Maximizando el Paralelismo: Ejecución de Tareas en Lote sobre PVM

Jorge Chaves
jchaves@exactus.com
Exactus Software Development
Costa Rica

Álvaro Rivera
alvaro.rivera@artinsoft.com
Artinsoft
Costa Rica

Francisco J. Torres-Rojas
torres@ic-itcr.ac.cr
Instituto Tecnológico de Costa Rica y
Universidad de Costa Rica
Costa Rica

Resumen: Parallel Virtual Machine (PVM) permite que una colección heterogénea de computadoras conectadas en red pueda ser vista como un solo recurso computacional o una gran máquina virtual. Este artículo muestra un mecanismo implementado sobre PVM para la ejecución de tareas en lote, capaz de establecer el momento en que cada una de las tareas invocadas termina su ejecución, permitiendo así un uso más eficiente de los recursos. Se detallan las funciones y mensajes nuevos introducidos en PVM que se mimetizan con la funcionalidad existente y permiten la coexistencia de demonios modificados con otros demonios PVM. Este artículo describe también con detalle los pasos a seguir para incluir nuevas funciones en el demonio PVM.
Palabras Clave: PVM, computación paralela, sistemas distribuidos, balance de carga.

Abstract: Parallel Virtual Machine (PVM) allows that a cluster of heterogeneous computers can be perceived as a large virtual machine. This paper describes a mechanism implemented on PVM that allows the execution of "batch processes", which in order to accomplish a more efficient resource usage, it is able to detect the instant when each one of the tasks finishes. The new functionality and the message protocol included to PVM are explained in detail. Since the proposed daemons can coexist with the standard daemons of PVM, the compatability with the existing functionality of PVM is guaranteed. It is also described the suggested procedure for including new functionality to PVM.
Palabras Clave: PVM, computación paralela, sistemas distribuidos, balance de carga.

1 Introducción

Actualmente la mayoría de empresas privadas, instituciones del estado y universidades cuentan con una gran cantidad y variedad de equipo de cómputo interconectado por medio de redes. En la gran mayoría de los casos, la capacidad de estos equipos es explotada en forma individual. Adicionalmente, el creciente número de aplicaciones de cómputo intensivo así como la necesidad de mayor poder de procesamiento a un precio más accesible [5] ha abierto las puertas al procesamiento paralelo y distribuido. Una de las alternativas que logra este objetivo es PVM, software que permite que una colección heterogénea de computadoras conectadas en red pueda ser vista como un solo recurso computacional [6]. PVM provee un mecanismo sencillo y eficiente de membresía en donde fácilmente se pueden agregar o eliminar nodos de la máquina virtual. De igual forma, el mecanismo que dispara tareas en nodos remotos es sencillo y transparente al usuario. Por otro lado, este último mecanismo no hace un uso eficiente de los recursos, dado que en forma nativa PVM separa las tareas utilizando un sencillo *round-robin*, con lo cual no se toma en cuenta la carga de trabajo de las máquinas que conforman la máquina virtual. De igual manera, PVM carece de un componente que permita establecer el momento en que una tarea dada termina su ejecución.

El presente artículo describe la funcionalidad implementada para que PVM sea capaz de ejecutar tareas en lote, señalando y describiendo los cambios específicos realizados sobre PVM. Un logro adicional es que las extensiones desarrolladas se integran a la arquitectura de programación original, mimetizándose con el resto de las estructuras y funciones de PVM. En la Sección **2**, se describirá en forma general la interfaz estándar de PVM para la invocación de procesos. En la Sección **3**, se contrasta en forma general la interfaz y funcionamiento de la interfaz desarrollada y en las subsecuentes secciones, a saber, desde la Sección **4** hasta la **7** se procede a detallar cada uno de los componentes y sus funciones para manejar y administrar una invocación en la implementación desarrollada.

2 Descripción General de la Invocación en PVM

Cuando un mandato PVM es invocado por una aplicación de usuario, se ejecuta únicamente el código suficiente para enviarle un mensaje a un proceso conocido como demonio de PVM (debido al nombre que se le da en UNIX a los programas del sistema [11, 13, 14]) con la solicitud del mandato y sus respectivos parámetros, como se muestra en la

Figura 1. Cada mensaje supone que el demonio está corriendo en la misma máquina física donde está el proceso que lo origina.

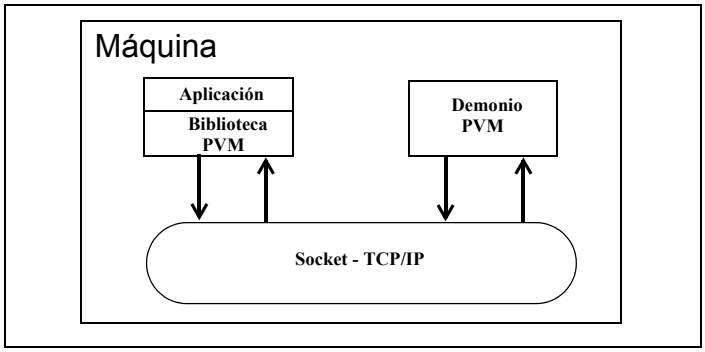

Figura 1. Comunicación entre el proceso usuario y el demonio PVM

Aunque internamente PVM se sustenta en las bibliotecas estándar de *sockets* [13] y el protocolo de comunicación TCP/IP [3,13], el usuario de PVM no necesita conocer el nombre de la máquina a la cual se desea enviar información, pues todas las máquinas reales se le presentan como una sola gran máquina virtual [7]. Una característica importante de los identificadores de tarea PVM es que parte del identificador determina al demonio de la máquina donde reside el proceso. Cuando la aplicación usuario ordena la creación de un proceso, éste puede ejecutarse en cualquiera de las máquinas pertenecientes a la máquina virtual PVM, pero la aplicación de usuario solo requiere comunicar el identificador para que el demonio establezca comunicación de demonio a demonio, hasta lograr aplicar cualquier solicitud de una función PVM a la tarea solicitada, en cualquier máquina donde ella se encuentre. Por ejemplo, la Figura 2 describe en forma básica el procesamiento de la invocación de la función `pvm_spawn`. Se menciona que en forma básica, pues PVM genera más mensajería si detecta, por ejemplo, que es la primera vez que la aplicación de usuario invoca a una función PVM..

