Clusterització

Remo Suppi Boldrito

PID_00174425

Universitat Oberta de Catalunya
www.uoc.edu

Índex

Introducció

Els avenços en la tecnologia han permès l'aparició de processadors ràpids i de baix cost i de xarxes altament eficients, cosa que ha afavorit un canvi de la relació preu/prestacions en favor de la utilització de sistemes de processadors interconnectats, en lloc d'un únic processador d'alta velocitat. Aquest tipus d'arquitectura es pot classificar en dues configuracions bàsiques:

1) Sistemes fortament acoblats (*tightly coupled systems*): són sistemes en què la memòria és compartida per tots els processadors (*shared memory systems*) i la memòria de tots ells "la veu" (el programador) com una única memòria.

2) Sistemes feblement acoblats (*loosely coupled systems*): no comparteixen memòria (cada processador en posseeix una) i es comuniquen mitjançant missatges passats a través d'una xarxa (*message passing systems*).

En el primer cas, són coneguts com a *sistemes paral·lels de còmput* (*parallel processing system*) i en el segon, com a *sistemes distribuïts de còmput* (*distributed computing systems*).

Un sistema distribuït és una col·lecció de processadors interconnectats mitjançant una xarxa en què cada un té els seus propis recursos (memòria i perifèrics) i es comuniquen intercanviant missatges per la xarxa.

En aquest mòdul es veuran diferents maneres de crear i programar un sistema de còmput distribuït, i també les eines i les biblioteques més importants per a complir aquest objectiu.

Objectius

En els materials didàctics d'aquest mòdul trobareu els continguts i les eines procedimentals per a aconseguir els objectius següents:

1. Analitzar les diferents infraestructures i eines per al còmput d'altes prestacions (inclosa la virtualització) (HPC).
2. Configurar i instal·lar un clúster d'HPC i les eines de monitoratge corresponents.
3. Instal·lar i desenvolupar programes d'exemples en les principals API de programació: Posix Threads, OpenMPI i OpenMP.
4. Instal·lar un clúster específic basat en una distribució ad hoc (Rocks).

1. Clusterització

La història dels sistemes informàtics és molt recent (es pot dir que comença en la dècada de 1960). Al principi, eren sistemes grans, pesats, cars, de pocs usuaris experts, no accessibles i lents. En la dècada de 1970, l'evolució va permetre millores substancials dutes a terme per tasques interactives (*interactive jobs*), temps compartit (*time sharing*) i terminals, i amb una considerable reducció de la grandària. La dècada de 1980 es caracteritza per un augment notable de les prestacions (fins avui en dia) i una reducció de la grandària en els anomenats *microordinadors*. La seva evolució s'ha produït gràcies a les estacions de treball (*workstations*) i els avenços en xarxes (LAN de 10 Mb/s i WAN de 56 kB/s el 1973 a LAN d'1/10 Gb/s i WAN amb ATM, *asynchronous transfer mode* d'1,2 Gb/s en l'actualitat), que és un factor fonamental en les aplicacions multimèdia actuals i d'un futur proper. Els sistemes distribuïts, per la seva banda, van entrar en la història en la dècada de 1970 (sistemes de 4 o 8 ordinadors) i el seu salt a la popularitat es va produir en la dècada de 1990. Si bé la seva administració, instal·lació i manteniment són complexos, perquè continuen creixent en grandària, les raons bàsiques de la seva popularitat són l'increment de prestacions que presenten en aplicacions intrínsecament distribuïdes (aplicacions que per la seva naturalesa són distribuïdes), la informació compartida per un conjunt d'usuaris, la compartició de recursos, l'alta tolerància a les fallades i la possibilitat d'expansió incremental (capacitat d'agregar més nodes per a augmentar les prestacions i de manera incremental). Un altre aspecte molt important actualment és la possibilitat, en aquesta evolució, de la virtualització. Les arquitectures cada vegada més eficients, amb sistemes multinucli, han permès que la virtualització de sistemes es transformi en una realitat amb tots els avantatges (i possibles desavantatges) que això comporta.

1.1. Virtualització

La virtualització és una tècnica basada en l'abstracció dels recursos d'una computadora, anomenada *Hypervisor* o VMM (*Virtual Machine Monitor*), que crea una capa de separació entre el maquinari de la màquina física (*host*) i el sistema operatiu de la màquina virtual (*virtual machine*, *guest*), i es un mitjà per a crear una "versió virtual" d'un dispositiu o recurs, com un servidor, un dispositiu d'emmagatzematge, una xarxa o fins i tot un sistema operatiu, en què es divideix el recurs en un o més entorns d'execució. Aquesta capa de programari (VMM) utilitza, gestiona i administra els quatre recursos principals d'un ordinador (CPU, memòria, xarxa i emmagatzemament) i els reparteix de manera dinàmica entre totes les màquines virtuals definides en el computador central.

D'aquesta manera ens permet tenir diversos ordinadors virtuals executant-se sobre el mateix ordinador físic.

La màquina virtual, en general, és un sistema operatiu complet que s'executa com si estigués instal·lat en una plataforma de maquinari autònoma.

Hi ha diferències entre els diversos tipus de virtualització, de les quals la més completa és aquella en què la màquina virtual simula un maquinari suficient per a permetre l'execució de manera aïllada d'un sistema operatiu virtual sense modificar i dissenyat per a la mateixa CPU (aquesta categoria també es denomina *virtualització del maquinari*.

Virtualització per maquinari

Els exemples més comuns de virtualització per maquinari són Xen, Qemu, VirtualBox, VMware Workstation/Server, Adeos i Virtual PC/Hyper-V. Alguns autors defenses l'existència d'una subcategoria dins de la virtualització per maquinari anomenada *paravirtualització*, que és una tècnica de virtualització amb una interfície de programari per a màquines virtuals similar, però no idèntica, al maquinari subjacent. Dins d'aquesta categoria entrarien KVM, QEMU, VirtualBox i VMware, entre d'altres.

Enllaç d'interès

Per saber més coses sobre virtualització podeu visitar: http://en.wikipedia.org/wiki/Virtualization.
Es pot consultar una llista completa de programes de virtualització per maquinari a: http://en.wikipedia.org/wiki/Comparison_of_platform_virtual_machines.

Un segon tipus de virtualització (parcial) és el que s'anomena *address space virtualization* (virtualització d'espai d'adreces). La màquina virtual simula múltiples instàncies de l'entorn subjacent del maquinari (però no de tot), particularment l'espai d'adreces. Aquest tipus de virtualització accepta compartir recursos i allotjar processos, però no permet instàncies separades de sistemes operatius virtuals i actualment es troba en desús.

Una tercera categoria és la denominada *virtualització compartida* del sistema operatiu, en la qual es virtualitzen servidors en la capa del sistema operatiu (nucli o *kernel*). Aquest mètode de virtualització crea particions aïllades o entorns virtuals (VE) en un únic servidor físic i instància de SO per maximitzar així els esforços d'administració del maquinari, el programari i el centre de dades. La virtualització del *Hypervisor* té una capa base (generalment un nucli de Linux) que es carrega directament en el servidor base. La següent capa superior mostra el maquinari que ha de virtualitzar-se perquè així pugui ser assignat a les VM en què existeix una còpia completa d'un sistema operatiu i l'aplicació. La diferència entre instal·lar dos sistemes operatius i virtualitzar dos sistemes operatius és que en el primer cas, tots els sistemes operatius que tinguem instal·lats funcionaran de la mateixa manera que si estiguessin installats en diferents ordinadors i necessitarem un gestor d'arrencada que, en encendre l'ordinador, ens permeti triar quin sistema operatiu volem utilitzar. En canvi, la virtualització permet canviar de sistema operatiu com si es tractés de qualsevol altre programa; no obstant això, aquesta agilitat té el desavantatge que fa que un sistema operatiu virtualitzat no sigui tan potent com un que ja estigués instal·lat.

Enllaços d'interès

Podeu trobar referències a les principals aportacions de la virtualització compartida a: http://en.wikipedia.org/wiki/Operating_system-level_virtualization.
Podeu trobar un guia ràpida d'instal·lació de Qemu [21] i la seva integració amb KQemu i KVM per a diferents distribucions de Debian a: http://wiki.debian.org/QEMU.
L'equivalent per a Xen [27] el podeu trobar a: http://wiki.debian.org/Xen.

1.2. Beowulf

Beowulf [23, 2, 25] és una arquitectura multiordinador que pot utilitzar-se per a aplicacions paral·leles/distribuïdes (APD). El sistema consisteix bàsicament en un servidor i un o més clients connectats (generalment) a través d'Ethernet i sense la utilització de cap maquinari específic. Per a explotar aquesta capacitat de còmput, cal que els programadors tinguin un model de programació distribuït que, si bé és possible mitjançant UNIX (sòcol, rpc), pot implicar un esforç considerable, ja que són models de programació a nivell de crides dels sistemes i llenguatge C, per exemple; tanmateix, aquesta línia de treball pot ser considerada de baix nivell. Un model més avançat en aquesta línia de treball són els **Posix Threads**, que permeten explotar sistemes de memòria compartida i multinuclis de manera simple i fàcil. La capa de programari (interfície de programació d'aplicacions, API) aportada per sistemes com ***Parallel Virtual Machine*** (PVM) i ***Message Passing Interface*** (MPI), facilita notablement l'abstracció del sistema i permet programar aplicacions paral·leles/distribuïdes de manera senzilla i simple. La forma bàsica de treball és mestre-treballadors (*master-workers*), en què hi ha un servidor que distribueix la tasca que faran els treballadors. En grans sistemes (per exemple, de 1.024 nodes) hi ha més d'un mestre i nodes dedicats a tasques especials com, per exemple, entrada/sortida o monitoratge. Una altra opció no menys interessant és **OpenMP**, una API per a la programació multiprocés de memòria compartida en múltiples plataformes. Aquesta capa de programari permet afegir concurrència als programes escrits en C, C++ i Fortran sobre la base del model d'execució *fork-join* i es compon d'un conjunt de directives de compilador, rutines de biblioteca i variables d'entorn que influencien el comportament en temps d'execució i proporcionen als programadors una interfície simple i flexible per al desenvolupament d'aplicacions paral·leles per a tot tipus de plataformes.

