Tutorial de un Cluster Beowulf Casero - Primera Parte

unixOZ
unizoz@hotmail.com

Cuando hablamos de clusters, se nos viene inmediatamente a la mente NASA, universidades y lugares con muchos recursos y conocimientos. Pero un cluster es solo dos o mas computadores funcionando como uno para desempeñar tareas de forma mas rá pida y eficiente. Generalmente no tenemos la necesidad de correr aplicaciones en paralelo como individuos ya que no hacemos cosas que requieran grandes velocidades; de todas formas, para nosotros los curiosos, es interesante ver como contruimos y corremos un "supercomputador" en nuestra casa.

1. Requerimientos

En mi caso, hice un cluster con 2 computadores, el primero (y mas poderoso) lo llame **unix.oz**, mientras que el otro se llamo **billy.oz**:

unix.oz:

```
AMD ATHLON K7 600 mhz
256 mb RAM
4.1 Gigabytes de DD
SiS 900 PCI (Tarjeta de Red)
Red Hat Linux 7.2
Kernel 2.4.7-14
```

billy.oz

```
CYRIX 233 mhz
32 mb RAM
2 Gigabytes de DD
Fast Ethernet 10/100 Genius
Red Hat Linux 7.2
Kernel 2.4.7-14
```

Como sólo tenía 2 computadores, no necesite un hub o switch, los conecte mediante un cable cruzado...

2. Configurando el cluster

Un cluster tipo Beowulf funciona con uno de dos librerías de trasnferencia de mensajes: MPI (Message Passing Interface) o con PVM (Parallel Virtual Machine).

Cuando son compiladas, estas librerías pasan información o data entre las má quinas (o nodos) del cluster. Ambos usan el protocolo de comunicacion TCP/IP. Tambié n usan el programa rsh (esto es muy inseguro desafortunadamente pero en un cluster casero no importa tanto) para iniciar las sesiones entra las má quinas. Este comando (rsh) permite correr comandos UNIX remotamente.

Yo preferi tratar con PVM, y lo siguiente se debe hacer para cada computador que se desee paralelizar, voy a usar unix.oz:

1. Entrar como root al sistema
2. En `/etc/hosts`

```
192.168.1.1     unix.oz         unix
192.168.1.2     billy.oz        billy
```

3. Crear el archivo `/etc/hosts.equiv`. Este contiene el nombre de las máquinas al que rsh puede acceder

```
unix.oz
billy.oz
```

4. Reiniciar y entrar como un usuario "normal".
5. Probar la red: desde unix.oz hice ping billy.oz.
   Si no funciona, deben arreglar la configuracion de su red, si funciona, pasar a paso número 6.
6. Ahora a probar rsh: desde unix.oz hice rsh usuario_de_billy "ls-l".
   Si funciona debería mostrar una lista del directorio de un usuario del otro computador. De lo contrario,
   es VITAL resolver ese problem.a
7. ***Si algo no funciona, no hay que continuar hasta arreglarlo ****

3. Instalar y configurar PVM

Yo usé la versión 3.4 de Parallel Virtual Machine (descargable de http://www.epm.ornl.gov/pvm/pvm_home.html).

1. La bajé a /home/unixoz y como uso Bash, añadí lo siguente a .bashrc PVM_ROOT=$HOME/pvm3 PVM_DPATH=$PVM_ROOT/lib/pvmd export PVM_ROOT PVM_DPATH *** Ojo: Esto puede variar dependiendo de la distro, kernel, etc. Si no usas Bash, ve en el README al descomprimir PVM****
2. Salir del sistema y entrar nuevamente a $HOME (/home/unixoz)
3. tar zxvf pvm-3.4.tgz
4. cd pvm3 y luego en $HOME/pvm3 correr make
5. Si todo sale bien hay que ejecutar pvm Deberías ver pvm> (para salir solo hay que poner halt)
6. Repetir el proceso con cada computador
7. Luego puse pvm y para ver cuantos computadores tengo en el cluster puse conf (solo me salió uno, ya que aun debemos agregar los otros)
8. Para agregar a billy.oz puse add billy.oz y ahora al poner conf me salen unix.oz y billy.oz AHORA YA TENEMOS UN CLUSTER DE TIPO BEOWULF!!!!!!!! :)

4. Recursos

Ahora debes aprender a escribir programas en paralelo o usar PVMPOV, un programa para crea imá genes "super" digitales y diseñado para clusters (http://www.povray.org) Pronto publicaré un tutorial para este, ya que aun no lo manejo del todo bien.