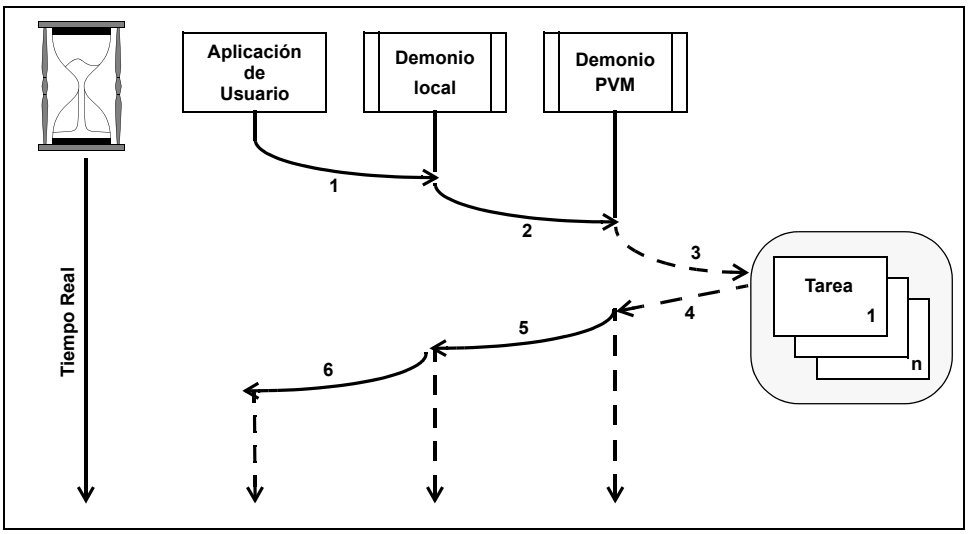

Figura 2. Comunicación para una llamada a `pvm_spawn()`

Para explicar la función de cada una de las entidades implicadas en la ejecución del `pvm_spawn` a continuación se describe como se originan cada una de las acciones que se llevan a cabo.

- **Paso 1:** La aplicación usuario invoca la ejecución de un nuevo programa utilizando la interfaz `pvm_spawn`, la cual envía un mensaje a través de un canal TCP/IP al demonio local. Mientras la llamada se resuelve, la aplicación

de usuario queda bloqueada automáticamente [11, 12, 13], esperando por el mensaje de respuesta que contiene el identificador de la tarea PVM recién creada.

- **Paso 2:** El demonio PVM local recibe un mensaje con una solicitud para creación de una o más tareas. A partir de la lista de demonios miembros de la máquina virtual, selecciona aquellos que cumplan con la solicitud y distribuye entre ellos la cantidad de tareas solicitadas. Él mismo puede ser uno de los demonios escogidos. El demonio envía un mensaje de tipo `DM_SPAWN` uno por uno a los demonios escogidos, permaneciendo bloqueado hasta recibir de cada uno de ellos los identificadores otorgados a los nuevos procesos.
- **Paso 3:** Llamada al sistema operativo solicitando la creación de un proceso.
- **Paso 4:** El demonio toma el identificador asignado por el sistema operativo y lo almacena en una de sus estructuras, además, le asigna un identificador propio de la máquina virtual.
- **Paso 5:** El demonio envía el identificador PVM al proceso que envió el mensaje `DM_SPAWN`.
- **Paso 6:** El demonio local recibe el identificador PVM que se le asignó a la tarea continuando el ciclo a partir del Paso 2 hasta conseguir disparar tantos procesos como se solicitaron y retorna, finalmente, todos los identificadores a la aplicación de usuario, la cual al recibir los valores se desbloquea y continua su ejecución.

En términos prácticos, un problema vinculado al hecho de disparar múltiples procesos es cómo recolectar y serializar su posterior salida de datos. Sin embargo, esto no representa ningún problema adicional, pues las plataformas computaciones usualmente proveen diferentes esquemas de comunicación entre procesos remotos tales como memoria compartida distribuida [12] o el sistema de mensajes [11, 12, 13]. En particular, PVM ofrece comunicación entre procesos por el sistema de mensajes, mediante el identificador de la tarea PVM y las primitivas `pvm_send` y `pvm_receive` [6]. PVM también cuenta con la función `pvm_catchout`, la cual permite a múltiples procesos PVM escribir en un solo archivo, y con la función `pvm_gather` que permite a los miembros de un grupo escribir una porción del resultado al cual luego tendrá acceso otro proceso denominado raíz [7, 9]. Esto permite que al utilizar operaciones como éstas, se pueda resolver la dependencia que puede existir entre los diferentes procesos a fin de abstraerlos como procesos independientes.

3 Interfaz para la Ejecución de un Proceso Batch

La función estándar y única de PVM para disparar procesos es la función `pvm_spawn` (cuya secuencia de pasos fue descrita en la sección anterior), con la interfaz que se puede apreciar en la Figura 3, la cual utiliza los siguientes parámetros: el nombre del programa a ejecutar, los parámetros a enviársele, otros dos parámetros que determinan en conjunto la colección de máquinas donde se disparará el proceso seleccionado (actuando como una especie de filtro), la cantidad de procesos a ejecutar en ellas y como parámetro de salida un puntero a un vector de enteros. Dada una invocación al mandato `pvm_spawn`, PVM responde disparando inmediatamente la cantidad de procesos solicitados y retorna en el arreglo de enteros los identificadores PVM de los procesos creados. Regresa el número total de procesos creados como resultado general de la función [7].

```
int numt = pvm_spawn (char *task,char ** argv, int flag, char *where,
                      int ntask, int *tids)
```

Figura 3. Interface de la función `pvm_spawn()`

La nueva función que da acceso al marco de ejecución propuesto en este trabajo se designó `pvm_batch` (Figura 4). Se conservaron los argumentos `flag` y `where`, se incorporó el parámetro `mode` que se explicará más adelante, y se combinaron los argumentos `task`, `ntask` y `argv` en un solo, aportando adicionalmente información respecto a algunas características de ejecución que podrían ser utilizadas posteriormente por módulos propuestos.

```
int id = pvm_batch (char *cmd,int flag,char *where,int mode)
```

Figura 4. Interface de la nueva función `pvm_batch()`

El argumento `cmd` es una hilera escrita en un pequeño lenguaje que describe como se debe paralelizar el código a ser usado (la especificación detallada de este lenguaje no será presentada en este artículo). Por ejemplo:

```
cc = pvm_batch("for i in [1..4] do foo(i)",0,"",1);
```