Una de les principals diferències entre Beowulf i un clúster d'estacions de treball (*cluster of workstations*, COW) és que Beowulf "es veu" com una única màquina on s'accedeix als nodes remotament, ja que no disposen de terminal (ni de teclat), mentre que un COW és una agrupació d'ordinadors que poden ser utilitzats tant pels usuaris de la COW com per altres usuaris de manera interactiva per mitjà de la pantalla i el teclat. Cal considerar que Beowulf no és un programari que transforma el codi de l'usuari en distribuït ni afecta el nucli del sistema operatiu (com, per exemple, Mosix). Simplement, és una forma d'agrupació (un clúster) de màquines que executen GNU/Linux i actuen com un superordinador. Òbviament, hi ha una gran quantitat d'eines que permeten obtenir una configuració més fàcil, biblioteques o modificacions al nucli per obtenir millors prestacions, però és possible construir un clúster Beowulf a partir d'un GNU/Linux estàndard i de programari convencional. La construcció d'un clúster Beowulf de dos nodes, per exemple, es pot dur a terme simplement amb les dues màquines connectades per Ethernet mitjançant un concentrador (*hub*), una distribució de GNU/Linux estàndard (Debian), el sistema d'arxius compartit (NFS) i tenint habilitats els serveis de xarxa com rsh o ssh. En aquestes condicions, es pot afirmar que es disposa d'un clúster simple de dos nodes.

1.2.1. Com s'han de configurar els nodes?

Primer cal modificar el fitxer `/etc/hosts` de cada node per tal que la línia de màquina física local només tingui el 127.0.0.1 i no inclogui cap nom de la màquina:

Per exemple:
127.0.0.1 localhost
I afegir les IP dels nodes (i per a tots els nodes), per exemple:
192.168.0.1 pirulo1
192.168.0.2 pirulo2
...
Cal crear un usuari (nteum) en tots els nodes, crear un grup i afegir aquest usuari al grup:
groupadd beowulf
adduser nteum beowulf
echo umask 007 >> /home/nteum/.bash_profile

Així, qualsevol arxiu creat per l'usuari nteum o qualsevol dins del grup serà modificable pel grup beowulf. S'ha de crear un servidor d'NFS (i els altres nodes seran clients d'aquest NFS). Sobre el servidor fem:

```
mkdir /mnt/nteum
chmod 770 /mnt/nteum
chown nteum:beowulf /mnt/nteum -R
```

Ara exportem aquest directori des del servidor:

```
cd /etc
cat >> exports
/mnt/wolf 192.168.0.100/192.168.0.255 (rw)
<control d>
```

S'ha de tenir en compte que la xarxa pròpia serà 192.168.0.xxx i que és una xarxa privada, és a dir, que el clúster no es veurà des d'Internet i s'hauran d'ajustar les configuracions perquè tots els nodes es vegin des de tots els altres (des dels tallafocs).

A continuació comprovem que els serveis estan funcionant (teniu en compte que l'ordre `chkconfig` pot no estar instal·lada en totes les distribucions):

```
chkconfig -add sshd
chkconfig -add nfs
chkconfig -add rexec
chkconfig -add rlogin
chkconfig -level 3 rsh on
chkconfig -level 3 nfs on
chkconfig -level 3 rexec on
chkconfig -level 3 rlogin on
```

En cas contrari, també és possible consultar en els directoris corresponents (`etc/rc3.d/`). Si bé estarem en un xarxa privada, per a treballar de manera segura és important treballar amb ssh en lloc de rsh, per la qual cosa hauríem de generar les claus per interconnectar en mode segur les màquines usuari

nteum sense contrasenya. Per a això, modifiquem (eliminem el comentari #), de `/etc/ssh/sshd_config` en les línies següents:

```
RSAAuthentication yes
AuthorizedKeysFile .ssh/authorized_keys
```

Reiniciem la màquina i ens connectem com a usuari nteum, ja que aquest usuari serà qui farà funcionar el clúster. Per a generar les claus:

```
ssh-keygen -b 1024 -f ~/.ssh/id_rsa -t rsa -N ""
```

En el directori `/home/nteum/.ssh` s'hauran creat l'arxiu `aneu_rsa` i l'arxiu `aneu_rsa.pub`. S'ha de copiar `aneu_rsa.pub` en un arxiu anomenat `authorized_keys` en el mateix directori. Després modifiquem els permisos amb `chmod 644 ~/.ssh/aut*` i `chmod 755 ~/.ssh`. Atès que solament el node principal es connectarà a la resta (i no al revés), aleshores només necessitem copiar la clau pública (`d_rsa.pub`) a cada node en el directori/arxiu `/home/nteum/.ssh/authorized_keys` de cada node. Sobre cada node, a més, s'haurà de muntar l'NFS agregant `/etc/fstab` a la línia següent: `pirulo1:/mnt/nteum /mnt/nteum nfs rw,hard,intr 0 0`.

A partir d'aquí ja tenim un clúster Beowulf per a executar aplicacions que podran ser PVM o MPI (com es veurà en els subapartats següents). Sobre FC existeix una aplicació (system-config-cluster) que permet configurar un clúster basat en una eina gràfica.

Enllaç d'interès

Per obtenir més informació consulteu: http://www.redhat.com /docs/manuals/enterprise /RHEL-5-manual /cluster_Administration /index.html.

1.3. Beneficis del còmput distribuït

Quins són els beneficis del còmput en paral·lel? Ho veurem amb un exemple [23]. Considerem un programa per a sumar nombres (per exemple, 4+5+6+...) anomenat `sumdis.c`:

```
#include <stdio.h>

int main (int argc, char** argv){
long inicial, final, resultat, tmp;
    if (argc < 2) {
   printf (" Ús: %s N.º inicial N.º final\n",argv[0]);
     return (4); }
    else {
   inicial = atol (argv[1]);
   final = atol (argv[2]);
   resultat = 0;}
    for (tmp = inicial; tmp <= final; tmp++){resultat += tmp; };
    printf("%f\n", resultat);
    return 0;
}
```

El compilem amb `gcc -o sumdis sumdis.c` i si mirem l'execució d'aquest programa amb, per exemple,

```
time ./sumdis 1 1000000
499941376000.000000

    real    0m0.006s
    user    0m0.004s
    sys     0m0.000s
```

es podrà observar que el temps en una màquina Debian 2.6.26 sobre una màquina virtual (Virtualbox) amb processador i7 és (aproximadament) 6 millèsimes de segon (real = 0,006 i usuari = 0,004). Si, en canvi, fem des d'1 a 16 milions, el temps real puja fins a 0,082 s, és a dir, 14 vegades més, la qual cosa, si es consideren 160 milions, el temps serà de l'ordre de desenes de minuts.

La idea bàsica del còmput distribuït és repartir la feina. Així, si disposem d'un clúster de 4 màquines (node1–node4) amb un servidor i on el fitxer es comparteix per NFS, seria interessant dividir l'execució mitjançant rsh (no recomanable, però com a exemple és acceptable), de manera que el primer sumi d'1 a 40.000.000; el segon, de 40.000.001 a 80.000.000; el tercer, de 80.000.001 a 120.000.000; i el quart, de 120.000.001 a 160.000.000. Les ordres següents mostren una possibilitat. Considerem que el sistema té el directori `/home` compartit per NFS i l'usuari nteum, que executarà l'*script*, té configurat adequadament el `.rhosts` per a accedir sense contrasenya al seu compte. A més, si s'ha activat el `tcpd` en `/etc/inetd.conf` en la línia de `rsh`, ha d'existir la corresponent en el `/etc/hosts.allow`, que permeti accedir a les quatre màquines del clúster:

```
mkfifo out1 Crea una cua fifo a /home/nteum
./distr.sh & time cat out1 | awk '{total + = $1 } END {printf "%lf", total}'
```

S'executa l'ordre `distr.sh`; es recol·lecten els resultats i se sumen mentre es mesura el temps d'execució. L'*script* de l'intèrpret d'ordres `distr.sh` pot ser quelcom semblant a:

```
rsh nodo1 /home/nteum/sumdis 1 40000000 > /home/nteum/out1 < /dev/null &
rsh nodo2 /home/nteum/sumdis 40000001 80000000 > /home/nteum/out1 < /dev/null &
rsh nodo3 /home/nteum/sumdis 80000001 120000000 > /home/nteum/out1 < /dev/null &
rsh nodo4 /home/nteum/sumdis 120000001 160000000 > /home/nteum/out1 < /dev/null &
```

Podrem observar que el temps es redueix notablement (aproximadament en un valor proper a 4) i no exactament de forma lineal, però molt propera. Òbviament, aquest exemple és molt simple i solament vàlid per a finalitats demostratives. Els programadors utilitzen biblioteques que els permeten realitzar el temps d'execució, la creació i comunicació de processos en un sistema distribuït (per exemple, PVM, MPI o OpenMP).

1.3.1. Com cal programar per a aprofitar la concurrència?

Existeixen diverses maneres d'expressar la concurrència en un programa. Les tres més comunes són:

1) Utilitzant fils (o processos) en el mateix processador (multiprogramació amb solapament del còmput i l'E/S).

2) Utilitzant fils (o processos) en sistemes multinucli.

3) Utilitzant processos en diferents processadors que es comuniquen per mitjà de missatges (MPS, *Message Passing System*).

