Universidad Nacional de La Plata

Septiembre de 2002

Facultades de Ingeniería
Departamento de Electrotecnia
Cátedra de Trabajo Final

Facultad de Informática
Trabajo de Grado

Consola de Computador

Paralelo en MPI

Informe Final

Autores: Mónica Beatriz Vitta y José Ignacio Gialonardo.
N° de Alumnos: 602/5 y 44336.
Director: Ing. Antonio Adrián Quijano.
Co-Director: Lic. y MS Fernando G. Tinetti.

Dedicatorias

Estoy agradecida con todas las personas que contribuyeron en la elaboración de este libro. Principal y primordialmente, al personal de la Facultad de Informática, que me permitió presentar un Trabajo de Grado realizado en otra Facultad (Facultad de Ingeniería) y en grupo con un alumno de la misma (es decir de otra carrera). A José Ignacio Gialonardo compañero en el desarrollo y la elaboración del trabajo desde el inicio y en su totalidad, a quién agradezco inmensamente su confianza depositada en mi persona (siendo alumna de otra carrera, en otra Facultad, con pocos conocimientos comunes a él), su amabilidad y comprensión para trabajar en conjunto y su dedicación totalmente incluyente para la realización y conclusión de nuestro propio trabajo, del cual nos atrevemos a proclamar que hemos cubierto con fortuna una nueva etapa en la que no habrá de faltarnos, la generosa compañía de las personas mencionadas a continuación que tan cordialmente se han identificado con nuestra labor. Al Ing. Antonio A. Quijano director del trabajo de gran talento, que me enseñó temas importantes por su amplia colaboración de conocimientos y experiencias brindadas en todos los temas implicados en el desarrollo de la consola, como así también, me proporcionó mucho material de lectura siendo de gran ayuda para entender mejor los temas, además de su apoyo moral y primordialmente su respeto, tanto en lo personal como en lo profesional. De igual manera agradezco a MSc. Fernando Tinetti co-director por su preparación que me brindó muchas sugerencias útiles, corrección de los contenidos presentados, tiempo y soluciones en los pasos a seguir para poder cumplir en tiempo y forma la presentación del material que implicó dicho proyecto, hasta la conclusión del trabajo. Un agradecimiento especial a todas las personas que trabajan en el CeTAD (Centro de Técnicas Analógico - Digitales), ambiente de realización del trabajo, por su amplia colaboración en otros aspectos de importancia que me permitieron desarrollar gustosa y gratamente el mismo, en este punto, quiero agradecer también al Ing. José Rapallini por su labor en la cátedra de Trabajo Final en la Facultad de Ingeniería, quién aprobó los informes parciales que mi compañero debía ir presentando, lo que me sirvió de guía para saber que lo trabajado, elaborado, planteado y presentado estaba correcto. Agradezco en particular a mi esposo, Alejandro, por su amor, valoración, talante y serenidad constante hasta el final de mi trabajo.

Mónica Beatriz Vitta.

Quiero agradecer en primer lugar a Dios por rodearme de personas que me han apoyado y animado durante la carrera, y especialmente en esta ultima etapa. Con esto quiero expresar mi gratitud a mi compañera de Trabajo Mónica Beatriz Vitta quien por su ayuda, compañerismo, dedicación, voluntad y positivismo ha sido de suma importancia para el desarrollo de este trabajo, también quiero agradecer la guía, paciencia, dedicación y tesón del Ing. Antonio Adrián Quijano, el cual considero, no solo el director de mi Trabajo Final, sino también una persona digna de ser imitada, tanto por sus capacidades profesionales como personales. Agradezco también la colaboración de mi codirector Lic. y MS Fernando G. Tinetti, quien aportó cosas muy valiosas para este trabajo, inclusive fue quien tuvo la iniciativa de crear una consola MPI. No quiero dejar de mencionar la colaboración, sumamente importante, sobre todo en los comienzos, del Ing. José Rapallini, quién proporcionó mucha ayuda para la decisión, y planificación del trabajo; quiero extender este agradecimiento a toda la cátedra de Trabajo Final, la cual ha tomado éste como un nuevo desafío para integrar alumnos de distintas facultades.

Agradezco a todos los que trabajan en el CeTAD (Centro de Técnicas Análogo-Digitales), nuestro lugar de trabajo, los que crearon un ambiente propicio para el desarrollo del mismo.

Quiero mencionar por último un profundo agradecimiento a mis padres, por su esfuerzo y apoyo, tanto económico como anímico, y a personas especiales que han estado muy cerca mío, como mi futura esposa, Estela, y mi hermana, a quienes agradezco su apoyo, amor, comprensión y paciencia.

A todos los mencionados y al resto de mi familia dedico este trabajo, en el cual hay invertido mucho tiempo, dedicación, paciencia y esmero. Gracias.

José Ignacio Gialonardo.

Resumen

Este trabajo describe el diseño de una consola de computador, que permite al usuario, configurar fácilmente un sistema de computación paralelo, sobre una red de estaciones de trabajo.

Se emplea la interfaz de pasaje de mensajes (MPI) como herramienta de programación paralela en sus conocidas implementaciones MPICH y LAM. Para la primera se consideran dos “dispositivos” (uno de ellos para generación estática de procesos, y el otro para el caso dinámico). Para LAM sólo existe la generación dinámica.

Se comienza con una breve discusión sobre computo paralelo, incluyendo los conceptos principales sobre redes y sus topologías. Luego se introducen los principios de MPICH y LAM considerados como bibliotecas de funciones externas de pasaje de mensajes aplicadas al lenguaje C.

Se ha estudiado la consola introducida con el sistema PVM previamente conocido para pasaje de mensajes, analizando su operación y comandos como la base inicial de nuestra nueva Consola MPI.

Finalmente se describe la Consola MPI con su instalación, condiciones operativas, comandos específicos y las ventajas que puede presentar al usuario de computo paralelo.

Se agregan dos breves apéndices, (respectivamente sobre sistema operativo UNIX y lenguaje C) que resumen conocimientos que se debieron adquirir para la realización del trabajo.

Abstract

This paper presents the design of a computer console allowing the user to configure a parallel computing system on a network of workstations (NOW) as a multicomputer platform.

The Message-Passing-Interface (MPI) was used as a software tool on both MPICH and LAM implementations. For the first we considered two different devices (one for static and the other for dynamic process generation); and LAM with only the dynamic behavior originally defined.

We begin with a short discussion on parallel computing, covering also the main concepts about networks and their topologies. Next we introduce the principles of MPICH and LAM as libraries of external procedures for message passing applied to the C programming language.

We studied the console previously introduced with the former PVM system of message passing, analyzing its operation and commands as a basis for the design of our new MPI console.

Finally with describe our console with its installation, operating conditions, specific commands, and the benefits it may introduce for the user of parallel computing.

Two appendixes are added (the first on the UNIX operating system and the second on the C programming language). They are a summary of some basic concepts necessary for our work.

Contenido

INDICE

Introducción

Este trabajo se refiere a la utilización de una red de estaciones de trabajo (Network of Workstations, NOW) en computación paralela. Es bien conocida la creciente utilización de este tipo de arquitectura, en razón de la buena relación que con ella se obtiene, entre "performance" y precio. Se basa en la modalidad de comunicación entre procesos por medio de **"pasaje de mensajes"** realizada mediante el uso de un lenguaje secuencial de alto nivel como lo es el "C", desde el cual se efectúan llamados a una biblioteca de funciones externas específicas para implementar dicho pasaje de mensajes. Es fundamental para ello, establecer un método para crear procesos que se ejecuten sobre diferentes computadores, existiendo dos casos típicos:

- *Creación estática de procesos* : Se los especifica **antes** de la ejecución, y el sistema procesará un número fijo de los mismos. Se presta especialmente a la programación "SPMD" (single program, múltiple data), con un único programa para todos los procesadores.
- *Creación dinámica de procesos:* Éstos pueden surgir y su ejecución iniciarse durante la ejecución de otros procesos, haciendo posible el modelo más general llamado MPMD (múltiple program, múltiple data), en el cual **diferentes** programas se ejecutan concurrentemente en los distintos procesadores. Esta creación dinámica es un recurso más potente que la estática. Pero tiene el inconveniente de determinar intervalos considerables durante dicha creación.

Han tenido gran difusión dos sistemas que permiten implementar el pasaje de mensajes:

- **PVM** (Parallel Virtual Machine), del Oak Ridge National Laboratories, que se obtiene sin cargo, crea procesos dinámicamente, y constituye un ambiente bastante completo para el desarrollo de computación paralela.
- **MPI** (Message Passing Interface), que es en realidad una norma que establece una amplia biblioteca de funciones (más de 120) con todas sus especificaciones (elaboradas por un comité muy numeroso de académicos y miembros de la industria). La norma no establece, a diferencia de PVM, las maneras de crear el Sistema Paralelo en base a una determinada red, y ello queda a cargo de las diversas **implementaciones** que se crean para cumplir la norma.

Existen diversas implementaciones de MPI. Entre ellas, las más conocidas y accesibles sin cargo son:

- **MPICH**, que crea procesos en forma estática (últimamente también dinámico).
- **LAM**, que puede tener un comportamiento dinámico.

Ninguna de estas implementaciones ofrece una "consola" a la manera de PVM, que permita a través de una serie de comandos propios, simplificar la tarea de constitución del "sistema paralelo" en base a nodos que se pueden agregar o quitar de la misma; dando en cada momento la configuración del sistema paralelo; y eliminarlo "matando" limpiamente los procesos en marcha.

Todo ello resulta en una operación mucho más cómoda y relativamente automatizada de las aplicaciones paralelas, en mayor grado que las mencionadas implementaciones de MPI, que invocan estas operaciones a través de la línea de comandos de UNIX.

Se trata, en consecuencia, de crear una consola que facilite la aplicación de MPI tanto por vía de MPICH como de LAM. El usuario deberá poder elegir una u otra de estas implementaciones, y a partir de allí:

- Identificar los computadores que se puedan utilizar de una red local dada.
- Poner en marcha el sistema paralelo, con determinados nodos.
- Monitorizar aplicaciones paralelas basadas en MPI, y hacer posible el análisis de "performance".
- Eliminar procesos y desactivar el sistema paralelo.

La consola mejora el desarrollo e implementación del sistema paralelo con MPI, y las tareas de mantenimiento y monitorización del mismo construido a partir de una red de PCs (Network of Workstations, NOW) con MPI.

A continuación haremos una breve síntesis de los contenidos de cada capitulo:

Capitulo 1: Mencionamos el concepto de arquitectura y los tipos de computadores paralelos, los objetivos del procesamiento en éstos, y además indicamos las formas de programación paralela y algunos de los lenguajes que se utilizan en éstas.

Capitulo 2: Explicamos de manera general y breve, el concepto de red de computadores, que les permite comunicarse entre sí, los casos donde es más conveniente la existencia de una red y sus distintas configuraciones para la transmisión de datos.

Capitulo 3: Describimos detalladamente la norma para el paso de mensajes MPI, que consiste en una librería de funciones, donde las básicas son definidas con su estructura para saber cómo invocarlas desde un programa en C. Luego definimos las implementaciones LAM y MPICH de MPI, explicando todos los pasos a realizar, desde su instalación hasta la aplicación de todos los comandos existentes en cada una de ellas.

Capitulo 4: Explicamos las características, funcionamiento y todas las funciones de la consola PVM, para luego considerar de ella los aspectos relevantes para el desarrollo de nuestra Consola MPI.

Capitulo 5: Detallamos paso a paso todos los requisitos que debemos tener en cuenta para el desarrollo de nuestra Consola MPI, comenzando con la definición de las variables de entorno y PATH que fueron necesarios, la creación de cuenta de usuario, la instalación de la consola propiamente dicha, el funcionamiento de la misma con todos los comandos creados para cada implementación y cada dispositivo; y por último enunciar los beneficios que nos brindó desarrollar nuestra Consola MPI.

Apéndice A: Hemos tratado de describir en forma superficial el sistema operativo UNIX, tanto a nivel de la estructura y el diseño del sistema como a nivel de usuario y de administración; además de describir el sistema de archivos, el control de los procesos, las herramientas de las que dispone el usuario cuando se encuentra dentro del UNIX, como así también sus comandos principales.

Apéndice B: Se ha descrito brevemente el lenguaje de alto nivel de programación “C”, mostrando la manera de codificar un programa fuente, utilizando librerías y funciones que ofrece dicho lenguaje, como así también las distintas estructuras de control que se pueden utilizar y la manera de trabajar con archivos, para el almacenamiento de los datos.

Capítulo 1

1. SISTEMAS DE PROCESAMIENTO PARALELO

1.1. Introducción

Consiste en ejecutar un programa en un mismo intervalo de tiempo sobre diferentes procesadores, siempre que dicho programa admita una descomposición en múltiples procesos, tal que cada proceso pueda ejecutarse en procesadores distintos.

Como existen muchas aplicaciones que necesitan procesar gran cantidad de datos en muy poco tiempo, la velocidad es crucial y la única forma de tratar estos problemas implica utilizar procesamiento paralelo, pues si varias operaciones se pueden ejecutar simultáneamente, el tiempo total de procesamiento se verá reducido.

Desde el punto de vista de las aplicaciones debemos comprender que: a mayor complejidad de cálculo y mayor compromiso con la ejecución en tiempo real, se hace imprescindible utilizar procesamiento paralelo para obtener tiempos de respuesta aceptables.

Por otra parte, la evolución en la tecnología tiende al procesamiento paralelo, debido a que: los costos favorecen la reducción de tamaño de los procesadores para incrementar el rendimiento, la máxima velocidad alcanzada por el reloj de cualquier procesador implica un límite al mínimo ciclo de operación de un procesador sincrónico, lo que hace inevitable el paralelismo, aunque la velocidad de procesamiento requiere componentes de tecnología especial que consumen más, lo cual disminuye la confiabilidad; en procesamiento paralelo se pueden utilizar tecnologías relativamente más lentas y a su vez más confiables, y distribuir el procesamiento implica distribuir la memoria local, lo cual incrementa el ancho de banda global alcanzable por el sistema.

En definitiva, los objetivos del procesamiento paralelo son:

1) Disminuir los tiempos de ejecución.
2) Incrementar la productividad.
3) Atender fenómenos del mundo real que suceden en paralelo.

Donde los ítems 1 y 2, determinan la medida del rendimiento del sistema.

El concepto de arquitectura paralela se asocia con varios computadores homogéneos o no, que utilizan sistemas operativos como UNIX o WINDOWS NT; también se podría usar LINUX (gratuito) que es una forma de UNIX, debido a que son sistemas muy potentes para intercomunicaciones. Vale aclarar que existen distintas versiones de UNIX, entre las que podemos nombrar: System V, BSD, AIX, etc., en particular UNIX es un sistema portable, flexible, con entorno programable, multiusuario y multitarea, que además

provee las bibliotecas de funciones MPI y PVM en forma gratuita para procesamiento paralelo, las que pueden ser invocadas desde lenguajes de programación tales como 'C', 'Fortran' y otros.

En la conocida clasificación de Flynn (1966) se encuentran los computadores MIMD (múltiple flujo de instrucción, múltiple flujo de datos), esta clase de computadores tienen n procesadores, y cada uno de ellos puede ejecutar su propia secuencia de instrucciones; cada una de éstas actúa (se ejecuta) en un procesador con diferentes datos de los que se utilizan en los demás. La forma en que se conectan los procesadores a la memoria y también entre sí, permite diferenciar al menos dos subclases: los *multiprocesadores* y los *multicomputadores*.

En lo que atañe al "software", existe la estructura de programación SPMD (simple programa, múltiples datos), en ella un único programa fuente es escrito y cada procesador ejecutará su copia personal de este programa, aunque en forma independiente y no sincrónica. El programa fuente puede ser construido tal que las partes del programa sean ejecutadas por ciertos computadores y no otros, dependiendo de la identidad de cada uno. Para una estructura maestro-esclavo, el programa debería tener partes para el maestro y partes para los esclavos.

1.2. Tipos de computadores paralelos

1.2.1. Sistema de multiprocesador paralelo de memoria compartida

Una manera de extender el modelo de simple procesador, es tener múltiples procesadores conectados a múltiples módulos de memoria, a los cuales puede tener acceso cada procesador; y a esto se lo llama configuración de memoria compartida. La conexión entre los procesadores y las memorias se hace a través de una red de interconexión.

Este sistema emplea un espacio simple de dirección de memoria, es decir que cada posición en el total del sistema de memoria principal, tiene una dirección única y estas direcciones son usadas por cada procesador para tener acceso a dicha posición. Para programar un multiprocesador de memoria compartida, se requiere tener código ejecutable almacenado en la memoria para ser ejecutado por cada procesador. Los datos para cada programa sólo son almacenados en la memoria compartida y desde aquí cada programa podrá tener acceso a todos los datos que necesite. Algunos lenguajes de programación paralela están basados sobre lenguajes secuenciales existentes, como Fortran y C.

1.2.2. Pasaje de mensajes en multicomputadores

Cada computador consiste de un procesador y una memoria local (no accesible a otros computadores). La memoria es distribuida entre los computadores y cada uno de ellos tiene su propio espacio de direccionamiento. La red de interconexión es provista para el envío de mensajes entre computadores. Estos mensajes pueden incluir datos que otros computadores puedan requerir para sus operaciones. Para el programador implica dividir el problema en partes, que serán ejecutadas simultáneamente para resolverlo. Comúnmente se utilizan librerías de rutinas de pasaje de mensajes que son llamadas por un programa secuencial convencional.

Si hay igual cantidad de procesos que computadores, se ejecuta un proceso por computador; y si hay más procesos que computadores, más de un proceso podrá ser ejecutado en estos últimos. Los procesos se comunicarán por pasaje de mensajes; esta será la única manera de distribuir datos y resultados entre procesos. En este punto es importante tener en cuenta el progreso tecnológico de los procesadores simples (cada vez más veloces), que pueden hacer obsoletos a los multiprocesadores; vale decir, es obviamente mejor usar un nuevo procesador simple que un viejo multiprocesador; siempre y cuando, el nuevo opere “n” veces mas rápido que cada uno de los “n” viejos procesadores del multiprocesador. Lógicamente esto es directamente proporcional a los costos.

1.2.3. Arquitecturas del pasaje de mensajes en multicomputadores

- **Redes de interconexión estática**

Son aquellas que tienen conexión física directa y fija entre computadores (nodos). Cada nodo contiene un procesador, memoria y una interfaz de comunicación con conexión a otros nodos. La conexión entre los nodos debe ser bidireccionable. Además en este tipo de red es aplicable el término: embbeding; que describe el mapeo de nodos de una red en otra red. Con respecto al pasaje de mensajes, en la mayoría de los sistemas es necesario rutear el mensaje a través de nodos intermediarios desde un nodo fuente a un nodo destino y existen diferentes métodos de comunicación:

- **Circuit switching**: establece el path y mantiene toda la conexión en el path ininterrumpida para el mensaje a pasar, desde la fuente al destino. Todos los links son reservados para la transferencia de mensajes, hasta que ésta sea completada, (es similar al sistema telefónico).
- **Packet switching**: el mensaje es dividido en paquetes de información, cada uno incluye la dirección fuente y destino para rutear al paquete a través de la red de interconexión, (lo cual lo hace similar al sistema de correo).

Una desventaja significativa es la “latencia”, que produce la espera de paquetes entre los nodos (ya que se almacenan en buffers) y que es aproximadamente proporcional al largo de la ruta; y a su vez en el circuit switching, se produce algo similar: “delay” que depende del número de links usados. Otro factor que aparece en la interconexion de redes es el “deadlock”, se produce cuando los paquetes no pueden avanzar hacia el próximo nodo porque están bloqueados por otros paquetes esperando avanzar ya que los buffers de nodos no están libres para aceptar paquetes.

Por último todo sistema de computación (incluyendo multicomputadores y multiprocesadores de memoria compartida) debe tener mecanismos y dispositivos de entrada-salida.

1.2.4. Computadores en red, como una plataforma de multicomputadores

Actualmente la mayoría de los computadores están o pueden ser conectados con otros computadores; formando una red local (LAN), por lo cual hoy en día están globalmente interconectados a través de Internet. La necesidad de tener estaciones de trabajo o computadores en red, brinda la posibilidad de usar algunas redes para programación en paralelo.

Las ventajas de esto son:

- Estaciones de trabajo de alta performance y PCs, disponibles a bajo costo.
- Los últimos computadores pueden fácilmente incorporarse al sistema de red.
- La existencia de software que puede ser usado o modificado.

1.2.5. Factores útiles para analizar la velocidad computacional

- **Procesos:** No importa la forma de (MIMD) multiprocesador o multicomputador, para lograr un aumento de velocidad a través del uso de paralelismo; esto es necesario para dividir la computación dentro de procesos que pueden ser ejecutados simultáneamente. El tamaño de un proceso puede ser descrito como: “granularidad”; donde cada proceso contiene un cierto número de instrucciones secuenciales y lleva un tiempo de ejecución. A veces la granularidad está definida como la medida del cómputo entre pasos de comunicación. Ella está relacionada con el número de procesadores que están siendo usados.

- **Factor “speedup”:** Es una medida de performance relativa entre un sistema paralelo y un sistema de procesador simple, y se lo define como:

$$S(n) = \frac{tiempo\ \ de\ \ ejecución\ \ usando\ un\ \ procesador\ \ simple}{tiempo\ \ de\ \ ejecución\ \ usando\ n\ \ procesadores}$$

- **Overhead:** Habrá factores severos que aparecerán como overhead en la versión paralela y limitan el speedup notablemente:

  - Periodos en los cuales no todos los procesadores pueden mejorar su capacidad de trabajo y son simplemente ociosos.
  - Computaciones extras en la versión paralela, que no aparecen en la versión secuencial.
  - Tiempo de comunicación para envío de mensajes.

- **Eficiencia:** Se la define como:

$$E = \frac{tiempo\ \ de\ \ ejecución\ \ usando\ un\ \ procesador}{tiempo\ \ de\ \ ejecución\ \ usando\ n\ \ procesadores\ \times\ n}$$

Para expresarlo en porcentaje, multiplicamos por 100:

$$E = \frac{S(n)}{n} \times 100\%$$

Donde la máxima eficiencia tiende al 100%, y ocurre cuando todos los procesadores están siendo usados a la vez; el factor "speedup" en este caso tiende a n.

- **Costo:** El costo (o trabajo) se define:

$$Costo = tiempo\ \ de\ \ ejecución\ \ \times numero\ \ total\ \ de\ \ procesadores\ \ usados$$

Un algoritmo paralelo tiene costo óptimo cuando, sobre un multiprocesador resulta proporcional al costo sobre un sistema de procesador simple.

- **Escalabilidad:** Este término es usado para indicar que un diseño de hardware permita al sistema ser incrementado en tamaño y en performance. Por supuesto, nosotros esperamos que todos los sistemas de multiprocesadores sean arquitecturalmente escalables (al igual que los fabricantes), pero depende fuertemente del diseño del sistema.

1.3. Lenguajes paralelos

Existen tres diferentes maneras para poder lograr la programación en un sistema multicomputador con comunicación y sincronismo por medio de pasaje de mensajes; ellas son:

- Diseñar un lenguaje especial para programación paralela. El único caso conocido es el lenguaje **OCCAM**, que fue introducido precisamente para el *transputer*, un procesador basado en pasaje de mensajes (Inmos) con características de mucho interés.
- Crear extensiones a lenguajes existentes de alto nivel, a fin de que con ellos se pueda hacer pasaje de mensajes. Dos de estos casos son **FORTRAN M** y **CC++** (una pequeña extensión a C++).
- Utilizar un lenguaje existente de alto nivel y constituir una librería de funciones externas para pasaje de mensajes.

A continuación hacemos una breve referencia de los lenguajes de programación y librerías usadas para pasaje de mensajes, exceptuando el lenguaje C, del cual se hace referencia en el apéndice B.

1.3.1. Lenguaje OCCAM

El nombre de este lenguaje proviene del filosofo y religioso Guillermo de Occam (1985), que acuñó profundas ideas filosóficas en frases simples:

"Hacer las cosas simples"

"No complicar las cosas más de lo necesario"

El lenguaje **OCCAM** proviene del lenguaje **CSP: Communicating Sequential Processes**, formulado por Hoare en 1978.

Es un lenguaje de programación de bajo nivel para computadores de procesamiento paralelo, y, específicamente, para el INMOS Transputer, que ejecuta el lenguaje Occam de manera casi directa. Este lenguaje está diseñado para manejar operaciones simultáneas.