La función pvm_batch puede deducir cuantas copias de foo se deben ejecutar, las cuales pueden ser hechas en cualquier orden y en cualquier instante (incluyendo la posibilidad de ejecutarse en paralelo). La investigación desarrollada permite que el proceso foo pueda realizarse en diferentes computadoras pertenecientes a la máquina virtual PVM, escogidos según criterios especificados por el usuario y criterios provenientes del componente de balance de carga. A diferencia de la función pvm_spawn, la llamada pvm_batch no dispara instantáneamente todos los procesos, sino que los ejecuta paulatinamente, alimentándose de la información de ejecuciones previas y de la cantidad restante de trabajo, mediante un mecanismo de balance de carga introducido con el fin de conseguir un menor tiempo **total** de retorno de esta llamada. Además, toda esta funcionalidad logra hacer transparente al usuario los detalles de implementación que requeriría coordinar si tuvieran que ser manejados como parte de su programa. En pvm_batch no es posible contar con todos los identificadores de las tareas al momento de ejecución de la interface, pues algunas de ellas empiezan a ejecutar hasta un momento posterior, por lo que las extensiones de pvm_batch lo que hacen es tomar la tarea en general, registrarla y retornar un identificador global para la tarea. Debido a esta particularidad, cierta cantidad de trabajo se ejecuta en *background* [14] pues al no contar con los identificadores de las tareas en forma inicial se tiene la aparente imposibilidad de no poder interactuar o conocer el avance de las tareas que están atendiendo el llamado, fue por lo que se dio el nombre a la interfaz de pvm_batch como si se tratase de una tarea de procesamiento por lotes [11, 13].

La función pvm_batch se acopla naturalmente a PVM y funciona en compatibilidad con sus otras funciones. Por ejemplo, cuando en PVM una tarea dispara a otra, la segunda tarea disparada puede preguntar cuál es el identificador de la tarea que la disparó, utilizando la función pvm_parent. De la misma forma, pvm_batch permite conocer a todas las tareas disparadas el identificador del proceso que las ejecutó. Aunque en el momento de la invocación a pvm_batch no se conocen los identificadores de las tareas disparadas, ellas pueden establecer comunicación posterior con el proceso padre y notificarle, de ser necesario, su identificador de tarea; el cual es obtenido mediante la función pvm_mytid proporcionada por PVM. Así, en la práctica, la imposibilidad mencionada anteriormente de interactuar propiamente con las tareas, no existe y más bien se restringe al hecho que la comunicación no puede ser iniciada por la tarea padre, debido a que inicialmente ésta no conoce los identificadores de sus procesos hijos. La función pvm_batch trabaja utilizando dos modalidades de ejecución las cuales brindan dos diferentes mecanismos de sincronización entre procesos ya sea con o sin bloqueo (ver Secciones **3.1** y **3.2**) [11, 12, 13], que permiten a su vez, aprovechar en menor o mayor medida el paralelismo al dejar al usuario la actividad de verificar si la tarea ha terminado o no. Esta interfaz podría resultar una alternativa más sencilla para usuarios de PVM que requieren utilizar la función pvm_scatter pues es probable que estos usuarios busquen cómo implementar algo que este marco de ejecución ya les provee, tal y como lo sugiere la documentación de PVM al mencionar que futuras versiones de esta función podrían permitir un mayor grado de paralelismo mediante un uso más eficiente de la arquitectura [9]. La función pvm_batch se incluyó en la biblioteca "libpvm3.lib" con las interfaces de las otras funciones PVM para que el usuario pueda incluir las declaraciones y compilar sus programas escritos en el lenguaje C, de la misma forma que utiliza cualquier otra función de PVM.

3.1 Modalidad de Ejecución con Bloqueo

La modalidad de ejecución de pvm_batch con bloqueo (mode igual a 0), deja al programa que realiza la invocación bloqueado hasta que finalice completamente la ejecución del mandato solicitado [13]. Esta modalidad de ejecución normalmente aprovecha el paralelismo que existe en el potencial de ejecutar una tarea en forma distribuida entre varios *workers*, al mismo tiempo que permite que la lógica de programación se mantenga con la lógica de un programa secuencial, pues al momento de ejecutar la siguiente instrucción se tiene la certeza de que la función anterior ya finalizó. Los pasos que se dan en la interacción existente entre el proceso y el demonio en la llamada a pvm_batch con bloqueo, ilustrados en la Figura 5, son:

- **Paso 1:** Al igual que el pvm_spawn, la nueva función envía al demonio local la solicitud de ejecución de un nuevo proceso *batch* y se queda bloqueado esperando por un identificador de proceso *batch*.