Aquests mètodes poden implementar-se sobre diferents configuracions de maquinari (memòria compartida o missatges) i, si bé tots dos mètodes tenen els seus avantatges i desavantatges, els principals problemes de la memòria compartida són les limitacions en la escalabilitat (ja que tots els nuclis/processadors utilitzen la mateixa memòria i el nombre d'aquests en el sistema està limitat per l'ample de banda de la memòria) i, en els sistemes de pas de missatges, la latència i velocitat dels missatges a la xarxa. El programador haurà d'avaluar quin tipus de prestacions necessita, les característiques de l'aplicació subjacent i el problema que es desitja solucionar. No obstant això, amb els avenços de les tecnologies de multinuclis i de xarxa, la popularitat (i quantitat) d'aquests sistemes ha crescut. Les API més comunes avui dia són Posix Threads i OpenMP per a memòria compartida i MPI (en les seves versions OpenMPI o Mpich) per a pas de missatges. Com hem esmentat anteriorment, existeix una altra biblioteca molt difosa per a passos de missatges, anomenada PVM, que però, amb la versatilitat i les prestacions que s'obtenen amb MPI, ha deixat relegada a aplicacions petites o per a aprendre a programar en sistemes distribuïts. Aquestes biblioteques, a més, no limiten la possibilitat d'utilitzar fils (encara que a nivell local) i tenir concurrència entre processament i entrada/sortida.

Per a crear una aplicació paral·lela/distribuïda, es pot partir de la versió sèrie o mirar l'estructura física del problema i determinar quines parts poden ser concurrents (independents). Les parts concurrents seran candidates a reescriure's com a codi paral·lel. A més, s'ha de valorar si és possible reemplaçar les funcions algebraiques per les seves versions paral·lelitzades (per exemple, ScaLapack *Scalable Linear Algebra Package*, disponible a Debian –scalapack-pvm, mpich-test, dev, scalapack1-pvm, mpich segons siguin per a PVM o MPI–). També és convenient esbrinar si hi ha alguna aplicació similar paral·lela (per exemple, per a PVM*) que pugui orientar-nos sobre el mode de construcció de l'aplicació paral·lela.

*http://www.epm.ornl.gov/pvm

Paral·lelitzar un programa no és una tasca fàcil, ja que s'ha de tenir en compte la **llei d'Amdahl**, que afirma que l'increment de velocitat (*speedup*) està limitat per la fracció de codi (f) que pot ser paral·lelitzat, de la següent manera:

$$\text{increment de velocitat} = \frac{1}{1-f}$$

Aquesta llei implica que amb una aplicació seqüencial, $f = 0$ i l'increment de velocitat = 1, mentre que amb tot el codi paral·lel, $f = 1$ i l'increment de velocitat es fa infinit(!). Si considerem valors possibles, un 90% ($f = 0,9$) del codi paral·lel significa un increment de velocitat igual a 10, però amb $f = 0,99$ l'increment de velocitat és igual a 100. Aquesta limitació es pot evitar amb algorismes escalables i diferents models de programació d'aplicació (paradigmes):

1) Mestre-treballador: el mestre inicia tots els treballadors i en coordina el treball i processos d'entrada/sortida.

2) *Single Process Multiple Data* (SPMD): mateix programa que s'executa amb diferents conjunts de dades.

3) Funcional: varis programes que realitzen una funció diferent en l'aplicació.

En resum, podem concloure:

1) Proliferació de màquines multitasca (multiusuari) connectades per xarxa amb serveis distribuïts (NFS i NIS YP).

2) Són sistemes heterogenis amb sistemes operatius de tipus NOS (*Networked Operating System*), que ofereixen una sèrie de serveis distribuïts i remots.

3) La programació d'aplicacions distribuïdes es pot efectuar a diferents nivells:

a) Utilitzant un model client-servidor i programant a baix nivell (sòcols) o utilitzant memòria compartida a baix nivell (Posix Threads).

b) El mateix model, però amb API d'"alt" nivell (OpenMP i MPI).

c) Utilitzant altres models de programació com, per exemple, la programació orientada a objectes distribuïts (RMI, CORBA, Agents, etc.).

1.4. Memòria compartida. Models de fils (*threading*)

Normalment, en una arquitectura client-servidor, els clients sol·liciten als servidors determinats serveis i esperen que aquests els contestin amb la major eficàcia possible. Per a sistemes distribuïts amb servidors amb una càrrega molt alta (per exemple, sistemes d'arxius de xarxa, bases de dades centralitzades o distribuïdes), el disseny del servidor es converteix en una qüestió crítica per a determinar el rendiment general del sistema distribuït. Un aspecte crucial en aquest sentit és trobar la manera òptima d'utilitzar l'E/S, tenint en compte el tipus de servei que ofereix, el temps de resposta esperat i la càrrega de clients. No existeix un disseny predeterminat per a cada servei i escollir el correcte dependrà dels objectius i restriccions del servei i de les necessitats dels clients.

Les preguntes que hem de respondre abans de triar un determinat disseny són: quant temps es triga en un procés de sol·licitud del client? Quantes d'aquestes sol·licituds és probable que arribin durant aquest temps? Quant temps pot esperar el client? E n quin grau afecta aquesta càrrega del servidor a les prestacions del sistema distribuït? A més, amb l'avenç de la tecnologia de processadors ens trobem amb què disposem de sistemes multinucli (múltiples nuclis d'execució) que poden executar seccions de codi independents. Si es dissenyen els programes en forma de múltiples seqüències d'execució i el sistema operatiu ho suporta (i GNU/Linux n'és un d'ells), l'execució dels programes es reduirà notablement i les prestacions s'incrementaran en forma (gairebé) lineal en funció dels nuclis de l'arquitectura.

1.4.1. Multifils (*multithreading*)

Les últimes tecnologies en programació per a aquest tipus d'aplicacions (i així ho demostra l'experiència) és que els dissenys més adequats són aquells que utilitzen models de multifils (*multithreading models*), en els quals el servidor té una organització interna de processos paral·lels o fils cooperants i concurrents.

> Un fil (*thread*) és una seqüència d'execució (fil d'execució) d'un programa, és a dir, diferents parts o rutines d'un programa que s'executen concurrentment en un únic processador i que accediran a les dades compartides al mateix temps.

Quins avantatges aporta això respecte a un programa seqüencial? Considerem que un programa té tres rutines A, B i C. En un programa seqüencial, la rutina C no s'executarà fins que s'hagin executat A i B. Si, en canvi, A, B i C són fils, les tres rutines s'executaran concurrentment i, si en elles hi ha E/S, tindrem concurrència d'execució amb E/S del mateix programa (procés), cosa que millorarà notablement les prestacions d'aquest programa. Generalment, els fils estan continguts dins d'un procés i diferents fils d'un mateix procés poden compartir alguns recursos, mentre que diferents processos no. L'execució de múltiples fils en paral·lel necessita el suport del sistema operatiu i en els processadors moderns existeixen optimitzacions del processador per a suportar models multifil (*multithreading*).

Generalment, existeixen quatre models de disseny per fils (en ordre de complexitat creixent):

1) Un fil i un client: en aquest cas el servidor entra en un bucle sense fi escoltant per un port i davant la petició d'un client s'executen els serveis en el mateix fil. Altres clients hauran d'esperar a que acabi el primer. És fàcil d'implementar però solament atén a un client a la vegada.

2) Un fil i diversos clients amb selecció: en aquest cas el servidor utilitza un sol fil, però pot acceptar múltiples clients i multiplexar el temps de CPU entre ells. Es necessita una gestió més complexa dels punts de comunicació (sòcols), però permet crear serveis més eficients, encara que presenta problemes quan els serveis necessiten una alta càrrega de CPU.

3) Un fil per client: és, probablement, el model més popular. El servidor espera peticions i crea un fil de servei per a atendre cada nova petició dels clients. Això genera simplicitat en el servei i una alta disponibilitat, però el sistema no escala amb el nombre de clients i pot saturar-se molt ràpidament, ja que el temps de CPU dedicat davant una gran càrrega de clients es redueix notablement i la gestió del sistema operatiu pot ser molt complexa.

4) Servidor amb fils en granja (*worker threads*): aquest mètode és més complex però millora l'escalabilitat dels anteriors. Existeix un nombre fix de fils treballadors (*workers*) als quals el fil principal distribueix el treball dels clients. El problema d'aquest mètode és l'elecció del nombre de treballadors: amb un nombre elevat, cauran les prestacions del sistema per saturació; amb un nombre massa baix, el servei serà deficient (els clients hauran d'esperar). Normalment caldrà sintonitzar l'aplicació per a treballar amb un determinat entorn distribuït.

Existeixen diferents maneres d'expressar en l'àmbit de programació amb fils: paral·lelisme a nivell de tasques o paral·lelisme a través de les dades. Triar el model adequat minimitza el temps necessari per a modificar, depurar i sintonitzar el codi. La solució a aquesta disjuntiva és descriure l'aplicació en termes de dos models basats en un treball en concret:

- Tasques paral·leles amb fils independents que poden atendre tasques independents de l'aplicació. Aquestes tasques independents seran encapsulades en fils que s'executaran asincrònicament i s'hauran d'utilitzar biblioteques com Posix Threads (Linux/Unix) o Win32 Thread API (Windows), que han estat dissenyades per a suportar concurrència nivell de tasca.

- Model de dades paral·leles per a calcular llaços intensius; és a dir, la mateixa operació ha de repetir-se un nombre elevat de vegades (per exemple, comparar una paraula amb les paraules d'un diccionari). Per a aquest cas és possible encarregar la tasca al compilador de l'aplicació o, si no és possible, que el programador descrigui el paral·lelisme utilitzant l'entorn OpenMP, que és una API que permet escriure aplicacions eficients sota aquest tipus de models.