OCCAM permite crear procesos autónomos que se comunican usando canales con nombre. La unidad del lenguaje es el proceso, un proceso es el equivalente a una sentencia en un programa secuencial. Los procesos pueden combinarse en procesos más grandes usando constructores (SEQ para definir un proceso secuencial y PAR para definir uno paralelo), también se pueden usar combinaciones de ellos; como en otros lenguajes existen **if, while** y **for** además de otros detalles que no corresponden al fin de este trabajo.

Un ejemplo de programación en OCCAM, para ver su aspecto es el siguiente:

```
#INCLUDE "hostio.inc"
PROC en.paralelo (CHAN OF SP fs, ts,  [] INT mem)
    #USE "hostio.lib"
    CHAN OF INT transporta.enteros :
    PROC consumidor ()
        INT n :
        SEQ
            transporta.enteros  ?  n
            WHILE n < 10
                SEQ
                    so.write.int ( fs,  ts,  n,  2 )
                    so.write.nl ( fs,  ts )
                    transporta.enteros  ?  n
    :
    PROC productor ()
        INT i :
        SEQ
            i := 0
            WHILE i < 10
            SEQ
            transporta.enteros  !  I
            i := i + 1
    :
    SEQ     --    programa o proceso principal
        PAR
            consumidor ( )
            productor ( )
        so.exit ( fs,  ts, sps.success )
:
```

1.3.2. Lenguaje FORTRAN

Uno de los lenguajes de programación más conocidos y utilizados hoy en día es el **FOR**mula **TRAN**slation.

Este fue desarrollado en 1957 y fue uno de los primeros lenguajes de alto nivel para computadores. Desde entonces, se han desarrollado unos 450 lenguajes de programación, pero aún así, el FORTRAN sigue siendo un lenguaje poderoso.
Ahora veamos un ejemplo de un programa en FORTRAN
Encuentre el área de un piso de un cuarto, si este tiene una longitud de 8 metros por 3 metros de ancho.
Procedimiento para la planeación del programa:

```
1- Comienzo (inicialización de las variables)
2- Asignar los datos a las variables A (largo)y B (ancho)
3- Calcule el área como C = A x B
4- Imprima el área
5- Fin
```

Programa en FORTRAN:

```
PROGRAM AREA
A = 8.0
B = 3.0
C = A * B
PRINT *, C
END
```

1.3.3. Librerías de funciones para pasaje de mensajes

Las librerías de funciones son utilizadas por lenguajes de programación de alto nivel como el **C** o **Fortran**, para permitir el pasaje de mensajes entre procesos y brindar a los programadores de aplicaciones muchas posibilidades para elaborar programas más simples y eficientes. Estas son las más utilizadas en el caso de nuestro interés, es decir en las redes de estaciones de trabajo y PCs.

Capítulo 2

2. REDES

2.1. Introducción

El propósito fundamental de la comunicación de datos es **intercambiar** información entre dos agentes. La información nace cuando los datos son interpretados. Los **objetivos** de las redes son: compartir recursos, alta fiabilidad, distribución de tareas, ahorro económico e intercambio de información.

Los elementos que componen una red son: el dispositivo de interconexión y los computadores conectados. La **comunicación** se realiza punto a punto entre dos nodos o por medio de ruteo ó direccionamiento entre dos o más redes.

Las redes se clasifican en **LAN** (Local Area Network) que no poseen estaciones intermedias, son “broadcast”, el medio es compartido y está administrada por una organización privada; y **WAN** (Wide Area Network) que poseen estaciones intermedias (routers y switches), son “unicast”, el medio de comunicación es punto a punto y está administrada por una empresa de comunicaciones.

2.2. Aplicación de las redes

Para dar una idea sobre algunos de los usos importantes de redes de computadores, distinguimos ahora brevemente tres casos: el acceso a programas remotos, el acceso a bases de datos remotas y facilidades de comunicación.

Muchas aplicaciones operan sobre redes por razones económicas: el llamar a un computador remoto mediante una red resulta mas económico que hacerlo directamente. La posibilidad de tener un precio más bajo se debe a que el enlace de una llamada telefónica normal utiliza un circuito caro y en exclusiva durante todo el tiempo que dura la llamada, en tanto que el acceso a través de una red, hace que sólo se ocupen los enlaces de larga distancia cuando se están transmitiendo los datos.

Otra forma que muestra el amplio potencial del uso de redes, es su empleo como medio de comunicación. Como por ejemplo, el tan conocido por todos, correo electrónico (e-mail), que se envía desde una terminal , a cualquier persona situada en cualquier parte del mundo que tenga este servicio.

2.3. Topologías de Redes

Para poder visualizar el sistema de comunicación en una red es conveniente utilizar el concepto de topología, o estructura física de la red. Las topologías describen la red físicamente y también nos dan información acerca del método de acceso que se usa (Ethernet, Token Ring, etc.).

Algunas posibles topologías son: Estrella, Anillo, Bus, y Árbol, las que se detallan a continuación:

Topología de estrella

En este esquema, todas las estaciones están conectadas por un cable a un módulo central (**hub**), y como es una conexión de punto a punto, necesita un cable desde cada computador al módulo central. Una ventaja de usar una red de estrella es que ningún punto de falla inhabilita a ninguna parte de la red, sólo a la porción en donde ocurre la falla, y la red se puede manejar de manera eficiente.

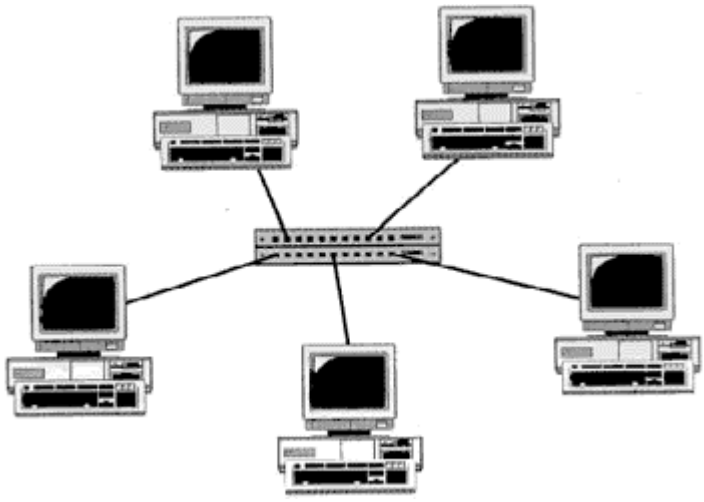


Figura 3.1 - Topología estrella

Topología de anillo

En esta configuración, todas las estaciones repiten la misma señal que fue enviada por la terminal transmisora, y lo hacen en un solo sentido en la red. El mensaje se transmite de terminal a terminal y se repite, bit por bit, por el repetidor que se encuentra conectado al controlador de red en cada terminal. Una desventaja con esta topología es que si algún repetidor falla, podría hacer que toda la red se caiga, aunque el controlador puede sacar el repetidor defectuoso de la red, evitando así algún desastre. Un buen ejemplo de este tipo de topología es el de Anillo de señal, que pasa una señal, o token a las terminales en la red. Si la terminal quiere transmitir alguna información, pide el token, o la señal y hasta que la tiene, puede transmitir. Claro, si la terminal no está utilizando el token, la pasa a la siguiente terminal que sigue en el anillo, y sigue circulando hasta que alguna terminal pide permiso para transmitir.

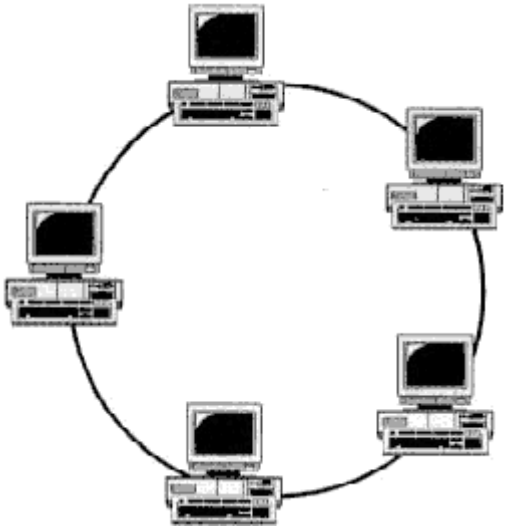


Figura 3.2 - Topología anillo

Topología de bus

También conocida como topología lineal de bus, es un diseño simple que utiliza un solo cable al cual todas las estaciones se conectan. La topología usa un medio de transmisión amplio (***broadcast***), ya que todas las estaciones pueden recibir las transmisiones emitidas por cualquier estación. Como es bastante simple la configuración, se puede implementar de manera barata. El problema inherente de este esquema es que si el cable se daña en cualquier punto, ninguna estación podrá transmitir.

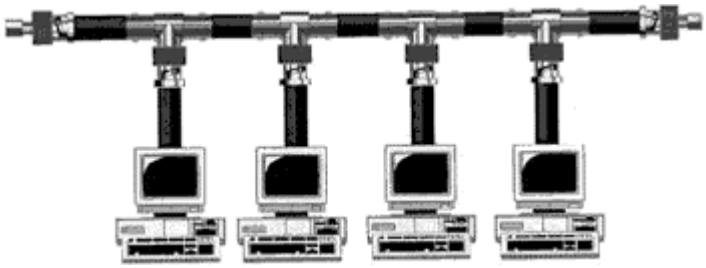


Figura 3.3 - Topología bus

Topología de árbol

Esta topología es un ejemplo generalizado del esquema de bus. El árbol tiene su primer nodo en la raíz, y se expande para afuera utilizando ramas, en donde se encuentran conectadas las demás terminales. Ésta topología permite que la red se expanda, y al mismo tiempo asegura que solamente exista una "ruta de datos" (data path) entre 2 terminales cualesquiera.

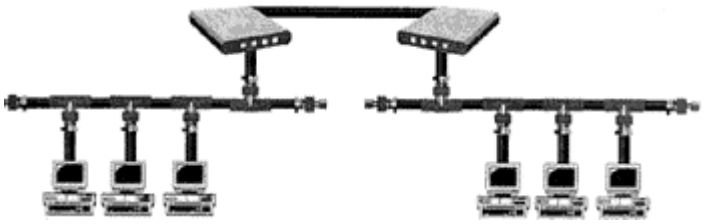


Figura 3.1 - Topología árbol

2.4. Transmisión de Datos

Depende básicamente de la calidad de la señal que se transmite y las características del medio de transmisión. La transmisión de datos ocurre entre transmisor y receptor sobre un medio de transmisión. Una señal digital tiene infinito ancho de banda, aunque la naturaleza del medio limita esto, provocando deterioro en la transmisión, que se puede deber a atenuación, distorsión o ruido.

2.5. Tipos de Transmisión

Puede ser en serie ó en paralelo. La transmisión en serie utiliza una sola línea, transmite un bit detrás de otro y es muy confiable para largas distancias.

La transmisión en paralelo realiza una transmisión simultanea de bits sobre un grupo de líneas y es utilizado en cortas distancias.

Capítulo 3

3. MPI Y ESTUDIO DE LAS IMPLEMENTACIONES LAM Y MPICH

3.1. Introducción a MPI

MPI (Interfaz para Pasaje de Mensajes) es una norma que ha sido creada por un amplio comité de expertos y usuarios con el objeto de definir la sintaxis y la semántica de un núcleo de más de un centenar de rutinas de librerías aplicables en lenguaje **C** o **Fortran**.

De esta forma los productores de software de procesamiento paralelo pueden implementar su propia versión de MPI siguiendo las especificaciones de este estándar.

Una implementación MPI no es un nuevo lenguaje de programación, sino una librería de funciones a la que podemos llamar desde C o Fortran. Está basada en el pasaje de mensajes, y sus ventajas más inmediatas son la implementación y portabilidad para una gran variedad de sistemas: desde computadores con memoria compartida hasta una red de estaciones de trabajo y PCs.

El pasaje de mensajes es considerado por algunos como una herramienta muy potente para la programación paralela, por el acercamiento a la estructura paralela de nuestro programa, manteniéndonos a salvo de los problemas de implementación en cada tipo de computador.

Por lo antes expuesto y dadas las posibilidades de que disponemos, utilizaremos una red de estaciones de trabajo y PCs heterogénea con sistemas operativos UNIX y LINUX respectivamente. Con el fin de procesar en paralelo, utilizaremos el lenguaje de programación "C" y la biblioteca de funciones MPI correspondiente. Específicamente haremos uso de dos implementaciones gratuitas de MPI que son utilizables sobre múltiples arquitecturas: se trata de LAM y MPICH, que son ambientes de desarrollo y programación de MPI orientado a arquitecturas heterogéneas de computadores distribuidos en una red local.

A continuación indicamos lo referente a los sistemas operativos y el lenguaje de programación mencionado: además se dan algunas aplicaciones de procesos paralelos, lógicamente haciendo uso de las librerías MPI, para evaluar la funcionalidad del sistema con el cual vamos a trabajar.

3.2. Características generales de MPI

- Para la confiabilidad requerida, hay que añadir dentro de los archivos: '**$HOME/.rhosts**', y '**/etc/hosts.equiv**' los computadores y los usuarios seguros y confiables que se necesiten:

computador1 ***usuario***
computador2 ***usuario***
computador3 ***usuario ...*** *, etc*

El archivo ***/etc/host.equiv*** es de uso general, y el que se encuentra en ***$HOME/.rhosts****,* es de uso particular. Ambos deben contener todos los computadores seguros y confiables que componen la red; y por lo tanto pueden ser incorporados al computador paralelo.
HOME es el directorio de usuario. Para más información al respecto ver *Archivos de configuración y confiabilidad del sistema* tratados en el apéndice A.

- Soporta paralelismo del tipo SPMD (Single Program Multiple Data) y MPMD (Multiple Program Multiple Data).

- La transferencia de mensajes se realiza de forma cooperativa, tanto el proceso emisor como el receptor participan en el traspaso.

- Permite establecer comunicación punto-a-punto (participan exactamente dos procesos) o comunicación colectiva (participan un número elevado de procesos simultáneamente).

- Proporciona cuatro modos de comunicación: ***standard****,* ***synchronous****,* ***buffered****,****ready****.*

  - ***standard***: la emisión de un mensaje se completa una vez que ha sido enviado, haya o no llegado al receptor.

  - ***synchronous***: el proceso de transferencia no se completa hasta que el emisor reciba la confirmación de que el receptor ha recibido el mensaje.

  - ***buffered***: el mensaje es copiado en un buffer del sistema para transmitirlo más tarde si fuese necesario. En este modo, la transferencia se completa inmediatamente.

  - ***ready***: el emisor deja el mensaje dentro de la red de comunicación y supone que el receptor lo estará esperando. Este modo también se completa inmediatamente.

3.3. Comandos de MPI

Para compilar un programa en MPI ejecutamos.

mpicc -o *nombre_del_ejecutable nombre_de_fuente*

Para ejecutar un programa en MPI, lo hacemos con:

mpirun -np *4 nombre_del_ejecutable*

donde **np** indica el número de computadores.

Otra forma de ejecutar un programa en MPI es:

mpirun -p4pg *nombre_archivo_pgfile nombre_del_ejecutable*

nombre_archivo_pgfile **:** Archivo que se utiliza para la ejecución de programas en MPI. En este se indica cada computador y la ubicación del programa a ejecutar en cada uno de ellos, además de indicar cual es el local (0) y cuales los remotos (1). Vale aclarar que este archivo puede tener cualquier nombre, aunque conviene alguno similar al mencionado para identificarlo fácilmente.

Un ejemplo de este, puede ser:

cetadfomec1	*0*	*/home/nombre_usuario/nombre_prog_ejecutable*
cetadfomec2	*1*	*/home/nombre_usuario/nombre_prog_ejecutable*
cetad	*1*	*/home2/cetad/nombre_usuario/nombre_prog_ejecutable*
paris	*1*	*/export/home/nombre_usuario/nombre_prog_ejecutable*
sofia	*1*	*/home/nombre_usuario/nombre_prog_ejecutable*

3.4. Rutinas de MPI con su estructura

MPI_Comm_world: grupo de comunicación global o general.
MPI_Char: tipo de dato predefinido en MPI, que identifica caracter con signo.
MPI_Max_Processor_Name: cantidad máxima de procesadores en MPI en el vector processor_name.
MPI_Status_Size: es una estructura, que comprende 3 campos llamados:
MPI_Source, MPI_Tag, y MPI_Error; donde cada uno contiene el código respectivo del mensaje recibido.
MPI_Init (&argc,&argv): comienza las llamadas a MPI.
MPI_Finalize(): termina con MPI. Recordar que después de esta llamada no podemos llamar a funciones MPI, ni siquiera de nuevo a MPI_Init().
MPI_Comm_size (comm, size): devuelve el número de procesos que están ejecutando nuestro programa.
MPI_Comm_rank (comm, rank): devuelve el rango del proceso en el programa que se ejecuta.
MPI_Send (buf, count, datatype, dest, tag, comm): manda un mensaje al proceso destino.
MPI_Recv (buf, count, datatype, source, tag, comm., status): recibe un mensaje de un proceso origen.
MPI_Get_processor_name (name, resultlen): da el nombre del procesador donde fue ejecutado en el momento de la llamada.
MPI_Bcast (buffer, count, datatype, root, comm): para comunicaciones colectivas, es decir para mandar un mensaje desde un proceso (proceso maestro) al resto de los procesos.
MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm): para comunicaciones colectivas, es decir para mandar mensajes desde todos los procesos a uno (proceso maestro).
MPI_Wtime (void): retorna un número de segundos en punto flotante, representando el lapso de un intervalo de tiempo, y garantiza que no cambia durante la vida del proceso.

3.5. Implementación LAM (Local Area Multicomputer)

LAM (Local Área Multicomputer): Es un ambiente de desarrollo y programación de MPI orientado a arquitecturas heterogéneas de computadores distribuidos en una red local. Con LAM, tanto un "Cluster" dedicado como una simple red local de computadores puede actuar como un computador multiprocesador.

3.5.1. Características principales de LAM

- Es una implementación completa del estándar MPI-1, y contiene muchas de las especificaciones de: MPI-2 client/server, dynamic process spawning, one-sided communication, C++ bindings, MPI I-O, etc.
- Soporta redes con estaciones de trabajo heterogéneas.
- Contiene herramientas para el debugging de procesos y mensajes.
- Tiene nodos LAM dinámicos.
- Tiene tolerancia a fallo.
- Produce comunicación rápida punto a punto.
- Está disponible gratuitamente.
- LAM es portable en la mayoría de los computadores UNIX: SUN (SUN OS y Solaris), SGI IRIS, IBM AIX, DEC OSF/1, HP-UX, y Linux.

3.5.2. Instalación

- **Desempaquetando la distribución.**

La distribución LAM se obtiene empaquetada en un archivo comprimido del tipo: ***lam-6.5.6.tar.Z, lam-6.5.6.tar.gz,*** o ***lam-6.5.6.tar.bz2***.
Estos están disponibles en la pagina web de LAM: http://www.lam-mpi.org/

Los archivos anteriores se descomprimen para extraer los códigos fuentes con:

$> gunzip -c lam-6.5.6.tar.gz | tar xf –

o

$> uncompress -c lam-6.5.6.tar.Z | tar xf –

o

$> bunzip2 -c lam-6.5.6.tar.bz2 | tar xf –

Lo anterior es según el archivo que se obtenga, en nuestro caso utilizamos la primera opción.

- **Compilación de LAM.**

El procedimiento que se siguió para la instalación de LAM fue el siguiente:

$> gunzip –c lam-6.5.6.tar.gz | tar xf –
$> cd lam-6.5.6

```
$> ./configure  - -prefix=/directorio/a/instalarlo  - -with-trillium
[ . . . salidas . . . ]
$> make
[ . . . salidas . . . ]
$> make install
[ . . . salidas . . . ]
$> make examples      # Este paso es opcional
[ . . . salidas . . . ]
```

Observación: el parámetro **- -*with-trillium*** es para incluir algunas funciones utilizadas luego en la Consola MPI (*doom y state*).

Algunas causas de fallas que tuvimos :

- Que el compilador de C++ no estaba instalado, entonces usamos la opción de configuración **--without-mpi2cpp** (para que no se instale)
- Que el compilador de Fortran no estaba instalado, por lo tanto usamos la opción de configuración **--without-fc** (para no instalarlo)
- **"mpicc"** no podía ser encontrado, luego tuvimos que definir el *path* en el ***.bashrc***
- Al ejecutar el comando ***lamboot*** no se formaba el "computador paralelo", por lo tanto tuvimos que crear el archivo de configuración ***.bashrc*** (en plataforma Linux) incluyendo los *path* y *variables de entorno* correspondientes.

Otras soluciones las encontramos con la ayuda proporcionada por la distribución LAM.

- **Requerimientos necesarios para trabajar con LAM.**

  - Computadores con plataforma Unix o Linux conectadas en Red.
  - Ejecutables de **LAM** (***recon, lamboot, mpirun, lamclean, wipe***).
  - Librerías **MPI**, compiladas e instaladas en cada arquitectura.
  - Archivos particulares de una aplicación específica debidamente compilados en cada computador.
  - Programas con comportamiento master/slave.
  - **MPI hostfile** (define qué computadores físicos componen el sistema paralelo). Usualmente lo llamamos ***hostfile***
  - Archivo ***$HOME/.rhosts***, (debe contener todos los computadores con los usuarios respectivos que serán incorporados al sistema paralelo).

- **Instalación y configuración bajo LINUX.**

  - El programa de instalación de paquetes (que se ejecuta en modo superusuario), coloca los ejecutables de **LAM** en ***/usr/local/lam-6.5.6/bin***. Comprobamos que el ejecutable estaba allí, ejecutando el comando:

    $> ***which mpirun***

- El programa de instalación coloca las librerías de **MPI** (***libmpi.a***) en el directorio ***/usr/local/lam-6.5.6/lib***. Comprobamos que las librerías estaban en el directorio indicado.
- Antes de ejecutar los programas que componen **MPI**, comprobamos que el **PATH** contenía el directorio ***/usr/local/lam-6.5.6/bin***. Esto lo pudimos hacer ejecutando el siguiente comando:

  $> ***echo $PATH***

3.5.3. Ejecución

- **Inicio de LAM**

Creamos un archivo (con el programa *vi*) que contenía una lista con los computadores que se iban a utilizar como parte del sistema paralelo. A este archivo lo llamamos ***hostfile***, y se puede observar mediante el comando *cat* como se muestra a continuación:

```
$> cat hostfile
#Usaremos un LAM de 3 nodos, a modo de ejemplo
sofia
cetadfomec1
cetadfomec2
```

En este listado debe estar contenido el computador desde donde se ejecutará el programa maestro (sofia en nuestro caso). Cada uno crea como usuario su propio archivo *hostfile* con los computadores que incluiría para el procesamiento paralelo. LAM automáticamente le asignará a cada nodo (computador) un identificador (nodeid) comenzando por '0' para el primer computador de la lista, '1' para el segundo y así sucesivamente.

- Utilizamos el programa ***recon*** para verificar que el "cluster" o grupo de computadores, podrá ser exitosamente inicializado con LAM.

  $> ***recon –v hostfile***

  El programa produjo la siguiente salida:

  ```
  recon: testing n0 (sofia)
  recon: testing n1 (cetadfomec1)
  recon: testing n2 (cetadfomec2)
  ```

  Y en consecuencia como no produjo un mensaje de error, entonces LAM pudo ser inicializado sin dificultades como sigue.

- Utilizamos ***lamboot*** para iniciar los "demonios" de LAM en los nodos especificados en el *hostfile*:

  $> ***lamboot –v hostfile***

  El programa produjo la siguiente salida:

LAM 6.5.6/MPI 2 C++/ROMIO –University of Notre Dame
hboot n0 (sofia)...
hboot n1 (cetadfomec1)...
hboot n2 (cetadfomec2)...

- Finalmente utilizamos el comando ***tping*** para asegurarnos que LAM estaba ejecutándose correctamente:

  $> ***tping –c1 N*** (*N* indica que utilice todos los nodos disponibles)

  El programa produjo la siguiente salida:

  1 byte from 3 nodes: 0.009 secs

- **Compilación**

Para ejecutar aplicaciones paralelas en el computador local o remoto, el ejecutable de dicha aplicación puede estar almacenado en cualquier directorio del computador desde donde se ejecutan los programas maestros. Una manera sencilla de trabajar con MPI es crear estos mismos directorios en los computadores remotos, de manera que no sea necesario decirle a MPI el directorio donde debe buscar las aplicaciones esclavas en los computadores remotos, pues asume que es el mismo que utiliza el computador maestro. Adicionalmente, MPI permite también transmitir por red el programa ejecutable a los computadores remotos, con la ventaja de que no es estrictamente necesario tener el programa compilado en todos los computadores, aunque se tiene un costo de tiempo de transmisión del mismo por la red.