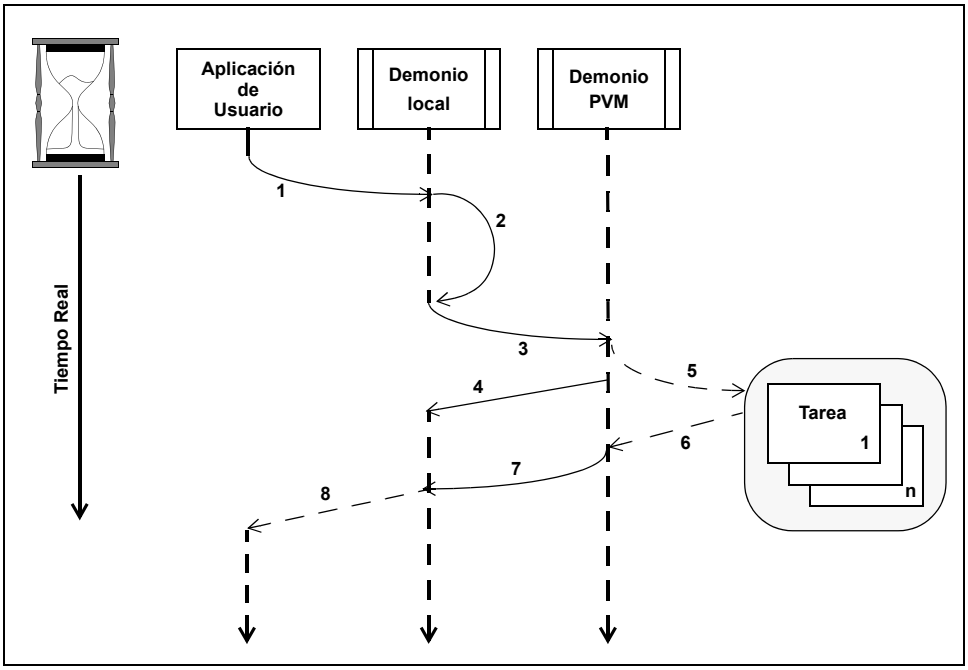

Figura 5. Comunicación para una llamada con bloqueo a `pvm_batch()`

- **Paso 2:** El demonio recibe la solicitud de una tarea *batch*, verifica y registra el código asociado al mandato tipo *batch* y crea un nuevo identificador de la misma forma que los genera PVM para cualquier otra tarea. Se envía un mensaje a sí mismo solicitando que se realice el proceso de asignación de cargas de trabajo.
- **Paso 3:** El procedimiento en el demonio encargado de realizar la asignación de cargas de trabajo es activado para realizar los cálculos de un proceso *batch* en particular. El proceso de asignación de carga construye la lista de posibles demonios que puedan atender la solicitud e invoca al componente de balance de carga para dar una asignación de trabajo a cada uno de ellos con la ayuda del mecanismo de ejecución del lenguaje. Para cada uno de los demonios elegidos para satisfacer la asignación de trabajo se envía un mensaje ordinario de tipo `DM_SPAWN`.
- **Paso 4:** El demonio tal y como reacciona originalmente con el mensaje `DM_SPAWN`, crea la tarea y retorna al demonio que originalmente hizo la solicitud el identificador de la nueva tarea PVM.
- **Paso 5:** El demonio crea una tarea local y ejecuta las acciones necesarias para vigilar cuando termina la ejecución de la tarea.
- **Paso 6:** Cuando finaliza el proceso *Worker* que llevó a cabo la asignación de trabajo se activa al demonio PVM para comunicar tal finalización a la administración del proceso *batch*.
- **Paso 7:** El demonio de la máquina donde acaba de finalizar la asignación de trabajo envía un mensaje de solicitud de balance de carga al demonio donde se encuentra la administración de proceso *batch*, indicándole de cuál proceso *batch* se trata y el identificador de la tarea. Finalmente, este recibe el mensaje y se inicia nuevamente la secuencia descrita a partir del Paso 3.
- **Paso 8:** Este procedimiento finaliza cuando el demonio que administra la tarea recibe un mensaje de solicitud de balance, procedente del demonio que atendió la última asignación de trabajo ejecutada y con la cual se finaliza la tarea. Si este es el caso, simplemente se limpian las estructuras utilizadas para la administración de proceso y se envía un mensaje con el identificador de tarea al proceso que realizó originalmente la llamada a `pvm_batch` desbloqueándolo y permitiéndole así continuar con su ejecución.

3.2 Modalidad de Ejecución sin Bloqueo

La modalidad de ejecución de `pvm_batch` sin bloqueo (`mode` diferente de 0), invoca al mandato e inmediatamente continua con la ejecución del programa sin esperar el tiempo necesario para que finalice la invocación del mandato *batch*. Este mandato permite al proceso principal realizar más procesamiento en forma paralela mientras se ejecuta el proceso *batch*. Note la semejanza con un proceso que se ejecuta en *background* [14]. La Figura 6 describe la

interacción existente entre el proceso y el demonio en la llamada a `pvm_batch` sin bloqueo, la cual es muy semejante a la llamada con bloqueo descrita en la sección anterior, salvo que retorna el identificador del proceso *batch* como segundo paso en lugar de que éste sea el último paso del proceso.

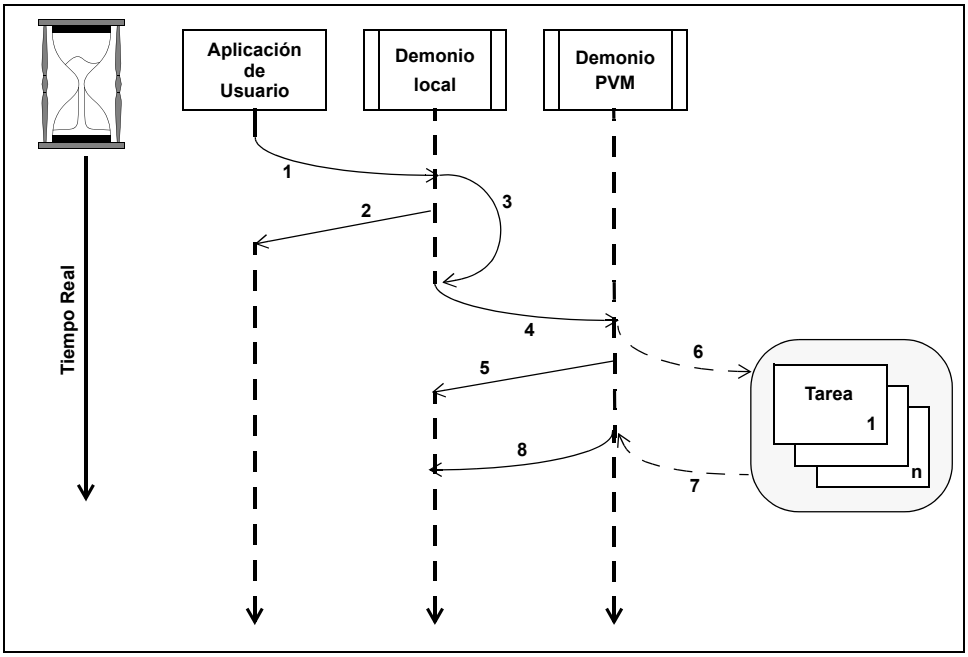

Figura 6. Comunicación para una llamada sin bloqueo a `pvm_batch()`

Esta modalidad de ejecución permite aprovechar mejor el paralelismo, pero presenta la problemática relativa a la sincronización [13] entre los procesos, por lo que se incluyó también la función `pvm_wait` en el marco de ejecución, la cual bloquea al proceso que realiza la invocación a esta llamada mientras esté activo el proceso *batch* asociado con el identificador de tarea *batch* indicado en el parámetro:

```
int bOk = pvm_wait (int id)
```

En caso que el identificador no exista o que la tarea ya hubiese terminado, entonces `pvm_wait` finaliza inmediatamente y retorna el control al proceso que realizó la invocación.

4 Mensajes de la Aplicación Usuario al Demonio

Cada una de las funciones a las que tiene acceso el usuario de PVM, así como ciertos subconjuntos de ellas, están representadas por diferentes marcas que se deben enviar al inicio de un mensaje para que el demonio pueda distinguir de qué petición se trata y procesarla en la forma adecuada. Cada una de las marcas es seguida por el conjunto de parámetros específicos y dependientes de cada marca y por lo tanto, de cada función PVM. Todas las marcas de los mensajes recibidos por el demonio que provienen de las aplicaciones locales se nombran con el prefijo `TM`, del inglés *Task Mmessage*. Estos mensajes, si bien utilizan algunas funciones utilitarias de PVM, se basan en el mecanismo estándar de *sockets* [13] el cual establece comunicación desde la aplicación usuario con el demonio de PVM, quien siempre está esperando por una conexión en un puerto estándar. Para la función `pvm_batch` se incluyó el mensaje `TM_BATCH`, mostrado en la Figura 7, el cual empaca en forma posterior a la marca, un parámetro de tipo *string*, seguido de un entero, luego otro *string* y finalmente un entero. También empaca variables de ambiente, tanto de PVM como del usuario, que luego serán establecidas como parte del contexto donde físicamente se ejecuta la aplicación. Nótese que al tratarse de un mensaje tipo `TM` es un mensaje que será enviado de aplicaciones usuarios al demonio local.