Una aplicació d'informació personal (*Personal Information Manager*) és un bon exemple d'una aplicació que conté concurrència a nivell de tasques (per exemple, accés a la base de dades, llibreta d'adreces, calendari, etc.). Això podria ser en pseudocodi:

```
Function addressBook;
Function inBox;
Function calendar;
Program PIM {
   CreateThread (addressBook);
   CreateThread (inBox);
   CreateThread (calendar); }
```

Podem observar que existeixen tres execucions concurrents sense relació entre elles. Un altre exemple d'operacions amb paral·lelisme de dades podria ser un corrector d'ortografia, que en pseudocodi seria: `Function SpellCheck {loop (word = 1, words_in_file) compareToDictionary (word);}`

S'ha de tenir en compte que tots dos models (fils paral·lels i dades paral·leles) poden existir en una mateixa aplicació. A continuació es mostrarà el codi d'un productor de dades i un consumidor de dades basat en Posix Threads. Per a compilar sobre Linux, per exemple, s'ha d'utilitzar `gcc -o pc pc.c -lptread`.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define QUEUESIZE 10
#define LOOP 20

void *producer (void *args);
void *consumer (void *args);
typedef struct { /* Estructura de la memòria intermèdia compartida i descriptors de fils */
int buf[QUEUESIZE]; long head, tail; int full, empty;
pthread_mutex_t *mut; pthread_cond_t *notFull, *notEmpty;
    } queue;
queue *queueInit (void);   /* Prototipus de funció: inicialització de la memòria intermèdia */
void queueDelete (queue *q); /* Prototipus de funció: esborrat de la memòria intermèdia*/
void queueAdd (queue *q, int in); /* Prototipus de funció: inserir element en la memòria intermèdia */
void queueDel (queue *q, int *out); /* Prototipus de funció: treure element de la memòria intermèdia */

int main () {
queue *fifo; pthread_t pro, con; fifo = queueInit ();
if (fifo ==  NULL) { fprintf (stderr, " Error en crear memòria intermèdia.\n");  exit (1);  }
pthread_create (&pro, NULL, producer, fifo);   /* Creació del fil productor */
pthread_create (&con, NULL, consumer, fifo); /* Creació del fil consumidor*/
pthread_join (pro, NULL);   /* main () espera fins que acaben tots dos fils */
pthread_join (con, NULL);
queueDelete (fifo);       /* Eliminació de la memòria intermèdia compartida */
return 0; }               /* Fi */

void *producer (void *q) { /*Funció del productor */
queue *fifo; int i;
fifo = (queue *)q;
for (i = 0; i < LOOP; i++) {    /* Insereixo en la memòria intermèdia elements=LOOP*/
pthread_mutex_lock (fifo->mut);   /* Semàfor per a entrar a inserir */
while (fifo->full) {
printf ("Productor: queue FULL.\n");
pthread_cond_wait (fifo->notFull, fifo->mut);  }
/* Bloqueig del productor si la memòria intermèdia està plena, alliberant el semàfor mut
    per tal que pugui entrar el consumidor. Continuarà quan el consumidor executi
    pthread_cond_signal (fifo->notFull);*/
         queueAdd (fifo, i); /* Insereixo element en la memòria intermèdia */
                   pthread_mutex_unlock (fifo->mut); /* Allibero el semàfor */
         pthread_cond_signal (fifo->notEmpty);/*Desbloquejo consumidor si està bloquejat*/
         usleep (100000);  /* Dormo 100 mseg per a permetre que el consumidor s'activi */
}
return (NULL); }
```

```
void *consumer (void *q) { /*Funció del consumidor */
queue *fifo; int i, d;
fifo = (queue *)q;
for (i = 0; i < LOOP; i++) { /* Trec de la memòria intermèdia elements=LOOP*/
pthread_mutex_lock (fifo->mut); /* Semàfor per a entrar a treure */
while (fifo->empty) {
printf (" Consumidor: queue EMPTY.\n");
pthread_cond_wait (fifo->notEmpty, fifo->mut); }
    /* Bloqueig del consumidor si la memòria intermèdia està buida, alliberant el semàfor mut
    per tal que pugui entrar el productor. Continuarà quan el consumidor executi
    pthread_cond_signal (fifo->notEmpty);*/
queueDel (fifo, &d); /* Trec element de la memòria intermèdia */
pthread_mutex_unlock (fifo->mut); /* Allibero el semàfor */
pthread_cond_signal (fifo->notFull); /*Desbloqueig productor si està bloquejat*/
printf (" Consumidor: Rebut %d.\n", d);
usleep(200000);/* Dormo 200 mseg per a permetre que el productor s'activi */
}
return (NULL); }

queue *queueInit (void) {
queue *q;
q = (queue *)malloc (sizeof (queue)); /* Creació de la memòria intermèdia */
if (q == NULL) return (NULL);
q->empty = 1;  q->full = 0;  q->head = 0;  q->tail = 0;
q->mut = (pthread_mutex_t *) malloc (sizeof (pthread_mutex_t));
pthread_mutex_init (q->mut, NULL); /* Creació del semàfor */
q->notFull = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
pthread_cond_init (q->notFull, NULL); /* Creació de la variable condicional notFull*/
q->notEmpty = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
pthread_cond_init (q->notEmpty, NULL);   /* Creació de la variable condicional notEmpty*/
return (q); }

void queueDelete (queue *q) {
pthread_mutex_destroy (q->mut);  free (q->mut);
pthread_cond_destroy (q->notFull); free (q->notFull);
pthread_cond_destroy (q->notEmpty); free (q->notEmpty);
free (q); }

void queueAdd (queue *q, int in) {
q->buf[q->tail] = in; q->tail++;
if (q->tail == QUEUESIZE) q->tail = 0;
if (q->tail == q->head) q->full = 1;
q->empty = 0;
return; }

void queueDel (queue *q, int *out){
*out = q->buf[q->head]; q->head++;
if (q->head == QUEUESIZE) q->head = 0;
if (q->head == q->tail) q->empty = 1;
q->full = 0;
return; }
```

1.5. OpenMP

L'OpenMP (Open-Multi Processing) és una interfície de programació d'aplicacions (API) amb suport multiplataforma per a la programació en C/C++ i Fortran de processos amb ús de memòria compartida sobre plataformes Linux/Unix (i també Windows). Aquesta infraestructura es compon d'un conjunt de directives del compilador, rutines de la biblioteca i variables d'entorn que permeten aprofitar recursos compartits en memòria i en temps d'execució. Definit conjuntament per un grup dels principals fabricants de maquinari i programari, OpenMP permet utilitzar un model escalable i portàtil de programació, que proporciona als usuaris una interfície simple i flexible per al

desenvolupament, sobre plataformes paral·leles, des d'aplicacions d'escriptori fins a aplicacions d'altes prestacions sobre superordinadors. Una aplicació construïda amb el model híbrid de la programació paral·lela pot executar-se en un ordinador utilitzant tant OpenMP com Message Passing Interface (MPI) [1].

OpenMP és una implementació multifil, mitjançant la qual un fil mestre divideix la tasques sobre un conjunt de fils treballadors. Aquests fils s'executen simultàniament i l'entorn d'execució realitza l'assignació d'aquests als diferents processadors de l'arquitectura. La secció del codi que està dissenyada per a funcionar en paral·lel està marcada amb una directiva de preprocessament que crearà els fils abans que la secció s'executi. Cada fil tindrà un identificador (`aneu`) que s'obtindrà a partir d'una funció (`omp_get_thread_num()` en C/C++) i, després de l'execució paral·lela, els fils s'uniran de nou en la seva execució sobre el fil mestre, que continuarà amb l'execució del programa. Per defecte, cada fil executarà una secció paral·lela de codi independent però es poden declarar seccions de "treball compartit" per a dividir una tasca entre els fils, de manera que cada fil executi part del codi assignat. D'aquesta forma, és possible tenir en un programa OpenMP paral·lelisme de dades i paral·lelisme de tasques convivint conjuntament.

Els principals elements d'OpenMP són les sentències per a la creació de fils, la distribució de càrrega de treball, la gestió de dades d'entorn, la sincronització de fils i les rutines a nivell d'usuari. OpenMP utilitza en C/C++ les directives de preprocessament conegudes com a *pragma* (`#pragma omp `) per a diferents construccions. Així per exemple, `omp parallel` s'utilitza per a crear fils addicionals per tal d'executar el treball indicat en la sentència paral·lela on el procés original és el fil mestre (`aneu=0`). El conegut programa que imprimeix "Hello, world" utilitzant OpenMP i multifils és*:

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

int main(int argc, char* argv[]){
   #pragma omp parallel
   printf("Hello, world.\n");
   return 0;}
```

***Compileu amb** `gcc -fopenmp -o hell hello.c`**; s'ha de tenir instal·lada la biblioteca GCC OpenMP (GOMP)**
`apt-get install libgomp1`

Per a especificar *work-sharing constructs* s'utilitza:

- **omp for o omp do**: reparteix les iteracions d'un llaç en múltiples fils.
- **sections**: assigna blocs de codi independents consecutius a diferents fils.
- **single**: especifica que el bloc de codi serà executat per un sol fil amb una sincronització (*barrier*) al final del mateix.
- **master**: similar a `single`, però el codi del bloc serà executat pel fil mestre i no hi ha sincronització implicada al final.

Per exemple, per a inicialitzar els valors d'una matriu en paral·lel utilitzant fils per a fer una porció de la feina (compileu, per exemple, amb `gcc -fopenmp -o init2 init2.c`):

```
#include <stdio.h>
#include <omp.h>
#define N 1000000
int main(int argc, char *argv[]) {
   float a[N]; long i;
    #pragma omp parallel for
   for (i=0;i<N;i++)  a[i]= 2*i;
return 0;
}
```

Si executem amb el *pragma* i després ho comentem i calculem el temps d'execució (`estafi ./*nit2`), veiem que el temps d'execució passa de 0.006 s a 0.008 s, la qual cosa mostra la utilització del nucli doble del processador.