- Creamos un directorio llamado MPI en el directorio HOME de cada computador que vamos a utilizar, mediante:

  $> ***mkdir $HOME/MPI***

  En la práctica utilizamos un programa llamado *pi_mpi.c*, así como el correspondiente archivo ***Makefile*** que se encuentra en el directorio:

  /usr/local/lam-6.5.6/PI/

- Copiamos el Makefile y el programa pi_mpi.c en el directorio de trabajo MPI que creamos en el paso anterior con el comando *cp*:

  $> ***cp /usr/local/lam-6.5.6/PI/* $HOME/MPI***

- Para compilarlo, solamente ejecutamos el comando ***make***:

  $> ***make***

El archivo *Makefile* ya contiene toda la información necesaria para compilar el programa. Este proceso tuvo que repetirse en todos los computadores que componen nuestra red. Alternativamente, podemos hacer un ftp a estos

computadores y copiar el programa compilado en el directorio de MPI (si trabajamos con computadores de arquitecturas iguales).

- **Ejecución de aplicaciones paralelas**

Una vez compilado el programa paralelo que íbamos a utilizar, pudimos ejecutarlo en todos los nodos utilizando el comando ***mpirun .*** LAM, a diferencia de PVM (que consideraremos más tarde), necesita que se le especifique el número de procesos a ejecutarse cuando se invoca el programa. Ejecutamos el siguiente comando:

$> ***mpirun –v n0-2 pi_mpi***

En este caso le estamos diciendo a LAM que ejecute el programa *pi_mpi* en los nodos 0..2 (parámetro n0-2) que corresponden a los tres computadores listados en el archivo *hostfile*. El parámetro *–v* solo le indica a *mpirun* que brinde información en la consola. Alternativamente si queremos lanzar un número determinado de procesos, podemos hacerlo de la siguiente manera:

$> ***mpirun –v –c 10 pi_mpi***

En este caso le estamos diciendo a LAM que ejecute 10 copias del programa *pi_mpi* en todos los computadores listados en *hostfile* empezando por el primero en la lista y siguiendo un orden circular. En este ejemplo, si utilizamos 3 computadores, levantará 4 procesos en el primero y 3 procesos en los 2 restantes.

Vale aclarar que se realizaron varias pruebas considerando las alternativas anteriores, observando los resultados arrojados por éstas.

- **Monitoreo de las aplicaciones**

Para monitorear el estado de los procesos y mensajes en cada nodo, se pudo utilizar en cualquier momento el comando ***mpitask***:

$> ***mpitask***

El mismo brinda información de los procesos origen y destino, los identificadores de mensaje, tipos de datos transmitidos en los mensajes y las funciones invocadas.

- **Eliminación de procesos**

Todos los procesos y mensajes de LAM pueden ser eliminados sin abortar LAM a través del comando ***lamclean***. Es recomendable que este comando siempre se utilice después de terminar la sesión de trabajo con LAM:

$> ***lamclean –v***

La ejecución de éste comando produjo la siguiente salida:

killing processes, done
sweeping processes, done
closing files, done
sweeping traces, done

- **Terminación de la ejecución**

El programa ***wipe*** elimina todos los procesos y demonios de LAM en los computadores que estamos utilizando. Esto solo debe ejecutarse cuando se termine de trabajar con LAM.

$> ***wipe –v hostfile***

El programa produjo la siguiente salida:

tkill n0 (sofia)...
tkill n1 (cetadfomec1)...
tkill n2 (cetadfomec2)...

Cabe mencionar que el comando ***wipe*** ha sido desaprobado frente al comando ***lamhalt ,*** solamente en ciertas situaciones cuando este último tiene fallas, el primero es útil, este es el motivo por el cual se utilizó en la práctica el primero, para prevenir fallas. De todas maneras ***lamhalt*** se utiliza de igual forma que el anterior, como se muestra a continuación:

$> ***lamhalt –v hostfile***

3.6. Implementación MPICH

MPICH: (**MPI** and **Ch**ameleon) es una implementación de la librería de pasaje de mensajes MPI escrita por el Argonne National Laboratory y la Mississippi State University.

3.6.1. Características principales de MPICH

- La principal característica de la librería MPICH es su adaptabilidad a gran número de plataformas, incluyendo clusters de workstations y procesadores masivamente paralelos (MPP)
- Permite trabajar con lenguaje C, C++ y FORTRAN.
- Para utilizar esta librería hay que incluir en el PATH las variables de entorno **$MPICH** y **$MPICH_MPL_BIN**.

3.6.2. Instalación y prueba

A continuación enumeramos un conjunto básico de pasos para preparar y probar MPICH. Se dan luego detalles e instrucciones para una vista más completa de las características de MPICH, incluyendo instalación, validación, cotas de referencias, y uso de las herramientas de evaluación de performance.

- MPICH pudo obtenerse por ftp anónimo desde el sitio ftp.mcs.anl.gov yendo al directorio pub/mpi y obteniendo el archivo **mpich.tar.gz**. este nombre de archivo es un enlace a la versión mas reciente de MPICH. Actualmente es de aproximadamente 4 Megabytes. El archivo es comprimido por **gzip** y agrupado por **tar**, así que puede ser desempaquetado con:

  *$> **gunzip -c mpich.tar.gz | tar xovf -*** (como se vio anteriormente)

  Si no hubiéramos tenido *gunzip*, pero si otro descompresor, entonces deberíamos haber obtenido **mpich.tar.Z**, y usar cualquiera, por ejemplo:

  *$> **zcat mpich.tar.Z | tar xovf -***
  o
  *$> **uncompress mpich.tar.Z***
  *$> **tar xvf mpich.tar***

  Esto hubiera creado un solo directorio llamado MPICH que contenía varios subdirectorios de la distribución entera, incluyendo todo el código fuente, alguna documentación (incluyendo la Guía de donde se tomó parte de esta explicación), páginas del *man* (manual), etc.

- ***cd mpich***

  En particular, se deben ver los siguientes archivos y directorios:

  Makefile.in: Plantilla para el *`Makefile'*, el cual será producido cuando ejecute *configure*.
  README: Información básica e instrucciones para configurar.
  aclocal.m4: Usado para compilar *`configure'* desde *`configure.in'*; no es necesario para más instalaciones. El archivo *aclocal tcl.m4* está incluido en *aclocal.m4*.
  ccbugs: Directorio para programas que testean el compilador C durante la configuración, para estar seguro que será capaz de compilar el sistema.
  configure: Es el script que usted ejecuta para crear Makefiles a través del sistema.
  configure.in: Entrada de *autoconf* que produce *configure*.
  doc Documentación y varias herramientas, como la Installation Guide y la User's Guide.
  examples: Directorio que contiene directorios de ejemplos de programas MPI.
  include: Las librerías *include*, tanto de usuario como del sistema.
  bin: Contiene programas script ejecutables, tales como *mpicc* y *mpirun*, usados para compilar y ejecutar programas MPI.
  lib: Contiene las librerías para MPI, MPE, y herramientas relacionadas.
  man: Páginas del manual para MPI, MPE, y rutinas internas.

mpe: El código fuente para las extensiones MPE para logging y gráficos X. El directorio *contrib* contiene ejemplos. Los mejores son los subdirectorios *mandel* y *mastermind*
mpid: El código fuente para varios *"devices"* que personalizan MPICH para un computador particular, sistema operativo, y entorno.
romio: El sistema *ROMIO parallel I/O*, el cual incluye una implementación del *MPI-2 parallel I/O standard*.
jumpshot: La fuente para el programa de visualización de performance *Jumpshot*
src: El código fuente para la parte portable de MPICH. Hay subdirectorios para varias partes de las especificaciones de MPI.
util: Programas de utilidad y archivos.
www: Páginas del manual en versión HTML.

- ***./configure***

  Con este comando el sistema reconoce la arquitectura y los dispositivos que se tratan de utilizar. Si los valores por defecto no son los que se desean, se seleccionan las opciones de *configure* . Fue mejor escoger un directorio para instalar MPICH y configurarlo con ese directorio y el dispositivo *ch_p4*. Por ejemplo:

  ./configure --prefix=/usr/local/mpich-1.2.3/ch_p4 - -with-device=ch_p4

  De igual manera, se puede configurar con otro directorio y otro dispositivo (como ser el directorio */usr/local/mpich-1.2.3/ch_p4mpd* y el dispositivo *ch_p4mpd*)

- **make >& make.log** (en sintaxis del C-shell).

  Produce la compilación y demora varios minutos, dependiendo de la carga del sistema y del servidor de archivos.

- (**Opcional**) Compilamos y ejecutamos un programa simple de prueba:

  $> cd examples/basic
  $> make cpi
  $> ln -s ../../bin/mpirun mpirun
  $> ./mpirun -np 4 cpi

  A esta altura hemos ejecutado un programa de MPI en nuestro sistema.

- El programa de visualización **nupshot** es una versión más rápida del **upshot**, pero requiere la versión 3.6 del código fuente de **tk**.

  $> **make nupshot**

  Este paso no fue necesario para el propósito de este trabajo.

- Para instalar MPICH en un lugar público para que otros puedan usarlo, usamos :

  *$> **make install***
  o
  *$> **bin/mpiinstall***

  Para instalar MPICH en un directorio especificado se puede utilizar la opción ***--prefix*** de ***configure***. La instalación consistió de los directorios *include, lib, bin, sbin, www*, y *man* y un pequeño directorio *examples* desde el cual los usuarios pueden copiar y modificar el *Makefile*. Si quisiéramos quitar la instalación, podemos ejecutar el script **sbin/mpiuninstall.**

- (**Optativo**) hasta aquí cada usuario puede compilar y ejecutar programas MPI, usando la instalación que se ha compilado en ***/usr/local/mpi-1.2.3*** (o dondequiera que se lo haya instalado). También pueden copiar el *Makefile* de ***/usr/local/mpi-1.2.3/examples*** y adaptarlo para su propio uso. La compañera *User's Guide [9],* disponible en postscript comprimida en el subdirectorio *doc*, da más información sobre compilación y ejecución de programas de MPI con MPICH. Ambos, la Guía de la Instalación (Installation Guide) y la Guía del Usuario (User's Guide) también están disponibles en la Web en el *http://www.mcs.anl.gov/mpi/mpich/docs.html* .

3.6.3. Ejecución

La ejecución se realizó a través de un script denominado ***mpirun***. Este utiliza las variables de entorno del Parallel Operation Environment (poe) para establecer el tipo de adaptador, el protocolo y el spool que se utilizan en cada ejecución.

Los valores posibles de estas variables de entorno son:

Adaptador: MP_EUIDEVICE={css0|en0}. Trabajamos con el *Switch* (css0) o con el *Ethernet* (en0).

Protocolo: MP_EUILIB={us|ip}. Trabajamos con protocolo User Space (us) o protocol IP (ip).

Pool: MP_RMPOOL=9. Para trabajar en interactivo en el SP2 siempre hay que utilizar el Pool9.

En nuestro caso utilizamos la configuración para una red Ethernet, con protocolo IP y Pool 9

Para ejecutar el programa hay que proceder de la siguiente manera:

$> ***mpirun -np {dígito} nombre_ejecutable***

donde {dígito} es el número de procesos que queremos utilizar.

Otra forma más eficaz de ejecutar un programa en MPICH es mediante la orden:

$> ***mpirun –p4pg pgfile nombre_ejecutable***

donde el modificador p4pg se encarga de indicar el tipo de red, y que debe leer un archivo "pgfile" donde se encuentran especificados los computadores que conformarán el sistema paralelo y cuantos procesos se ejecutarán en cada una de ellas.

El formato de *pgfile* debe contener los siguientes campos:

<nombre-computador> <#procs> <nombre-prog> [<login>]

Un ejemplo de este es el que utilizamos en nuestra práctica:

```
cetad          0   /home2/cetad/ignacio/prog
paris          1   /export/home/ignacio/prog
cetadfomec1    1   /home/ignacio/prog
cetadfomec2    1   /home/monica/prog     monica
```

Este archivo especifica 4 procesos, uno en cada computador. Nótese que el 0 en la primera línea esta allí para indicar que otros procesos no se inicien, además del propio iniciado por el usuario. Además en los casos donde el nombre de la cuenta de usuario es diferente es necesario utilizar el **login**. (caso del último computador del ejemplo).

Si necesitáramos ejecutar todos los procesos en un solo computador, como una primera prueba de un programa, podríamos hacerlo repitiendo el nombre en el archivo:

```
cetad      0   /home2/cetad/ignacio/prog
cetad      1   /home2/cetad/ignacio/prog
cetad      1   /home2/cetad/ignacio/prog
```

Así se ejecutarán tres procesos sobre cetad.

Para el caso de ***memoria compartida***, se puede ejecutar sobre un multiprocesador, 10 procesos, usando un archivo que contenga:

```
cetad      9   /home2/cetad/ignacio/prog
```

Nótese que uno de los 10 procesos es iniciado por el usuario directamente, y los otros nueve son especificados en este archivo. Para esto se necesita que MPICH sea configurado con la opción **–comm=shared**; tal como se verifica en el manual de instalación.

Si estamos localmente en el computador cetad y queremos iniciar un trabajo con un proceso en cetad y tres sobre paris, por ejemplo, donde los procesos se comunican a través de la memoria compartida, deberíamos usar:

```
cetad      0   /home2/ignacio/principal
paris      3   /export/home/ignacio/graficos
```

No es posible proporcionar distintos argumentos a la línea de comandos para diferentes procesos MPI sobre un computador.

Para consultar más opciones del *mpirun* podemos usar:

mpirun –help ó ***man mpirun***

- **Variables de entorno usadas en el dispositivo p4**

Hay varias variables de entorno que pueden ser usadas para ajustar la performance del dispositivo *ch_p4* . Estas variables podrían ser parte del ***'.profile'*** o ***'.cshrc'***)

P4 SOCKBUFSIZE. Incrementando este valor puede mejorar la performance en algunos sistemas. Sin embargo, bajo LINUX, particularmente sistemas LINUX con soporte TCP, esto puede incrementar la probabilidad de que MPICH no funcione.

P4 WINSHIFT. Es otro parámetro de red que es soportado por pocas plataformas.

P4 GLOBMEMSIZE. Es la cantidad de memoria en bytes reservada para la comunicación con la memoria compartida (cuando el MPICH es configurado con la opción -c*omm=shared*).

3.6.4. Compilación

Para compilar un programa en C que llama a rutinas MPI se procedió con la siguiente línea de comandos:

$> ***mpicc -o nombre-ejecutable nombre-programa.c***

Donde ***nombre-programa.c*** es el nombre del programa fuente y ***nombre-ejecutable*** es el nombre que uno da al ejecutable resultante de la compilación. El comando ***mpicc*** compila con el compilador del sistema operativo pero llama automáticamente a las librerías de MPICH.

3.6.5. Documentación

El comando *mpiman* en *mpich/bin* es una buena vinculación con las paginas del manual.

El directorio *mpich/www* contiene versiones HTML de las paginas del manual para MPI y MPE.

El directorio *mpich/doc* contiene una Guia de Instalación y también una Guía para el Usuario.

Se puede consultar el manual on line, especificando con el comando **man** la siguiente variable de entorno: **man -M $MPICH_MAN nombre_de_rutina** ó añadiendo este path a nuestra variable de entorno **MANPATH** (que también se puede incluir dentro de nuestro .***profile*** o ***.cshrc*** si se utiliza MPICH habitualmente), de la siguiente manera:

Si utilizamos **ksh**: export MANPATH="$MANPATH:$MPICH_MAN"
Si utilizamos **csh**: setenv MANPATH "$MANPATH":$MPICH_MAN"

3.7. Ejemplos de programas en "C" usando las librerías de MPI

greetings.c : Este programa manda un mensaje desde todos los procesos con rango distinto de 0 al proceso 0. El proceso 0 los saca por pantalla. No hay ninguna entrada y la salida es el contenido de los mensajes recibidos por el proceso 0, que los imprime. Las funciones de MPI utilizadas en el programa ***greetings***, utilizan sus propios tipos de datos, que no son definidos en C, aunque hay una correspondencia entre ellos. El pasaje de mensajes se realiza mediante dos funciones básicas en MPI, ellas son: **MPI_Send()** y **MPI_Recv()**. Además, se modificó dicho programa para que mostrara el nombre del computador del que recibía el mensaje; utilizando la función **MPI_Get_procesor_name()** de MPI.

Código del programa greetings.c

```
#include<stdio.h>
#include<math.h>
#include     "mpi.h"
main(int argc, char* argv[])
{
int       my_rank;        /* rank of process */
int       p;            /* number of processes*/
int       source;         /* rank of sender */
int       dest;           /* tag of receiver */
int       tag = 0;              /* storage for message */
char          message[100];      /* return status for */
char          processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Status status             /* receive */
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Get_processor_name(processor_name, &name);
 if (myid = = 0)
{
  sprintf(message, "Greetings from process %d,  which is on %s", my_rank,
processor_name);
 dest = 0;
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag,
MPI_COMM_WORLD);
          }
 else
{
   for (source = 1; source < p; source ++)
{
MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD,
&status);
printf("%s \n", message);
               }
          }
 MPI_Finalize();        /* Shut down MPI */
} /* main */
```

pi.c : calcula el valor del número π, con la mayor precisión posible, con el error producido y con el tiempo promedio de realización de dicho cálculo, dependiendo de la cantidad de intervalos o procesos que intervienen en el mismo (introducidos por el usuario), lo que permite obtener mayor exactitud en los resultados esperados, cuanto mayor sea la cantidad de intervalos; para lo cual se utilizaron primordialmente las sentencias **MPI_Bcast()** y **MPI_Reduce()**.

Código del programa pi.c

```
#include“mpi.h”
#include<stdio.h>
#include<math.h>
double f(a)
double a
{
return (4.0 / (1.0 + a*a));
}
int main(argc, argv)
int argc;
char *argv[];
{     int done = 0, n, myid, numprocs, I;
double PI25DT = 3.14159265358979323846264 3;
double mypi, pi, h, sum, x;
double startwtime, endwtime;
int namelen;
char processor_name [MPI_MAX_PROCESSOR_NAME];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Get_processor_name(processor_name, &namelen);
fprintf(stderr, “process %d on %s \n”, myid, processor_name);
n = 0;
while (!done)     {
      if (myid = = 0)   {
          printf(“Enter the number of intervals: (0 quits) ”);
          scanf(“%d”, &n);
          startwtime = MPI_Wtime();   }
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      if (n = = 0)
          done = 1;
      else       {
           h == 1.0 / (double) n;
           sum = 0.0;
           for (i = myid + 1; i <= n; i += numprocs)   {
           x = h * ((double)i – 0.5 );
           sum += f(x);   }
            mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                                                MPI_COMM_WORLD);
```

```
if (myid = = 0)     {
    printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi – PI25DT));
    endwtime = MPI_Wtime();
    printf("wall clock time = %f \n", endwtime-startwtime);     }
                }
            }
MPI_Finalize();
}
```

pingpong_mpi.c: transmite un mensaje entre dos computadores solamente, indicando el tiempo promedio de dicha transmisión. Además de las sentencias de envío y recepción de mensajes de MPI, se utilizaron las sentencias de tiempo, de comienzo y de fin, usando **MPI_Wtime()**, para poder calcular el tiempo promedio de ejecución; destacamos que para realizar esto se requiere que ambas sentencias se ejecuten en el mismo proceso.

Código del programa pingpong_mpi.c

```
#include<stdio.h>
#include<string.h>
#include<mpi.h>
#include<math.h>
#define  BUFSIZE  64
char          buf[BUFSIZE];
double        tiempo,startime,endtime;

int main(argc, argv)
int   argc;
char      *argv[];
{
  int size, rank;
 MPI_Status      status;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
if (2 != size)
                {
                MPI_Finalize();
                return(1);
                }
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (0 = = rank)
                        {
 strcpy(buf, "Este es el mensaje que se transmite a traves de computadores");
 startime = MPI_Wtime();
 MPI_Send(buf, BUFSIZE, MPI_CHAR, 1, 11, MPI_COMM_WORLD);
 MPI_Recv(buf, BUFSIZE, MPI_CHAR, 1, 11, MPI_COMM_WORLD, &status);
 endtime = MPI_Wtime();
 tiempo =(endtime-startime) / 2;
 printf("tiempo = %f \n", tiempo);
 printf(" %s \n", buf);
                        }
```

```
else
    {
 MPI_Recv(buf, BUFSIZE, MPI_CHAR, 0, 11, MPI_COMM_WORLD, &status);
 MPI_Send(buf, BUFSIZE, MPI_CHAR, 0, 11, MPI_COMM_WORLD);
 printf(" %s \n", buf);
    }
MPI_Finalize();
return(0);
}
```

Capítulo 4

4. CONSIDERACIONES PARA REALIZAR LA CONSOLA EN MPI

4.1. Un antecedente importante: la consola PVM

PVM (Parallel Virtual Machine): es un conjunto de herramientas del Oak Ridge National Laboratory para paralelizar con memoria distribuida en “Fortran”, “C” y “C++”. Permite crear un sistema paralelo (máquina virtual paralela) a partir de una colección de computadores y redes que pueden ser heterogéneas. Suministra rutinas para el pasaje de mensajes, gestión de tareas, de computadores, etc.

4.1.1. Características principales

- Es un software de dominio público, que cumple los requerimientos que necesita la computación distribuida.
- Reduce el "*wall clock execution time*".
- Es de fácil instalación y uso.
- Posee una interfaz de programación simple y completa.
- Soporta redes con estaciones de trabajo heterogéneas.
- Permite combinaciones de redes locales con las de área extendida.
- Es fácil la **definición** y la posterior **modificación** de la propia Máquina Virtual Paralela.
- Se pueden programar diferentes componentes, en diferentes lenguajes.
- Tolerancia a fallo.
- Cada aplicación "decide" dónde y cuándo sus componentes son ejecutados y determina su propio control y dependencia.
- Está disponible gratuitamente.
- PVM es utilizable en la mayoría de los computadores UNIX: SUN (SUN OS and Solaris), SGI IRIS, IBM AIX, DEC OSF/1, HP-UX, LINUX; y además, sobre WINDOWS NT.

- **Tiene tres componentes básicos**

  - El proceso **daemon (pvmd3)**: Proceso UNIX encargado de controlar el funcionamiento de los procesos de usuario en la aplicación PVM y de coordinar las comunicaciones. Los procesos de usuario se comunican unos con otros a través de los *daemons*. Primero se comunican con el *daemon* local por medio de la librería de funciones de interfaz. Luego el *daemon* local manda/recibe mensajes a/de los *daemons* de los hosts remotos.

  - La librería de rutinas de interfase (libpvm3.a, libfpvm3.a y libgpvm3.a) libpvm3.a: Librería en lenguaje C de rutinas de interfaz;

son simples llamadas a funciones que el programador puede incluir en la aplicación paralela.

Las rutinas de la librería no se conectan directamente con otros procesos, sino que mandan y reciben la configuración de la máquina virtual paralela.
libgpvm3.a: Se requiere para usar grupos dinámicos.
libfpvm3.a: Utilizada para programas en Fortran

- La **consola de PVM** que actúa a modo de intérprete de comandos, proporcionando una interfaz simple entre el usuario y el *daemon* ***pvmd3***. Además se ha introducido ***xpvm*** que proporciona una serie de herramientas gráficas que permiten configurar la red, arrancar procesos y monitorizar la ejecución de los procesos en el tiempo.

4.1.2. Ejecución de la consola PVM

Aquí analizaremos la consola de PVM con el fin de analizar los comandos para el posterior desarrollo de la Consola MPI. Para ello tomamos en cuenta las dos formas de iniciar la consola de PVM:

- **Una es ejecutando**

$ > ***pvm***

Este programa arranca la consola PVM así como el *daemon* ***pvmd*** si no está ya corriendo. La consola puede ser invocada un número arbitrario de veces en cualquiera de los computadores que forman la máquina virtual. Inicialmente, la máquina virtual se compone únicamente del host desde el que se lanza la consola.
Comprobamos la configuración de la máquina virtual usando el comando

pvm> ***conf***

Que muestra los nombres de los computadores conectados en la máquina virtual (en este momento sólo el host desde el que se ejecutó PVM).
A continuación, agregamos a la máquina virtual, otro computador que pertenece a la red local. Para eso, usamos el comando:

pvm> ***add <host_name>***

Y comprobamos que el computador ha sido añadido a la máquina virtual utilizando *conf*.
Eliminamos cualquier computador utilizando:

pvm> ***delete <host_name>***

Y comprobamos la nueva configuración.
Existe una ayuda para la consola PVM, que explica los comandos principales, y a la que se tiene acceso ejecutando

pvm> ***help***

Para terminar PVM hay dos formas:

pvm> ***quit***

Cierra la consola PVM pero mantiene el daemon ***pvmd*** ejecutándose en background, pero este procedimiento todavía permite ejecutar aplicaciones paralelas desde línea de comando.
Otros comandos de interés que se pueden ejecutar desde la consola son:

pvm> ***spawn:*** Inicia un proceso PVM.

pvm> ***ps:*** Muestra el estado de la aplicación; con el flag **-a**, muestra todas las aplicaciones.

pvm> ***kill:*** Termina el proceso PVM especificado.