Tutorial de un Cluster Beowulf Casero [2da parte]

unixOZ
unixoz@tux.cl

En la primera parte de este tutorial vimos como crear y configurar un cluster beowulf para usar una herramienta llamada PVM. En este artículo veremos que cosas podemos hacer con nuestro recié n creado cluster. Una de las principales utilidades son las librerías para la programación en C, y FORTRAN. En este artículo solo veremos la parte de C, ya que FORTRAN es un lenguaje que no manejo. Es importante adelantar que lo que viene es para personas que ya tienen un nivel intermedio o avanzado de la programación en C, esto NO es un tutorial de C.

1. Tecnicas Basicas de la Programación Paralela

Creando aplicaciones para PVM continúa con lo tradicional de la programación de memoria distribuida de multiprocesadores, como es el nCUBE de Intel. La computación en paralelo puede ser visto desde tres

aspectos bá sicos, todas basadas en la organización de las tareas que requieran el uso de nuestro cluster. El modelo má s común para programar con PVM es el **crowd** o multitud; es un conjunto de procesos similares (tienen el mismo código) pero que ejecutan distintas fracciones de un todo, o la tarea total.

El segundo es el té rmino llamado **tree** o á rbol, aquí los procesos son copiados de forma tripartita, creando una relación padre e hijo. Esta té cnica, aunque no es muy usada, permite encajar aplicaciones de las que no sabemos nada apriori.

La tercera es el **hybrid** o híbrido, y es una combinación de las dos anteriores; en cualquier punto de la ejecución de una aplicación la relación entre los procesos de carga puede cambiar. Estas tres clasificaciones nos pueden ayudar con la topología de nuestro cluster, en mi caso, gracias a esto puedo añadir un tercer nodo fá cilmente, sin afectar la configuración de unix.oz y billy.oz. A continuación está n las dos primeras de forma má s detallada.

2. Cómputos CROWD

Estas generalmente usan tres fases. El primero es la inicialización del grupo; el segundo es el cómputo mismo; y el tercero es el conjunto del output (lo que devuelve el programa al usuario), durante esta fase el grupo de nodos pueden terminar una sesión. A continuación se puede ver a través de un programa de multiplicación de matrices el uso de esto por medio del algoritomo de Pipe Multiply Roll:

{Matrix Multiplication Using Pipe-Multiply-Roll Algorithm}

```
{Processor 0 starts up other processes}
if (<my processor number> = 0) then
    for i := 1 to MeshDimension*MeshDimension
        pvm_spawn(<component name>, . .)
    endfor
endif

forall processors Pij, 0 <= i,j < MeshDimension
    for k := 0 to MeshDimension-1
        {Pipe.}
        if myrow = (mycolumn+k) mod MeshDimension
            {Send A to all Pxy, x = myrow, y <> mycolumn}
            pvm_mcast((Pxy, x = myrow, y <> mycolumn),999)
        else
            pvm_recv(999)   {Receive A}
        endif

        {Multiply.  Running totals maintained in C.}
        Multiply(A,B,C)

        {Roll.}
        {Send B to Pxy, x = myrow-1, y = mycolumn}
        pvm_send((Pxy, x = myrow-1, y = mycolumn),888)
        pvm_recv(888)       {Receive B}
    endfor
endfor
```