Ja que OpenMP és un model de memòria compartida, moltes variables en el codi són visibles per a tots els fils per defecte. Però de vegades és necessari tenir variables privades i passar valors entre blocs seqüencials del codi i blocs paral·lels, per la qual cosa cal definir atributs a les dades (*data clauses*) que permetin diferents situacions:

- **shared**: les dades a la regió paral·lela són compartides i accessibles per tots els fils simultàniament.

- **private**: les dades a la regió paral·lela són privades per a cada fil, i cadascun té una còpia d'elles sobre una variable temporal.

- **default**: permet que el programador defineixi com seran les dades dins de la regió paral·lela (`shared`, `private` o `none`).

Un altre aspecte interessant d'OpenMP són les directives de sincronització:

- **critical section**: el codi emmarcat serà executat per fils, però solament un a la vegada (no hi haurà execució simultània) i es manté l'exclusió mútua.

- **atomic**: similar a `critical section`, però avisa al compilador perquè usi instruccions de maquinari especials de sincronització i així obtenir millors prestacions.

- **ordered**: el bloc és executat en l'ordre com si es tractés d'un programa seqüencial.

- **barrier**: cada fil espera que la resta hagi acabat la seva execució (implica sincronització de tots els fils al final del codi).

- **nowait**: especifica que els fils que acabin el treball assignat poden continuar.

A més a més, OpenMP subministra sentències per a la planificació (*scheduling*) del tipus `schedule(type, chunk)` (on el tipus pot ser `static`, `dynamic` o `guided`), o proporciona control sobre la sentència `if`, la qual cosa permetrà definir si es paral·lelitza o no en funció de si l'expressió és veritable o no. OpenMP també proporciona un conjunt de funcions de biblioteca, com per exemple:

- **omp_set_num_threads**: defineix el nombre de fils a utilitzar a la següent regió paral·lela.
- **omp_get_num_threads**: obté el nombre de fils que s'estan utilitzant en una regió paral·lela.
- **omp_get_max_threads**: obté la màxima quantitat possible de fils.
- **omp_get_thread_num**: retorna el nombre del fil.
- **omp_get_num_procs**: retorna el màxim nombre de processadors que es poden assignar al programa.
- **omp_in_parallel**: retorna un valor diferent de zero si s'executa dins d'una regió paral·lela.

A continuació veurem alguns exemples simples (compileu amb la instrucció `gcc -fopenmp -o out_file input_file.c`):

```
/* Programa simple multifils amb OpenMP */
#include <omp.h>
int main()  {
     int iam =0, np = 1;

     #pragma omp parallel private(iam, np)
     #if defined (_OPENMP)
         np = omp_get_num_threads();
         iam = omp_get_thread_num();
     #endif

     printf("Hello from thread %d out of %d \n", iam, np);
     }
 }

 /* Programa simple amb fils imbricats amb OpenMP */
 #include <omp.h>
 #include <stdio.h>
     main(){
         int x=0,nt,tid,ris;

         omp_set_nested(2);
         ris=omp_get_nested();
         if (ris) printf("Parale\lgem{}isme imbricat actiu %d\n", ris);
         omp_set_num_threads(25);
         #pragma omp parallel private (nt,tid) {
             tid = omp_get_thread_num();
             printf("Thread %d\n",tid);
             nt = omp_get_num_threads();
             if (omp_get_thread_num()==1)
             printf("Número de Threads: %d\n",nt);
             }
}

 /* Programa simple d'integració amb OpenMP */
 #include <omp.h>
 #include <stdio.h>
 #define N 100
     main() {
```

```
        double local, pi=0.0, w; long i;
        w = 1.0 / N;
        #pragma omp parallel private(i, local)
            {
            #pragma omp single
            pi = 0.0;
            #pragma omp for reduction(+: pi)
            for (i = 0; i < N; i++) {
              local = (i + 0.5)*w;
              pi = pi + 4.0/(1.0 + local*local);
              printf ("Pi: %f\n",pi);
                }
            }
        }
```

```
/* Programa simple de reducció amb OpenMP */
#include <omp.h>
#include <stdio.h>
#define NUM_THREADS 2
    main () {
    int i; double ZZ, res=0.0;
        omp_set_num_threads(NUM_THREADS);
     #pragma omp parallel for reduction(+:res) private(ZZ)
        for (i=0; i< 1000; i++) {
       ZZ = i*i;
       res = res + ZZ;
       printf("ZZ: %f, res: %f\n", ZZ, res);}
              }
        }
```

Nota

Es recomana veure també els exemples que es poden trobar en la referència [17].

1.6. MPI, *Message Passing Interface*

La definició de l'API de MPI [15, 16] ha estat el treball resultant de l'MPI Forum (MPIF), que és un consorci de més de 40 organitzacions. MPI té influències de diferents arquitectures, llenguatges i treballs en el món del paral·lelisme, com per exemple: WRC (Ibm), Intel NX/2, Express, nCUBE, Vertex, p4, Parmac i contribucions de ZipCode, Chimp, PVM, Chamaleon i PICL.

El principal objectiu del MPIF va ser dissenyar una API, sense relació particular amb cap compilador ni biblioteca, que permetés la comunicació eficient (*memory-to-memory copy*), el còmput i la comunicació concurrents i la descàrrega de comunicació,sempre que existeixi un coprocessador de comunicacions. A més, es demanava que suportés el desenvolupament en ambients heterogenis, amb interfície C i F77 (incloent C++, F90), on la comunicació fos fiable i les fallades, resoltes pel sistema. L'API també havia de tenir interfície per a diferents entorns (PVM, NX, Express, p4, etc.), disposar una implementació adaptable a diferents plataformes amb canvis insignificants i que no interfereixi amb el sistema operatiu (*thread-safety*). Aquesta API va ser dissenyada especialment per a programadors que utilitzessin el *Message Passing Paradigm* (MPP) en C i F77, per a aprofitar-ne la característica més rellevant: la portabilitat. L'MPP es pot executar sobre màquines multiprocessador, xarxes d'estacions de treball i fins i tot, sobre màquines de memòria compartida. La versió MPI1 (la versió més estesa) no suporta creació (*spawn*) dinàmica de tasques, mentre que MPI2 (versió que implementa OpenMPI) sí que ho fa.

Molts aspectes han estat dissenyats per a aprofitar els avantatges del maquinari de comunicacions sobre SPC (*scalable parallel computers*) i l'estàndard ha estat acceptat per la majoria dels fabricants de maquinari en paral·lel i distribuït (SGI, SUN, Cray, HPConvex, IBM, etc.). Existeixen versions lliures (per exemple, Mpich i openMPI) que són totalment compatibles amb les implementacions comercials realitzades pels fabricants de maquinari iinclouen comunicacions punt a punt, operacions col·lectives i grups de processos, context de comunicacions i topologia, suport per a F77 i C i un entorn de control, administració i anàlisi de perfils.

Tanmateix, també existeixen alguns punts que poden presentar certs problemes en determinades arquitectures com són la memòria compartida, l'execució remota, les eines de construcció de programes, la depuració, el control de fils, l'administració de tasques i les funcions d'entrada/sortida concurrents (la major part d'aquests problemes de falta d'eines estan resolts en la versió 2 de l'API MPI2). El funcionament en MPI1, com que no té creació dinàmica de processos, és molt simple, ja que els processos/tasques s'executen de manera autònoma amb el seu propi codi estil MIMD (*multiple instruction multiple data*) i comunicant-se via trucades MPI. El codi pot ser seqüencial o multifil (concurrents) i MPI funciona sense interferir amb el sistema operatiu, és a dir, es poden utilitzar trucades a MPI en fils concurrents, ja que les trucades són reentrants.

Per a la instal·lació de MPI es recomana utilitzar la distribució, ja que la seva compilació és summament complexa (per les dependències que necessita d'altres paquets). Debian inclou la versió Mpich 1.2.x i també OpenMPI 1.2.x (també s'inclou la versió MPD –versió de *multipurpose daemon* que inclou suport per a la gestió i control de processos en forma escalable–). La millor elecció serà OpenMPI, ja que combina les tecnologies i els recursos de diversos projectes (FT-MPI, LA-MPI, LAM/MPI i PACX-*MPI) i suporta totalment l'estàndard MPI-2. Entre d'altres característiques d'OpenMPI tenim: és conforme a MPI-2, concurrent i no interfereix amb el sistema operatiu, creació dinàmica de processos, alt rendiment i gestió de treballs tolerants a fallades, instrumentació en temps d'execució, planificador de tasques, etc. Per a això, cal instal·lar els paquets `libopenmpi-dev`, `libopenmpi1`, `openmpi-bin`, `openmpi-common` i `openmpi-doc` (Recordeu que cal tenir activat el *senar-free* com a font de programari en Debian). A més, aquesta distribució inclou una altra implementació de MPI anomenada LAM (paquets `lam*` i documentació en `/usr/doc/lam-runtime/release.html`). Les implementacions són equivalents des del punt de vista de MPI però es gestionen de manera diferent.

1.6.1. Configuració d'un conjunt de màquines per a fer un clúster adaptat a OpenMPI

Per a configurar un conjunt de màquines per tal de fer un clúster adaptat a OpenMPI [29] s'han de seguir els següents passos:

1) Cal tenir les màquines "visibles" (per exemple, mitjançant un `ping`) a través de TCP/IP (IP pública/privada).

2) És recomanable que totes les màquines tinguin la mateixa arquitectura de processador, així és més fàcil distribuir el codi, amb versions similars de Linux (si pot ser amb el mateix tipus de distribució).