- **Otra es, utilizando el archivo *hostfile***

  - Se debe crear un archivo que contenga una lista con los *hosts* que se van a utilizar para la máquina virtual paralela, en nuestra cuenta, en el directorio ***HOME*** del usuario. A este archivo lo llamamos ***hostfile*** y vemos su contenido con el comando *cat*

    $ > ***cat hostfile***

    sofia dx = /home/ignacio/pvm3/lib/pvmd
    paris dx = /export/home/paris/ignacio/pvm3/lib/pvmd
    cetadfomec1 dx = /usr/local/pvm3/lib/pvmd

    dx indica, para cada computador, el camino de ubicación del *daemon* **pvmd**

  - Ahora utilizamos *pvm* para iniciar ó arrancar PVM en los nodos especificados:

    $ > ***pvm hostfile***

  - Ahora utilizamos *conf* para que muestre la configuración:

    pvm> ***conf***

  - Luego utilizamos *quit* para salir de la consola, aunque quedan ejecutándose los daemons para poder ejecutar algún programa:

    pvm> ***quit***

  - Finalmente podemos utilizar el comando *xpvm* para interfaz gráfica:

pvm> ***xpvm***

4.1.3. Terminar la ejecución

El programa ***halt*** elimina todos los procesos y daemons de PVM en los computadores que estamos utilizando. Sólo debe ejecutarse cuando se termine de trabajar con PVM.

pvm> halt

Concluye la ejecución del *daemon* (por lo tanto de la máquina virtual) y de la consola PVM.

También se puede terminar PVM usando el comando ***kill*** de UNIX, en línea de comando y asegurándose de que todos los archivos ***/tmp/pvmd.uid*** han sido borrados en todos los computadores que componían la máquina virtual. De todos modos, no es recomendable utilizar esta opción, salvo que no quede más remedio (bloqueo de la consola).

4.1.4. Ejecución de programas

Para ejecutar un programa en PVM, solo se debe escribir el nombre del programa. Por ejemplo, para ejecutar el programa hello, compilado con anterioridad con ***aimk***, solo debemos hacer:

$ > ***hello***

y esto ejecutará el programa ***hello*** en el computador local y ***hello_other*** en el resto de los computadores.

4.2. Análisis de la consola PVM

De observar la implementación anterior, podemos realizar una serie de consideraciones para desarrollar la Consola en MPI.

En primer lugar pasaremos a estudiar las funciones más destacadas y necesarias para la creación de la Consola MPI, basándonos, lógicamente en la consola PVM ya existente y mostrada con anterioridad.

- **Estudio de las funciones a implementar**

La siguiente es la lista de las funciones existentes en la consola PVM para ser consideradas en la implementación de la nueva consola:

add *hostname* **:** Agrega computadores a la máquina virtual, donde *hostname* es el nombre del computador a agregar.
conf : Muestra una lista con la configuración de la máquina paralela virtual.
delete *hostname* **:** Borra computadores a la máquina virtual donde hostname es el nombre del computador a borrar.
export : Agrega variables de entorno para generar una lista exportable.
halt : Detiene todos los daemons.
help ***[comando]*****:** Imprime ayuda acerca del comando.

id : Imprime el número de identificación de tarea.
jobs : Muestra una lista de los trabajos que se están ejecutando.
kill *task_id* **:** Termina una tarea o proceso determinado sin salir de la consola.
mstat *host* **:** Muestra el estado de los computadores.
ps –a : Muestra una lista de todos los procesos.
pstat *tid* **:** Muestra el estado de un proceso.
quit : Sale de la consola sin matar los daemons.
reset : Mata todos los procesos.
setenv : Muestra las variables de entorno y su asignación.
sig *nº task* **:** Manda una señal a la tarea especificada (1=Hangup, 3=Quit, etc.)
spawn ***[****opt****]*** *file* **:** Genera tarea o proceso *file*.
opt: host, ARCH, etc.
unexport : Remueve variables de entorno creadas con **export**.
version : Muestra versión de la implementación que esta corriendo.

Algunas de las funciones listadas con anterioridad se implementarán utilizando lenguaje 'C' y lenguaje shell (***Korn, Bourne, Bourne Again, ó Cshell,*** *según* corresponda).

Con lenguaje shell trataremos especialmente aquellas funciones que se puedan implementar usando en forma directa funciones de cada implementación MPICH o LAM, y para cambiar el formato de las salidas respectivas; y con lenguaje 'C' desarrollaremos aquellas funciones que no tienen similitudes con las de las implementaciones en MPI.

4.3. Consideraciones previas a la implementación de los comandos de la Consola MPI

Como se ha visto en la sección anterior, entre la lista de funciones correspondientes a la consola PVM, las que se analizarán en detalle posteriormente, (que de ahora en mas las llamaremos comandos para no confundirnos con las funciones utilizadas por "C") existen algunas que son de imprescindible utilidad, aunque en otras no hemos encontrado tales virtudes, por tal razón es que antes de continuar con el funcionamiento especifico de la Consola MPI, es necesario aclarar y justificar, la/s razón/es para descartar ciertos comandos, usar algunos, y crear otros.

Para comprender lo antes dicho enumeramos a continuación tales comandos divididos en estos tres grupos, explicando brevemente que nos motivó a tomar tales decisiones:

- **Comandos descartados:**

Entre los comandos que han sido descartados por nosotros a la hora de desarrollar la consola MPI, podemos mencionar los siguientes:

export : Agrega variables de entorno para generar una lista exportable.
id : Imprime el número de identificación de tarea.

mstat *host* **:** Muestra el estado de los computadores.
ps –a : Muestra una lista de todos los procesos.
pstat *tid* **:** Muestra el estado de un proceso.
sig *nº task* **:** Manda una señal a la tarea especificada.
spawn ***[opt]*** *file* **:** Genera tarea o proceso *file*.
opt: host, ARCH, etc.
unexport : Remueve variables de entorno creadas con **export**.

La justificación de porque no hemos incluido en la Consola MPI el comando **export**, es que este es de poca utilidad, dado que todas las variables exportables se deben definir antes de ejecutar la misma, además de ser un comando no propio para la facilitación del procesamiento paralelo, sino mas bien para facilitar la configuración del sistema operativo, lo cual no es el principal objetivo de esta consola; por obvios motivos también se ha descartado el comando **unexport**.

El comando **id** puede ser util, cuando se necesita matar un proceso determinado, ya que para esto es necesario conocer su número de identificación, pero esto se ha conseguido con el comando **jobs**, el cual brinda tal información de todos los procesos MPI, cuando es necesaria.

El comando **mstat**, es un comando que muchas veces resulta redundante, pues dependiendo de la implementación, proporciona una información que se puede obtener con **jobs**, una consideración similar es también válida para el comando **pstat**.

Si bien **ps –a**, muestra todos los procesos, no agrega mucha mas información que la mostrada por el comando **jobs**, para los objetivos de la Consola MPI.

sig manda una señal a la tarea especificada, pero en nuestra experiencia, esto nunca fue demasiado utilizado, motivo por el cual, no lo hemos incluido, al igual que el comando **spawn**, el cual se usa para generar procesos en forma dinamica.

- **Comandos usados:**

Los comandos usados en nuestra Consola, que existen también en la consola PVM, son los siguientes:

add: Agrega computadores al sistema paralelo.
conf: Muestra la configuración del sistema paralelo.
delete: Borra computadores del sistema paralelo.
halt: Sale de la consola y detiene todos los demonios (si existen).
help: Muestra información útil acerca de un comando.
jobs: Lista los trabajos que se están ejecutando.
kill: Termina una tarea ó proceso.
quit: Sale de la consola, sin matar los demonios (si existen).
reset: Mata todos los procesos y demonios (si existen).
setenv: Muestra las variables de entorno y su asignación.
version: Muestra la versión de la implementación.

- **Comandos creados:**

También hemos creído conveniente la necesidad de crear ciertos comandos que faciliten aún más la utilización de las implementaciones MPICH y LAM. Ellos se indican a continuación:

avm: Lista todos los computadores disponibles.
comp: Compila programas MPI sobre consola.
recon: Identifica los computadores disponibles.
run: Ejecuta programas MPI sobre consola.
test: Envía un mensaje a través de los computadores.

El comando **avm** se ha desarrollado para mostrar una lista rápida de todos aquellos computadores que podrán ser habilitados o agregados, según la implementación y dispositivo que se utilice, vale decir, que **avm** no analiza cuales computadores están disponibles, sino solo muestra el resultado de los computadores disponibles hallados en el último reconocimiento (con **recon**).

Para compilar programas en C, haciendo uso de las librerías MPI, hemos creado el comando **comp** permitiendo con este la unificación de los compiladores utilizados por las distintas implementaciones usadas en la consola. De igual manera logramos unificar la ejecución de programas MPI sobre consola, con el comando **run**.

El comando **recon** se desarrolló para permitir al usuario una herramienta útil que realice el reconocimiento, en cualquier momento, de los computadores disponibles, el resultado de este comando es lo que muestra el comando **avm** mencionado anteriormente.

Con el objetivo de verificar la comunicación entre los computadores, creamos el comando **test**, el cual envía un mensaje a través de ellos.

- **Lista definitiva de comandos**

A continuación mostramos la lista de todos los comandos que hemos resuelto implementar en nuestra Consola MPI:

add: Agrega computadores al sistema paralelo.
avm: Lista todos los computadores disponibles.
comp: Compila programas MPI sobre consola.
conf: Muestra la configuración del sistema paralelo.
delete: Borra computadores del sistema paralelo.
halt: Sale de la consola y detiene todos los demonios (si existen).
help: Muestra información útil acerca de un comando.
jobs: Lista los trabajos que se están ejecutando.
kill: Termina una tarea ó proceso.
quit: Sale de la consola, sin matar los demonios (si existen).
recon: Identifica los computadores disponibles.
reset: Mata todos los procesos y demonios (si existen).
run: Ejecuta programas MPI sobre consola.
setenv: Muestra las variables de entorno y su asignación.
test: Envía un mensaje a través de los computadores.
version: Muestra la versión de la implementación.

Capitulo 5

5. DESARROLLO DE CONSOLA DE COMPUTADOR PARALELO EN MPI

5.1. Condiciones para el funcionamiento de la Consola MPI

- **Requerimientos**

  - Se debe contar con computadores ***UNIX*** o ***LINUX*** conectados en red.
  - Antes de ejecutar MPICH ó LAM desde consola, se deben definir y asignar ciertas variables de entorno y *PATH* (en el archivo ***.profile***, ***.bashrc .cshrc*** según el shell que se utilice), para indicar el PATH donde reside MPICH, LAM, los ejecutables de la Consola MPI, los programas fuentes y los correspondientes ejecutables, como se indican a continuación.
    Para los archivos **.profile** y **.bashrc** (usado en Bourne shell, BourneAgain shell o Korn shell) son válidas las siguientes definiciones para las variables de entorno:

    HOME=/home/usuario

    MPI_ROOT=/usr/local/mpich-1.2.3 y

    LAMHOME=/usr/local/lam-6.5.6

    Y los PATH necesarios pueden ser agregados así:

    PATH=:$PATH:$HOME/src:$HOME/bin:$HOME/consola/func:

    export PATH HOME MPI_ROOT LAMHOME

    Si el shell utilizado es csh, agregar las siguientes líneas al archivo ***.cshrc***:

    setenv HOME /home/usuario

    setenv MPI_ROOT /usr/local/mpich-1.2.3

    setenv LAMHOME /usr/local/lam-6.5.6

    setpath ~/src ~/bin ~/consola/func $path

    Si se utiliza el shell **bash** (BourneAgain Shell), las variables de entorno y PATH se definen como en el primer caso.

  - Tener instaladas las implementaciones de MPI: ***MPICH*** (con los dispositivos ***ch_p4*** y ***ch_p4mpd***) y ***LAM*** en cada computador.

- Ambas implementaciones, compiladas e instaladas para cada arquitectura, con lo cual:
  - El programa de instalación coloca las librerías de MPICH: (***libmpi***) en el directorio ***/usr/local/mpich-1.2.3/lib*** y las librerías de LAM: (***liblam***) en el directorio ***/usr/local/lam-6.5.6/lib***.
    Se debe comprobar que las librerías estén en el directorio indicado.

- Crear el archivo **'*allred*'** (particular de la consola) que contiene todas los computadores que componen la red local que se desea utilizar.

- Crear el archivo: ***'machines'*** (particular de la consola), donde se define qué computadores físicos compondrán el computador paralelo.

- Se debe contar con el ejecutable de la Consola MPI denominado ***mpi*** el cual se encuentra en el directorio **/consola/func** dentro del HOME de usuario.

- El comando ***rsh***, debe funcionar en forma automática (sin pedir password), si se cuenta con los archivos indicados en el punto ***confiabilidad entre computadores***, explicado a continuación.

- **Creación de una cuenta de usuario**

  El usuario de la Consola MPI debe tener una cuenta propia, para ello se debe pedir al administrador del sistema que cree una; para mas información ver el punto A.8 en el apéndice A.

- **Confiabilidad entre computadores**

  Para brindar la confiabilidad requerida por las implementaciones MPI, es necesario crear o configurar los archivos de seguridad **hosts.equiv** y **.rhosts** cuyos contenidos y ubicaciones se detallaron en la sección 3.2 del capitulo 3.

5.2. Instalación de la consola MPI

La instalación de la Consola MPI es muy simple, ya que una vez realizados los cambios en los archivos de configuración, tales como *.profile ó .cshrc, etc* según corresponda, se procede a la instalación propiamente dicha.

En primer lugar, se debe montar el diskette mediante el comando **mount** de UNIX, para ello debe tener permiso de ejecución del mismo; probablemente deba ser **root**, en este caso, ejecutar ***su -p***, para preservar las variables de entorno de usuario; dicho procedimiento se indica a continuación:

```
$> mount  –o r  /dev/fd0  /mnt/floppy
```

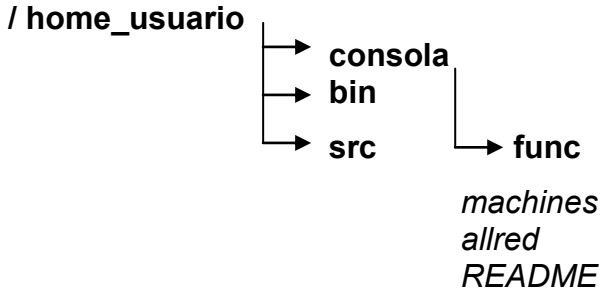
Aclaramos que lo anterior es para montar un diskette en el directorio */mnt/floppy*, aunque también puede ser montado en otro directorio existente.

A continuación, se debe ingresar al diskette con: ***cd /mnt/floppy***, y aquí ejecutar **instalar**, con los parámetros como se detallan a continuación:

$> ***instalar lugar_de_instalación nombre_usuario nombre_grupo***

lugar_de_instalación: generalmente es el HOME de usuario. Por ejemplo: */home/monica.*
nombre_usuario: es el nombre con que el sistema lo identifica.
nombre_grupo: es el grupo al cual pertenece como usuario.

En este paso se crearán los directorios y archivos necesarios, quedando de la forma como se ilustra a continuación.



Vale aclarar que los directorios **src** y **bin** en el HOME de usuario no serán creados durante la instalación, sino cuando se utilice el comando de compilación ***comp*** de la Consola MPI, el cual los creará automáticamente.

Por último, es conveniente desmontar el diskette con el comando **umount**, cambiando previamente de directorio, como se indica a continuación:

$> ***cd***

$> ***umount /dev/fd0***

5.3. Funcionamiento de la Consola MPI

5.3.1. Estructura de la Consola MPI

En lo que se refiere a la estructura de la Consola MPI, indicaremos en primer lugar que esta consola consta básicamente de un programa principal, desarrollado en lenguaje C, y un conjunto de funciones, realizadas en lenguaje shell, las cuales son llamadas desde el primero, utilizando las denominadas *llamadas al sistema*.

A continuación se muestra la estructura en forma esquemática:

Figura 5.1 – Estructura general de consola

Aquí podemos observar que el programa principal utiliza un conjunto de funciones, las cuales implementan cada comando de la consola, e inclusive para distintas implementaciones de MPI, en nuestro caso, para MPICH, con sus dos variantes de dispositivo *ch_p4* y *ch_p4mpd*, y la implementación LAM, las que se han estudiado con anterioridad. De aquí surge la necesidad de realizar tres grupos de 16 funciones para: MPICH con *ch_p4,* MPICH con *ch*_p4mpd, y LAM, como se ejemplifica gráficamente a continuación:

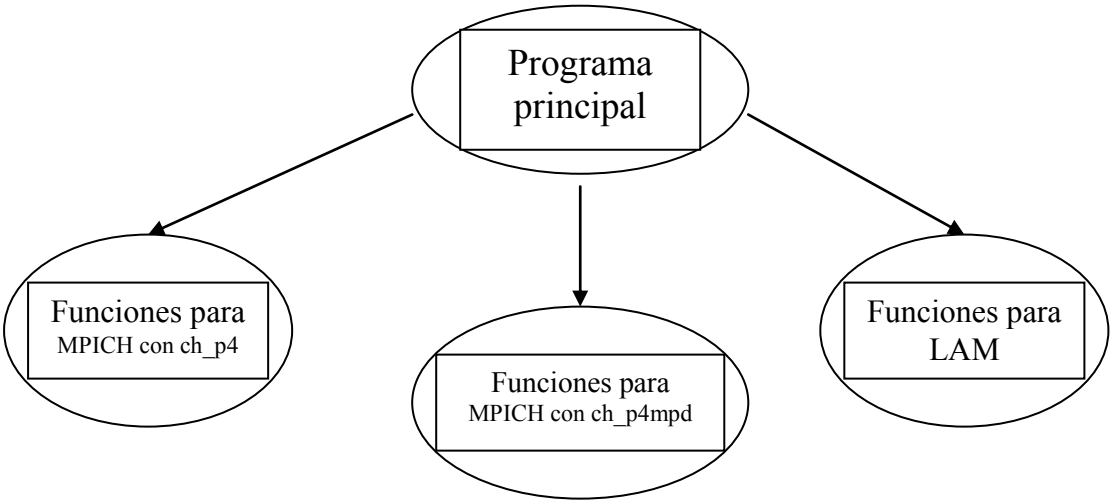


Figura 5.2 – Estructura de consola

Al iniciar la consola, escribiendo en la línea de comandos ***mpi*** se ejecuta el programa *mpi* cuyo código fuente es *mpi.c* , éste tiene algunos parámetros que pueden ser utilizados según lo que se desea:

En primer lugar el parámetro ***–f*** , se utiliza para ingresar a la Consola MPI en forma forzada, este parámetro es especialmente útil para cuando se ha salido de la Consola MPI en forma incorrecta, es decir, si se ha salido sin usar los comandos ***halt*** o ***quit****;* para más detalle ver sección 5.3.4 de este capitulo.

Otra opción muy interesante es la de poder iniciar la Consola MPI, con un conjunto de computadores preestablecidos, los cuales deben figurar en un archivo, que hemos llamado *machines*, con sus nombres respectivos, es decir crear un archivo con los nombres de los computadores a agregar, encolumnados de manera tal que quede un nombre por línea.

Y por último el parámetro ***–help***, en línea de comandos esto seria:

mpi -help

que muestra una breve ayuda del uso de los parámetros para el programa *mpi*, como se indica a continuación:

```
mpi - Inicia la Consola MPI.

SINTAXIS - mpi [-help | -f | machines]

-f       Inicia la Consola MPI forzadamente en caso de que no se haya
         salido de la Consola MPI correctamente.

-help    Muestra esta ayuda.

machines Inicia la Consola MPI con todos los computadores que contiene
         el archivo 'machines'.
```

Una forma de visualizar más detalladamente el funcionamiento de la consola (programa ***mpi.c***), es mediante un diagrama de flujo, como el que se ilustra a continuación:

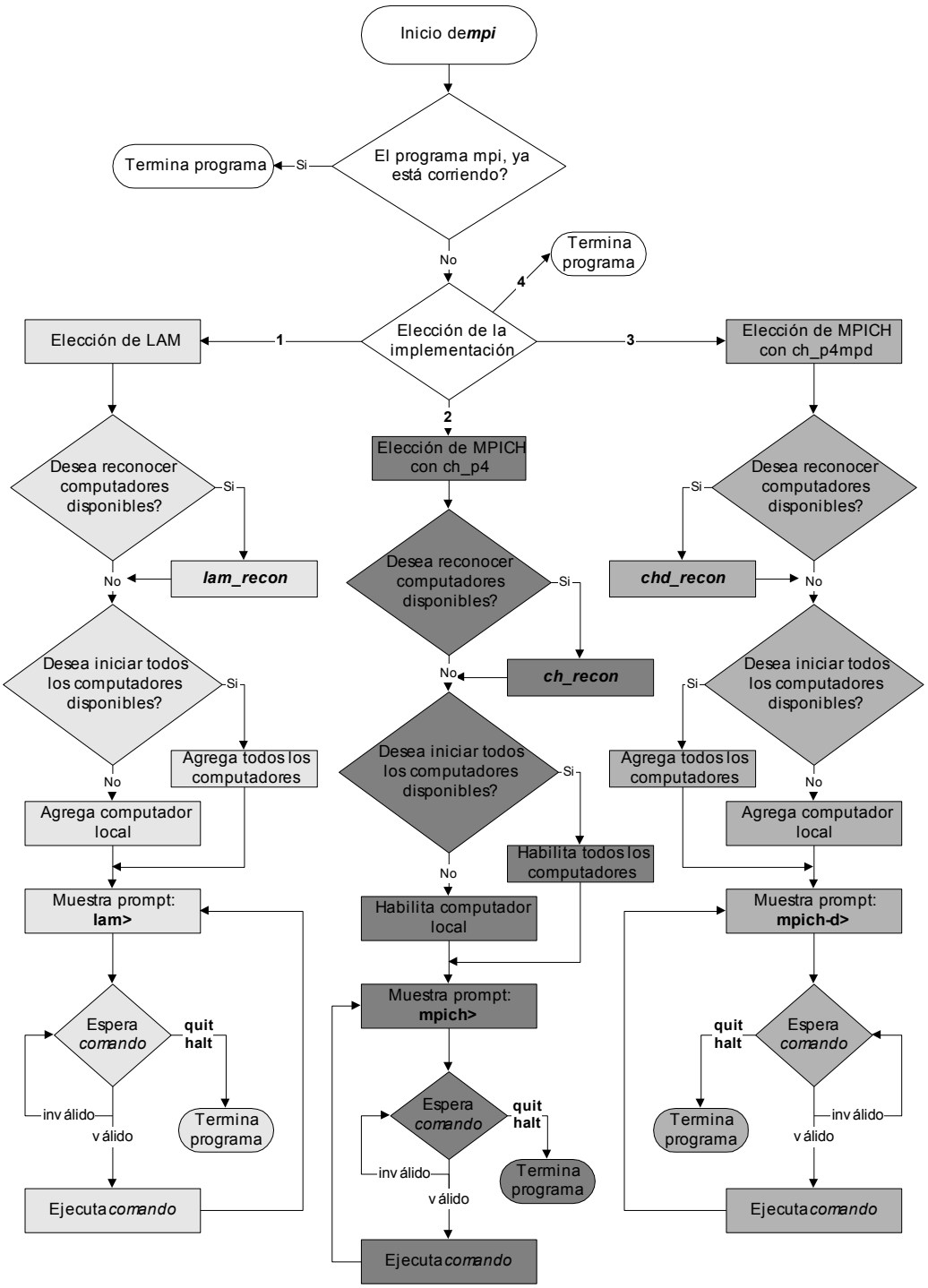


Figura 5.3 – Diagrama de flujo del programa *mpi.c*

La elección de las implementaciones se realiza mediante tres números: 1, 2 o 3; con ello se ingresa a la implementación correspondiente, las que hemos distinguido con distintas tonalidades de grises.

