  }
  fclose(f);
  printf("\n se desmontaron %d servidores. \n", n_servs);
  pvm_exit();

  f = fopen("/usr/pvm3/SAD/TablaTids.SAD.tmp","w");
  fclose(f);
  return (n_servs);
}/* DesmontarServidores */

main(){
  DesmontarServidores();
}
```

Los resultados obtenidos al ejecutar el programa son:

```
Desmontando el servidor en uam1...
servidor desmontado
Desmontando el servidor en uam2...
servidor desmontado
Desmontando el servidor en uam3...
servidor desmontado
Desmontando el servidor en uam4...
servidor desmontado
se desmontaron 4 servidores.
```

PROGRAMA 12

```
/* Programa Joinleave.c
   join and leave a lot ( Unión y Permisos en lotes)
*/

#include <stdio.h>
#include "pvm3.h"
#define MAXJOINS 10000
int
main(argc, argv)
int argc;
char *argv[];
{
    int mytid, mygid, njoins;
    int cnt = 0;
    mytid = pvm_mytid();
    fprintf(stderr, "%s 0x%x enrolled\n", argv[0], mytid);
    if (mytid < 0) {
            pvm_perror(argv[0]);
            return -1;
            }
     if (argc > 1)
       njoins = atoi(argv[1]);
     else
       njoins = MAXJOINS;

    /* join a group */
    while (cnt ++ <= njoins) {
            if((mygid = pvm_joingroup(" ")) < 0) {
                    pvm_perror("JoinLeave");
                    break;
                    }
            if((mygid = pvm_lvgroup("JoinLeave")) < 0) {
                    pvm_perror("JoinLeave");
                    break;
                    }
            if ((cnt%100)==0) printf("Joined %d times\n", cnt);
            }

}
```

Los resultados al ejecutar el programa son:

```
joinleave 0X40045 enrolled

joined 100 times
joined 200 times
joined 300 times
joined 400 times
joined 500 times
joined 600 times
joined 700 times
joined 800 times
joined 900 times
joined 1000 times
```

PROGRAMA 13

```
/* Programa tst.c
   test groups (Pruebas de grupo)
*/

int
main(argc, argv)
int argc;
char *argv[];
{
    int mytid, mygid1, mygid2, mygid3;

    mytid = pvm_mytid();

    mygid1 = pvm_joingroup("prognosis");
    mygid2 = pvm_joingroup("negative");

    printf("joined prognosis %d, negative %d\n", mygid1, mygid2);

    mygid1 = pvm_lvgroup("prognosis");
    mygid2 = pvm_lvgroup("negative");

    printf("left prognosis %d, negative %d\n", mygid1, mygid2);
    pvm_exit();

}
```

Los resultados de ejecutar el programa son:

```
joined prognosis 0, negative 0
left     prognosis 0, negative 0
```

PROGRAMA 14

```
/*
 * Example of group Reduce, Scatter, and Gather functions - J.M. Donato
 *
 * This example calculates the sum of squares of the first N integers
 * in three different ways where
 *
 *   N = (number of processors)*(number of elements per row)
 *
 * Note:  This is obviously not an efficient way to compute the
 *        sum of squares, but it is a cutesy example and test case.
 */

#include <stdio.h>
#include "pvm3.h"

#define MAXNDATA  20
#define MAXNPROCS 16
#define DFLTNDATA 5
#define DFLTNPROCS 4
#define TASK_NAME "trsg"

#define min(x, y) ( ((x)<(y))? (x) : (y) )
#define max(x, y) ( ((x)>(y))? (x) : (y) )

extern void PvmMin();
extern void PvmMax();
extern void PvmSum();
extern void PvmProduct();

void MaxWithLoc();
```

```
main()
{
 int myginst, i, j, gsize, count, nprocs, msgtag, datatype;
 int info_product, info_user;
 int tids[MAXNPROCS], myrow[MAXNDATA],
matrix[MAXNDATA*MAXNPROCS];
 float values[2];
 int midpoint, bigN, Sum1=0, Sum2=0, SumSquares, rootginst;
 int PSum = 0, PartSums[MAXNPROCS], dupls[MAXNDATA];
 char *gname = "group_rsg";

 /* join the group */
 myginst = pvm_joingroup(gname);
 pvm_setopt(PvmAutoErr, 1);

 /* I am the first group member, get input, start up copies of myself */
 if ( myginst == 0 )
 {
  if (pvm_parent() == PvmNoParent)
  {
   printf("\n *** Example use of PVM Reduce, Scatter, and Gather *** ");
   printf("\n Number of processors to use (1-%d)? : ", MAXNPROCS);
   scanf("%d", &nprocs);
   if (nprocs > MAXNPROCS) nprocs = MAXNPROCS;
   printf(" Number of elements per row to use (1-%d)? : ", MAXNDATA);
   scanf("%d", &count);
   if (count > MAXNDATA) count = MAXNDATA;
   printf(" INPUT values: nprocs = %d, count = %d \n", nprocs, count);
  }
  else
  {
   count = DFLTNDATA;
   nprocs = DFLTNPROCS;
  }

  tids[0] = pvm_mytid();

  if (nprocs > 1)
   pvm_spawn(TASK_NAME, (char**)0, 0, "", nprocs-1, &tids[1]);

  /* wait until they have all started, then send input values */
  while (gsize = pvm_gsize(gname) < nprocs) sleep(1);
  pvm_initsend(PvmDataDefault);
  pvm_pkint(&nprocs, 1, 1);
```