3) S'ha de generar un mateix usuari (per exemple, mpiuser) a totes les màquines i el mateix directori `$HOME` muntat per NFS.

4) Nosaltres anomenarem les màquines `slave1`, `slave2`, etc. (ja que després resultarà més fàcil fer les configuracions) però es poden anomenar com prefereixi cadascú.

5) Una de les màquines serà el mestre i la resta, *slaveX*.

6) Cal instal·lar en tots els nodes (suposem que tenim Debian): openmpi-bin, openmpi-common, libopenmpi1, libopenmpi-dev (aquesta última no és necessària per als nodes esclaus). Cal verificar que en totes les distribucions es treballa amb la mateixa versió d'OpenMPI.

7) A Debian els executables estan en `/usr/bin`, però si estan en un camí diferent haurà d'agregar-se a mpiuser i també verificar que `LD_LIBRARY_PATH` apunta a `/usr/lib`.

8) En cada node esclau ha d'instal·lar-se l'SSH server (heu d'instal·lar el paquet `openssh-server`) i sobre el mestre, el client (paquet `openssh-client`).

9) Cal crear les claus públiques i privades fent `ssh-keygen -t dsa` i copiar

```
cp /home/mpiuser/.ssh/id_dsa.pub /home/mpiuser/.ssh/authorized_keys
```

Si no es comparteix el directori, cal assegurar que cada esclau sap que l'usuari mpiuser es pot connectar, per exemple, fent des del *slaveX*

```
scp /home/mpiuser/.ssh/id_dsa.pub mpiuser@slave1:.ssh/authorized_keys
```

i canviar els permisos

```
chmod 700 /home/mpiuser/.ssh; chmod 600 /home/mpiuser/.ssh/authorized_keys
```

Es pot verificar que això funciona fent, per exemple, `ssh slave1`.

10) Cal configurar la llista de les màquines sobre les quals s'executarà el programa, per exemple `/home/mpiuser/.mpi_hostfile` i amb el següent contingut:

```
# The Hostfile for Open MPI
# The master node, slots=2 is used because it is a dual-processor machine.
localhost slots=2
# The following slave nodes are single processor machines:
slave1
slave2
slave3
```

11) OpenMPI permet utilitzar diferents llenguatges, però aquí utilitzarem C. Per a fer-ho, cal executar sobre el mestre `mpicc testprogram.c`. Si es desitja veure que incorpora mpicc, es pot fer `mpicc -showme`.

12) S'ha de verificar que els *slaveX* no sol·liciten cap contrasenya (podem indicar la contrasenya per a la clau ssh amb `ssh-add ~/.ssh/id_dsa`).

13) Per a executar en local podríem fer `mpirun -np 2 ./myprogram` i per a executar sobre els nodes remots (per exemple, 5 processos) `mpirun -np 2 -hostfile ./mpi_hostfile ./myprogram`.

Sobre Debian es pot instal·lar el paquet `update-cluster` per a ajudar en la seva administració (cal anar amb compte, ja que aquest paquet sobreescriu arxius importants). És important observar que `np` és el nombre de processos o processadors en què s'executarà el programa i es pot posar el nombre que es desitgi, ja que OpenMPI intentarà distribuir els processos de forma equilibrada entre totes les màquines. Si hi ha més processos que processadors, OpenMPIMpich utilitzarà les característiques d'intercanvi de tasques de GNU/Linux per a simular l'execució paral·lela. A continuació es veuran dos exemples: Srtest, un programa simple per a establir comunicacions entre processos punt a punt, i cpi, que calcula el valor del número π de manera distribuïda (per integració).

```
/* Srtest Progam */
#include "mpi.h"
#include <stdio.h>
#include <string.h>
#define BUFLEN 512

int main(int argc, char *argv[]){
    int myid, numprocs, next, namelen;
    char buffer[BUFLEN], processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;
    MPI_Init(&argc,&argv);  /* Sempre s'ha de posar MPI abans d'altres trucades */
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);    /*Integra el procés en un grup de comunicacions*/
    MPI_Get_processor_name(processor_name,&namelen); /*Obté el nom del processador*/

    fprintf(stderr,"Procés %d sobre %s\n", myid, processor_name);
    fprintf(stderr,"Procés %d de %d\n", myid, numprocs);
    strcpy(buffer,"hello there");
    if (myid == numprocs-1) next = 0;
    else next = myid+1;

    if (myid == 0) { /*Si es l'inicial, envia cadena de memòria intermèdia*/
        printf("%d sending '%s' \n",myid,buffer);fflush(stdout);
    MPI_Send(buffer, strlen(buffer)+1, MPI_CHAR, next, 99, MPI_COMM_WORLD);
        /*Blocking Send, 1:memòria intermèdia, 2:mida, 3:tipus, 4:destinació, 5:etiqueta, 6:context*/
    printf("%d receiving \n",myid);fflush(stdout);
    MPI_Recv(buffer, BUFLEN, MPI_CHAR, MPI_ANY_SOURCE, 99, MPI_COMM_WORLD,&status);
    printf("%d received '%s' \n",myid,buffer);fflush(stdout);
       /* mpdprintf(001,"%d receiving \n",myid); */
        }
    else {
       printf("%d receiving  \n",myid);fflush(stdout);
   MPI_Recv(buffer, BUFLEN, MPI_CHAR, MPI_ANY_SOURCE, 99, MPI_COMM_WORLD,&status);
       /* Blocking Recv, 1:memòria intermèdia, 2:mida, 3:tipus, 4:font, 5:etiqueta, 6:context, 7:estat*/
   printf("%d received '%s' \n",myid,buffer);fflush(stdout);
   /* mpdprintf(001,"%d receiving \n",myid); */
   MPI_Send(buffer, strlen(buffer)+1, MPI_CHAR, next, 99, MPI_COMM_WORLD);
   printf("%d sent '%s' \n",myid,buffer);fflush(stdout);
       }
    MPI_Barrier(MPI_COMM_WORLD); /*Sincronitza tots els processos*/
```

```
        MPI_Finalize(); /*Allibera els recursos i finalitza*/
        return (0);
        }


    /* CPI Program */
    #include "mpi.h"
    #include <stdio.h>
    #include <math.h>
    double f( double );
    double f( double a) { return (4.0 / (1.0 + a*a)); }
    int main( int argc, char *argv[] ) {
        int done = 0, n, myid, numprocs, i;
        double PI25DT = 3.14159265358979323846264643;
        double mypi, pi, h, sum, x; double startwtime = 0.0, endwtime;
        int namelen; char processor_name[MPI_MAX_PROCESSOR_NAME];
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs); /*Indica el número de processos en el grup*/
        MPI_Comm_rank(MPI_COMM_WORLD,&myid); /*Id del procés*/
        MPI_Get_processor_name(processor_name,&namelen); /*Nom del procés*/
        fprintf(stderr, "Procés %d sobre %s\n", myid, processor_name);
        n = 0;
        while (!done) {
            if (myid ==0) { /*Si és el primer...*/
                if (n ==0) n = 100; else n = 0;
                startwtime = MPI_Wtime();} /* Time Clock */
            MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD); /*Difusió a la resta*/
                /*Envia des del 4 arg. a tots els processos del grup. La resta que no són 0
                copiaran la memòria intermèdia des de 4 o arg -procés 0-*/
                /*1:memòria intermèdia, 2:mida, 3:tipus, 5:grup */
            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);
                    /* Combina els elements del Send Buffer de cada procés del grup utilitzant
                    l'operació MPI_SUM i retorna el resultat en el Recv Buffer. Ha se ser trucada
                    per tots els processos del grup utilitzant els mateixos arguments*/
                    /*1:sendbuffer, 2:recvbuffer, 3:mida, 4:tipus, 5:oper, 6:arrel, 7:context*/
                if (myid == 0){ /*Només el P0 imprimeix el resultat*/
                    printf("Pi és aproximadament %.16f, l'error és %.16f\n", pi,fabs(pi - PI25DT));
                    endwtime = MPI_Wtime();
                    printf("Temps d'execució = %f\n", endwtime-startwtime); }
                }
        }
        MPI_Finalize(); /*Allibera recursos i finalitza*/
        return 0;
    }
```

Per a visualitzar l'execució d'un codi paral·lel/distribuït en MPI existeix una aplicació anomenada XMPI (a Debian xmpi) que permet "veure", durant l'execució, l'estat de l'execució de les aplicacions sobre MPI. També és possible instal·lar una biblioteca, `libxmpi4`, `libxmpi4-dev`, que implementa el protocol XMPI per a analitzar gràficament programes MPI amb més detalls que els oferts per xmpi.

1.7. Rocks Cluster

Rocks Cluster és una distribució de Linux per a clústers de computadors d'alt rendiment. Les versions actuals de Rocks Cluster estan basades en CentOS i, com a instal·lador, Anaconda amb certes modificacions, que simplifica la

instal·lació "en massa" en molts ordinadors. Rocks Cluster inclou moltes eines (com MPI) que no formen part de CentOS però són els components integrals que fan un grup d'ordinadors en un clúster. Les instal·lacions es poden personalitzar amb paquets de programari addicionals, utilitzant CD especials (anomenats *roll CD*). Els *rolls* estenen el sistema integrant automàticament els mecanismes de gestió i empaquetament usats pel programari bàsic, i simplifiquen molt la instal·lació i configuració d'un gran nombre de computadors. S'han creat una gran quantitat de *rolls*, com el SGE roll, el Condor roll, el Llustre roll, el Java roll i el Ganglia roll. Rocks Cluster és una de les distribucions més emprades en l'àmbit de clústers, per la seva facilitat d'instal·lació i incorporació de nous nodes, i incorpora gran quantitat de programes per al manteniment i monitoratge del clúster.

Enllaços d'interès

Trobareu una llista detallada de les eines incloses en Rocks Cluster a: http://www.rocksclusters.org/roll-documentation/base/5.4/x8109.html.