Cuando se usa el término “empaca” se hace referencia al hecho de que PVM establece sus propias funciones para

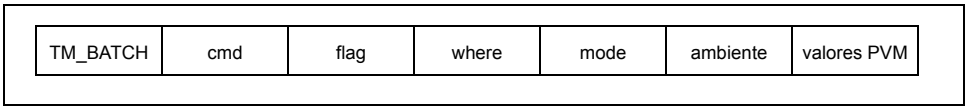

Figura 7. Mensaje de la aplicación de usuario al demonio

enviar cada tipo de datos con la intención de estandarizar y hacer transparentes detalles de la arquitectura de las diferentes máquinas que pertenecen a la máquina virtual PVM [13]. Una vez enviado el mensaje al demonio PVM, la tarea de usuario se queda bloqueada y esperando a que el proceso demonio realice el procesamiento del mensaje (ver Sección **5**) y retorne el mensaje que contiene el identificador del proceso *batch*, o en su defecto, un valor que corresponde a un error (se diferencian pues los mensajes de error son valores enteros negativos mientras que los identificadores de proceso son valores positivos diferentes de cero). En general, todo caso en que la función PVM tenga un valor de retorno, éste también es enviado del demonio a la tarea a través de mensajería de *sockets*, pero es esperado en la tarea de usuario como un simple mensaje de respuesta y no como un mensaje con marca de tipo `TM`.

5 Procesamiento de Mensajes de Tipo Tarea

Para el procesamiento de un mensaje de tipo *tarea* (con el prefijo `TM`), PVM se basa en el hecho de que estos mensajes en el fondo son un valor numérico entero constante y consecutivo, que es utilizado para referenciar un arreglo llamado `loclswitch`, que contiene en cada posición del arreglo y en forma correspondiente al valor del mensaje, la dirección del procedimiento o código en lenguaje `C` que lo atenderá. Cada uno de estos procedimientos recibe como tipos definidos en el código PVM, un parámetro con información relativa a la tarea que esta realizando la actividad y otro con el mensaje originalmente enviado por el proceso usuario que invocó la llamada PVM. El procedimiento que atiende específicamente el llamado a una función PVM debe conocer la cantidad de parámetros, su tipo y su orden. Este presume que vienen correctamente en el mensaje y procede a desempacarlos.

Cuando se hace un llamado a cualquier función PVM, el demonio verifica si la invocación es realizada por una tarea previamente registrada y de no ser así, establece mediante el mecanismo de mensajes de tipo demonio, las acciones para registrar la nueva tarea, enviándose mensajes a sí mismo para luego proceder a procesar ordinariamente el llamado original. En el caso de `pvm_batch`, el procedimiento desarrollado para atender la llamada de un mensaje de tipo `TM_BATCH` se llama `tm_batch` siguiendo el estándar utilizado en el código fuente de PVM.

6 Procesamiento de Mensajes TM_BATCH

El procedimiento `tm_batch` es llamado cada vez que el demonio PVM recibe un mensaje de tipo `TM_BATCH` por lo que éste procede a tomar cada uno de los parámetros del mensaje para iniciar su procesamiento. En el diseño aquí expuesto, se estableció que la administración de los procesos tipo *batch* sería descentralizada y que cada demonio sería responsable de administrar los procesos *batch* de las tareas que corren en la misma máquina real y que han invocado a `pvm_batch`. Dado lo anterior, `tm_batch` toma el primer parámetro correspondiente al programa `batch` enviado por el usuario y se invoca con él a la función `setBatchProcess`, la cual realiza el análisis sintáctico [1] y de semántica estática [7, 15]. En el caso de tratarse de un programa al cual no se le detectan errores, entonces se registra en las estructuras para el control de proceso como una tarea de tipo *batch* y se le asigna un identificador de tarea, aunque propiamente no se trata de una tarea PVM.

Para responder al programa que invocó a la llamada `pvm_batch` siempre se retorna un entero. En caso de error se empaca y retorna un valor negativo en concordancia con los mensajes de error definidos por el lenguaje y éste es retornado a la aplicación usuario inmediatamente. Por el contrario, si no hubo ningún error se procede a generar un identificador de tarea único y asignárselo a la solicitud como identificador de proceso *batch*, el cual es siempre un valor positivo y diferente de cero. Este identificador se retorna inmediatamente a la aplicación usuario solo si se trata de una llamada sin bloqueo. A diferencia del `tm_spawn`, encargado de procesar las solicitudes de `pvm_spawn`, el procedimiento `tm_batch` no procede a realizar las gestiones para disparar inmediatamente todos los procesos solicitados, sino que envía un mensaje al demonio local para indicarle que un proceso recién ha incluido. El mensaje enviado al demonio es un nuevo mensaje de tipo demonio llamado `DM_BALANCE`. En general, en cualquier parte del código del demonio PVM se puede enviar un mensaje al demonio de la misma máquina en que se encuentra una tarea, a partir establecer como dirección destino el resultado de aplicar la operación *or* de *bits* al identificador de la

tarea y la constante llamada `TIDPVMD` que contiene un valor máscara, lo cual es en el fondo, lo mismo a lo que tiene acceso una aplicación usuario a mediante la función `pvm_tidtohost` [7, 9]. El `tm_batch` adicionalmente extrae el parámetro entero llamado `flag` y el parámetro `where`, para ser almacenados en la estructura del proceso y utilizarlos cada vez que sea necesario distribuir cargas de trabajo entre los demonios PVM disponibles.

7 Mensaje de Solicitud de Balance

En forma análoga a los mensajes que provienen de las tareas, la arquitectura PVM también cuenta con un mecanismo para envío de mensajes de un demonio a otro o a sí mismo. Estos son los mensajes tipo DM (*Daemon Message*). Cada mensaje de tipo DM está asociado a una constante que sirve como referencia para acceder un arreglo llamado `netswitch`, que contiene la dirección de memoria de la función encargada de procesar un mensaje.