```
 pvm_pkint(&count, 1, 1);
 pvm_bcast(gname, msgtag=17);
}
else
{
 /* receive the input values */
 pvm_recv(-1, msgtag=17);
 pvm_upkint(&nprocs, 1, 1);
 pvm_upkint(&count, 1, 1);
}

rootginst = 0;  /* determine the group root */

/* init the matrix values on the root processor */
if (myginst == rootginst)
 for (j=0; j<nprocs; j++)
   for (i=0; i<count; i++)
     matrix[j *count + i] = j *count + i + 1;

/* scatter rows of matrix to each processor */
pvm_scatter(myrow, matrix, count, PVM_INT, msgtag=19, gname, rootginst);

/* this should end up squaring each value on each processor */
for (i=0; i<count; i++) dupls[i] = (myginst *count + i + 1);
datatype = PVM_INT;
PvmProduct(&datatype, myrow, dupls, &count, &info_product);
if ((myginst == rootginst) && (info_product < 0))
   printf(" ERROR: %d on PvmProduct call \n", info_product);

/* do partial sum on each proc */
for (i=0; i<count; i++) PSum += myrow[i];

/* gather partial sums to the rootginst */
pvm_gather(PartSums, &PSum, 1, PVM_INT, msgtag=21, gname, rootginst);

/* do a global sum over myrow, the result goes to rootginst */
pvm_reduce(PvmSum, myrow, count, PVM_INT, msgtag=23, gname, rootginst);

/* init values and include location information on each processor */
midpoint = nprocs/2;
values[0] = -(myginst - midpoint)*(myginst-midpoint) + count;
values[1] = myginst;    /* location information */
```

```
/* use a user-defined function in reduce, send answer to rootginst */
info_user = pvm_reduce(MaxWithLoc, values, 2, PVM_FLOAT,
                msgtag=25, gname, rootginst);

bigN = nprocs *count;
if (myginst == rootginst)
{
 /* Complete the Sum of Squares using different methods */
 for (i=0; i<nprocs; i++) Sum1 += PartSums[i];
 for (i=0; i<count;  i++) Sum2 += myrow[i];
 SumSquares = bigN*(bigN+1)*(2*bigN+1)/6;
 if ( (Sum1 == SumSquares) && (Sum2 == SumSquares))
  printf("\n Test OK: Sum of Squares of first %d integers is %d \n",
       bigN, Sum1);
 else
  printf("\n %s% d% s% d% s% d, \n% s% d \n",
       "ERROR: The Sum of Squares of the first ", bigN,
       " integers \n was calculated by Sum1 as ", Sum1,
       " and by Sum2 as ", Sum2,
       " for both it should have been ", SumSquares);

 if (info_user < 0)
  printf(" ERROR: %d on Reduce with User Function \n", info_user);

 if ((values[0] != count) || (values[1] != midpoint))
  printf(" ERROR: Should have (%f, %f), but have (%f, %f): \n",
        count, midpoint, values[0], values[1]);
 else
  printf(" Test Ok: Received (%f, %f): ", values[0], values[1]);
 printf("\n");
}

/* sync up again, leave group, exit pvm */
pvm_barrier(gname, nprocs);
pvm_lvgroup(gname);
pvm_exit();
exit(1);

} /* end main() */
```

```
/*
 This function returns the elementwise maximum of two vectors
 along with location information.

 The first num/2 values of x and y are the data values to compare.
 The second num/2 values of x and y are location information
 which is kept with the maximum value determined.
 In the case of a tie in data values, the smaller location
 is kept to insure the associativity of the operation.
*/

void MaxWithLoc(datatype, x, y, num, info)
int *datatype, *num, *info;
float *x, *y;
{
 int i, count;
 count = (*num) / 2;

 if (*datatype != PVM_FLOAT) { *info = PvmBadParam; return; }

 for (i=0; i<count; i++)
  if (y[i] > x[i])
  {
   x[i]      = y[i];
   x[i+count] = y[i+count];
  }
  else
   if (y[i] == x[i]) x[i+count] = min(x[i+count], y[i+count]);

 *info = PvmOk;
 return;

} /* end MaxWithLoc() */
```

Los resultados al ejecutar el programa son los siguientes:

```
/* Al usar los valores iniciales 2 y 2 obtenemos lo siguiente */

    *** Example use of PVM Reduce, Scatter, and Gather ***
    Number of processors to use (1 - 16)? : 2
    Number of elements per row to use (1 - 20)? : 2
    INPUT values: nprocs = 2, count = 2
```

Test OK: Sum of Squares of first 4 integers is 30
Test OK: Received (2.000000, 1.000000)

/* Al usar los valores iniciales 1 y 5 obtenemos lo siguiente */

*** Example use of PVM Reduce, Scatter, and Gather ***
Number of processors to use (1 - 16)? : 1
Number of elements per row to use (1 - 20)? : 5
INPUT values: nprocs = 5, count = 55
Test OK: Sum of Squares of first 4 integers is 30
Test OK: Received (2.000000, 1.000000)

BIBLIOTECA DE FUNCIONES PVM

5. BIBLIOTECA DE FUNCIONES PVM (EN ESPAÑOL)

Este apéndice contiene una lista en orden alfabético de todas las rutinas PVM3. Cada rutina se describe a detalle para ser usadas en lenguaje C. Así como ejemplos de cada rutina, escritos en esté lenguaje.

pvm_addhosts( )

Adiciona uno o más hosts en la máquina virtual.

Synopsis

int info = pvm_addhosts (char **hosts, int nhost, int *infos)

Parámetros

hosts: Arreglo de apuntadores a una cadena de caracteres conteniendo el nombre de la máquina a ser adicionada.
nhost: Entero que especifica el número de hosts a adicionar.
infos: Arreglo de enteros de longitud nhost que contiene el código del status retornado por la rutina para el hosts individual. Un valor >0 indica un error.
info: Entero que retorna el código de status para la rutina. Un valor menor que nhost indica un fracaso parcial, un valor menor que 1 indica fracaso total.

Ejemplo

C:

```
static char *hosts[ ] = {
            "sparky",
            "thud.cs.utk.edu",
};
info = pvm_addhosts (hosts, 2, infos);
```

pvm_barrier( )

Grupo de llamadas a procesos, hasta que dichos procesos de el grupo son llamados por él.

Synopsis

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

Parámetros

group: Cadena de caracteres del nombre del grupo. El grupo debe de existir y las llamadas a procesos deben ser miembros de el grupo.

count: Entero que especifica el número de miembros del grupo que debe llamar a pvm_barrier después todos ellos son relacionados. Además no requiere de un contador que cuente el número total de miembros de el grupo especificado.

info: Entero del código de status que retorna la rutina. Un valor menor que cero indica un error.

Ejemplo

C:

```
inum = pvm_joingroup ("worker");
        .
        .
info = pvm_barrier ("worker", 5);
```

pvm_bcast( )

Propaga los datos en el buffer activo de mensajes.

Synopsis

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

Parámetros

group: Cadena de caracteres con el nombre del grupo para el grupo existente.

msgtag: Identificador entero de los mensajes, sustituyendo a los usuarios. Msgtag debe ser >=0.

info: Entero que retorna el código de status para la rutina. Un valor menor que cero indica error.

Ejemplo

C:

```
info = pvm_initsend (PvmDataRaw);
info = pvm_pkint (array, 10, 1);
msgtag = 5;
info = pvm_bcast ("worker", msgtag);
```

pvm_bufinfo( )

Retorna información según lo solicitado por el buffer de mensajes.

Synopsis

```
int info = pvm_bufinfo (int bufid, int *bytes, int *msgtag, int *tid)
```

Parámetros

bufid: Entero que especifica un identificador en particular del buffer de mensajes.
bytes: Entero que retorna la longitud en bytes de todo el mensaje.
msgtag: Entero que retorna la etiqueta del mensaje.
tid: Entero que retorna el origen del mensaje.
info: Entero que retorna el código de status para la rutina. Un valor menor que cero indica error.

Ejemplo

C:

```
bufid = pvm_recv (-1, -1);
info = pvm_bufinfo ( bufid, &bytes, &type, &source );
```

pvm_catchout( )

Captura la salida de las tareas hijo.

Synopsis

```
#include <stdio.h>
int bufid = pvm_catchout (FILE **ff)
```

Parámetros

ff: Descriptor de archivos en el cual se escribe lo recogido en la salida.

Ejemplo

C:

```
#include <stdio.h>
pvm_catchout (stdout);
```

pvm_config( )

Retorna información sobre la configuración de la máquina virtual en uso.

Synopsis

```
int info = pvm_config (int *nhost, int *narch, struct pvmhostinfo **hostp)
struct pvmhostinfo{
                        int   hi_tid;
                        char *hi_name;
                        char *hi_arch;
                        int    hi_speed;
}hostp;
```

Parámetros

nhost: Entero que retorna el número de hosts (pvmd) en la máquina virtual.
narch: Entero que retorna el número de los diferentes formatos de datos en uso.
hostp: Puntero a un arreglo de estructuras que contiene información acerca de cada hosts, incluyendo el ID de la tarea pvmd, nombre, arquitectura y la velocidad relativa.
dtid: Entero que retorna el ID de tarea pvmd para este host.
name: Cadena de caracteres que retorna el nombre de este host.
arch: Cadena de caracteres que retorna el nombre de la arquitectura del host.
speed: Entero que retorna la velocidad relativa de este host. El valor por default es 1000.
info: Entero que retorna el código de status para la rutina. Un valor menor que cero indica error.

Ejemplo

C:

```
info = pvm_config (&nhost, &narch, &hostp);
```

pvm_delhosts( )

Borra uno o más hosts de la máquina virtual.

Synopsis

int info = pvm_delhosts (char **hosts, int nhost, int *infos)

Parámetros

hosts: Arreglo de apuntadores a una cadena de caracteres conteniendo el nombre de la máquina a ser suprimida.
nhost: Entero que especifica el número de hosts a suprimir.
infos: Arreglo de enteros de longitud nhost que contiene el código del status retornado por la rutina para el hosts individual. Un valor >0 indica un error.
info: Entero que retorna el código de status para la rutina. Un valor menor que nhost indica un fracaso parcial, un valor menor que 1 indica fracaso total.

Ejemplo

C:

```
static char *hosts[ ] = {
            "sparky",
            "thud.cs.utk.edu",
};
info = pvm_delhosts (hosts, 2);
```

pvm_exit( )

Afecta al pvmd local en el que se procesa la salida de PVM.

Synopsis

int info = pvm_exit (void)

Parámetros

info: Entero del código de status retornado por la rutina. Un valor >0 indica un error.

Ejemplo

C:

```
/* Programa done */
pvm_exit ( );
exit ( );
```

pvm_freebuf( )

Coloca el mensaje en el buffer.

Synopsis

int info = pvm_freebuf (int bufid)

Parámetros

bufid: Entero que identifica el buffer del mensaje.
info: Entero del código de status retornado por la rutina. Un valor >0 indica un error.

Ejemplo

C:

```
bufid = pvm_mkbuf (PvmDataDefault);
        .
        .
info = pvm_freebuf (bufid);
```

pvm_getinst( )

Retorna el número de instancia en un grupo de un proceso PVM.

Synopsis

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

Parámetros

group: Cadena de caracteres de un grupo de nombres de un grupo existente.
tid: Entero identificador de la tarea de un proceso PVM.
inum: Entero que retorna el número de instancia para la rutina. El número de instancia esta inicialmente en cero y de ahí cuenta hacia arriba. Un valor menor que cero indica error.

Ejemplo

C:

```
inum = pvm_getinst ("worker", pvm_mytid( ) );
----------------------
inum = pvm_getinst ("worker", tid [i] );
```

pvm_getopt( )

Muestra varias opciones para libpvm.

Synopsis

int val = pvm_getopt (int what)

Parámetros

what: Entero que define lo que se muestra. Las opciones son:

Option value		MEANING
PvmRoute	1	routing policy
PvmDebugMask	2	debugmask
PvmAutoErr	3	auto error reporting
PvmOutputTid	4	stdout device for children
PvmOutputCode	5	output msgtag
PvmTraceTid	6	trace device for children
PvmTrace Code	7	trace msgtag
PvmFragSize	8	message fragment size
PvmResvTids	9	Allow messages to reserved tags and TIDs

val: Entero que especifica el valor de la opción . Los valores predifinidos para la ruta son:

Option value	MEANING
PvmDontRoute	1
PvmAllowDirect	2
PvmRouteDirect	3

Ejemplo

C:

```
route_method = pvm_getopt (PvmRoute);
```

pvm_getrbuf( )

Retorna el identificador del buffer de recibo activo. Guarda el estado del buffer anterior, y regresar el identificador del buffer activo previo.

Synopsis

int bufid = pvm_getrbuf (void)

Parámetros

bufid: Entero que retorna el identificador del buffer de recibo activo.

Ejemplo

C:

```
bufid = pvm_getrbuf ( );
```

pvm_getsbuf( )

Retorna el identificador del buffer de envío activo.

Synopsis

int bufid = pvm_getsbuf (void)

Parámetros

bufid: Entero que retorna el identificador del buffer de recibo activo.

Ejemplo

C:

```
bufid = pvm_getsbuf ( );
```

pvm_gettid()

Retorna el tid del identificador de proceso para el nombre del grupo y el número de instancia.

Synopsis

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

Parámetros

group: Cadena de caracteres que contiene el nombre de un grupo existente.
inum: Entero de el número de instancia de el proceso en el grupo.
tid: Entero que retorna el identificador de tarea.

Ejemplo

C:

```
tid = pvm_gettid ("worker", 0 );
```

pvm_gsize( )

Retorna el número de miembros presentes en el grupo de nombres.

Synopsis

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

Parámetros

group: Cadena de caracteres del grupo de nombres de un grupo exixtente.
size: Entero que retorna el número de miembros presentes en el grupo. Un valor >0 indica error.

Ejemplo

C:

```
size = pvm_gsize ("worker");
```

pvm_halt( )

Cierra el sistema PVM completamente.

Synopsis

```
int info = pvm_halt (void)
```

Parámetros

info: Entero que retorna el status de error.

pvm_hostsync( )

Obtiene el tiempo transcurrido en el host de PVM.

Synopsis

```
#include <sys/time.h>
int info = pvm_hostsync ( int host, struct timeval *clk, struct timeval *delta)
```

Parámetros

host: TID del host
(clk or clksec and clkusec): Retorna una muestra del tiempo transcurrido en el host.
(delta or deltasec and deltausec): Retorna la diferencia entre el reloj local y el reloj del host remoto.

Pvm_initsend( )

Limpia el buffer de envío y crea uno nuevo para encapsular el nuevo mensaje.

Synopsis

```
int bufid = pvm_initsend ( int encoding )
```

Parámetros

encoding: Entero que especifica la siguiente combinación de mensajes.
Las opciones en C son:

Encoding	Value	MEANING
PvmDataDefault	0	XDR
PvmDataRaw	1	no encoding
PvmDataInPlace	2	data left in place

bufid: Entero que retorna el contenido de el identificador del buffer del mensaje. Un valor menor que cero indica error.

Ejemplo

C:

```
bufid = pvm_initsend (PvmDataDefault);
info = pvm_pkint (array, 10, 1);
msgtag = 3;
info = pvm_send (tid, msgtag);
```

pvm_joingroup( )

Registra las llamadas a procesos en un nombre de grupo.

Synopsis

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

Parámetros

group: Cadena de caracteres del nombre del grupo de un grupo existente.
inum: Entero que retorna el número de instancia para la rutina. El número de instancia comienza de cero y hacia adelante. Un valor menor a cero indica error.

Ejemplo

C:

```
inum = pvm_joingroup("worker");
```

pvm_kill( )

Termina un proceso especifico de PVM.

Synopsis

```
int info = pvm_kill (int tid)
```

Parámetros

tid: Entero del identificador de tarea de el proceso a ser terminado.
info: Entero que retorna el código de status por la rutina. Un valor menor que cero indica un error.

Ejemplo

C:

```
info = pvm_kill (tid);
```

pvm_lvgroup()

Quita de el registro la llamada al proceso de el nombre de grupo.

Synopsis

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

Parámetros

group: Cadena de caracteres del nombre de el grupo de un grupo existente.
info: Entero que retorna el código de status por la rutina. Un valor menor que cero indica un error.

Ejemplo

C:

```
info = pvm_lvgroup ("worker");
```

pvm_mcast( )

Múltiples lanzamientos de datos en el buffer activo señalando una tarea.

Synopsis

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

Parámetros

ntask: Entero que especifica el número de tareas que puede ser vistas.
tids: Arreglo de enteros de longitud menor que ntask conteniendo los Ids de las tareas que pueden ser vistas.
msgtag: Entero de la etiqueta del mensaje sustituido por el usuario. msgtag podría ser mayor o igual a cero.
info: Entero que retorna el código de status retornado por la rutina. Un valor menor que cero indica error.

Ejemplo

C:

```
info = pvm_initsend (PvmDataRaw);
info = pvm_pkint (array, 10, 1);
msgtag = 5;
info = pvm_mcast (tids, ntask, msgtag);
```

pvm_mkbuf( )

Crea un nuevo buffer de mensajes.

Synopsis

int bufid = pvm_mkbuf (int encoding)

Parámetros

encoding: Entero que especifica la combinación de encapsulamiento de buffers. Las opciones en C son:

Encoding	Value	MEANING
PvmDataDefault	0	XDR
PvmDataRaw	1	no encoding
PvmDataInPlace	2	data left in place

bufid: Entero que retorna el identificador del buffer de mensaje. Un valor menor que cero indica un error.

Ejemplo

C:

```
bufid = pvm_mkbuf (PvmDataRaw);
/* send message */
info = pvm _freebuf (bufid);
```

pvm_mstat( )

Retorna el status de un hosts en la máquina virtual.

Synopsis

int mstat = pvm_mstat (char *host)

Parámetros

host: Cadena de caracteres que contienen el nombre del host.
mstat: Entero que retorna el status de la máquina:

Value	MEANING
PvmOk	host is OK
PvmNoHost	host is not in virtual machine encoding
PvmHostFail	host is unreachable (and thus possibly failed)

Ejemplo

C:

```
mstat = pvm_mstat ("msr.ornl.gov");
```

pvm_mytid( )

Retorna el tid de el proceso.

Synopsis

int tid = pvm_mytid (void)

Parámetros

tid: Entero identificador de la tarea es retornado por la llamada a un proceso PVM. Un valor menor que cero indica error.

Ejemplo

C:

```
tid = pvm_mytid( );
```

pvm_notify( )

Pide notificación de un evento de PVM, tal como una falla de host.

Synopsis

int info = pvm_notify(int what, int msgtag, int cnt, int *tids)

Parámetros

what: Identificador tipo entero de que evento puede disparar la notificación.
msgtag: Etiqueta de mensaje tipo entero para ser usada en notificación.
cnt: Valor entero especificando la longitud del arreglo de los tids para PvmTaskExit y PvmHostDelete. Para PvmHostAdd especifica el número de veces a notificar.
tids: Arreglo de enteros de longitud ntask que contiene una lista de tids de tareas o de pvm a ser notificados.
info: Valor entero que indica estado, regresado por la rutina, si es menor de cero indica error.

Ejemplo

C:

```
info = pvm_notify (PvmHostAdd, 9999, 1, dummy);
```

pvm_nrecv( )

Verifica que haya un mensaje no bloqueador con la etiqueta msgtag.

Synopsis

int bufid = pvm_nrecv(int tid, int msgtag)

Parámetros

tid: Identificador de tareas del proceso que se está mandando.
msgtag: Etiqueta de mensaje dada por el usuario. msgtag debe ser mayor o igual a cero.
bufid: Valor de tipo entero que regresa el nuevo buffer activo identificador. Valor menor a cero indica error.

Ejemplo

C:

```
tid = pvm_parent( );
msgtag = 4;
arrived = pvm_nrecv (tid, msgtag);
if (arrived > 0)
  info = pvm_upkint (tid_array, 10, 1);
else
   /* go do other computing */
```

pvm_pk*( )

Empaqueta el buffer activo de mensajes con los tipos de datos preespecificados.

Synopsis

```
int info = pvm_packf(const char *fmt, ...)
int info = pvm_pkbyte(char *xp, int nitem, int stride)
int info = pvm_pkcplx(float *cp, int nitem, int stride)
int info = pvm_pkdcplx(double *zp, int nitem, int stride)
int info = pvm_pkdouble(double *dp, int nitem, int stride)
int info = pvm_pkfloat(float *fp, int nitem, int stride)
int info = pvm_pkint(int *ip, int nitem, int stride)
int info = pvm_pkuint(unsigned int *ip, int nitem, int stride)
int info = pvm_pkushort(unsigned short *ip, int nitem, int stride)
int info = pvm_pkulong(unsigned long *ip, int nitem, int stride)
int info = pvm_pklong(long *ip, int nitem, int stride)
int info = pvm_pk(short *jp, int nitem, int stride)
int info = pvm_pk(char *sp)
```

Parámetros

fmt: Expresión de tipo printf (de C) que especifica que empacar.
nitem: El total de datos a empacar (no el número de bytes).
stride: El tipo de empaquetación a ser usado mientras se empacan los datos.
xp: Apuntador al principio del bloque de bytes.
cp: Arreglo complejo de por lo menos nitem*stride datos de longitud
zp: Arreglo de doble precisión.
dp: Arreglo real de doble precisión.
fp: Arreglo real de punto flotante.
ip: Arreglo de enteros.
jp: Arreglo de dobles enteros.

sp: Apuntador a una cadena terminada en null.
what: Valor entero que especifica el tipo de dato empacado.
info: Valor entero que indica estado, regresado por la rutina, si es menor de cero indica error.

Ejemplo

C:
```
info = pvm_initsend(PvmDataDefault);
info = pvm_pkstr ("initial data");
info = pvm_pkint (&size, 1, 1);
info = pvm_pkint (array, size, 1);
info = pvm_pkdouble (matrix, size*size, 1);
msgtag = 3;
info = pvm_send (tid, msgtag);
```

pvm_parent( )

Regresa el tid del proceso que inició al mismo.

Synopsis

```
int tid = pvm_parent(void)
```

Parámetros

tid: Tipo entero, indica el identificador de tarea, en este caso del proceso padre.

Ejemplo

C:
```
tid = pvm_parent ( );
```

pvm_perror( )

Imprime el estado de error de la última llamada a PVM.

Synopsis

int info = pvm_error(char *msg)

Parámetros

msg: Cadena de caracteres dada por el usuario que será anexada al principio del mensaje de error de la última llamada a PVM.

Info: Valor entero que indica estado, regresado por la rutina, si es menor de cero indica error.

Ejemplo

C:

```
if (pvm_send (tid, msgtag );
pvm_perror( );
```

pvm_precv( )

Recibe un mensaje directamente al buffer.

Synopsis

int info = pvm_precv (int tid, int msgtag, char *buf, int len, int datatype, int atid, int atag, int alen)

Parámetros

tid: Identificador de tareas tipo entero para mandar procesos.

mstag: Etiqueta de mensaje, debe ser mayor o igual a cero.

buf: Apuntador al buffer a donde recibir.

len: Longitud del buffer (en múltiplos del tamaño del tipo de dato.

datatype: El tipo de dato al cual apunta buf.

atid: Regresa el tid actual del que envía.

atag: Regresa la etiqueta de mensaje actual.

alen: Longitud del mensaje actual.

info: Valor entero que indica estado, regresado por la rutina, si es menor de cero indica error.

Ejemplo

C:

```
info = pvm_precv (tid, msgtag, array, cnt, PVM_FLOAT, &src, &atag, &acnt);
```

pvm_probe( )

Revisa a ver si ha llegado un mensaje.

Synopsis

```
int bufid = pvm_probe(int tid, int msgtag)
```

Parámetros

tid: Identificador de tareas tipo entero para mandar procesos.
msgtag: Etiqueta de tipo entero que es dada por el usuario.
bufid: Valor que regresa un entero que da un buffer identificador.

Ejemplo

C:

```
tid = pvm_parent ( );
msgtag = 4;
arrived = pvm_probe (tid, msgtag);
if (arrived >0)
   info = pvm_bufinfo (arrived, &len, &tag, &tid);
else
/* go do other computing */
```

pvm_psend( )

Empaca y manda datos en una sola llamada.

Synopsis

```
int info = pvm_psend (int tid, int msgtag, char *buf, int len, int datatype)
```

Parámetros

tid: Identificador de tareas tipo entero para mandar procesos.
msgtag: Etiqueta de mensaje tipo entero dada por el usuario.
buf: Apuntador al buffer de donde mandar.

len: Longitud del buffer (en múltiplos del tamaño del tipo de dato.
datatype: El tipo de dato al cual buf está apuntando.
info: Valor entero que indica estado, regresado por la rutina, si es menor de cero indica error.

Ejemplo

C:

```
info = pvm_psend (tid, msgtag, array, 1000, PVM_FLOAT);
```

pvm_pstat( )

Regresa el estado del proceso PVM especificado.

Synopsis

```
int status = pmv_pstat(int tid)
```

Parámetros

tid: Identificador de tareas tipo entero para mandar procesos.
status: Valor de tipo entero regresado indicando el estado dado en el parámetro.

Ejemplo

C:

```
tid = pvm_parent ( );
status = pvm_pstat(tid);
```

pvm_recv( )

Recibe un mensaje.

Synopsis

```
int bufid = pvm_recv(int tid, int msgtag)
```

Parámetros

tid: Identificador de tareas tipo entero para mandar procesos.
msgtag: Etiqueta de mensaje tipo entero dada por el usuario.
bufid: Valor entero que tiene el identificador del buffer activo receptor.

Ejemplo

C:

```
tid = pvm_parent ( );
msgtag = 4;
bufid = pvm_recv (tid, msgtag);
info = pvm_upkint (tid_array, 10, 1);
info = pvm_upkint (problem_size, 1, 1);
info = pvm_upfloat (input_array, 100, 1);
```

pvm_recvf( )

Modifica la comparación de funciones usadas al aceptar mensajes.

Synopsis

```
int (*old) () = pvm_recvf (int (*new)(int bufid, int tid, int tag))
```

Parámetros

tid: Entero identificador de la tarea llamada en el proceso sustituida por el usuario.
tag: Entero de la etiqueta del mensaje sustituida por el usuario.
bufid: Entero del identificador del buffer de mensajes.

pvm_reduce( )

Realiza operaciones globales aritméticas en el grupo.

Synopsis

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

Parámetros

func: Función que define la operación a ejecutar con lo datos globales. Las predefinidas son: PvmMax, PvmMin, PvmSum y PvmProduct. El usuario puede definir su propia función.
data: Puntero al inicio de las direcciones de un arreglo de valores locales.
count: Entero que especifica el número de elementos en el arreglo de datos.
datatype: Entero que especifica el tipo de todos los datos del arreglo.

msgtag: Entero de la etiqueta del mensaje sustituida por el usuario. msgtag podría ser mayor o igual que cero.
group: Cadena de caracteres del nombre de grupo de un grupo existente.
root: Entero del número de instancia del miembro del grupo que obtienen el resultado.
info: Entero que retorna el código de status de la rutina. Un valor menor que cero indica error.

Ejemplo

C:

```
info = pvm_reduce (PvmMax, &myvals, 10, PVM_INT, msgtag, "workers",
                   roottid);
```

pvm_reg_hoster( )

Registra a esta tarea como responsable de agregar nuevos hosts de PVM.

Synopsis

```
#include <pvmsdpro.h>
int info = pvm_reg_hoster(void)
```

Parámetros

info: Código de estado, de tipo entero regresado por la rutina.

pvm_reg_rm( )

Registra a esta tarea como manejador de los recursos de PVM.

Synopsis

```
#include <pvmsdpro.h>
int info = pvm_reg_rm(struct hostinfo **hip)
  struct hostinfo {
          int hi_tid;
          char *hi_name;
          char *hi_arch;
          int hi_speed;
    } hip ;
```

Parámetros

hip: Apuntador a un arreglo que contiene información acerca de cada host, incluyendo a si tid de pvmd (el demonio), nombre, arquitectura, y velocidad relativa.
Info: Código de estado, de tipo entero regresado por la rutina.

pvm_reg_tasker( )

Registra a esta tarea como responsable de iniciar nuevas tareas de PVM.

Synopsis