5.3.2. Breve reseña de comandos

A continuación se hará una breve presentación de cada comando para facilitar la familiarización con la Consola MPI, primeramente, como se realiza el inicio a la misma y luego una breve reseña de cada comando.

Aclaramos que cada comando se tratará aquí de forma general, sin entrar en los detalles de cada uno de acuerdo a la implementación elegida, cosa que si será abordada posteriormente.

La forma de iniciar la Consola MPI es escribiendo en linea de comandos:

$ > ***mpi***

Este programa arranca la Consola MPI mostrando una pantalla con cuatro opciones como se muestra a continuación:

```
 ************************************************************************
 *                                                                      *
 *       Elija la opción de implementación MPI que desea usar:          *
 *                                                                      *
 *        1) Para utilizar LAM.                                         *
 *        2) Para utilizar MPICH estático.                              *
 *        3) Para utilizar MPICH dinámico mediante 'daemons'.           *
 *        4) Para SALIR de la consola.                                  *
 *                                                                      *
 ************************************************************************
Opción:_
```

De esta forma podemos elegir la opción de implementación a utilizar y en caso de que la implementación elegida no esté funcionando, el programa preguntará, si desea realizar un reconocimiento de los computadores disponibles, y luego de ello, si desea iniciar la Consola MPI con todas ellas, como se muestra a continuación:

```
ADVERTENCIA - Recomendamos realizar un reconocimiento de los
              computadores disponibles en este momento.


¿Desea hacer el reconocimiento ahora? (S/N):s


computador1.........................Disponible
computador2.........................No disponible
   ...
computadorN.........................Disponible


¿Desea iniciar la consola con todos los computadores disponibles?
 De lo contrario solo se iniciará con el computador LOCAL (S/N):_
```

En ambos casos (Si o No), se inicia el ***demonio*** local si así lo requiere la implementación y dispositivo seleccionado anteriormente (si ya no está corriendo).

La Consola MPI, puede ser invocada un número arbitrario de veces en cualquiera de los computadores que forman el sistema paralelo.

Inicialmente, el sistema paralelo se compone únicamente del computador desde el que se inicia la Consola MPI.

Comprobamos la configuración del sistema paralelo usando el comando ***conf*** como indicamos a continuación:

mpi> ***conf***

Que muestra los nombres de los computadores agregados/habilitados (según dispositivo) hasta el momento, hasta ahora sólo el computador desde el que se ejecutó ***mpi***.

Vale aclarar que el prompt **"mpi>"** utilizado es genérico, pero realmente en la Consola MPI se mostrará un prompt acorde a la implementación y dispositivo seleccionado, es decir:

Para MPICH con dispositivo ch_p4: **mpich>**

Para MPICH con dispositivo ch_p4mpd: **mpich-d>**

Para LAM: **lam>**

A continuación, agregamos/habilitamos otro computador que pertenece a la red local. Para ello, usamos el comando ***add***:

mpi> ***add <nombre_computador>***

Y comprobamos que el computador ha sido añadido/habilitado, utilizando ***conf*** nuevamente.

Se puede eliminar/deshablitar cualquier computador utilizando ***delete***:

mpi> ***delete < nombre_computador >***

Podemos comprobar la nueva configuración con ***conf***.

También existe, una forma de mostrar los procesos que están corriendo ó ejecutándose en las distintos computadores que conforman el computador paralelo, utilizando el comando ***jobs*** de la siguiente forma:

mpi > ***jobs***

Para terminar un proceso determinado, es decir matarlo, se utiliza el comando ***kill*** como se muestra a continuación:

mpi> ***kill <nombre_programa/número_PID>***

Existe el comando ***reset*** que mata todos los procesos y demonios (solo para el dispositivo *ch_p4mpd y* LAM), que se estén ejecutando en el computador paralelo y a su vez hace una limpieza de toda basura a nivel de comandos, funciones, variables, archivos, etc.; quedando dentro de la Consola MPI y pudiendo continuar limpiamente con las funciones de comandos. Se realiza este proceso ingresando el comando de la siguiente manera:

mpi> ***reset***

Para realizar un reconocimiento de los computadores disponibles, se puede utilizar el comando ***recon***, que es equivalente al reconocimiento que se debería efectuar cuando se ingresa a la Consola MPI, luego de haber elegido una de las implementaciones establecidas (mostradas en el punto 5.3.2 de este capítulo), en el que se advierte la conveniencia de hacerlo porque en distintos momentos los computadores disponibles pueden no ser los mismas. La forma de realizarlo desde la Consola MPI es la siguiente:

mpi> ***recon***

El comando ***avm*** lista cuales son todos los computadores disponibles (pero no necesariamente habilitados/agregados). Este comando siempre muestra el conjunto de computadores disponibles hallados en el último reconocimiento, es decir, muestra el resultado del último ***recon*** y se ejecuta como se muestra a continuación:

mpi> ***avm***

Para averiguar los computadores disponibles actualmente se lo puede hacer con el comando ***recon***, explicado anteriormente. El comando ***avm*** muestra los nombres de los computadores disponibles y que pueden ser habilitados/agregados con el comando ***add***.

Existe un simple comando, ***version***, que muestra la versión correspondiente a la implementación MPI seleccionada al iniciar la Consola, y puede visualizarse ejecutando:

mpi> ***version***

Para ver todas las variables de entorno con su asignación (incluidas las definidas para que la Consola funcione) lo hacemos mediante el comando ***setenv*** (no confundir éste con el comando *setenv* de UNIX usado en C-shell), de la siguiente manera:

mpi> ***setenv***

Existe una ayuda general para la consola MPI, que explica los comandos, y es consultada ejecutando ***help***:

mpi> ***help***

Para compilar un programa en MPI, se utiliza el comando ***comp,*** y se debe indicar el nombre del ejecutable seguido del nombre del fuente. Por ejemplo, para compilar el programa hello.c, se debe hacer:

mpi> ***comp hello hello.c***

y esto compilará el programa ***hello*** en el computador *LOCAL* y en el resto de los computadores, colocando el ejecutable en la carpeta **bin** dentro de la cuenta de usuario.

Para ejecutar un programa en MPI, se utiliza el comando: ***run*** y esto se puede realizar indicando solo el nombre del programa ejecutable. Por ejemplo, para ejecutar el programa hello, compilado con anterioridad, se debe hacer:

mpi> ***run hello***

y esto ejecutará el programa ***hello*** en el computador *LOCAL* y en el resto de los computadores agregados/habilitados.

Existe un comando ***test*** que envía un mensaje a través de los computadores comprobando la comunicación entre ellos. Este comando necesita como parámetro un número, el cual indica el tamaño en bytes del mensaje, o la cantidad de veces que se envía el mismo a través del anillo, según la implementación.

mpi> ***test <número>***

Para terminar MPI, hay dos formas, ***quit*** o ***halt***

- El comando ***quit*** sale de la Consola MPI sin matar los demonios (si existen), y todavía permite que sigan ejecutándose programas de aplicaciones paralelas ya iniciados.

  mpi> ***quit***

- El programa ***halt*** sale de la Consola MPI y elimina todos los procesos y *demonios* (si existen) en los computadores que estamos utilizando. Esto solo debe ejecutarse cuando se termine de trabajar con MPI.

  mpi> ***halt***

5.3.3. Explicación minuciosa de cada comando

- **Para implementación MPICH con dispositivo *ch_p4***

Una vez iniciada la consola se pueden ejecutar los siguientes comandos que aquí se detallarán minuciosamente:

Comenzamos con el comando ***add*** el cual permite habilitar un computador o varios a la vez, o también un computador o varios con n procesos cada uno, o combinación de los anteriores. Un ejemplo de esto es:

mpich> ***add cetad sofia cetadfomec2***

En este caso se habilitarán los tres computadores indicados con un proceso cada uno.

mpich> ***add cetad:1 sofia:3 cetadfomec2:5***

En este caso se habilitará *cetad* con un solo proceso, *sofia* con 3 procesos y *cetadfomec2* con 5 procesos. Vale aclarar que el primer computador se ha habilitado con un proceso al igual que en el caso anterior, por lo tanto para habilitar con un solo proceso es mas simple indicar solo el nombre del computador.

mpich> ***add cetad:2 sofia cetadfomec2:5 prited***

En este ejemplo se habilitarán los cuatro computadores que se muestran, el primero con dos procesos, el segundo con un proceso, el tercero con cinco procesos y el cuarto con un solo proceso.

El comando **avm** se puede ejecutar en cualquier momento, y no requiere de argumentos

mpich> ***avm***

El comando **comp**, utilizado para compilar los programas de C basados en MPI, necesita de dos argumentos, el nombre del ejecutable y el nombre del programa fuente, como se muestra a continuación:

mpich> ***comp ejecutable programa.c***

Una segunda opción muy interesante es la de usar el parámetro **–a**, el cual permite compilar el programa, pero no solo en el computador local, sino en todos los computadores disponibles en la red. La forma de lograrlo es:

mpich> ***comp -a ejecutable programa.c***

El siguiente comando que vamos a explicar es el **conf**, que no necesita de ningún parámetro ni argumento, por lo tanto, solo se ejecuta con:

mpich> ***conf***

Para deshabilitar los computadores utilizamos el comando **delete**, el cual requiere como argumento el nombre del computador, aunque se le pueden indicar mas de uno, por ejemplo:

mpich> ***delete cetad sofia***

en este ejemplo se deshabilitarán los computadores *cetad* y *sofia* con todos sus procesos.

El comando **halt** no necesita argumentos y solo se debe ejecutar cuando no se desea seguir utilizando la implementación.

mpich> ***halt***

Para ver la ayuda de todos los comandos de la consola, se ejecuta el comando **help** sin argumentos, o bien si se desea ver la ayuda de un comando en particular requiere como argumento el nombre del comando, es decir:

mpich> ***help [comando]***

Para saber qué procesos se están ejecutando usamos el comando **jobs** sin argumentos. Por lo tanto se ejecuta como sigue:

mpich> ***jobs***

Para matar los procesos con el comando **kill**, necesitamos indicarle como argumento el nombre del programa a matar.

mpich> ***kill <nombre_programa>***

Para salir de la consola sin borrar la configuración debemos utilizar:

mpich> ***quit***

El comando **recon** tampoco hace uso de argumentos para reconocer que computadores están disponibles, entonces:

mpich> ***recon***

Para borrar las configuraciones y matar todos los procesos sin salir de la consola usamos

mpich> ***reset***

Al comando **run** se le debe especificar el nombre del programa ejecutable como argumento, por ejemplo, para ejecutar el programa *hello* escribimos:

mpich> ***run hello***

Para mostrar las variables de entorno solo se debe indicar el comando:

mpich> ***setenv***

Para el comando **test** es necesario especificar como argumento el número de bytes del mensaje, es decir:

mpich> ***test <nº bytes>***

El comando **version** no utiliza argumentos

mpich> ***version***

- **Para implementación MPICH con dispositivo *ch_p4mpd***

mpich-d> ***add cetad sofia cetadfomec2***

En este caso se agregarán los tres computadores indicados con un proceso cada uno.

El comando **avm** se puede ejecutar en cualquier momento, y no requiere de argumentos

mpich-d> ***avm***

El comando **comp**, utilizado para compilar los programas de C basados en MPI, necesita de dos argumentos, el nombre del ejecutable y el nombre del programa fuente, como se muestra a continuación:

mpich-d> ***comp ejecutable programa.c***

Una segunda opción muy interesante es la de usar el parámetro **–a**, el cual permite compilar el programa, pero no solo en el computador local, sino en todos los computadores disponibles en la red. La forma de lograr esto es:

mpich-d> ***comp -a ejecutable programa.c***

El siguiente comando que vamos a explicar es el **conf**, este tiene la opción del parámetro **–**a, el cual permite ver en detalle la configuración del anillo, por lo tanto para cada caso se ejecutará:

mpich-d> ***conf***
ó
mpich-d> ***conf –a***

Para borrar los computadores utilizamos el comando **delete**, el cual requiere como argumento el nombre del computador, aunque se le pueden indicar mas de uno, por ejemplo:

mpich-d> ***delete cetad sofia***

en este ejemplo se borrarán los computadores *cetad* y *sofia*.

El comando **halt** no necesita argumentos y solo se debe ejecutar cuando no se desea seguir utilizando la implementación.

mpich-d> ***halt***

Para ver la ayuda de todos los comandos de la consola, se ejecuta el comando **help** sin argumentos, o bien si se desea ver la ayuda de un comando en particular se requiere como argumento el nombre del comando, es decir:

mpich-d> ***help [comando]***

Para saber qué procesos se están ejecutando usamos el comando **jobs** sin argumentos. Por lo tanto se ejecuta como sigue:

mpich-d> ***jobs***

Para matar un determinado proceso es necesario indicar como argumento del comando **kill** el número de *jobid*, el cual se puede ver ejecutando el comando **jobs**:

mpich-d> ***kill <nº jobid>***

Para salir de la consola sin matar los demonios debemos utilizar:

mpich-d> ***quit***

El comando **recon** tampoco hace uso de argumentos para reconocer que computadores están disponibles para ser agregados, entonces:

mpich-d> ***recon***

Para matar los demonios, borrar temporarios y matar todos los procesos sin salir de la consola usamos

mpich-d> ***reset***

Al comando **run** se le debe especificar el nombre del programa ejecutable como argumento, por ejemplo, para ejecutar el programa *hello* escribimos:

mpich-d> ***run hello***

Para mostrar las variables de entorno solo se debe indicar el comando:

mpich-d> ***setenv***

Para el comando **test** es necesario especificar como argumento el número de veces que el mensaje recorrerá el anillo, es decir:

mpich-d> ***test <nº veces>***

El comando **version** no utiliza argumentos

mpich-d> ***version***

- **Para implementación LAM**

Para el comando **add** se necesita como argumento el nombre del computador a agregar, existiendo también la posibilidad, como en las otras implementaciones, de ingresar varios argumentos a la vez. Por ejemplo:

lam> ***add paris cetad sofia cetadfomec1***

En este caso se agregarán los cuatro computadores indicados: *paris, cetad, sofia y cetadfomec1*.

El comando **avm** se puede ejecutar en cualquier momento, y no requiere de argumentos.

lam> ***avm***

El comando **comp**, utilizado para compilar los programas de C basados en MPI, necesita de dos argumentos, el nombre del ejecutable y el nombre del programa fuente, como se muestra a continuación:

lam> ***comp ejecutable programa.c***

Una segunda opción muy interesante es la de usar el parámetro **–a**, el cual permite compilar el programa, no solo en el computador local, sino en todos los computadores disponibles en la red. La forma de lograr esto es:

lam> ***comp -a ejecutable programa.c***

El siguiente comando que vamos a explicar es el **conf**, este no necesita de ningún parámetro ni argumento, por lo tanto, solo se ejecuta escribiendo:

lam> ***conf***

Para borrar los computadores utilizamos el comando **delete**, el cual requiere como argumento el nombre del computador, aunque se le pueden indicar mas de uno, por ejemplo:

lam> ***delete cetad paris***

En este ejemplo se borrarán los computadores *cetad* y *paris*.

El comando **halt** no necesita argumentos y solo se debe ejecutar cuando no se desea seguir utilizando la implementación, esto es:

lam> ***halt***

Para ver la ayuda de todos los comandos de la consola, se ejecuta el comando **help** sin argumentos, o bien si se desea ver la ayuda de un comando en particular se requiere como argumento el nombre del comando, es decir:

lam> ***help [comando]***

Para saber que procesos se están ejecutando usamos el comando **jobs** sin argumentos. Por lo tanto se ejecuta como sigue:

lam> ***jobs***

Para matar un determinado proceso es necesario indicar como argumento del comando **kill** el número de PID (identificador del proceso), el cual se puede ver ejecutando el comando **jobs**:

lam> ***kill <nº PID>***

Para salir de la consola sin matar los demonios debemos utilizar:

lam> ***quit***

El comando **recon** tampoco hace uso de argumentos para reconocer que computadores están disponibles para ser agregados, entonces se usará:

lam> ***recon***

Para borrar temporarios y matar todos los procesos sin salir de la consola usamos:

lam> ***reset***

Al comando **run** se le debe especificar el nombre del programa ejecutable como argumento, por ejemplo, para ejecutar el programa *hello* escribimos:

lam> ***run hello***

Además **run** brinda la posibilidad de especificar el número de procesos en la ejecución de un programa, por ejemplo, para ejecutar el programa hello con 10 procesos, indicamos:

lam> ***run 10 hello***

Para mostrar las variables de entorno solo se debe indicar el comando **setenv** como se muestra a continuación:

lam> ***setenv***

Para el comando **test** es necesario especificar como argumento el número de bytes que contendrá el mensaje que recorrerá el anillo, es decir:

lam> ***test <nº bytes>***

El comando **version** no utiliza argumentos, entonces:

lam> ***version***

5.3.4. Consideraciones al crear la Consola MPI y cada comando

- **La Consola MPI**

Se buscó primeramente la generación de un prompt acorde a la implementación que se esté usando, para ello se procedió a la creación de un programa en lenguaje "C", desde el cual hacemos llamadas a las distintas funciones, que implementan cada comando; se detecta qué implementación se elige y se guarda el prompt a utilizar en una variable interna llamada PS1 (no es la usada por UNIX o LINUX), donde PS1 tomará distintos valores.

Seguidamente, al imprimir el prompt, se espera que se ingrese algún comando de la consola, que se puede determinar, mediante el uso de la sentencia de control anidada **if then else**.

También se ha previsto la inicialización de la Consola MPI (programa *mpi*), con el uso de ciertos parámetros o un argumento, ellos son: los parámetros ***–help***, ***–f*** y el argumento ***machines*.**

Con respecto al parámetro ***–help*** no necesitamos explicar demasiado, pues solo muestra una pequeña ayuda del uso de **mpi**; en cuanto al parámetro ***–f***, es interesante mencionar, que ha sido creado para evitar que la consola no pueda reiniciarse por causa de no haber salido antes adecuadamente, es decir, sin hacer uso de los comandos **quit** o **halt**, pues mientras la consola está activa se genera un archivo temporario llamado *cons-mpi~,* para controlar si existe una consola funcionando o no, con el fin de asegurarnos que no haya más de una consola activa. Como ya hemos dicho, si se termina la ejecución de la consola de alguna forma imprevista, sería necesario eliminar primero el archivo *cons-mpi~* para poder iniciar la consola nuevamente; es por eso que se ha provisto el parámetro ***–f***, que lo realiza automáticamente.

Por último, el argumento ***machines,*** es un archivo que puede ser creado por el usuario, el cual contiene todos los computadores que se desean inicializar al arrancar la consola. Este archivo puede tener cualquier nombre, pero es necesario, que contenga los nombres de los computadores en forma de columna, es decir, solo un nombre por renglón.

El programa **mpi** también inicializa los archivos *pgfile_ch, pgfile_chd* y *pgfile_lam*, luego utilizados, y presenta una pantalla de bienvenida al usuario. Se presiona ENTER para continuar, y se pasa a una segunda pantalla que permite la elección de la implementación MPI a utilizar, lo cual es fundamental para poder seleccionar el **prompt** a mostrar y la familia de **funciones** a utilizar posteriormente.

Seguido a la elección anterior, se realiza una pregunta para saber si se desea realizar un reconocimiento de los computadores disponibles o no, en caso afirmativo se ejecuta internamente el comando **recon** de la implementación elegida; una segunda pregunta aparece indicando si se quiere iniciar la Consola MPI con todos los computadores reconocidos en el paso anterior; si la respuesta es negativa, se inicia la consola solo con el computador local; de lo contrario se utiliza el archivo *pgfile_* * (el * cambia de acuerdo a la implementación elegida) generado por el comando **recon** como si fuera el archivo *machines;* en consecuencia se iniciará la consola con todas los computadores disponibles.

Vale aclarar que como la implementación LAM a veces producía errores cuando se iniciaba con todos los computadores a la vez, se ha incluido un mensaje para esta situación.

- **Para MPICH con dispositivo *ch_p4***

add: al desarrollar este comando, se tuvo en cuenta que al tratarse de un dispositivo que no hace uso de *demonios,* sólo necesitamos guardar los nombres de los computadores a habilitar en un archivo, que llamamos *pgtmp_ch;* pero para habilitar efectivamente un computador, es necesario que exista otro archivo llamado *pgfile_ch*, donde figuran todos los computadores disponibles para esta implementación y dispositivo. Este último, se crea automáticamente cuando se elige realizar un reconocimiento de los computadores disponibles al iniciar la consola por primera vez.

Además de los archivos nombrados, se genera otro denominado *pgfile* que tiene una configuración especial y se utilizará mas tarde para la ejecución de programas MPI desde consola.

También se ha previsto la necesidad del agregado de muchos computadores a la vez e inclusive varias veces un mismo computador como se ha descripto anteriormente.

El mismo comando 'add' cada vez que se ejecuta cuenta el número de computadores que están habilitados, y este número se guarda en un archivo denominado *n_proc*, que se utilizará mas tarde como parámetro interno para la ejecución de programas en consola.

Otra consideración que se tuvo en cuenta es la de testear si el computador local se encuentra entre los computadores ya habilitados, de lo contrario se habilitará automáticamente, mostrando en pantalla la siguiente linea:

```
'nombre_computador' también será habilitado (COMPUTADOR LOCAL).
```

De esta forma nos aseguramos que siempre estará presente el computador local.

avm: Al crear este comando se pensó en una rápida información sobre los computadores disponibles para ser utilizados desde la consola. El comando 'avm' comprueba la existencia y muestra el contenido de un archivo llamado *pgfile_ch;* el mismo es generado cuando se realiza un reconocimiento de computadores disponibles, o sea al inicio de la consola o bien ejecutando el comando ***recon***.

Es importante aclarar que si no se ha hecho un reconocimiento antes de ejecutar este comando, aparecerá la inscripción:

```
No hay computadores disponibles.
```

Es esa la causa de la advertencia que se muestra al iniciar la Consola MPI.

comp: Este comando es el utilizado para compilar cada programa fuente escrito en lenguaje C, para esto utiliza el ejecutable del compilador *mpicc* proporcionado por la implementación MPICH cuando se instala con el dispositivo *ch_p4*; este comando utiliza dos parámetros, *prog* y *prog.c* (nombre de ejecutable y fuente respectivamente).

Además ubica los ejecutables en la carpeta **bin** del HOME de usuario, y lee los fuentes desde cualquier lugar al que se pueda tener acceso. Para asegurar el perfecto funcionamiento del compilador, recomendamos ubicar los

programas fuentes de C en el directorio **src** dentro del HOME de usuario (ambos directorios serán creados por la ejecución del comando '*comp*').

Lo más interesante de este comando es la opción **–a** (por *all=todo*) con la que se puede compilar un determinado programa en todos los computadores disponibles. Para ello es necesario que el programa fuente esté presente y accesible en el computador local. Para esta última opción los programas fuentes serán copiados al resto de los computadores y al local en el directorio **src**, y los ejecutables ubicados en el directorio **bin** (ambos directorios serán creados automáticamente por el comando '*comp*').

conf: Para realizar este comando se necesitó primeramente analizar cuales computadores, de los disponibles, se encuentran habilitados, guardar esta lista en un archivo, para luego contar de cuantos se trata, este número es necesario para mostrar el siguiente mensaje:

```
Hay habilitados 8 computador(es):
```

Para indicar cuantos procesos realizará un solo computador, se necesitó nuevamente constatar qué computadores de los disponibles se encuentran habilitados, y luego contar, cuantas veces aparece el mismo computador. Por último vale mencionar que se fueron almacenando en una variable todos los procesos contados en los pasos anteriores para poder indicar al final el total de procesos que serán creados. Una salida típica sería:

```
Hay habilitados 3 computador(es):

cetad para 1 proceso.
sofia para 3 procesos.
paris para 2 procesos.

Total de procesos: 6
```

delete: en la creación del comando 'delete' se tuvo en cuenta el análisis del archivo *pgtmp_ch* nombrado en la explicación del comando **add**; en este archivo se encuentran los nombres de los computadores habilitados, por lo tanto para poder eliminar un computador, es necesario que éste exista en el archivo mencionado, y por ello se realiza tal comprobación antes de eliminar un computador; además, también se borra este computador del archivo especial *pgfile*, para que no sea utilizado en el momento de la ejecución.