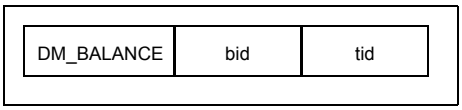

Figura 8. Mensaje de solicitud de balance

Como se muestra en la Figura 8, el mensaje de solicitud de balance incluye dos parámetros de tipo entero. El propósito del procedimiento que atiende un mensaje de tipo balance es realizar la administración del proceso *batch*, para lo cual recibe mediante un mensaje con la notificación de que se requiere continuar con el procesamiento de una tarea de tipo *batch*, la cual es identificada mediante el primer parámetro, así como la razón de tal solicitud en el segundo parámetro. Una llamada a una tarea *tm_batch* puede darse a raíz de dos motivos: ya sea porque recién se registró un nuevo proceso *batch* o que alguno de los *workers* encargados de realizar procesamiento para un proceso ya finalizó su asignación de trabajo. El parámetro `tid` corresponde al identificador de la asignación de trabajo que ha finalizado y como tal es el motivo de la solicitud de balance para asignación de carga de nuevo trabajo. Cada asignación de trabajo está asociada a un identificador entero mayor que cero. Se utiliza un cero en el parámetro `tid` para indicar que la solicitud es debida a la reciente inclusión de un nuevo proceso *batch* y no debido a la finalización de una asignación de carga.

Con base en los argumentos recibidos en el mensaje DM_BALANCE se procede a realizar las siguientes acciones:

- Utilizando el identificador de proceso *batch* se recupera su información de las estructuras de control de procesos *batch* para su uso y actualización.
- Si la solicitud se origina por la finalización de una asignación, se busca la asignación en las estructuras del balance de carga y a partir de la fecha y hora actual, se establece el tiempo total que requirió la tarea para ser realizada; adicionalmente se cambia el estado de la máquina real donde se estaba ejecutando esa asignación para que sea considerada como disponible
- Luego, utilizando los valores de `flag` y `where` almacenados como parte de la información del proceso, se realiza la selección de las posibles máquinas adscritas a la máquina virtual PVM que son candidatas para asignarles trabajo parte de la tarea *batch.* Esta selección se hace en forma idéntica a la realizada por el proceso `tm_spawn`. Nótese que al hacer esta selección cada vez que se hace una solicitud de balance, esta incluye las posibles máquinas que se hayan unido o salido de la máquina virtual PVM en el lapso de tiempo transcurrido desde la última solicitud de balance.
- Seguidamente, el proceso de balance de carga verifica si se puede considerar que alguna máquina ha fallado en virtud de la política de recuperación de errores.
- Luego, el algoritmo de balance de carga procede a calcular la cantidad de trabajo a asignar a cada una de las máquinas candidatas y solicita al mecanismo de ejecución del programa *batch* que se active y haga avanzar el programa hasta generar una hilera con el siguiente mandato o bloque de mandatos con todos los parámetros evaluados, utilizando la interfaz `getBatchCommand` de manera tal que se acerque a la cantidad de trabajo solicitada por el balance de carga.
- La hilera retornada por el mecanismo de ejecución es tratada por el balance de carga antes de incorporarla a sus estructuras de control. El mecanismo de ejecución genera una hilera que satisface el requerimiento y la retorna junto con la cantidad de trabajo exacto que la evaluación del lenguaje logró obtener.

- Es el mecanismo de ejecución quien detecta que ya no existe trabajo por ejecutar al determinar que el programa *batch* ya se ha ejecutado completamente. No importa cuantas veces se insista en obtener un intervalo de cualquier tamaño este siempre retornará una hilera nula y una asignación de cero cuando ya no existe trabajo pendiente.
- Finalmente, para cada una de las asignaciones a dispararse, se realiza un proceso muy semejante al realizado por `tm_spawn` enviando un mensaje de tipo `DM_SPAWN` con parámetros equivalentes a los descritos en la Sección **3**, pero se consigna específicamente la máquina real en donde se debe ejecutar el proceso. Conjuntamente al nombre del proceso, sus argumentos, se incluyen como variables de ambiente para el proceso, dos variables llamadas `PVMBATCHID` y `PVMTASKID` con valores enteros que informarán a la tarea, el identificador del proceso `batch` y un valor consecutivo único para identificar la asignación de trabajo. Se adoptó esta estrategia para no cambiar a PVM la cantidad o significado de los parámetros que reciben los demonios en una función tan importante como `dm_spawn`. De esta forma, si bien los demonios contienen modificaciones con todo el trabajo necesario para incluir el marco de ejecución de `pvm_batch`, no es necesario que se tenga que cambiar o compilar nuevamente las invocaciones de los programas usuario preexistentes que usan `pvm_spawn`.
- Justo antes de disparar los nuevos procesos, el programa de balance de carga actualiza el tiempo de inicio de la tarea.

Es importante mencionar, que al lanzar una tarea utilizando el mensaje `DM_SPAWN`, tal como lo hace la función `tm_spawn`, el demonio PVM habilita la recepción de mensajes para poder recibir el identificador del proceso recién creado. Esto hace que no solo se active la recepción de este mensaje, si no que permite que se reciban mensajes de cualquier otro tipo, en particular otro mensaje de tipo `DM_BALANCE`, el cual invocaría al procedimiento `tm_balance` que actualmente ya está en ejecución.

Debido a que la función `dm_balance` no es reentrante, en particular las secciones de que hacen llamados a funciones PVM para enviar o recibir mensajes, fue necesario recurrir a semáforos como mecanismo para brindar exclusión mutua al procesamiento de este mensaje. Además esto explica el comportamiento errático en presencia de *threads* de PVM notado durante el transcurso del otras investigaciones [4], pero que en su momento, solo se pudo atribuir en forma muy general a un error desconocido de PVM. El hallazgo de esta característica, hace reconsiderar que más que un error se trata de una limitación o requerimiento: si se utilizan las interfaces de PVM en las aplicaciones de usuario junto con *threads*, es el usuario quien tiene que considerar la interface o más específicamente las funciones de comunicación y estructuras de datos como un recurso en competencia y brindarles exclusión mutua, dado que las funciones de interface son exhaustivas en uso de las funciones utilitarias de comunicación de mensajes que en general no son reentrantes.