Les característiques principals de Rocks Cluster són:

1) Facilitat d'instal·lació, ja que només cal completar la instal·lació d'un node anomenat *node mestre* (*frontend*), la resta s'instal·la amb Avalache, que és un programa P2P que ho fa de forma automàtica i evita haver d'instal·lar els nodes un a un.

2) Disponibilitat de programes (conjunt molt ampli de programes que no cal compilar i transportar) i facilitat de manteniment (solament s'ha de mantenir el node mestre).

3) Disseny modular i eficient, pensat per a minimitzar el tràfic de xarxa i utilitzar el disc dur propi de cada node per a compartir solament la informació mínima i imprescindible.

La instal·lació es pot seguir pas a pas des del lloc web de la distribució [24] i els autors garanteixen que no es triga més d'una hora per a fer una instal·lació bàsica. Rocks Cluster permet, en l'etapa d'instal·lació, diferents mòduls de programari, els *rolls*, que contenen tot allò necessari per a instal·lar i configurar el sistema amb aquestes noves "addicions" de forma automàtica i, a més, decidir en el node mestre com serà la instal·lació en els nodes esclaus, quins *rolls* estaran actius i quina arquitectura s'usarà. Per al manteniment, inclou un sistema de còpia de seguretat de l'estat, anomenat Roll Restore. Aquest *roll* guarda els arxius de configuració i *scripts* (i fins i tot es poden afegir els arxius que es desitgi).

1.7.1. Guia ràpida d'instal·lació

Aquest és un resum breu de la instal·lació proposada a [24] per a la versió 5.4 i es parteix de la base que el node mestre té (mínim) 20 GB de disc dur, 1 GB de RAM, arquitectura x86-64 i 2 interfícies de xarxa; per als nodes 20 GB de disc dur, 512 MB de RAM i 1 interfície de xarxa. Després d'obtenir el disc d'arrencada del nucli seguim els passos següents:

Enllaços d'interès

Es pot descarregar el disc d'arrencada del nucli de: http://www.rocksclusters.org/wordpress/?page_id=80

1) Arrenquem el node mestre i veurem una pantalla en la qual introduïm `build`. Ens demanarà la configuració de la xarxa (IPV4 o IPV6).

2) El següent pas és seleccionar els `*rolls` (per exemple, seleccionant els "CD/DVD-based *Roll" i introduint els següents *rolls*) i marcar en les pantalles successives quins són els que desitgem.

3) La següent pantalla ens demanarà informació sobre el clúster (és important definir bé el nom de la màquina, *Fully-Qualified Host Name*, ja que en cas contrari fallarà la connexió amb diferents serveis) i també informació per a la xarxa privada que connectarà el node mestre amb els nodes i la xarxa pública (per exemple, la que connectarà el node mestre amb Internet) així com DNS i passarel·les.

4) A continuació se sol·licitarà la contrasenya per a l'usuari principal, la configuració del servei de temps i les particions del disc (es recomana escollir "auto").

5) Després de formatar els discos, sol·licitarà els CD de ***rolls* indicats, installarà els paquets i farà una nova arrencada del node mestre.

6) Per a instal·lar els nodes s'ha d'entrar com a usuari principal en el node mestre i executar `insert-ethers`, que capturarà les peticions de DHCP dels nodes i els agregarà a la base de dades del node mestre, i seleccionar l'opció "Compute" (per defecte, consulteu la documentació per a les altres opcions).

7) Encenem el primer node i en l'ordre d'arrencada de la BIOS generalment es tindrà CD, PXE (Network Boot), Hard Disk, (si l'ordinador no suporta PXE, llavors feu l'arrencada del node amb el *Kernel Roll CD*). En el node mestre es veurà la petició i el sistema ho agregarà com a `compute-0-0` i començarà la descàrrega i instal·lació dels arxius. Si la instal·lació falla, s'hauran de reiniciar els serveis httpd, mysqld i autofs en el node mestre.

8) A partir d'aquest punt es pot monitorar la instal·lació executant la instrucció `rocks-console`, per exemple amb `rocks-console compute-0-0`. Després que s'hagi instal·lat tot, es pot sortir de `insert-ethers` pressionant la tecla "F8". Si es disposa de dos bastidors (*racks*), després d'instal·lar el primer es pot començar amb el segon fent `insert-ethers -cabinet=1`, els quals rebran els noms `compute-1-0`, `compute-1-1`, etc.

9) A partir de la informació generada en consola per `rocks list host`, generarem la informació per a l'arxiu `machines.conf`, que tindrà un aspecte com:

```
nteum slot=2
compute-0-0 slots=2
compute-0-1 slots=2
compute-0-2 slots=2
compute-0-3 slots=2
```

i que després haurem d'executar amb

```
mpirun -np 10 -hostfile ./machines.conf ./mpi_program_to_execute.
```

1.8. Monitoratge del clúster

1.8.1. Ganglia

Ganglia [8] és una eina que permet monitorar de forma escalable i distribuïda l'estat d'un clúster. Mostra a l'usuari les estadístiques de forma remota (per exemple, les mitjanes de càrrega de la CPU o la utilització de la xarxa) de totes les màquines que conformen el clúster, basant-se en un disseny jeràrquic i utilitzant multidestinació (*multicast*) per a escoltar/anunciar el protocol amb la finalitat de controlar l'estat dins dels grups i connexions punt a punt entre els nodes del clúster per tal d'intercanviar l'estat. Ganglia utilitza XML per a la representació de dades, XDR per al transport compacte i portàtil de dades i RRDtool per a l'emmagatzematge i visualització de dades. El sistema es compon de dos dimonis (gmond i gmetad), una pàgina de visualització basada en PHP i algunes eines d'utilitat.

Gmond és un dimoni multifil que s'executa en cada node del clúster que es desitja supervisar (no és necessari tenir un sistema d'arxius NFS o base de dades ni mantenir comptes especials dels arxius de configuració). Les tasques de Gmond són: monitorar els canvis d'estat en l'amfitrió (*host*), enviar els canvis pertinents, escoltar l'estat d'altres nodes (a través d'un canal d'unidestinació –*unicast*– o de multidestinació –*multicast*–) i respondre a les peticions d'un XML de l'estat del clúster. La federació dels nodes es realitza amb un arbre de connexions punt a punt entre els nodes determinats (representatius) del clúster per a agregar l'estat de la resta de nodes. En cada node de l'arbre s'executa el dimoni **Gmetad**, que periòdicament sol·licita les dades dels restants nodes, analitza el XML, guarda tots els paràmetres numèrics i exporta l'XML agregat per un sòcol TCP. Les fonts de dades poden ser dimonis gmond, en representació de determinats grups, o altres dimonis gmetad, en representació de conjunts de grups. Finalment, la web de Ganglia proporciona una vista de la informació recollida dels nodes del clúster en temps real. Per exemple, es pot veure la utilització de la CPU durant l'última hora, dia, setmana, mes o any i mostra gràfics similars per a l'ús de memòria, ús de disc, estadístiques de la xarxa, nombre de processos en execució i tots els altres indicadors de Ganglia.

Instal·lació de Ganglia (sobre Debian)

En aquest cas farem la instal·lació sobre un Debian Lenny, però és similar per a altres distribucions:

1) Ganglia utilitza RRDTool per a generar i mostrar els gràfics, de manera que hem d'executar `apt-get install rrdtool librrds-perl librrd2-dev` i per a visualitzar diagrames de seccions, `apt-get install php5-gd`.

2) Excepte pel node mestre, Ganglia està inclòs en el dipòsit Debian i sobre el servidor web hem d'instal·lar `apt-get install ganglia-monitor`

`gmetad`. Sobre tots els altres nodes solament es necessita tenir instal·lat el paquet `ganglia-monitor`.

3) Per al node mestre s'ha de compilar el paquet sencer de Ganglia de la pàgina web*, per exemple fent, dins del directori arrel:

*http://ganglia.info/?page_id=66

```
wget http://downloads.sourceforge.net/ganglia/ganglia-3.1.7.tar.gz,
```

canviant la versió segons correspongui, i després fer

```
tar xvzf ganglia*.tar.gz.
```

4) Abans de configurar el paquet font (i depenent de la instal·lació que es tingui), s'hauran d'incloure, mitjançant l'apt-get o Synaptic, les biblioteques següents: `libapr1-dev`, `libconfuse-dev`, `libpcre3-dev`.

5) Dins del directori creat (`$HOME/ganglia-3.1.7/`, per exemple) executeu

```
./configure -prefix=/opt/ganglia -enable-gexec -with-gmetad -sysconfdir=/etc/ganglia
```

o similar (vegeu `./configure -help` per a les diferents opcions) i en aquest cas utilitzarem el directori `/opt` per a instal·lar-lo. Solucioneu els diferents problemes que pugui donar la configuració per falta de biblioteques necessàries (el procediment es fàcil mirant la biblioteca que falta i instal·lant-la a traves del apt-get o Synaptic). Quan la configuració finalitzi, es mostrarà quelcom com:

```
Welcome to..
      ______                    ___
     / ____/___ _____  ____ _/ (_)___ _
    / / __/ __ `/ __ \/ __ `/ / / __ `/
   / /_/ / /_/ / / / / /_/ / / / /_/ /
   \____/\__,_/_/ /_/\__, /_/_/\__,_/
                    /____/
Copyright (c) 2005 University of California, Berkeley
Version: 3.1.7 Library: Release 3.1.7 0:0:0
Type "make" to compile.
```

6) Cal fer un `make` i després un `make install` (es podrà comprovar que s'ha instal·lat a `/opt/ganglia`).