Por último se realiza una contabilización de los computadores que quedan en *pgtmp_ch* para actualizar y almacenar tal número en el archivo *n_proc*.

ch_dtarch: esta función es de uso interno, y solo se usa cuando se realiza un reconocimiento de los computadores disponibles al inicio de la Consola MPI; debe detectar las arquitecturas de los computadores conectados a la red, haciendo uso de la función ***tarch*** provista por MPICH. El resultado se guarda en archivos, con nombres de acuerdo al nombre de cada computador **.ARQ(nombre_computador),** y cuya generación es automática.

halt: este comando cuenta el número de computadores existentes en el *pgtmp_ch* y con esta información muestra cuantos computadores se han borrado.

help: se ha realizado un comando 'help' que funciona de dos maneras, la primera es cuando solo se ejecuta help en la consola, donde se muestra una ayuda básica y general de todos los comandos, y una segunda forma es la de utilizar el nombre de un comando como argumento de 'help'; de este modo reunimos toda la ayuda de la Consola MPI para este dispositivo en una sola función llamada 'ch_help'.

jobs: este comando se desarrolló considerando primeramente cuales de los computadores disponibles están habilitados, almacenando luego este dato en un archivo denominado *maqs*.

Por otro lado como cuando se ejecuta un programa, también se genera un archivo llamado *prog,* que contiene el programa que se está ejecutando, el comando 'jobs' comprueba la existencia, y luego utiliza los dos archivos antes mencionados, generando un mensaje cuya forma es:

```
Están corriendo los siguientes trabajos:

En cetad está(n) corriendo cpi
En sofia está(n) corriendo cpi
En paris está(n) corriendo cpi
```

kill: lo primero que se consideró para la creación de este comando fue analizar el tipo de arquitectura correspondiente a cada computador disponible, con el fin de ejecutar los comandos adecuados para la obtención del PID del proceso a matar. Este es un dato necesario para poder eliminar el proceso en cuestión, y se realiza tanto para el computador local, como para el resto de los computadores que componen la red.

Este comando utiliza el comando 'kill' del sistema, para matar cualquier proceso que se le indique, siempre que se tenga permiso.

quit: este comando solo hace que se salga de un bucle, imprime el prompt y espera que se escriba un comando, utilizado solo dentro del programa *mpi.*

recon: este comando verifica si existe el archivo *allred,* que contiene todos los computadores que componen la red. Si el mismo existe, se procede a verificar si en cada computador existe el ejecutable más importante de ésta implementación con el dispositivo *ch_p4*, el ejecutable *mpirun*, si es hallado en el lugar correcto, se indica como **Disponible**, de lo contrario se indicará como **No disponible**.

reset: aquí este comando sólo borra el contenido de los archivos *pgfile*, *pgtmp_ch* y *n_proc* mostrando el mensaje:

```
Se ha reseteado el sistema paralelo.
```

run: al ejecutarse este comando, se guarda en un archivo *prog* el programa ejecutado (para ser utilizado luego por el comando '*jobs*') a la vez que se introduce el nombre del ejecutable en un archivo *pgfile.* Este último es conformado (por el comando '*add*') para ser utilizado aquí, al llamar a *mpirun,* ejecutable proporcionado por MPICH. Para preservar el archivo anterior se

copia el modificado como *pgfilex*, que es el utilizado para la ejecución mediante *mpirun.*

Aquí solo se utilizó una de las formas de ejecución de *mpirun*, ya que existen varias modalidades; la utilizada por este comando es:

mpirun –v –p4pg pgfilex nombre_ejecutable

Se ha adoptado esta forma porque es la más flexible, pues permite gobernar varios procesos e indicar los procesadores que los ejecuten. Esta modalidad hace uso del archivo *pgfilex* donde se especifican tales datos, que es generado automáticamente.

setenv: este comando solo invoca el comando *env* del sistema.

test: el comando '*test*' se ha pensado para verificar la correcta comunicación entre los computadores disponibles, no haciendo otra cosa que un *ping* a cada computador, con una medida de paquete arbitraria, que es especificada como argumento del comando '*test*'.

version: este comando muestra la versión de la implementación mpich que se está utilizando; para lograrlo se utiliza el archivo *README* desde donde la propia implementación está instalada, y luego este archivo es procesado para detectar la información requerida.

- **Para MPICH con dispositivo *ch_p4mpd***

add: al crear este comando, se tuvo en cuenta primeramente si se ha proporcionado algún argumento o no, y en caso afirmativo se procede a verificar la existencia del archivo *pgtmp_chd.* Luego se va tomando de a un argumento por vez, y se van evaluando para saber si corresponden a un computador que ya fue agregado o no, lo cual se realiza comprobando si el nombre del computador existe o no en el archivo *pgtmp_chd.*

A continuación se verifica si se trata de un computador disponible o no. Verificando la existencia de su nombre en el archivo *.pgarq_chd*; creado cuando se realiza un reconocimiento de los computadores de la red; el mismo contiene todos los computadores de la misma arquitectura que la local, y que tengan instalada la implementación y dispositivo correspondiente.

Una vez comprobado esto se mata el anillo existente con el comando *mpdallexit* del dispositivo usado y se modifica el archivo *pgtmp_chd,* agregando el computador o computadores que se pasan como argumentos del comando '*add*'. Aquí también se cuenta el número de computadores presentes en el archivo *pgtmp_chd* y este número se almacena en un archivo llamado *n_proc*; finalmente se inicia de nuevo el anillo con todos los computadores mediante el comando *startdaemons* del dispositivo usado.

Se ha considerado también la posibilidad de que se agregue el computador local, en caso que no haya sido agregado hasta el momento, pues es indispensable para el funcionamiento del anillo. Esto se hará automáticamente, indicándose el siguiente mensaje:

```
'nombre_computador' también será agregado al anillo (COMPUTADOR LOCAL).
```

avm: este comando funciona igual al '*avm*' anterior, con la diferencia que éste utiliza los computadores presentes en el archivo *pgfile_chd*.

comp: Este comando, al igual que el anterior, es utilizado para compilar cada programa fuente escrito en lenguaje C, pero en este caso utiliza el ejecutable del compilador *mpicc* proporcionado por la implementación MPICH cuando se instala con el dispositivo *ch_p4mpd*; este comando también utiliza dos parámetros, *prog* y *prog.c* (nombre de ejecutable y fuente respectivamente).

Del mismo modo, ubica los ejecutables en el directorio **bin** del HOME de usuario, y los fuentes los lee desde cualquier lugar que se pueda tener acceso; los programas fuentes de C se copiarán en el directorio **src** dentro del HOME de usuario (ambos directorios serán creados con la ejecución del comando '*comp*').

Lo importante de este comando es la opción **–a** (por *all=todo*), como en el otro caso, con la que se puede compilar un determinado programa en todos los computadores disponibles; es necesario que el programa fuente este presente y accesible en el computador local. Para esta última opción los programas fuentes serán copiados al resto de los computadores en el directorio **src**, (inclusive en el directorio **src** del local), y los ejecutables ubicados en el directorio **bin** (ambos directorios serán creados automáticamente por el comando '*comp*').

conf: el comando '*conf* para el dispositivo *ch_p4mpd* se ha desarrollado considerando la existencia de un comando propio de esta implementación y dispositivo, tal como lo es el *mpdtrace* que ofrece una lista con todos los computadores que conforman el anillo. Para algunos comentarios primeramente se tuvo en cuenta el archivo *n_proc* para mostrar cuantos computadores forman el anillo actualmente, ya que este archivo contiene el número de computadores agregados en cada instante, como así también se consideró que cuando éste tenga el valor 0, se muestre en pantalla el siguiente mensaje:

```
El anillo no tiene ningún computador.
```

El parámetro '**-a**' de este comando hace que él muestre en detalle los computadores que forman el anillo, es decir, que con este parámetro no se realiza ningún tipo de filtrado a la salida del comando *mpdtrace,* mientras que sin éste, solo nos quedamos con el nombre de los computadores.

delete: para el comando '*delete*' se tuvo en cuenta que borrar el computador local rompía el anillo (si existen otros computadores), y era necesario reiniciar la Consola MPI para que aquel se volviera a formar. Por lo tanto se analiza en este comando si entre los computadores a borrar figura el local; si es así, tal acción no se llevará a cabo y aparecerá una indicación como la que se muestra a continuación:

```
No se puede borrar COMPUTADOR LOCAL, pues no es único en el anillo.
```

Esta aclaración de que "no es único en el anillo", es porque el comando sí permite borrar el computador local cuando éste es el último.

Para eliminar computadores, se analiza si se le ha especificado algún argumento (es decir algún nombre de computador), si es así, se verifica la existencia del archivo *pgtmp_chd,* en el cual figuran los computadores que forman el anillo.

Si el o los computadores a eliminar, cuyos nombres se han pasado como argumentos, existen en el archivo anterior; se mata todo el anillo, utilizando el comando *mpdallexit*, y luego se borran el o los nombres de los computadores correspondientes en dicho archivo; una vez borrados, se actualiza el número de computadores que quedan, y se almacena en *n_proc;* luego se inicia el anillo con la nueva configuración, utilizando el comando *startdaemons.* Esto último, permite iniciar el anillo con todos los computadores especificados en el archivo *pgtmp_chd* antes mencionado.

chd_dtarch: esta función es de uso interno y no aparece en la consola como comando, pero es utilizada por el comando '*recon*', siendo la encargada de detectar la arquitectura de todos los computadores disponibles de la red, para esta implementación y dispositivo; almacenando tal información (la arquitectura) en archivos ocultos cuyos nombres están formados por un punto inicial seguido de "ARQ" y pegado a ARQ el nombre del computador correspondiente. Se crea además, un archivo llamado *.pgarq_chd* en el cual se encuentra la lista de los computadores disponibles, y que tienen la misma arquitectura que la local. Esto es necesario puesto que este dispositivo de MPICH solo funciona para un sistema homogéneo de computadores, es decir, que los computadores deben ser de la misma arquitectura.

halt: este comando es el que hace que la Consola MPI termine, y a la vez mate todos lo demonios existentes, para esto, se ha utilizado nuevamente el comando del dispositivo llamado *mpdallexit*, el cual se encarga de eliminar todos los demonios que estaban corriendo hasta el momento. Además, el comando '*halt*', se encarga de eliminar los archivos *pgtmp_chd* y *n_proc* entre otros.

help: el comando '*help*' de este dispositivo funciona exactamente igual al anteriormente explicado, con la diferencia que aquí se llama a una función diferente a la anterior, llamada *chd_help*, ya que los comandos de esta tienen pequeñas diferencias, y por supuesto también sus ayudas

jobs: el comando 'jobs' implementado por la función *chd_jobs* hace uso del comando *mpdlistjobs* de MPD (demonio multipropósito), para invocarlo y filtrar partes de la salida, para uniformarlas en la consola.

kill: '*kill*' como en el caso anterior invoca el comando *mpdkilljob* el cual exige como argumento el número de *jobid*, por lo tanto aquí se han agregado algunas ayudas para especificar que este se puede obtener ejecutando el comando '*jobs*' antes explicado, como así también otros comentarios que hacen mas interactivo el trabajo en consola.

quit: como en el caso anterior '*quit*' no es una función, aunque exista como comando de la Consola MPI, y sirve para salir de la consola sin hacer otra cosa, que mostrar en pantalla, que los demonios siguen corriendo.

recon: el comando '*recon*' es el encargado de realizar un reconocimiento a través de la red para averiguar que computadores están o no disponibles, para ser utilizados por esta implementación y dispositivo; para ello utiliza el archivo *allred* en el cual figuran todos los computadores de la red, y comprueba uno por uno si tienen el ejecutable *mpd* ubicado en *$MPI_ROOT/ch_p4mpd/bin.* Si esta búsqueda es satisfactoria se mostrará "**Disponible**" de lo contrario "**No disponible**".

Se comprueba la comunicación por red a cada computador, haciendo uso del comando '*rsh*' de UNIX; de esta forma se verifica la perfecta y muy necesaria comunicación entre los computadores, y estos resultados se guardan en el archivo *pgfile_chd* listando todos los computadores disponibles.

Con este comando también se llama a la función *ch_dtarch* explicada anteriormente, con el objetivo de detectar los computadores que tengan la misma arquitectura que la local.

Vale comentar que este comando se ejecuta automáticamente cuando se inicia la consola por primera vez, así como cuando se lo invoca desde la Consola MPI.

reset: el comando '*reset*' mata todos los demonios y borra todos los archivos temporarios, para ello utiliza el comando de MPD llamado *mpdcleanup* ejecutando éste en cada computador mediante el comando *rsh* de UNIX. De esta forma nos aseguramos que no queden restos de algún proceso anterior que impida o traiga problemas en ejecuciones futuras.

run: este comando implementado mediante la función *chd_run* permite ejecutar programas MPI desde consola, para ello se han previsto aquí dos alternativas válidas. La primera y más simple es la de pasar como único argumento del comando, el nombre del programa a ejecutar (previamente compilado); de esta forma se utilizará un parámetro de ejecución por defecto que es el número de procesos a crear, tomándose éste igual a la cantidad de computadores que se van a utilizar; pero la segunda alternativa permite indicar éste número de procesos, de forma tal que podemos indicar mas procesos que computadores, ésta segunda alternativa es opcional pero así permitimos cierta flexibilidad a la hora de ejecutar programas sobre la Consola MPI.

setenv: este comando tiene la misma función que el del otro dispositivo.

test: permite comprobar la intercomunicación del anillo formado, para ello se utilizó el comando *mpdringtest* de MPD, el cual envía un mensaje a través del anillo una cantidad de veces igual al argumento que espera el comando '*test*'. Lo único que se hizo a continuación es filtrar la salida, para tratar de uniformar las salidas con otras implementaciones.

version: aquí se utiliza una función igual a la de la implementación MPICH con dispositivo ch_p4.

- **Para LAM**

add: el comando '*add*' como los anteriores es el encargado de agregar computadores al sistema paralelo, para realizar esto, verifica si existe el archivo *pgtmp_lam* y si el comando tiene algún argumento, en caso afirmativo se comprueba si el computador a agregar existe en el archivo *pgtmp_lam*, si es así significa que ya fue agregado y se mostrará el siguiente mensaje:

```
ERROR – 'nombre_computador' ya fue agregado.
```

Una precaución que se ha tenido en cuenta es la de comprobar si hay nodos inválidos en el anillo, es decir si se han quitado computadores que no hayan sido los últimamente agregados; entonces se procederá a ocupar ese lugar con el computador que se quiere agregar; de esta forma no se creará un nuevo nodo si existe un lugar vacío antes.

También se comprueba si el computador a agregar está en el archivo *pgfile_lam* creado por el comando '*recon*'; en caso afirmativo (que significa disponible) se indicará en pantalla:

```
'nombre_computador' será agregado al anillo.
```

Y aquí se invocará el comando *lamgrow* de LAM, el cual permite agregar un computador al sistema paralelo, luego de ejecutar este comando lo que se hace es añadir el computador agregado al archivo *pgtmp_lam* en el cual se mantendrá un registro de los computadores que conforman el anillo, por último se cuenta el número de computadores (o líneas) del archivo *pgtmp_lam* y se almacena éste en el archivo *n_proc.*

Como comentario final aclaramos que todo mensaje de error que se produzca se guardará en un archivo llamado ERROR.

avm: el comando '*avm*' sirve para visualizar los computadores disponibles a ser utilizados por esta implementación, para esto comprueba si el archivo *pgfile_lam*, (generado automáticamente por el comando '*recon*') existe y tiene un tamaño mayor que cero. En caso afirmativo se edita mostrando tales computadores, y de lo contrario se indica el mensaje:

```
No hay computadores disponibles.
```

comp: este comando, como en las demás implementaciones, es el encargado de compilar programas fuentes escritos en "C", y funciona de igual manera que los anteriores, pero con la diferencia que en este caso se utiliza el comando '*mpicc*' proporcionado por LAM.

conf: el comando '*conf*' es el encargado de mostrar los computadores agregados al sistema paralelo, para ello analiza cuantos computadores tiene el anillo, verificando el contenido del archivo *n_proc*; si el contenido es 0 o está vacío, entonces se muestra en pantalla el siguiente mensaje:

```
El anillo no tiene ningún computador.
```

De lo contrario, se indican cuantos computadores lo forman, y utilizando el comando *lamnodes* de LAM, se muestra la lista de los nombres de cada computador, como se ve a continuación:

```
El anillo tiene  3 computador(es):

n0      cetad
n1      sofia
n2      paris
```

delete: este comando en primer lugar verifica si tiene o no argumentos, si ellos se han indicado, son analizados uno por uno; de lo contrario se indica el siguiente mensaje:

```
Falta el nombre del o los computador(es).
```

Antes de procesar cada argumento se verifica si existe el archivo *pgtmp_lam,* pues de lo contrario no existirían computadores para eliminar; en caso afirmativo se realiza una verificación para saber si cada argumento corresponde a un computador existente o no*,* entonces se procede a quitar tal computador del anillo usando el comando *lamshrink* con ciertas precauciones, para que automáticamente se detecte el número de nodo correspondiente, puesto que dicho dato es necesario para este comando.

Cualquier error es guardado en un archivo ERROR, para no complicar la salida del comando en consola.

Una vez eliminado el computador del anillo, se procede a borrarlo del archivo *pgtmp_lam,* realizándose luego un recuento de los computadores que quedan, y almacenando este valor en el archivo *n_proc*.

Una consideración que se ha hecho es la de avisar cuando se elimina el computador local, puesto que se mata el demonio local y se rompe el anillo, en cuyo caso se deberá reiniciar la consola.

Finalmente, se comprueba si el computador eliminado fue el último, es decir, si el archivo *pgtmp_lam* queda vacío; si es así, se muestra un mensaje indicando esto y se elimina el archivo vacío.

halt: el comando '*halt*' además de terminar el programa *mpi* que es la consola en sí, mata todos los demonios, utilizando el comando de LAM llamado *wipe*; luego lo que hace es eliminar todos los archivos temporarios.

help: funciona exactamente igual a los comandos '*help*' anteriores con la única diferencia que se han distinguido algunas variantes en la ayuda de ciertos comandos; éste está desarrollado a través de la función *lam_help*.

jobs: el comando '*jobs*' lista los trabajos que se están ejecutando utilizando el comando *state* de LAM, la única precaución que se ha tenido en cuenta a la hora de desarrollarlo es comprobar si se están ejecutando o no trabajos, e indicar en cada caso la salida correspondiente por pantalla.

kill: este comando es el utilizado para matar cualquier proceso de LAM, para lo cual hace uso del comando proporcionado por LAM llamado *doom,* que toma como argumento el PID del proceso a matar, el que se obtiene con '*jobs*'.

Para verificar si efectivamente se ha matado un proceso o no, se analiza la salida del comando *doom*, que se redirige al archivo *killjobs;* éste posteriormente se examina para ver si está vacío o no, pues vacío significa que no se ha matado nada, de lo contrario se guarda el proceso eliminado.

quit: '*quit*' es el comando que permite salir de la Consola MPI sin matar los demonios, para esto al igual que los "quit" anteriores solo se usa para terminar el programa *mpi*.

recon: el comando '*recon*' se utiliza como en los anteriores para detectar qué computadores de los que están en la red, es decir en el archivo *allred,* están disponibles o no; para esto analiza cada computador del *allred* y si se halla el ejecutable *lamboot* de LAM en el lugar adecuado, se marca como "**Disponible**" y se agrega en el archivo llamado *pgfile_lam*, que es utilizado por los demás comandos; de lo contrario se marca como "**No disponible**".

Aquí no es necesario el reconocimiento de cada arquitectura, puesto que LAM se puede utilizar para una red heterogénea, sin la necesidad de especificar la arquitectura de cada computador.

reset: este comando es el encargado de limpiar todos los procesos, temporarios y demás cosas que puedan perjudicar una futura ejecución, está implementado haciendo uso del comando *lamclean* de LAM, que se encarga de todo.

run: el comando '*run*' como en los casos anteriores se utiliza para ejecutar programas de LAM; en esta oportunidad se hace uso del comando *mpirun* de LAM, pudiéndose especificar el número de procesos a formar, siendo esto opcional, ya que solo es suficiente como parámetro el nombre del ejecutable, con lo cual se generará por defecto un proceso en cada computador agregado al sistema paralelo.

setenv: este comando es idéntico a los anteriores del mismo nombre.

test: Utiliza el comando *tping* de LAM, el cual envía un mensaje alrededor del anillo, con la cantidad de bytes que se le especifiquen; este es el único argumento necesario para el comando '*test*'.

version: este comando llama a un pequeño programa, el cual hace uso de las librerías de LAM, para hallar e imprimir su versión.

5.4. Ventajas que brinda el uso de la Consola MPI

La **Consola MPI** actúa como intérprete de comandos, permitiendo así al usuario una interfaz simple que le permita utilizar cada implementación, (MPICH con los dispositivos ch_p4 o ch_p4mpd y LAM) de modo tal, que no se tenga que cambiar de comandos para trabajar con cada una de ellas.

En la Consola MPI se proporciona una serie de comandos sumamente útiles para la configuración del sistema paralelo, la iniciación de procesos, y también para consultar en cualquier instante el estado de los mismos; además agregamos comandos que nos parecieron útiles a la hora de trabajar con

procesamiento paralelo, tales como: *avm*, *comp*, *recon*, *run* y *test*, los que ya se han explicado y detallado anteriormente, y que permiten averiguar que computadores están disponibles en la red para utilizarlos (*recon*), realizar pruebas con los computadores ya agregados, verificando la correcta comunicación entre ellos (*test*), como así también, permitir al usuario de la Consola MPI, compilar los programas escritos en "C", sin salir de la misma, permitiéndole si lo desea, hacer este trabajo en todos los computadores disponibles (*comp -a*), y también permitir desde la consola la ejecución de los programas compilados (*run*).

Por todo lo mencionado, comprobamos las vantajas del uso de la Consola MPI; y entre todas ellas vale destacar, la uniformidad de los comandos, aunque se estén utilizando distintas implementaciones en la consola, lo que permite al usuario familiarizarse más rápidamente con ellos, ya que manejar cada implementación por separado, y fuera de la consola, exige utilizar diferentes comandos para cada una de ellas.

Una dificultad solucionada, es la existencia de comandos que son iguales (en las distintas implementaciones), pero lógicamente no funcionan de igual forma, entre ellos podemos mencionar: *mpicc*, *mpirun*, etc., lo que se ha resuelto en la Consola MPI, llamando directamente al ejecutable e indicándole la ruta completa; de esta forma, también evitamos al usuario tener que definir todos los PATH necesarios para el uso de cada implementación.

Apéndice A

A. SISTEMA OPERATIVO UNIX

A.1. Introducción

Hoy en día, desde los microcomputadores con recursos limitados, a los maxicomputadores, de grandes prestaciones, utilizan el sistema UNIX. Este crecimiento se está acelerando cada vez más, conforme más y más usuarios se habitúan a la sorprendente flexibilidad, potencia y elegancia del sistema.

- *Portabilidad:* proporciona el entorno requerido para permitir el fácil traslado de las aplicaciones desde los microcomputadores a los maxicomputadores.
- *Flexibilidad:* El sistema ha sido adaptado a aplicaciones tan divergentes como la automatización de fábricas, los sistemas de conmutación telefónica y los juegos.
- *Potencia:* Como la sintaxis de ordenes es clara y concisa, permite a los usuarios hacer muchas cosas rápida y sencillamente, cosas que ni siquiera son posibles con otros sistemas operativos.
- *Multiusuario y multitarea:* Debido a que UNIX es un entorno multitarea de tiempo compartido, puede hacer más de una cosa a la vez fácilmente. En un sistema UNIX personal un usuario puede estar editando un archivo, enviando un correo electrónico a otro computador, imprimiendo otro archivo, simultáneamente. Está diseñado para manejar sin esfuerzo las necesidades múltiples y simultáneas de su usuario. También es un entorno multiusuario, que soporta las actividades de mas de una persona a la vez.
- *Elegancia:* Una vez que los usuarios comprenden algunos de los conceptos básicos del sistema, pueden realizar muchas y grandes tareas de un modo sencillo y hermoso.

Estos factores dejan claro que el sistema continuará creciendo y desarrollándose, y la mayoría de los usuarios llegarán a disponer de su propio sistema UNIX sobre su microcomputador personal.

A.2. El saber del sistema UNIX

UNIX esta constituido por un núcleo ("kernel"), que es el corazón del sistema operativo, el sistema de archivos, un método jerárquico de directorios para la organización de archivos en disco, y el "shell" o cápsula, la interfaz de usuario que provee la forma en que el usuario comanda el sistema.