3. Cómputos TREE

Para mostrar mejor como esto crea un proceso parecido a la estructura de un á rbol podemos usar un algoritmo de **Parallel Sorting** del MIT donde un proceso procesa, para luego compilar un segundo proceso. Ahora hay dos procesos, y ambos se copian nuevamente para generar dos má s y así sucesivamente, creando una estructura de un á rbol. Cada proceso es independiente del otro. Este es el código desarrollado por la MIT:

```
{ Spawn and partition list based on a broadcast tree pattern. }
for i := 1 to N, such that 2^N = NumProcs
        forall processors P such that P < 2^i
                pvm_spawn(...) {process id P XOR 2^i}
```

```
                if P < 2^(i-1) then
                        midpt: = PartitionList(list);
                        {Send list[0..midpt] to P XOR 2^i}
                        pvm_send((P XOR 2^i),999)
                        list := list[midpt+1..MAXSIZE]
                else
                        pvm_recv(999)    {receive the list}
                endif
        endfor
endfor

{ Sort remaining list. }
Quicksort(list[midpt+1..MAXSIZE])

{ Gather/merge sorted sub-lists. }
for i := N downto 1, such that 2^N = NumProcs
        forall processors P such that P < 2^i
                if P > 2^(i-1) then
                        pvm_send((P XOR 2^i),888)
                        {Send list to P XOR 2^i}
                else
                        pvm_recv(888)    {receive temp list}
                        merge templist into list
                endif
        endfor
endfor
```

4. C Para Aplicaciones con PVM

Ahora que ya conocemos lo bá sico de la programación en paralelo, podemos comenzar a explorar las librerías que nos oferece PVM. Cada programa de PVM debe incluir la librería está ndard, o sea debemos añadir al comienzo de nuestro programa:

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

Las funciones de PVM generalmente se llaman con:

```
info=pvm_mytid()
```

info es un int (número entero) que devuelve esa funcion, si `pvm_mytid()` devolverá una número negativo si ocurre un error. Cuando finaliza un programa PVM, es recomendable usar la función `pvm_exit()`.

Para poder escribir un programa en paralelo, las tareas deben ser ejecutados en diferentes procesadores, para esto se usa `pvm_spawn()`. Esa función se usa de referencia para los computos tree. Se llama:

```
      numt=pvm_spawn(mi_tarea, NULL, PvmTaskDefault, 0, n_task, tids)
```

La pvm_spawn() copia mi_tarea en el nodo que elige PVM. PVM tiene mucha información a la que podemos acceder, como con pvm_parent(), pvm_config(), pvm_tasks(), etc.

5. Compilando Nuestros Programas

Para compilar mi_programa.c, se usa las siguentes extensiones gené ricas que nos ofrece PVM:

```
gcc -L directorio/donde esta pvm3/lib/ARQUITECTURA mi_programa.c -lpvm3 -o foo
```

Donde ARQUITECTURA generalmente sera LINUX, al no ser que estemos usando otro sistema operativo como BSD, DARWIN, MACINTOSH, etc. Despué s de compilar debemos poner el ejecutable en el /directorio donde esta pvm3/bin/ARQUITECURA. La compilación se debe hacer separadamente en cada arquitectura de los nodos de nuestro cluster. Luego debemos dejar corriendo el demonio de PVM (pvmd) en cada nodo, con el comando pvm y luego ponemos quit para acceder a la consola:

```
pvm> quit
```

Finalmente podemos correr nuestra aplicación en paralelo. :)

Aún queda mucho que ver sobre la programación de C en paralelo y PVM, en la tercera parte y final de este tutorial veremos como se comunican nuestras tareas, balanceo de carga, etc. Para poder ver ejemplos de código fuente estan en /directorio donde esta pvm3/examples

Tutorial de un Cluster Beowulf Casero [3ra parte]

unixOZ
unixoz@tux.cl

1. Comunicación entre Tareas

Ya es la hora de ponerse a programar aplicaciones que realmente valen la pena. Para esto es imprescindible conocer las distintas formas de ponder hacer que las tareas se puedan comunicar entre si; con PVM, esto se hace por medio de las **message-passing**.

Para poder mandar un mensaje de A a B, A debe llamar pvm_initsend(). Esto limpia el buffer y especifca un mensaje; bufid=pvm_initsend(PvmDataDefault) es generalmente usado para esto.

Despué s del inicio, la tarea para mandar debe tomar toda la información que se quiera mandar y convertirla a un buffer especial para distribuirlo entre las taras (ie: A & B); esto se hace con la función pvm_pack() (la cual es muy parecida a la famosa printf()). Tambié n hay funciones para arreglos (arrays en inglé s) de tipos de datos únicos; como la pvm_pkdbl(). Para mayor información recomiendo buscar en las pá ginas man. Cuando ya esta todo esto listo, la tarea esta preparada para ser enviada, esto se hace por medio de pvm_send().

```
info=pvm_send(tid, msgtag)
```

Esta función enviará el mensaje en el buffer a la tarea con la task id (ver 2da parte) de la tid. Esta nombra la tarea con la variable (tipo int) msgtag. Un mensaje tag es útil para decirle a la tarea cual recibirá el mensaje que tipo de dato va a recibir.