8 Ejecución de una Asignación de Trabajo

La ejecución de las tareas que llevan a cabo una asignación de trabajo para la llamada `pvm_batch` se llevan a cabo, en última instancia, por el procedimiento llamado `forkexec`, el cual es el mismo que realiza la ejecución de los procesos de la llamada `pvm_spawn`. Para el correcto funcionamiento de un proceso `pvm_batch` es necesario que el administrador del proceso *batch* sea notificado cuando un proceso que realiza una asignación de trabajo ha finalizado, mediante el proceso descrito en la Sección 6. Dado este requerimiento, el procedimiento `forkexec` ha sido la única función dentro de esta investigación que fue necesario modificar respecto a su código original, pues en todos los demás casos lo que se hizo fue agregar nueva funcionalidad que convive con el resto del demonio PVM y no una modificación como en este caso. El reto con respecto a la modificación de esta función consistió en tener el menor impacto posible sobre la codificación original de PVM. Es un hecho, que para notificar al administrador del proceso *batch*, este procedimiento necesita conocer información tal como la máquina donde se encuentra el demonio de administración del proceso, el identificador del proceso *batch*, así como el identificador de la asignación de trabajo a que corresponde la tarea que está pronto a ejecutarse.

El mecanismo más sencillo para dar a conocer estos valores pudo haber sido incluirlos entre los parámetros de `DM_SPAWN`, lo que hubiese hecho necesario modificar el `tm_spawn` mismo y además hubiese determinado que la estructura del mensaje establecido en esta investigación difiriera de la de un demonio PVM original, haciendo incompatibles la coexistencia en una máquina virtual PVM del demonio original con otro demonio que contiene las modificaciones propuestas en esta investigación. No obstante lo anterior, se estableció un mecanismo igualmente

sencillo pero muy creativo que hace llegar al `forkexec` la información necesaria, al incluirlas como variables de ambiente para la ejecución de la aplicación, permitiéndose así la coexistencia de demonio original con el demonio que incluye y maneja procesos *batch*. Dichas variables son `PVMBATCHID` y `PVMTASKID`, que como se mencionó en la sección anterior, son incluidas como variables de ambiente por el procedimiento `tm_balance`. Además, como también se mencionó anteriormente, si se conoce el identificador de una tarea se puede enviar un mensaje al demonio local de esa tarea. Dado que muy convenientemente se asignaron identificadores PVM a los proceso *batch*, se puede hacer llegar el mensaje al demonio donde estaría esa tarea y por lo tanto, donde reside el proceso de administración del proceso *batch*.

En el caso que un demonio sin modificaciones dispare la ejecución de `forexec` en un demonio modificado, no existe ninguna diferencia funcional pues el `forkexec` modificado conserva iguales capacidades que el original. Esto implica, que se puede satisfacer cualquier invocación a `pvm_spawn` en cualquier demonio, modificado o no, en forma completamente normal. En el caso contrario, si un demonio modificado dispara la ejecución de `forkexec` en un demonio sin modificaciones, éste último no incluiría la funcionalidad para informar la finalización de la tarea al administrador del proceso *batch*, por lo que eventualmente, el administrador del proceso lo marcará como un demonio con fallo aunque éste haya finalizado satisfactoriamente y no utilizará más ese demonio como una opción para ejecutar una asignación de trabajo. Dado que el `forkexec` es dependiente del mecanismo de invocación de procesos que tiene el sistema operativo de la máquina real, PVM tiene en forma general dos diferentes códigos para realizar el `forkexec`, los cuales son incorporados en PVM según directivas de compilación en función de la plataforma de que se trate. En forma equivalente, fue necesario realizar dos diferentes modificaciones para identificar y notificar cuando una tarea responsable de realizar una asignación de trabajo ha finalizado. Uno para arquitecturas basadas en sistema operativo UNIX (incluyendo a Linux) y otra para sistemas operativos de 32 *bits* basados en Windows.

8.1 Modificación de forkexec para Ambiente UNIX

En el caso de las arquitecturas basadas en UNIX se solucionó utilizando las llamadas al sistema `fork` y `wait`. Como se muestra en la Figura 9, el `forkexec` original utiliza la llamada a la función `fork`, para crear dos procesos idénticos. El padre continua sin reparo a la ejecución del hijo, registrando en las estructuras la creación del nuevo proceso e informando el `id` otorgado al demonio que originalmente solicitó su creación. Mientras que el proceso hijo, reemplaza con la tarea solicitada el espacio del proceso que antes ocupaba una copia del demonio PVM mediante la llamada a `execv`, en caso de que esta llamada no funcione y no sea posible tal sustitución, el código incluye un `exit` para terminar la ejecución de la copia del proceso demonio [3,11, 13].

`forkexec` **original**	`forkexec` **modificado**

`pid = fork();`
`if (!pid) {`
`execv(path, argv);`
`exit(1);`
`}`
`//Actualizar estructuras`
`//Continuar demonio PVM`
	`pid = fork();`
	`if (!pid) {`
	`pid2 = fork();`
	`if (!pid2) {`
	`execv (path, argv);`
	`exit(1);`
	`} else {`
	`wait(&estatus);`
	`if (estatus == 0)`
	`{`
	`//Notificar`
	`}`
	`exit(0);`
	`}`
	`}`
	`//Actualizar estructuras`
	`//Continuar demonio PVM`

Figura 9. Pseudocódigo de la modificación del forexec para Unix

La modificación hecha recurre nuevamente al `fork`, haciendo que el proceso padre continúe sin cambios como en el original, pero que el proceso hijo, mediante el uso nuevamente del `fork`, se separe en dos procesos. Uno que realizará lo que en el proceso original hacia antes el hijo y el otro que esperará por la señal de que el otro proceso hijo terminó utilizando la llamada al sistema `wait`. Cuando el `execv` haya sido exitoso y el proceso que fue ejecutado finalice sin reportar un error, será registrado en la variable de `estatus`. Así, si `estatus` denota que el proceso que atendió la asignación de tarea se realizó correctamente, después de la llamada `wait`, se procede a notificar mediante la mensajería PVM al demonio que administra la ejecución del proceso *batch*. Esto lo realiza enviando un mensaje `DM_BALANCE` con el identificador de proceso *batch* y el identificador de la asignación de trabajo recibidas inicialmente como parte de las variables de ambiente. Cuando se conoce que la ejecución no fue exitosa, no se envía ningún mensaje, por lo que eventualmente el mecanismo de recuperación de fallos en el administrador del proceso *batch* se percatará de la situación y programará nuevamente esta asignación de trabajo.

8.2 Modificación de forkexec para Ambiente Windows

En el caso de la modificación para plataformas basadas en sistema operativo Windows de 32 *bits*, se crea una estructura en memoria donde se almacena información sobre el proceso *batch* y el identificador de asignación de tarea. Posteriormente, se crea un *thread* en estado suspendido y se le proporciona como datos la estructura en memoria mencionada, como se observa en la Figura 10 [2, 10].