Este sistema permite a los usuarios invocar a un computador remoto, esto hace que el sistema sea ideal para aplicaciones de servidor, en las cuales la interfaz primaria de usuario no está asociada con el computador UNIX centralizado, sino que se ejecuta directamente sobre un terminal “inteligente” remoto. Se trata de una arquitectura moderna aplicada a entornos de red de área local en oficinas comerciales de tamaño pequeño a moderado. El nuevo

sistema de ventanas X es un buen ejemplo del uso de terminales inteligentes con servidores centralizados. El sistema UNIX va contra la tendencia actual de hacer los sistemas operativos “invisibles” al usuario. La interfaz de usuario orientada a ventanas y menús ayuda a ocultar las órdenes, sistemas de archivos y herramientas administrativas del usuario. Al utilizar el sistema UNIX, sin embargo, mientras mas concientes sean de las funciones internas del sistema, mejor pueden controlarlo para su propio beneficio y para mejorar su productividad. La implementación original fue codificada en lenguaje ensamblador, pero pronto se desarrolló el lenguaje de programación C dentro del grupo, comenzando en 1971. El lenguaje C fue utilizado casi inmediatamente en la continuación del desarrollo del sistema UNIX, y en 1973 el núcleo se recodificó en C. Hoy solo unas cuantas subrutinas del núcleo de alto rendimiento están escritas en lenguaje ensamblador. Este fue el primer intento de codificar un sistema operativo entero en un lenguaje de alto nivel y la portabilidad que se consiguió está ampliamente considerada como una de las razones principales de la popularidad que el sistema UNIX goza hoy.

El sistema UNIX puede soportar cualquier lenguaje que tenga un compilador o intérprete y una interfase de sistema que defina las peticiones del usuario de los servicios del sistema operativo.

Felizmente, las versiones recientes han sido destinadas al usuario menos experimentado, sin reducir la potencia del sistema ni sus otras ventajas. Sus mejoras han ocurrido principalmente en las siguientes áreas:

- *Robustez:* el sistema ha sido endurecido de modo que se requiere muy poco mantenimiento de software para mantenerlo ajustado y operando a rendimiento máximo. Muchas tareas rutinarias del administrador, tales como, el arranque del sistema, se han hecho automáticas.
- *Consistencia:* casi todas las órdenes han evolucionado a lo largo de los años para adoptar una sintaxis mas consistente, de modo que la utilización confusa o inconsciente de diferentes órdenes se ha reducido.
- *Documentación:* el Manual del usuario de UNIX, aún cuando sigue siendo un material de referencia conciso, ha evolucionado para ser mas consistente, con mas ejemplos de utilización de órdenes y mas material explicativo que en versiones mas antiguas.
- *Agentes de usuario:* la mayoría de las implementaciones proporcionan ahora herramientas simplificadas para ayudar a la configuración y administración del sistema.

A.3. Componentes del sistema

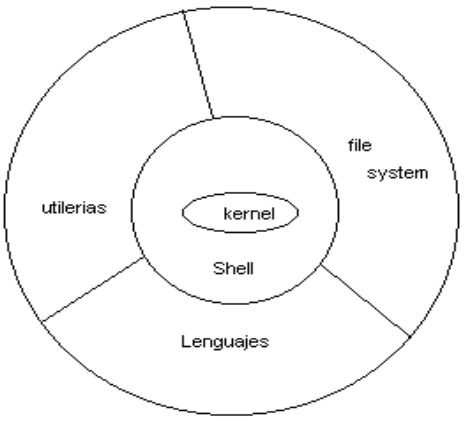


Figura 1.2 - Arquitectura del sistema UNIX.

- **Kernel:** Es la parte central del sistema operativo, en el se encuentra la mayor parte del código de la dependencia del hardware de SO, que supervisa a todos los dispositivos, comunicación, controla la ejecución de procesos y asigna memoria.
- **Shell:** Es la interfase entre el usuario y kernel, también funciona como un interpretador de comandos es decir, es el responsable de interpretar y ejecutarlos.
- **File System:** El sistema de archivos es el encargado de manipular las estructuras de los dispositivos de almacenamiento. Es el medio por el cual organiza todos los datos desde la raíz (Root) que se simboliza por una /. Un sistema de archivos generalmente contiene:
  - Método de acceso
  - Manejo de archivos
  - Manejo de almacenamiento
  - Mecanismos de integridad de archivos
- **Lenguajes:** Todos los sistemas tienen lenguajes de programación, como C, Pascal, Fortran, clipper, etc.
- **Utilerías:** Estas son las que ayudan en el manejo de los dispositivos, o software como: gráficas, calendario, juegos, etc.

A.4. Tipos de Shell

Algunos de los tipos de shell mas comunes son:

- **Bourne Again Shell** (/usr/bin/bash): Utilizado por los sistemas Linux
- **Bourne Shell** (/sbin/sh): Escrito por Steve Bourne de los laboratorios Bell, este es el Shell por default por la mayoría de los sistemas UNIX.
- **C Shell** (/usr/bin/csh): Desarrollado por Bill Joe de Sun Micro Systems, tiene una sintaxis similar a la del lenguaje C.
- **Korn Shell** (/usr/bin/ksh): Escrito por David Bourne de los laboratorios Bell. Provee una serie de herramientas que se agregan a las proporcionadas por el Bourne Shell, además de incluir varias características del C Shell.

A.5. Programación Shell

Un programa en shell es un conjunto de comandos de UNIX agrupados dentro de un archivo. Este archivo debe contar con permisos de lectura y ejecución (rx). Para ser invocado, solo se debe escribir su nombre en el prompt del shell y oprimir la tecla ENTER o RETURN. Comúnmente, los programas en shell se utilizan para automatizar tareas rutinarias dentro del sistema.

- **Parámetros posicionales**

  Al iniciar el programa shell, se crean variables referidas como parámetros posicionales. El nombre del programa es el parámetro posicional $0, el primer argumento de la línea de comandos es $1, el segundo es $2 y así hasta el noveno parámetro posicional que es $9.

- **Mecanismos de entrecomillado**

  Los caracteres <, >, *, ¿, [y] tienen un significado especial para el shell. Para quitar dicho significado, se requiere de algún mecanismo de entrecomillado:

  - Las comillas sencillas (‘):Las comillas sencillas le quitan el significado a todos los caracteres especiales que se encuentran dentro de ellas.
  - Las comillas dobles (“): Las comillas dobles le quitan el significado especial a todos los caracteres especiales que se encuentran dentro de ellos, con la excepción de los caracteres **$**, \, **‘**, **{**, **}** y **“**.
  - La diagonal invertida (\): La diagonal invertida le quita el significado especial a cualquier carácter que le preceda.

- **Variables del Shell**

Una variable es un nombre que representa un valor. El shell tiene varios tipos de variables que son:

- Variables definidas como parámetros posicionales.
- Variables definidas por el usuario.
- Variables del shell.

- **Variables de entorno definidas por el usuario**

El shell también reconoce variables alfanuméricas, a las cuales un valor de cadena puede ser asignado, indicándolo entre comillas dobles y no deben existir espacios a ambos lados del signo igual (=).

Más de una asignación puede aparecer en una sentencia, tomando en cuenta que las asignaciones se ejecutan de derecha a izquierda.

Para concatenar el valor de una variable se puede encerrar el nombre de la variable entre llaves ({ }) y anteponiendo el signo pesos ($). Ejemplo:

```
$ agp=’esto es un encadena’
$ echo “${agp}miento de cadenas”
esto es un encadenamiento de cadenas
$
```

- **Variables de entorno definidas por el Shell**

Las siguientes variables son definidas por el shell y todas pueden ser accedidas y modificadas por el usuario:

- **HOME:** Especifica el nombre del directorio del inicio de sesión del usuario.
- **MAIL:** Es la variable que almacena el nombre de ruta del archivo, donde el correo electrónico es depositado: además el shell revisa periódicamente el arribo de nuevo correo y si este es encontrado, entonces el shell envía el mensaje “You have new mail”.

- **PATH:** Es la variable que especifica la ruta usada por el shell en la búsqueda de comandos: esta es una lista ordenada de nombres de ruta de directorios, donde se encuentran los comandos a ser ejecutados y esta lista debe ser separada por dos puntos (:), como ser, el shell que inicializa la variable PATH con la siguiente lista:

  PATH=*/usr/bin: /bin:.*

En este caso la trayectoria de búsqueda será en primer término el directorio /usr/bin, posteriormente el directorio /bin y después el directorio del usuario.

- **PS1:** PS1 es la variable que especifica el prompt primario cuyo valor por default es ($) seguido por un espacio. El valor de esta variable puede ser cambiado de la siguiente manera:

  $ PS1:"cetad>".
  cetad>

Donde ***cetad>*** es el nuevo prompt primario.

- **El carácter numeral (#):**

Los comentarios se hacen con el carácter numeral (#). Todo el texto que se encuentra desde el inicio del numeral, hasta el final de la línea, no se interpreta por el shell.

A.6. Editor vi

Una vez que se ha entrado al sistema, una de las tareas más comunes es la de crear archivos. Un editor de texto es un programa utilizado para almacenar y manipular información dentro del computador. La mayor parte de los sistemas UNIX poseen un editor de pantalla. Dos de los más populares son el *vi* y el *emacs*. El que se ha utilizado para la creación de los archivos que constituyen la parte práctica del proyecto, es el *vi*. Como todo editor de pantallas, el vi utiliza un protocolo especial para moverse por la pantalla, borrar, anexar, etc. Para invocarlo se teclea:

vi archivo

Que edita el archivo con nombre archivo. Si ya existiera, en pantalla aparecería su contenido actual. Una vez que se ha entrado, el entorno es diferente al ofrecido por el programa shell. Se podrán escribir textos, moverlos, realizar cambios, etc. Para salir del editor hay varias opciones. Una es introduciendo el comando:

ZZ ó :wq Enter → El archivo se grabará, y posteriormente se saldrá del *vi*.

:q! Enter → Si no se quieren grabar los cambios al salir

:w Enter → Si se quiere grabar los datos sin salir del editor

Algo muy importante que hay que tener en cuenta, es que el editor ofrece dos modos de operación. Cuando se accede al *vi*, se está en modo comando. Para insertar texto en el archivo, habrá que pasar al modo texto, introduciendo el comando adecuando (i, a, o). Cuando se finaliza con la inserción del texto, o quiere introducirse algún comando, mediante la tecla ESC, se pasará al modo comando.

A.7. Archivos de configuración y confiabilidad del sistema

.profile, .cshrc y .bash_profile (para los shells Bourne, C y Bourneagain respectivamente): Son archivos de configuración del sistema que normalmente cada usuario crea y mantiene individualmente; estos se ejecutan por el sistema durante la presentación inicial. Se utilizan generalmente para configurar las sesiones de presentación de acuerdo con las preferencias del usuario y para crear algunas de las variables del entorno que las ordenes esperan encontrar. Estos archivos **.profile .cshrc y .bash_profile** serán localizados siempre en el directorio HOME de cada usuario. Para visualizarlos se hace mediante la orden **ls -a**.

Seguidamente definimos algunos archivos de configuración y confiabilidad del sistema:

.profile: Contiene las variables de entorno de cada uno de los usuarios del sistema. (Es un archivo oculto de UNIX)

.rhosts: Es un archivo de confiabilidad donde se indican los nombres de los hosts con sus correspondientes nombres de usuarios, y permite ingresar a el computador donde se encuentra este archivo, a todos los usuarios de los hosts que se detallan en el mismo sin necesidad de usar una contraseña. (Es un archivo oculto de UNIX)

hosts.equiv: Este esta ubicado en el directorio /etc. y es igual a **.rhosts**. Este lo debe crear el administrador del sistema a nivel de **root**.

A.8. Descripción de comandos UNIX

La sintaxis general de los comandos es la que sigue:

comando [-opciones] [argumentos] [archivos]

" Todos los comandos sin excepción son con minúsculas"

Opciones: Estas opciones van precedidas por un (-) al inicio de las letras.

Argumentos: Proveen información adicional de la ayuda de este comando.

Archivos: Son los que se especifican en los nombres de los archivos requeridos por el comando (archivos en los cuales se quiere que tengan efecto los comandos).

- **Presentación login:** Una vez que su terminal esté conectado a el computador UNIX, puede tener que presionar uno o dos ENTER para despertar el sistema UNIX. Cuando lo haga, vera el "introductor de presentación": [login prompt]. Debido a que UNIX es un sistema multiusuario, su primera tarea es identificarse por medio de un id de

identificación (debe ser un nombre de usuario único, que no exceda los 8 caracteres y conste solo de letras y números; con el cual lo conocerá el sistema, le da acceso a sus archivos y establece su sesión de trabajo). En este sistema se distinguen mayúsculas de minúsculas, con lo cual no será lo mismo "nombre_usuario" que "NOMBRE_USUARIO" y deberá tener un carácter minúscula al principio. Por lo tanto el sistema le solicita su identificación con el comando: **login:** nombre_usuario y finaliza presionando ENTER. Luego de haber introducido su nombre, el sistema le solicitará una contraseña individual para demostrarle que es la persona autorizada para usar ese nombre de usuario (así protegerá sus archivos y otros datos privados frente a los demás) se la solicita con el comando **password:** y no muestra los caracteres de su contraseña. Si alguno de estos datos que introduce están incorrectos proporcionará una segunda oportunidad para su ingreso, y cuando ambos datos sean aceptados, verá algunos rótulos y mensajes iniciales indicándole un sistema UNIX multiusuario activo. Finalmente, el proceso de presentación se completa, y el sistema le devuelve el control imprimiendo el inductor **"$"** luego de ello el sistema queda a la espera de órdenes, es decir, ya puede comenzar a hablar con el procesador de órdenes: el **shell**.

- **Listar archivos:** tras la presentación el usuario se encontrará en un lugar específico dentro del sistema llamado su directorio propio [**home**], donde podrá examinar los archivos que existen en su propia área privada de trabajo de almacenamiento, por medio de la orden **ls** y una vez ejecutada la orden mostrará sus archivos nota y README y luego le devuelve el control al shell para la siguiente orden.

```
$ls
nota
README
$
```

Realmente, como la mayoría de las órdenes, tienen varias opciones, que se indican por medio de **argumentos** (llamados **opciones**) precedidos del signo **"-"**. Por ejemplo podría pedirle su lista de archivos a lo **"largo"** con la opción **"-l"** que da mayor información si le indica:

```
$ls –l
total 2
-rw-rw-rw-  1  nombre_usuario  users   138  Jul  8 19:34  README
-rwxrw-rw-  1  nombre_usuario  users   227  Apr 18 12:30  nota
$
```

Lo primero que aparece a la izquierda **"-rwxrw-rw-"**, indica los modos y permisos del archivo, mientras que **nombre_usuario** es el propietario del archivo, que pertenece al grupo "**users**". A continuación el **tamaño en bytes** del archivo, la fecha de creación y/o última modificación, y lo último de la línea es el **nombre del archivo**.

- **Creación de una cuenta de usuario:** En todos los casos debe hacerlo el administrador del sistema, a nivel de root y dependiendo de la plataforma, se utilizan los siguientes comandos:

  - Para las estaciones con plataforma UNIX:

  Hardware: SUN-SPARC5 Software: SUN OS 4.1
  Comando:
  add_user *login* 999 10 "*Nombre Usuario*" */home_de_usuario/bin/csh*

  Con este comando ya es suficiente para crear una cuenta de usuario, donde *login* es el identificador que el usuario deberá escoger, y *Nombre Usuario* es el nombre real del usuario, a los fines de identificación.

  Hardware: SUN-SPARC4 Software: SOLARIS2 V5.1
  Comando: **admintool &**

  Este comando abrirá una ventana solicitando ciertos datos que el administrador de red deberá completar de acuerdo con el usuario.

  Hardware: IBM-RS6000 Software: AIX
  Comando: **similar al anterior.**

  - Para las estaciones con plataforma LINUX:

  Hardware: PC-IBM-80586 Software: LINUX
  Comando: **adduser** -d *home/nombre_usuario* -r *nombre_usuario*

  Esta orden crea una cuenta de usuario, donde se especifica el directorio HOME del usuario y el login o nombre de usuario en *nombre_usuario.*

A.9. Lista alfabética de comandos mas utilizados en UNIX

cat: permite ver archivos, crearlos ó bien podemos concatenar archivos, solo de lectura.
cd /: (*change directory*), se utiliza para ir al directorio raíz.
cd ..: (*change directory*), se utiliza para salir del directorio actual, al nivel anterior.
cd *nombre*: (*change directory*), se utiliza para moverse de un directorio a otro llamado *nombre*, si está comprendido en el actual.
chmod: (change mode), concede permisos de lectura, escritura y ejecución sobre archivos y directorios a todos los usuarios; dichos permisos se pueden cambiar y los tipos de permisos son: **r:** permiso de lectura, **w:** permiso de escritura y **x:** permiso de ejecución.
chown: se utiliza para cambiar los permisos de un archivo o de un directorio.
chsh: me permite cambiar el shell, por otro disponible.
cmp: compara dos archivos que le indicamos.
cp: se utiliza para copiar archivos.
exit: se utiliza para salir de un host en el que entré en forma remota o para desloguearme.

find: busca los archivos que satisfacen la condición señalada a partir del directorio indicado.
hostname: se utiliza para conocer el nombre del host en el que estoy trabajando.
login: entra a un subdirectorio del directorio raíz (root), llamado *home directory,* que es donde van a residir los archivos del usuario.
logout: termina una sesión en terminal UNIX.
ls: lista el contenido del directorio en que nos encontramos (solo los nombres de los archivos).
ls -a: lista los archivos llamados ocultos o invisibles, cuyos nombres comienzan con punto (.), normalmente se los utilizan para almacenar información que el sistema utiliza automáticamente.
man *comando***:** visualiza las páginas de la ayuda de *comando*.
.*/programa***:** ejecuta el programa en el directorio actual.
mkdir: (*make directory)* se utiliza para crear nuevos directorios.
more: se utiliza para ver el contenido completo de un archivo por página, solo de lectura.
mount: se utiliza para montar un sistema de archivos en un punto de montaje especifico.
mv: se usa para renombrar ó mover un archivo ó directorio a otra dirección.
passwd: cambia nuestra clave de acceso
pwd: nos dice en que directorio estamos posicionados, lo que permite conocer el directorio activo ó de trabajo.
rcp: copia archivos de un host a otro, en forma remota.
rlogin: para acceder a otro host en forma remota.
rm: se utiliza para borrar archivos.
rmdir *nombre*: (*remove directory*) se utiliza para borrar un directorio especificado (*nombre*), pero este no debe contener información (es decir, debe estar vacío).
rusers: permite saber que usuarios están trabajando en ese momento en el sistema, y el computador que están utilizando.
sed: (string editor) reemplaza una cadena de caracteres indicados por otra mencionada.
su: para pasar a modo root desde el modo usuario.
tail: escribe las últimas líneas de un archivo.
tar: (tape archiver) agrupa archivos/directorios en un único archivo.
umount: desmonta un sistema de archivos de un punto especifico.
vi: editor de archivos o programas, y permite modificarlos.
who: muestra una lista de los usuarios que están conectados al sistema.
whoami: informa el usuario actual que esta logueado.
wc: cuenta los caracteres ó palabras de un archivo de texto.

Apéndice B

B. LENGUAJE DE PROGRAMACION 'C'

B.1. Introducción

C es un lenguaje de programación de propósito general que ofrece economía sintáctica, control de flujo y estructuras sencillas y un buen conjunto de operadores. No es un lenguaje de muy alto nivel y más bien un lenguaje pequeño, sencillo y no está especializado en ningún tipo de aplicación. Esto lo hace un lenguaje potente, con un campo de aplicación ilimitado y sobre todo, se aprende rápidamente. En poco tiempo, un programador puede utilizar la totalidad del lenguaje.

Este lenguaje ha sido estrechamente ligado al sistema operativo UNIX, puesto que fueron desarrollados conjuntamente. Sin embargo, este lenguaje no está ligado a ningún sistema operativo ni a ningún computador concreto. Se le suele llamar *lenguaje de programación de sistemas* debido a su utilidad para escribir compiladores y sistemas operativos, aunque de igual forma se pueden desarrollar cualquier tipo de aplicación.

Una aportación muy importante de ANSI consiste en la definición de un conjunto de librerías que acompañan al compilador y de las funciones contenidas en ellas. Muchas de las operaciones comunes con el sistema operativo se realizan a través de estas funciones. Una colección de archivos de encabezamiento, *headers*, en los que se definen los tipos de datos y funciones incluidas en cada librería. Los programas que utilizan estas bibliotecas para interactuar con el sistema operativo obtendrán un comportamiento equivalente en otro sistema.

B.2. Conceptos básicos - Estructura de un programa en C

Todo programa en C consta de uno o mas módulos llamados *funciones.* Una de las funciones se llama **main**. El programa siempre comenzará por la ejecución de la función **main**, la cual puede acceder a las demás funciones. Las definiciones de las funciones adicionales se deben realizar aparte, precediendo o siguiendo a **main.**

Cada función debe contener:

- Una *cabecera* de la función, que consta del nombre de la función, seguido de una lista opcional de argumentos encerrados entre paréntesis.
- Una lista de *declaración* de argumentos, si se incluyen estos en la cabecera.
- Una *instrucción compuesta,* que contiene el resto de la función.

Los argumentos son símbolos que representan información que se le pasa a la función desde otra parte del programa, también se llaman *parámetros*.

Cada instrucción compuesta se encierra con un par de llaves **{ }**. Las llaves pueden contener combinaciones de instrucciones elementales (denominadas *instrucciones de expresión*) y otras compuestas. Así las instrucciones compuestas pueden estar anidadas, una dentro de otra, cada instrucción elemental debe terminar en punto y coma (**;**).

Los *comentarios* pueden aparecer en cualquier parte del programa, mientras estén situados entre los delimitadores ***/**** y ****/*** (por ejemplo: */** Este es un comentario **/*). Los comentarios son útiles para indicar los elementos principales de un programa o para aplicar la lógica subyacente de estos.

La mejor forma de aprender un lenguaje es programando con él.

Un programa sencillo, es el siguiente:

```
#include <stdio.h>

main()
{
   printf("Hola amigos!\n");
}
```

Con el visualizamos el mensaje Hola amigos! en el terminal. En la primera línea indica que se tengan en cuenta las funciones y tipos definidos en la librería stdio (*standard input/output*). Estas definiciones se encuentran en el archivo *header* stdio.h. Ahora, en la función main se incluye una única sentencia que llama a la función printf. Esta toma como argumento una cadena de caracteres, que se imprimen van encerradas entre dobles comillas **" "**. El símbolo **\n** indica un cambio de línea. Hay un grupo de símbolos, que son tratados como caracteres individuales.

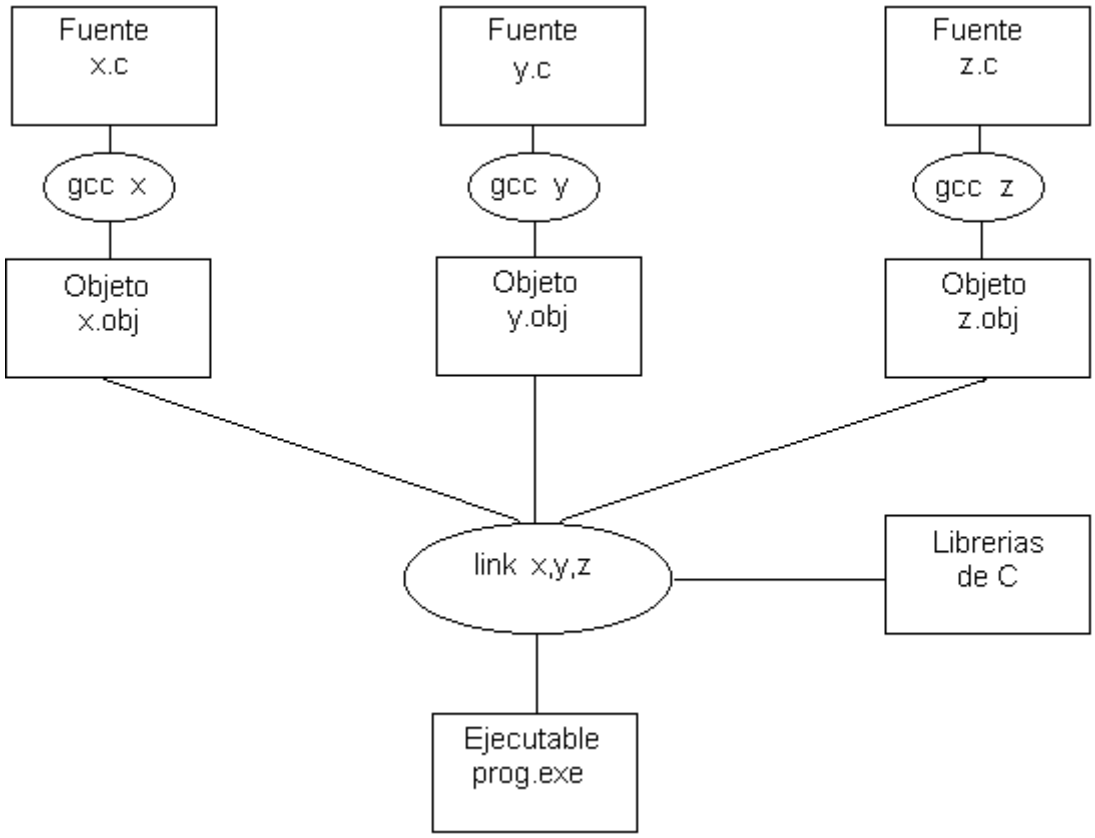
Solo se expone una visión general de las características básicas de la mayoría de los programas en C.