Por ejemplo, un mensaje de tag 9 puede significar la suma de los números en el mensaje, mientras que un tag 3 puede significar la su división. pvm_mcast() es una función bastante parecida ya que hace los mismo que la pvm_send(), solo que la pvm_cast() toma varios tids en vez de uno. Conviene usarla cuando se envien varios mensajes a un conjunto de tareas.

La tarea que recibe el mensaje llama a la función pvm_recv() para recibirlo. bufid=pvm(tid,msgtag) esperará que la tarea llame a la función anterior. Con la nueva PVM3.3 se añadió la pvm_trecv(), la cual se deja de esperar a la tarea que reciba en un periodo de tiempo determinado, así permitiendo mayor fluidez en los programas. Tambié n esta la pvm_probe(), esta sólo le dice a la tarea que ha llegado un mensaje, y luego se termina.

Ya cuando la tarea ha recibido un mensaje, debe desempacar la información con pvm_unpack() (la opuesta a pvm_pack()).

2. Balanceo de Carga

El balanceo de carga es muy importante para las aplicaciones. Asegurarse que cada nodo esta haciendo la tarea que debe y como lo debe hacer puede hacer o deshacer un cluster beowulf.

La forma má s fá cil se llama **balance de carga está tica**. Este mé todo consiste en la división de tareas para ser procesadas sólo una vez. La división es mejor hacerla antes que comience el trabajo.

Otra forma de balancear la carga de tareas es la **Pool of Task Paradigm** o el paradigma del Conjunto de Tareas. Consiste en un programa maestro / esclavo (donde el maestro es el administrador de tareas); el maestro manda a los esclavos trabajos. Este sistema no se recomienda implementarlo en clusters con programas que requieren una comunicación entre tareas, ya que las tareas paran y comienzan en tiempos

determinados.

3. Consideraciones en Cuanto a Rendimiento

La programación en paralelo puede ser una actividad entretenida e interesante para todo aficionado de la programación, pero hay algunas cosas que pueden hacer que nuestros programas fallen o no funcionen como nosotros queramos.

La granularidad. Es decir la proporción entre la cantidad de bytes recibidos por un proceso a el número decimal de operaciones o tareas que realiza.

```
int NumeroBytesRecibidos() : float NumeroOperacionesRealizadas()
```

Cantidad de mensajes enviados. Se demora menos el enviar pocos mensajes pesados o con mucha información que enviar varios mensajes con poca información. Esto es cierto, excepto cuando hay programas en donde se especifica lo contrario. De todos modos, la cantidad de mensajes es específico para cada plataforma y programa.

Algunas aplicaciones funcionan mejor con **paralelismo funcional**, mientras que otras para **paralelismo de datos**. En la primera, distintas máquinas hacen distintas tareas (dependiendo de su capacidad).

Por ejemplo un supercomputador puede resolver un problema de matemático o de física vectorial, una estación de trabajo gráfico puede usarse para visualizar datos en tiempo real.

En el **paralelismo de datos** la información es distribuida a todas las tareas del cluster (o máquina virtual). Las operaciones son pasadas entre procesos hasta que el problema se resuelva.

Tomar encuenta la red también es importante en un cluster. El poder de procesamiento depende de cada computador (hay diferencias mínimas hasta en procesadores de la misma marca y misma velocidad). Esto puede alterar el dinamísmo de las cargas de las máquinas o nodos. Para solucionar esto, recomiendo usar un programa de balance de carga.

4. Recursos

4.1 Depuradores

- Xmdb - Depurador para programas paralelas (recomendado para principiantes)
  http://www-c8.lanl.gov/dist_comp2/mdb/mdb.html
- P2D2 - Depurador portable, creado por la NASA
  http://www.nas.nasa.gov/NAS/Tools/Projects/P2D2

4.2 Balances de Carga & Organizadores de Tareas

- CONDOR - http://www.cs.wisc.edu/condor
- CraySoft NQE - http://www.cray.com/craysoft

Para terminar...

...cluster Beowulf en casa de unixOZ

 última actualización 8.07.2002 (log)