```
#include <pvmsdpro.h>
int info = pvm_reg_tasker( )
```

Parámetros

info: Código de estado, de tipo entero regresado por la rutina.

pvm_send( )

Manda datos al buffer de mensajes activo.

Synopsis

```
int info = pvm_send (int tid, int msgtag)
```

Parámetros

tid: Identificador de tareas tipo entero para mandar procesos.
msgtag: Etiqueta de mensaje tipo entero dada por el usuario.
info: Código de estado, de tipo entero regresado por la rutina.

Ejemplo

C:

```
info = pvm_initsend (PvmDataDefault);
info = pvm_pkint (array, 10, 1);
msgtag = 3;
info = pvm_send (tid, msgtag);
```

pvm_sendsig( )

Manda una señal a otro proceso PVM.

Synopsis

int info = pvm_sendsig (int tid, int signum)

Parámetros

tid: Identificador de tareas tipo entero para mandar procesos.
signum: Número entero de la señal.
info: Código de estado, de tipo entero regresado por la rutina.

Ejemplo

C:

```
tid = pvm_parent ( );
info = pvm_sendsig (tid, SIGKILL);
```

pvm_setopt( )

Define varias opciones para libpvm.

Synopsis

int oldval = pvm_setopt (int what, int val)

Parámetros

what: Entero que define lo que se fija inicialmente. Las opciones son:

Option value		MEANING
PvmRoute	1	routing policy
PvmDebugMask	2	debugmask
PvmAutoErr	3	auto error reporting
PvmOutputTid	4	stdout device for children
PvmOutputCode	5	output msgtag
PvmTraceTid	6	trace device for children
PvmTraceCode	7	trace msgtag
PvmFragSize	8	message fragment size
PvmResvTids	9	Allow messages to reserved tags and TIDs
PvmSelfOutputTid	10	Stdout destination

PvmSelfOutputCode	11	Output message tag
PvmSelfTraceTid	12	Trace data destination
PvmSelfTraceCode	13	Trace message tag

val: Entero que especifica la nueva opción a colocar. Los valores predefinidos para la ruta son:

Option value	MEANING
PvmDontRoute	1
PvmAllowDirect	2
PvmRouteDirect	3

oldval: Entero que retorna el escenario previo de la opción.

Ejemplo

C:

```
oldval = pvm_setopt (PvmRoute, PvmRouteDirect);
```

pvm_setrbuf( )

Esta rutinas definen el buffer de recibo activo en bufid y guarda el estado del buffer anterior, y regresar el identificador del buffer activo previo oldbuf.

Synopsis

```
int oldbuf = pvm_setrbuf (int bufid)
```

Parámetros

bufid: Entero que especifica el identificador del buffer para el nuevo buffer activo de recibo.

oldbuf: Entero que retorna el identificador del buffer para el anterior buffer activo de recibo.

Ejemplo

C:

```
rbuf1 = pvm_setrbuf (rbuf2);
```

pvm_setsbuf( )

Cambia el buffer activo de envío.

Synopsis

int oldbuf = pvm_setsbuf(int bufid)

Parámetros

bufid: Valor entero con el identificador del para el nuevo buffer activo.
oldbuf: Contiene al identificador del antiguo buffer activo.

Ejemplo

C:

```
sbuf1 = pvm_setsbuf (sbuf2);
```

pvm_spawn( )

Inicia uno o más nuevos procesos PVM.

Synopsis

```
int numt = pvm_spawn (char *task, char **argv, int flag, char *where, int ntask, int
                      *tids)
```

Parámetros

task: Una cadena de caracteres que contiene el nombre del archivo ejecutable a ejecutarse como proceso PVM.
argv: Apuntador a un arreglo de argumentos para el archivo ejecutable, el final del arreglo se especifica con NULL.
flag: Valor de tipo entero indicando las opciones del spawn.
where: Cadena de caracteres especificando donde iniciar el proceso de PVM.
ntask: Entero especificando el número de copias del ejecutable a iniciar.
tids: Arreglo de enteros de longitud ntask.
numt: Valor entero regresado con el número de tareas iniciadas.

Ejemplo

C:

```
numt = pvm_spawn ("host", 0, PvmTaskHost, "sparky", 1, &tid[0]);
numt = pvm_spawn ("host", 0, (PvmTaskHost + PvmTaskDebug), "sparky", 1,
                  &tid[0]);
numt = pvm_spawn ("node", 0, PvmTaskArch, "RIOS", 1, &tid[i]);
numt = pvm_spawn ("FEM1", agrs, 0, 0, 16, tids);
numt = pvm_spawn ("pde", 0, PvmTaskHost, "paragon.ornl", 512, tids);
```

pvm_tasks( )

Regresa información acerca de las tareas corriendo en la máquina virtual.

Synopsis

```
int info = pvm_tasks(int where, int *ntask, struct pvmtaskinfo **taskp)
struct pvmtaskinfo {
        int ti_tid;
        int ti_ptid;
        int ti_host;
        int ti_flag;
        char *ti_a_out;
        int ti_pid;
} taskp;
```

Parámetros

where: Cadena de caracteres especificando sobre que tareas regresar información.
ntask: Entero que regresa el número de tareas para las cuales se está reportando.
taskp: Apuntador a un arreglo de estructuras las cuales contienen información acerca de la tarea.
tid: Entero que contiene el número de la tarea.
ptid: Entero que regresa el número de la tarea del padre.
dtid: Entero que tiene el tid del demonio de donde se está ejecutando esta tarea.
flag: Da el estado de la tarea.
aout: Cadena de caracteres que contiene el nombre de la tarea. Tareas iniciadas manualmente regresan un contenido en blanco.
info: Contiene el código de estado regresado por la rutina, es de tipo entero.

Ejemplo

C:

```
info = pvm_tasks (0, &ntask, &taskp);
```

pvm_tidtohost( )

Regresa el identificador del host en el cual la tarea especificada se ejecuta.

Synopsis

```
int dtid = pvm_tidtohost(int tid)
```

Parámetros

tid: Identificador de tipo entero.
dtid: Tid del demonio en el host solicitado.

Ejemplo

C:

```
host = pvm_tidtohost (tid[0]);
```

pvm_trecv( )

Recepción con interrupción.

Synopsis

```
int bufid = pvm_trecv ( int tid, int msgtag, struct timeal *tmount)
```

Parámetros

tid: Entero que corresponde al identificador de tarea del proceso enviado.
msgtag: Entero que corresponde a la etiqueta del mensaje; sería ¿=0.
tmount: Tiempo de espera antes de retornar el mensaje.
bufid: Entero que retorna el valor de el nuevo identificador del buffer activo de recibo . Un valor menor que cero indica error.

Ejemplo

C:

```
struct timeval tmount;
tid = pvm_parent( );
msgtag = 4;
if ((bufid = pvm_trecv (tid, msgtag, &tmount)) > 0){
    pvm_upkint (tid_array, 10, 1);
    pvm_upkint (problem_size, 1, 1);
    pvm_upkfloat (input_array, 100, 1);
}
```

pvm_upk*( )

Desempaca a el buffer de mensajes activo a los arreglos según el tipo de dato prescrito.

Synopsis

```
int info = pvm_unpackf (const char *fmt, ...)
int info = pvm_upkbyte (char *xp, int nitem, int stride)
int info = pvm_upkcplx (float *cp, int nitem, int stride)
int info = pvm_upkdcplx (double *zp, int nitem, int stride)
int info = pvm_upkdouble (double *dp, int nitem, int stride)
int info = pvm_upkfloat (float *fp, int nitem, int stride)
int info = pvm_upkint (int *ip, int nitem, int stride)
int info = pvm_upklong (long *ip, int nitem, int stride)
int info = pvm_upkshort (short *jp, int nitem, int stride)
int info = pvm_upkstr (char *sp, int nitem, int stride)
```

Parámetros

fmt: Expresión de tipo printf (de C) que especifica que desempacar.
nitem: El total de datos a desempacar (no el número de bytes).
stride: El tipo de empaquetación a ser usado mientras se desempacan los datos.
xp: Apuntador al principio del bloque de bytes.
cp: Arreglo complejo de por lo menos nitem*stride datos de longitud.
zp: Arreglo de doble precisión.
dp: Arreglo real de doble precisión.
fp: Arreglo real de punto flotante.
ip: Arreglo de enteros.
jp: Arreglo de dobles enteros.
sp: Apuntador a una cadena terminada en null.
what: Valor entero que especifica el tipo de dato desempacado.

Info: Valor entero que indica estado, regresado por la rutina, si es menor de cero indica error.

Ejemplo

C:

```
info = pvm_recv (tid, msgtag);
info = pvm_upkstr (string);
info = pvm_upkint (&size, 1, 1);
info = pvm_upkint (array, size, 1);
info = pvm_upkdouble (matrix, size*size, 1);
```

PERPECTIVAS Y PASOS A SEGUIR

Las perspectivas de este proyecto son mostrar de una manera sencilla la forma de trabajar con PVM y crear un pequeño manual de usuario (en español) para que cualquier persona que este interesada en este tipo de lenguaje de programación pueda entenderlo sin ningún problema. Así como el crear una fuente bibliográfica de programas, para dar una visión (muy general) de la manera de operar con este lenguaje, así como el que dichos programas sean de utilidad para un futuro próximo.

Para poder llevar acabo lo anterior fue necesario familiarizarnos con las instrucciones básicas del sistema operativo UNIX y el editor de textos Vi para después de esto aprender la manera en que trabaja PVM, el demonio de PVM, pero sobretodo el archivo Makefile.aimk que es un archivo muy importante, con el cual es posible la compilación de nuestros programas. Para finalmente ejecutarlos.

CONCLUSIONES:

En base a todo lo anterior, se puede decir que PVM es un lenguaje muy sencillo de manejar, y que la parte que estuvo un poco más difícil de entender fue la que se refiere al archivo Makefile.aimk, ya que es en ella donde se declaraban las rutas de las librerias a utilizar, el tipo de compilador, la ruta donde se envían los archivos ejecutables y la más importante, la forma de declarar nuestro programa para que este pudiera ser compilado y crearse su ejecutable.

El tipo de arquitectura que se utilizo fue una estación de trabajo SUN4.

Espero que todo lo anterior sirva como base para futuras investigaciones sobre PVM y que pueda servir de apoyo para los que apenas comienzan a familiarizarse con este lenguaje de programación.

BIBLIOGRAFIA

1. PVM3 User's guide and reference manual

2. URL=http://fciencias.ens.uabc.mx/~rnajera/so2/falla.html

3. URL=http://lola.ii.uam.es/~juan/docencia/pvm/tutorial.html

4. URL=http://fciencias.ens.uabc.mx/~marcia/reportefinal.htm#codificacionparalelo

5. URL=http://www.cenapadne.br/cursos/pvm3/pvm29.html

6. URL=http://www.cacalc.ula.ve/turoriales/load/node15.html

7. URL= http:/ fciencias.ens.uabc.mx/~malba/school/pvm.html

8. URL= http://fciencias.ens.uabc.mx/~vmunguia/so

9. URL= http://fciencias.ens.uabc.mx/~ssaucedo/tareas/so/pvm.html