B.3. El desarrollo de un programa

Un programa C puede estar formado por diferentes módulos o fuentes. Es conveniente mantener los fuentes de un tamaño no muy grande, para que la compilación sea rápida. También, al dividirse un programa en partes, puede facilitar la legibilidad del programa y su estructuración. Los diferentes fuentes son compilados de forma separada, únicamente los fuentes que han sido modificados desde la última compilación, y después combinados con las librerías necesarias para formar el programa en su versión ejecutable.

Los comandos necesarios para compilar, *linkar* y ejecutar un programa dependen del sistema operativo y debemos dirigirnos a los manuales correspondientes para conocer la sintaxis exacta. En nuestro caso utilizamos los siguientes comandos:

```
gcc prog
gcc modulo1, modulo2
link prog, modulo1, modulo2
prog
```



FiguraB.1 - Creación de un programa en C.

B.4. Funciones

Un programa C está formado por un conjunto de funciones que al menos contiene la función main. Una función se declara con el nombre de la función precedido del tipo de valor que retorna y una lista de argumentos encerrados entre paréntesis. El cuerpo de la función está formado por un conjunto de declaraciones y de sentencias comprendidas entre llaves. Veamos un ejemplo de utilización de funciones:

```
#include <stdio.h>
#define VALOR 5
#define FACT        120

int fact_i ( int v )
{
    int r = 1, i = 0;
    while ( i <= v ) {
                    r = r * i;
                    i = i + 1;
                    }
    return r;
}

int fact_r ( int v )
```

```
{
    if ( v == 0 )
        return 1;
    else
        return v * fact_r(v-1);
}

main()
{
    int  r, valor = VALOR;
    if ( (r = fact_i(valor)) != fact_r(valor) )
        printf("Codificación errónea!!.\n");
    else
        if ( r == FACT )
            printf("Codificación correcta.\n");
        else
            printf("Algo falla!!.\n");
}
```

Se definen dos funciones, fact_i y fact_r, además de la función main. Ambas toman como parámetro un valor entero y devuelven otro entero. La primera calcula el factorial de un número de forma iterativa, mientras que la segunda hace lo mismo de forma recursiva.

Todas las líneas que comienzan con el símbolo # indican una directiva del precompilador. Antes de realizar la compilación en C se llama a un precompilador cuya misión es procesar el texto y realizar ciertas sustituciones textuales. Hemos visto que la directiva #include incluye el texto contenido en un archivo en el fuente que estamos compilando. De forma parecida, #define *nombre texto* sustituye todas las apariciones de *nombre* por *texto*. Así, en el fuente, la palabra VALOR se sustituye por el número 5.

El valor que debe devolver una función se indica con la palabra return. La evaluación de la expresión debe dar un valor del mismo tipo de dato que el que se ha definido como resultado. La declaración de una variable puede incluir una inicialización en la misma declaración. Se debe tener muy en cuenta que en C todos los argumentos son pasados 'por valor'. No existe el concepto de paso de parámetros 'por variable' o 'por referencia'. Veamos un ejemplo:

```
int incr ( int v )
{
    return v + 1;
}

main()
{
    int  a, b;
    b = 3;
    a = incr(b);

/*  a = 4 mientras que  b = 3, no ha cambiado después de la llamada  */
```

```
}
```

En el ejemplo anterior el valor del parámetro de la función **incr**, aunque se modifique dentro de la función, no cambia el valor de la variable b de la función main. Todo el texto comprendido entre los caracteres **/*** y ***/** son comentarios al programa y son ignorados por el compilador. En un fuente C los comentarios no se pueden anidar.

B.5. Control de flujo

La sentencia de control básica es **if** (<e>) **then** <s> **else** <t>. En ella se evalúa una expresión condicional y si se cumple, se ejecuta la sentencia *s*; si no, se ejecuta la sentencia *t*. La segunda parte de la condición, else <t>, es opcional.

```
int cero ( double a )
{
    if ( a == 0.0 )
        return (TRUE);
    else
        return (FALSE);
}
```

En el caso que <e> no sea una expresión condicional y sea aritmética, se considera falso si vale 0; y si no, verdadero. Hay casos en los que se deben evaluar múltiples condiciones y únicamente se debe evaluar una de ellas. Se puede programar con un grupo de sentencias **if then else** anidadas, aunque ello puede ser engorroso y de complicada lectura. Para evitarlo nos puede ayudar la sentencia **switch**. Su utilización es:

```
switch (valor) {
    case    valor1: <sentencias>
    case    valor2: <sentencias>
    ...
    default:    <sentencias>

}
```

Cuando se encuentra una sentencia case que concuerda con el valor del switch se ejecutan las sentencias que le siguen y todas las demás a partir de ahí, a no ser que se introduzca una sentencia **break** para salir de la sentencia switch. Por ejemplo,

```
ver_opcion ( char c )
{
    switch(c)
        {
        case 'a':printf("Op A\n");    break;
        case 'b':printf("Op B\n");    break;
        case 'c':
```

```
        case 'd':printf("Op C o D\n");     break;
        default: printf("Op ?\n");
    }
}
```

Otras sentencias de control de flujo son las que nos permiten realizar iteraciones sobre un conjunto de sentencias. En C tenemos tres formas principales de realizar iteraciones. La sentencia **while** (<e>) <s> es seguramente la más utilizada. La sentencia, o grupo de sentencias <s> se ejecuta mientras la evaluación de la expresión <e> sea verdadera.

```
long raiz ( long valor )
{
    long     r = 1;

    while ( r * r <= valor )
        r++;

    return r;
}
```

Otra sentencia iterativa, que permite inicializar los controles del bucle es la sentencia **for** (<i>; <e>; <p>) <s>. La sentencia for se puede escribir también como:

```
<i>;
while ( <e> )
    {
    <s>;
    <p>;
    }
```

El ejemplo anterior se podría escribir como:

```
long raiz ( long valor )
{
    long     r;
    for ( r = 1; r * r <= valor; r++ );
    return r;
}
```

Una variación de la sentencia while es: **do** <s> **while** (<e>); En ella la sentencia se ejecuta al menos una vez, antes de que se evalúe la expresión condicional.

Otras sentencias interesantes, aunque menos utilizadas son **break** y **continue**. **break** provoca que se termine la ejecución de una iteración o para salir de la sentencia **switch**, como ya hemos visto. En cambio, **continue** provoca que se comience una nueva iteración, evaluándose la expresión de control.

Veamos dos ejemplos:

```
final_countdown ()
{
     int  count = 10;

     while ( count-- > 1 )
          {
          if ( count == 4 )
               start_engines();
          if ( status() == WARNING )
               break;
          printf("%d ", count );
          }
     if ( count == 0 )
          {
          launch();
          printf("Shuttle launched\n");
          }
     else
          {
          printf("WARNING condition received.\n");
          printf("Count held at T - %d\n", count );
          }
}

d2 ()

{
     int  f;

     for (f = 1; f <= 50; f++ )
          {
          if (f % 2 == 0 )
               continue;
          printf("%d", f );
          }
}
```

B.6. Funciones de entrada y salida por pantalla

En este apartado y los siguientes vamos a ver algunas de las funciones más importantes que nos proporcionan las librerías definidas por ANSI y su utilización. Como hemos visto hasta ahora, el lenguaje C no proporciona ningún mecanismo de comunicación ni con el usuario ni con el sistema operativo. Ello es realizado a través de las librerías.

El archivo de declaraciones que normalmente más se utiliza es el **stdio.h**. Vamos a ver algunas funciones definidas en él.

Una función que ya hemos utilizado y que, ella y sus variantes, es la más utilizada para la salida de información es **printf**. Esta permite formatear al enviar datos a la salida estándar del sistema operativo.

```
#include <stdio.h>

int printf ( const char *format [, argumentos, ...] );
```

Acepta un string de formato y cualquier número de argumentos. Estos argumentos se aplican a cada uno de los especificadores de formato contenidos en format. Un especificador de formato toma la forma:

%[flags][width][.prec][h|l] type.

Otra función similar a printf pero para la entrada de datos es scanf. Esta toma los datos de la entrada estándar del sistema operativo. En este caso, la lista de argumentos debe estar formada por punteros, que indican dónde depositar los valores.

```
#include <stdio.h>

int scanf ( const char *format [, argumentos, ...] );
```

Hay dos funciones que trabajan con **strings**. La primera lee un string de la entrada estándar y la segunda lo imprime en el dispositivo de salida estándar.

```
#include <stdio.h>

char *gets ( char *s );
int puts ( char *s );
```

También hay funciones de lectura y escritura de caracteres individuales.

```
#include <stdio.h>

int getchar ( void );
int putchar ( int c );
```

Veamos, por ejemplo, un programa que copia la entrada estándar a la salida.

```
#include <stdio.h>

main()
{
    int  c;

    while ( (c = getchar()) != EOF )
        putchar(c);
}
```

B.7. Operaciones con archivos

La entrada y salida a archivos es uno de los aspectos más delicados de cualquier lenguaje de programación, pues suelen estar estrechamente integradas con el sistema operativo. Los servicios ofrecidos por los sistemas operativos varían enormemente de un sistema a otro. Las librerías de C proporcionan un gran conjunto de funciones, muchas de ellas descritas en el libro de Kernighan y Ritchie y otras derivadas de los servicios que ofrece el Unix.

En C hay dos tipos de funciones de entrada/salida a archivos. Las primeras son derivadas del sistema operativo Unix y trabajan sin buffer. Las segundas son las que fueron estandarizadas por ANSI y utilizan un buffer intermedio. Además, hacen distinciones si trabajan con archivos binarios o de texto. Veremos las segundas, que son las más utilizadas.

Las funciones de C no hacen distinción si trabajan con un terminal, cinta o archivos situados en un disco. Todas las operaciones se realizan a través de ***streams***. Un ***stream*** está formado por una serie ordenada de bytes. Leer o escribir de un archivo implica leer o escribir del ***stream***. Para realizar operaciones se debe asociar un ***stream*** con un archivo, mediante la declaración de un puntero a una estructura **FILE**. En esta estructura se almacena toda la información para interactuar con el sistema operativo. Este puntero es inicializado mediante la llamada a la función **fopen()**, para abrir un archivo.

Cuando se ejecuta todo programa desarrollado en C hay tres ***streams*** abiertos automáticamente. Estos son ***stdin***, ***stdout*** y ***stderr***. Normalmente estos *streams* trabajan con el terminal, aunque el sistema operativo permite redireccionarlos. Las funciones **printf()** y **scanf()**, utilizan ***stdout*** y ***stdin*** respectivamente.

Los datos de los archivos pueden ser accedidos en uno de los dos formatos: texto o binario. Un *text stream* consiste en una serie de lineas de texto acabadas con un carácter ***newline***. En modo binario un archivo es una colección de **bytes** sin ninguna estructura especial.

Para utilizar las funciones de archivos se debe incluir el archivo **stdio.h**. Este define los prototipos de todas las funciones, la declaración de la estructura FILE y algunas macros. Una macro importante es **EOF**, que es el valor devuelto por muchas funciones cuando se llega al final del archivo.

Los pasos a seguir para operar con un archivo son: abrir, realizar el tratamiento y cerrar. Para abrir un archivo se utiliza la función **fopen**. Esta toma dos ***strings*** como parámetros. El primero indica el nombre del archivo que deseamos abrir y el segundo indica el modo de acceso.

```
#include <stdio.h>
FILE *fopen ( const char *filename, const char *mode );
```

Si se desea especificar un archivo binario, se añade una b al modo: **"wb+".** Si el archivo se abre correctamente, la función devuelve un puntero a una estructura **FILE**. Si no, devuelve **NULL**. La función **fprintf** se comporta exactamente igual a **printf**, excepto que toma un argumento más que indica el stream por el que se debe realizar la salida. De hecho, la llamada **printf("x")** es equivalente a **fprintf (stdout, "x")**.

```
FILE    *f;

if ((f = fopen( "login.com", "r" )) == NULL )
     printf("ERROR: no puedo abrir el archivo\n");
```

Para cerrar un archivo se utiliza la función **fclose**. Esta toma como argumento el puntero que nos proporcionó la función **fopen**.

```
#include <stdio.h>

int fclose ( FILE *stream );
```

Devuelve 0 si el archivo se cierra correctamente, EOF si se produce algún error.

Una vez que conocemos como abrir y cerrar archivos, vamos a ver cómo leer y escribir en ellos. Hay funciones para trabajar con caracteres, líneas y bloques.

Las funciones que trabajan con caracteres son **fgetc** y **fputc**. La primera lee un carácter de un stream y la segunda lo estribe.

```
#include <stdio.h>

int fgetc ( FILE *stream );

int fputc ( int c, FILE *stream );
```

La primera lee el siguiente carácter del stream y los devuelve convertido a entero sin signo. Si no hay carácter, devuelve **EOF**. La segunda, fputc, devuelve el propio carácter si no hay error. Si lo hay, devuelve el carácter EOF. Hay una tercera función, **feof** que devuelve cero si no se ha llegado al final del stream. Veamos un ejemplo de cómo se copia un archivo carácter a carácter.

```
while ( !feof(infile))

     fputc ( fgetc ( infile ), outfile );
```

Suponemos que *infile* y *outfile* son los streams asociados al archivo de entrada y al de salida, que están abiertos y luego los cerramos para completar los cambios realizados.

Otras funciones nos permiten realizar operaciones con archivos de texto trabajando línea a línea.

```
#include <stdio.h>
char *fgets ( char *s, int n, FILE *stream );

int fputs ( const char *s, FILE *stream );
```

La función **fgets** lee caracteres del stream hasta que encuentra un final de línea o se lee el carácter n-1. Mantiene el carácter \n en el string y añade el carácter \0. Devuelve la dirección del string o NULL si se produce algún error.

La función **fputs** copia el string al stream, no añade ni elimina caracteres \n y no copia la marca de final de string \0.

Hay dos funciones que nos permiten trabajar en bloques. Podemos considerar un bloque como un array. Debemos especificar el tamaño de cada elemento y el número de elementos.

```
#include <stdio.h>

size_t fread ( void *p, size_t s, size_t n, FILE *f );

size_t fwrite ( void *p, size_t s, size_t n, FILE *f );
```

A las anteriores funciones se les pasa un puntero genérico p con la dirección del área de datos que se desea leer o escribir, el tamaño de cada elemento s, y el número de elementos n, además del stream. Ambas devuelven el número de elementos leídos o escritos, que debe ser el mismo que le hemos indicado, en el caso en que no se haya producido ningún error.

B.8. Parámetros de la línea de ordenes

En esta sección se explicará el uso de los paréntesis vacíos en la primera línea de la función **main**, es decir, **main ()**. Estos paréntesis pueden contener argumentos especiales que permiten pasarles parámetros a *main* desde el sistema operativo. La mayoría de las versiones de C permiten dos argumentos, que tradicionalmente se llaman ***argc*** y ***argv***. El primero, *argc,* debe ser una variable entera, mientras que el segundo, *argv*, es una formación de punteros a carácter, es decir, una formación de cadenas de caracteres. Cada cadena de esta formación representará un parámetro que es pasado a *main*. El valor de *argc* indicará el número de parámetros pasados.

Consideremos el siguiente programa sencillo en C, que se ejecuta desde una línea de órdenes.

```
# include <stdio.h>

main ( int argc, char *argv [ ] )
{
 int cont;
 printf("argc = %d\n", argc );
 for (cont = 0; cont < argc; ++cont )
      printf("argv[%d] = %s\n", cont, argv[cont] );
}
```

Este programa permitirá introducir un número no especificado de parámetros desde la línea de órdenes. Cuando el programa es ejecutado, el valor actual de *argc* y los elementos de *argv* serán mostrados en líneas de salidas separadas

Supongamos, por ejemplo, que el nombre del programa es *muestra* y la línea de órdenes que inicia la ejecución es:

```
muestra rojo azul blanco
```

Entonces la ejecución del programa dará como resultado la siguiente salida:

argc = 4
argv[0] = muestra.exe
argv[1] = rojo
argv[2] = azul
argv[3] = blanco

La salida nos indica que cuatro elementos han sido introducidos desde la línea de órdenes, el primero es el nombre del programa, *muestra.exe*, segido por tres parámetros, *rojo, azul* y *blanco*. Cada uno es un elemento en la formación *argv*.(Notar que *muestra.exe* es el resultado de la compilación del código fuente *muestra.c*)

Análogamente, si la línea de órdenes es:

muestra rojo "azul blanco"

La salida será:

argc = 3
argv[0] = muestra.exe
argv[1] = rojo
argv[2] = azul blanco

En este caso la cadena de caracteres *"azul blanco"* será interpretada como un parámetro simple, debido a las comillas.

Una vez que los parámetros han sido introducidos, pueden ser utilizados dentro del programa como se desee. Una aplicación común es especificar nombres de archivos de datos como parámetros de la línea de órdenes. Esta técnica es la que se ha utilizada en el código *mpi.c* para poder iniciar la Consola MPI con todos los computadores especificados en un archivo, a lo cual nos referimos en el capítulo 5.

Conclusiones

Como conclusión del trabajo realizado podemos decir que hemos logrado definir una consola, la cual sirve como una herramienta útil para realizar tareas sumamente importantes, de manera más clara y automática como ser la ejecución y monitorización de aplicaciones paralelas basadas en MPI, que se ejecutan en una red local de computadores, entre las cuales mencionamos: la identificación de los computadores que se pueden utilizar de una red local para ejecutar una aplicación con MPI, monitorización de una aplicación paralela basada en MPI, *reset* del sistema paralelo (matando limpiamente los procesos), unificación del comando *mpirun* para las distintas implementaciones (MPICH y LAM) en un solo comando de consola denominado *run*, de igual manera hemos logrado unificar el comando *mpicc* utilizado para la compilación de los programas, bajo el nombre de comando de consola *comp*, es importante mencionar que fue un gran desafío el desarrollo de cada comando implementado por distintas funciones, pues cada una de ellas debió ser pensada y desarrollada individualmente, ya que cada implementación utiliza distintos comandos.

Aclaramos que en un primer momento trabajamos con la implementación MPICH con el dispositivo *ch_p4*, el que tiene un comportamiento estático, y la implementación LAM, la cual tiene un comportamiento dinámico (mediante daemons), pero luego utilizamos también el dispositivo *ch_p4mpd* de la implementación MPICH, que al igual que LAM es dinámico por el uso de daemons. Esto demandó mayor esfuerzo, dado que fue necesario estudiar y analizar mas conceptos de los previstos, demandando un grupo de funciones para atender a esta implementación con su dispositivo.

En resumen, logramos crear un espacio único de identificación de comandos en el sistema paralelo, lo cual permite al usuario familiarizarse mas rápidamente con este, dado que solo debe conocer los comandos de la Consola MPI.

Glosario alfabético de términos

array: vector que almacena un conjunto de elementos.
bit: dígito binario (0,1).
broadcast: transmisión, envío (de 1 a varios procesos).
buf o buffer: unidad de almacenamiento de mensajes.
byte: conjunto de ocho bits, que representa un carácter.
circuit switching: conectando circuítos.
deadlock: bloqueo de un proceso.
delay: demorar, retardar.
embbeding: empotrado, embebido, encastrado.
EOF: end on file, espacio fin de fichero.
file: archivo.
hardware: conjunto de unidades físicas que conforman un computador.
heterogéneas: trabajan bajo distintas plataformas o sistemas operativos.
homogéneas: trabajan bajo una misma plataforma o sistema operativo.
host: nombre del computador.
Internet: red de redes, es una enorme red global de computadoras conectadas entre sí con el objeto de compartir e intercambiar la información que contienen, independientemente del lugar del mundo en que se encuentre el usuario y de la ubicación de la computadora de la que se saca dicha información.
LAM: Local Área Multicomputer, traducido significa: varios computadores en red de área local.
LAN: Local Área Network, traducido significa: red de trabajo de área local, que permite compartir bases de datos, programas y periféricos entre computadores que estén dentro de un mismo edificio o en edificios colindantes (no mas de miles de metros).
links: enlaces.
login: es el identificador o nombre del usuario.
MPI: Message Passing Interface, traducido significa: Interfaz para Pasaje de Mensajes .
name: nombre.
overhead: por encima, arriba.
packet switching: conectando paquetes.
password: contraseña.
path: camino.
red: interconexión de computadores por medio de un camino físico, llamado: protocolo de comunicaciones.
root: supervisor o superusuario.
router: computador que interconecta dos o más redes y transfiere paquetes entre ellas.
shell: es el software que atiende a las órdenes tecleadas en el terminal y las traduce a instrucciones en la sintaxis interna del sistema.
software: conjunto de información ó datos.
speedup: aceleración.

stream: conjunto de cadenas de caracteres.
string: cadenas de caracteres.
switch: conector.

Referencias bibliográficas

[01].- Baker, L., Smith, B. J., "Parallel Programming", McGraw-Hill, New York, 1996.

[02].- Chalmers, A., Tidmus, J., "Practical Parallel Processing", International Thomson Computer Press, London, 1996.

[03].- Foster, I., "Designing and Building Parallel Programs", Addison-Wesley, Reading, Massachusetts, 1995.

[04].- Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, B., Sunderam V., "PVM: Parallel Virtual Machine - A User´s Guide and Tutorial for Network Parallel Computing", The MIT Press, Cambridge, Massachussets, 1994.

[05].- Gropp, W., Lusk, E., Skjellum, A., "Using MPI, Portable Parallel Programming with the Message Passing Interface", The MIT Press, Cambridge, Massachussets, 1997.

[06].- Message Passing Interface Forum, "MPI: A Message Passing Interface Standard", University of Tennessee, Knoxville, Tennessee, June 12, 1995

[07].- Pacheco, P., "Parallel Programming with MPI", Morgan Kaufmann, San Francisco, California, 1997.

[08].- Patterson, D., A., Hennessy, J., L., "Computer Architecture, A Quantitative Approach", 2nd ed., Morgan Kaufmann, San Francisco, California, 1996.

[09].- Stallings, W., "Computer Organization and Architecture", Fifth Edition, Prentice Hall, Upper Saddle River, New Jersey, 2000.

[10].- Tinetti, F. G., De Giusti, A. E., "Procesamiento Paralelo – Conceptos de Arquitecturas y Algoritmos", Exacta.

[11].- Wilkinson, B., Allen, M., "Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers", Prentice Hall, Upper Saddle River, New Jersey, 1999.

[12].- Stephen Coffin, “UNIX – Manual de referencia – Sistema V. Version 3”, OSBORNE/McGraw-Hill, España, 1989.

[13].- Byron Gottfried, “Programación en C”, Segunda Edición, McGraw-Hill, Madrid, España, 1997.

[14].- Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., Dongarra, J., “*MPI: The Complete Reference*”, The MIT Press, Cambridge, Massachussets, 1997.

[15].- http://www.faces.ula.ve/~ieac/manual/comand.html

[16].- http://www.geocities.com/SiliconValley/Haven/7414/UNIX/unix.html

[17].- http://www.abcdatos.com/tutoriales/sistemasoperativos/unix/index.html

[18].- http://ciberia.ya.com/miswebsri/

[19].- http://www.uib.es/c-calculo/cursc.htm

[20].- http://www.caos.uab.es/soremo/doc/csh.html

[21].- http://www.mpi-forum.org

[22].- http://www.lam-mpi.org

[23].- http://www.cnb.uam.es/~carazo/practica_mpi.html

[24].- http://www.csc.ucm.es/csc/Usuarios/Formacion/Conline/IntroMPI/
[25].- http://www.monografias.com