`forkexec` **original**	`forkexec` **modificado**

`pid = CreateProcess(…);`
`//Actualizar estructuras`
`//Continuar demonio PVM`
	`tDta.bpid =`
	`getenv(bpid);`
	`tDta.btid =`
	`getenv(btid);`
	`tid = CreateThread(…);`
	`tDta.pid =`
	`CreateProcess(…);`
	`ResumeThread(tDta.pid);`
	`pid = tDta.pid`
	`//Actualizar estructuras`
	`//Continuar demonio PVM`

Figura 10. Pseudocódigo de la modificación del forexec para Windows

Luego se crea el proceso que atenderá la asignación de la tarea *batch*, se almacena el identificador del proceso en la estructura en memoria y se reactiva la ejecución del *thread*. La única función de este *thread* es monitorear el proceso que se le indica en el parámetro de datos para verificar que continua activo, lo cual se realiza mediante una invocación periódica a la llamada al sistema `GetExitCodeProcess`, como se muestra a continuación [10].

```
while(GetExitCodeProcess (tDta->pid, &estatus) && estatus == STILL_ACTIVE)Sleep(1000);
```

Cuando se constata que el proceso ha finalizado, el código restante del *thread* procede a enviar un mensaje `DM_BALANCE` al demonio encargado de la administración del proceso `batch`.

Aunque aparentemente también la modificación para Windows resulta ser sencilla, incluye el detalle de que ahora el demonio tiene dos hilos en ejecución y que en un momento especifico podrían estar compitiendo por el uso de la función `sendmessage` de PVM para enviar el mensaje `DM_BALANCE`, por lo que para evitar éste contratiempo fue necesario tratar a la función como una región crítica y garantizarle exclusión mutua mediante el uso de un semáforo [10, 11, 13].

9 Conclusiones

Este artículo presenta una descripción de la operación que utiliza PVM para la invocación de un proceso. Se estableció en forma general las características y pasos seguidos por las modificaciones realizadas a PVM en la implementación desarrollada.

La posibilidad de disparar tareas en lote en conjunto con la capacidad de determinar el momento justo en que termina cada tarea brinda grandes oportunidades en cuanto a la adecuada utilización de recursos. Aunque el componente de balance de carga no fue descrito con detalle, se puede derivar fácilmente el beneficio que se obtiene al contar con un mecanismo que dispare tareas con cierta carga de trabajo en los nodos de la máquina virtual que han reportado, mediante la terminación de sus tareas asignadas, una mejor eficiencia en la ejecución de dichas tareas.

Esta investigación podría acoplarse a muchas de las tecnologías ya existentes para realizar computación distribuida o paralela sobre HNOW (*Heterogeneous Networks of Workstations*) tales como CORBA, RMI, RPC, DCOM; pero se utilizó PVM para instrumentar una prueba de los cambios propuestos en el mecanismo de ejecución debido a las facilidades que esta plataforma brinda.

10 Referencias

1 **Aho, Alfred; Sethi, Ravi y Ullman, Jeffrey**, "*Compiladores, Principios, técnicas y herramientas*", Addison-Wesley Iberoamerica, U.S.A., 1990

2 **B. Nichols, D. Buttlar and J. P. Farrell**, "*Pthreads Programming*", O'Reilly, 1996

3 **Comer, Douglas**. "*Redes Globales de Información con Internet y TCP/IP: Principios básicos, protocolos y arquitectura"*, México: Prentice-Hall, 1996.

4 **Chaves, Jorge; González, Alejandro y Rivera, Álvaro.** "*Un modelo para procesamiento distribuido utilizando PVM"* Tiempo Compartido, 2001, Vol 5, N° 1. 35-40.

5 **Dan L. Clark, Jeremy Casas, Steve W. Otto, Robert M. Prouty, Jonathan Walpole.** *"Scheduling of Parallel Jobs on Dynamic, Heterogenous Networks*", Technical report, Dept. of Comp. Sci., Oregon Graduate Institute of Science and Technology, http://www.cse.ogi.edu/DISC/projects/cpe/, 1995.

6 **G.A Geist et al,** *"PVM 3 user's guide and reference manual*", Technical Report ORNI/TM12187, Oak Ridge National Laboratory, 1994

7 **Geist, Al; Beguelin, Adam; Dongarra, Jack; Jiang, Weicheng y Sunderam, Vaidy**. *"PVM: Parallel Virtual Machine A user's Guide and Tutorial for Networked Parallel Computing*" E.U.U.U.: MIT Press, 1994.

8 **Kernighan, Brian y Ritchie, Dennis.** *"El lenguaje de programación C",* México, Prentice-Hall 1991

9 **Cray Research, Inc**, "*Message Passing Toolkit: PVM Programmer's Manual*", Technical Report SR–2196 1.2. , 1998

10 **R. J. Simon**, *"Windows NT WIN32 API Superbible*", Waite Group Press, 1998

11 **Silberschatz, Abraham y Galvin, Peter**. *"Operating System Concepts*", E.E.U.U.: Addison-Wesley, 1997.

12 **Singhal, Mukesh y Shivaratri, Niranjan G**. *"Advanced concepts in Operating Systems: distributed, database, and multiprocesos operating systems*", E.E.U.U.: MacGraw Hill, 1994.

13 **Tanenbaum, Andrew**. *"Sistemas Operativos Distribuidos*". México: Prentice-Hall. 1996.

14 ---, *"Berkeley UNIX User's Manual Reference Guide"*, 4.3 BSD. Computer Systems Research Group, Computer Science Division, EECS, University of California, Berkeley, April 1986

15 **Watt, David**. *"Programming Language Processors*". Inglaterra: Prentice Hall, 1993

16 **Cheng-Zhong XU, Francis C.M LAU**. *"Iterative Dynamic Load Balancing in Multicomputers*". Journal of Operational Research Society. Vol 45, No 7, July 1994, pp 786-796

17 **Haiying Cai, Oliver Mquelin, Prasad Kakulavarapu, Guan R. Gao**. *"Design and Evaluation of Dynamic Load Balancing Schemes under a Fine-grain Multithreaded Execution Model"*. Technical Report. Proceedings of the Workshop on Multithreading Execution, Architecture and Compilation, 1999.

18 **Matthias Neeracher**. *"Scheduling for Heterogeneous Opportunistic Workstation Clusters"*. PhD Thesis. Swiss Federal Institute of Technology, Zurich, 1998.

19 **Stella C.S Porto, Celso C. Ribeiro**. *"Parallel Stragegies for Task Scheduling Algorithms Using PVM*". Technical Report. PUCRioInf-MCC06/95. Abril, 1995

20 **T. L. Casavant and J. G. Kuhl**. *"A taxonomy of scheduling in generalpurpose distributed computing systems*". IEEE Trans. on Software Eng., 14(2):141–154, February 1988.