7) Finalment s'ha de configurar el servidor web executant primer l'ordre `mkdir /var/www/ganglia`, copiant el contingut del directori `./web` al directori del servidor `cp web/* /var/www/ganglia` i fent la configuració del directori adequat en Apache. Per a fer això, és necessari que busqueu en l'arxiu `/etc/apatxe2/sites-enabled/000-default` la secció de codi del directori `/var/www` i agregar una redirecció. Aquesta secció haurà de quedar com:

```
<Directory /var/www/>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
```

```
        Order allow,deny
        allow from all
        RedirectMatch ^/$ /ganglia/
    </Directory>
```

A continuació solament fa falta reiniciar Apache amb `/etc/init.d/apatxe2 restart` i amb un navegador anar a `http://localhost/ganglia` per a tenir la visualització del node local!

8) Tenint en compte que ja hem instal·lat el *Ganglia meta daemon gmetad* sobre el node de capçalera en el segon pas (`apt-get install gmetad`) haurem de configurar les opcions bàsiques a `/etc/gmetad.conf`. Els canvis són pocs i les opcions estan comentades per defecte: `authority`, per a indicar la URL (per exemple, `host.dominio.org/ganglia`), sobretot si s'està darrere d'un tallafocs amb NAT a través de `iptables`. `trusted_hosts`, si el servidor web té múltiples dominis, podrien ser llistats en aquesta opció (en cas contrari, deixeu-ho buit). `gridname` és el nom amb el qual es vol visualitzar la instal·lació. `rrd_rootdir` si desitja canviar la destinació on s'emmagatzemen les dades, ja que Ganglia necessita emmagatzemar-ne gran quantitat (la destinació per defecte és `/var/lib/ganglia/rrds`). Finalment, cal reiniciar gmetad amb `/etc/init.d/gmetad restart`.

9) A cada amfitrió que es desitgi monitorar s'haurà d'executar el monitor gmon, incloent l'amfitrió que executa gmetad si també es desitja monitorar. Per a això, cal instal·lar en cada node `apt-get install ganglia-monitor` i modificar l'arxiu `/etc/gmond.conf` perquè es pugui connectar al servidor que recollirà les dades. Editeu els següents valors: `name` és el nom del clúster al que està associat aquest node. `mcast_if` serveix per a seleccionar quina interfície de xarxa usarà el node per a connectar-se, si és que té múltiples. `num_nodes` és el nombre de nodes en el clúster. Finalment, cal reiniciar gmon amb `/etc/init.d/ganglia-monitor restart`.

10) Després d'haver configurat tots els nodes, cal reiniciar el gmetad sobre el servidor del web amb `/etc/init.d/gmetad restart` i esperar uns segons per a veure els canvis en els nodes.

1.8.2. Cacti

Cacti [3] és una solució per a la visualització d'estadístiques de xarxa i va ser dissenyada per a aprofitar el poder d'emmagatzematge i la funcionalitat de generar gràfiques que posseeix RRDtool. Aquesta eina, desenvolupada en PHP, ofereix diferents formes de visualització, gràfics avançats i disposa d'una interfície d'usuari fàcil d'usar, fet que la fa interessant tant per a xarxes LAN com per a xarxes complexes amb centenars de dispositius.

La seva instal·lació és simple i requereix tenir instal·lat MySQL prèviament. Després, fent `apt-get install cacti` s'instal·larà el paquet i al final ens demanarà si volem instal·lar la interfície amb la base de dades, sol·licitant-nos

el nom d'usuari i contrasenya de la mateixa, i la contrasenya de l'usuari admin de la interfície de Cacti. Després haurem de copiar el lloc de Cacti en el directori `/var/www` fent `cp -r /usr/share/cacti/site /var/www/cacti` i a continuació, connectar-nos a `http://localhost/cacti`. A l'inici ens preguntarà pels ajustos en la instal·lació, ens demanarà que canviem la contrasenya i a continuació, ens mostrarà la consola d'administració per a crear els gràfics que desitgem i ajustar mitjançant la pestanya "Settings" un conjunt d'opcions, entre elles, la freqüència d'actualització.

1.9. Introducció a la metacomputació o la computació distribuïda o en reixeta

Els requeriments de còmput necessaris per a certes aplicacions són tan grans que requereixen milers d'hores per a poder executar-se en entorns de clústers. Aquestes aplicacions han donat lloc a la generació d'ordinadors virtuals en xarxa, metacomputació (*metacomputing*) o computació distribuïda o en graella (*grid computing*). Aquesta tecnologia ha permès connectar entorns d'execució, xarxes d'alta velocitat, bases de dades, instruments, etc., distribuïts geogràficament. Això permet obtenir una potència de processament amb excellents resultats, que no seria econòmicament possible d'una altra forma. En són exemples d'aplicació experiments com l'I-WAY Networking (que connecta superordinadors de 17 llocs diferents) a Amèrica del Nord, DataGrid o CrossGrid a Europa i IrisGrid a Espanya. Aquests metaordinadors, o ordinadors en graella, tenen molt en comú amb els sistemes paral·lels i distribuïts (SPD), però també difereixen en aspectes importants. Si bé estan connectats per xarxes, aquestes poden ser de diferents característiques, no es pot assegurar el servei i estan localitzades en dominis diferents. El model de programació i les interfícies han de ser radicalment diferents (pel que fa a la dels sistemes distribuïts) i adequades per al còmput d'altes prestacions. Igual que en SPD, les aplicacions de metacomputació requereixen una planificació de les comunicacions per a aconseguir les prestacions desitjades; però atesa la seva naturalesa dinàmica, són necessàries noves eines i tècniques. És a dir, mentre que la metacomputació pot formar-se sobre la base dels SPD, per a aquests cal crear noves eines, mecanismes i tècniques [7].

Si es té en compte solament l'aspecte de potència de càlcul, llavors podem veure que existeixen diverses solucions en funció de la magnitud i les característiques del problema. En primer lloc, es podria pensar en un superordinador (servidor), però presenten problemes, com la falta de escalabilitat, equips i manteniment costós, còmput de pics (passen molt temps desaprofitats) i problemes de fiabilitat. L'alternativa econòmica és un conjunt d'ordinadors interconnectats mitjançant una xarxa d'altes prestacions *(Fast Ethernet –LAN– o Myrinet –SANT–*), la qual cosa formaria un clúster d'estacions dedicat a computació paral·lela/distribuïda (SPD) amb un rendiment molt alt (relació cost/rendiment de 3 a 15 vegades). Però aquests sistemes presenten inconvenients com el cost elevat de les comunicacions, el manteniment, el model de

programació, etc. No obstant això, és una solució excel·lent per a aplicacions de gra mitjà o HTC, *High Estafi Computing* (computació d'alta productivitat). Un altre concepte interessant és el de *Intranet Computing*, que significa la utilització dels equips d'una xarxa local (per exemple, una xarxa de classe C) per a executar treballs seqüencials o paral·lels amb l'ajut d'una eina d'administració i càrrega; és a dir, és el següent pas a un clúster i permet l'explotació de potència computacional distribuïda en una gran xarxa local amb els consegüents avantatges, en augmentar l'aprofitament dels recursos (cicles de CPU a baix cost), la millora de l'escalabilitat i una administració no massa complexa. Per a aquest tipus de solucions existeix programari com Oracle (Sun) Grid Engine de Sun Microsystems [19], Condor de la Universitat de Wisconsin (tots dos gratuïts) [28] o LSF de Platform Computing (comercial) [20]. L'opció de l'*Intranet Computing* presenta alguns inconvenients com ara la impossibilitat de gestionar recursos fora del domini d'administració.

Algunes de les eines esmentades (Condor o SGE) permeten la col·laboració entre diferents subnodes del sistema, però tots ells han de tenir la mateixa estructura administrativa, les mateixes polítiques de seguretat i la mateixa filosofia de gestió de recursos. Si bé representa un pas endavant en l'obtenció de còmput a baix preu, solament gestionen la CPU i no les dades compartides entre els subnodes. A més, els protocols i interfícies són de propietat i no estan basats en cap estàndard obert, no es poden amortitzar els recursos quan estan desaprofitats ni es pot compartir recurs amb altres organitzacions [2, 5, 4]. El creixement dels ordinadors i de la tecnologia de xarxes en l'última dècada ha motivat el desenvolupament d'una de les arquitectures per a HPC més rellevants de tots els temps: la computació distribuïda en Internet o *grid computing* (GC). La computació distribuïda ha transformat les grans infraestructures amb l'objectiu de compartir recursos a Internet de manera uniforme, transparent, segura, eficient i fiable. Aquesta tecnologia és complementària a les anteriors, ja que permet interconnectar recursos en diferents dominis d'administració respectant les seves polítiques internes de seguretat i el seu programari de gestió de recursos en la intranet. Segons un dels seus precursors, Ian Foster, en el seu article "What is the Grid? A Three Point Checklist" (2002), una graella (*grid*) és un sistema que:

1) coordina recursos que no estan subjectes a un control centralitzat,

2) utilitza protocols i interfícies estàndards, obertes i de propòsits generals, i

3) genera qualitats de servei no trivials.

Entre els beneficis que presenta aquesta nova tecnologia, es poden destacar el lloguer de recursos, l'amortització de recursos propis, gran potència sense necessitat d'invertir en recursos i instal·lacions, col·laboració i compartició entre institucions i organitzacions virtuals, etc. [13].

El projecte Globus [9] és un dels més representatius en aquest sentit, ja que és el precursor del desenvolupament d'una eina per a la metacomputació o

computació distribuïda i que aporta avenços considerables a les àrees de la comunicació, la informació, la localització i la planificació de recursos, l'autenticació i l'accés a les dades. És a dir, Globus permet compartir recursos localitzats en diferents dominis d'administració, amb diferents polítiques de seguretat i gestió de recursos i està format per un paquet de programari (*middleware*) que inclou un conjunt de biblioteques, serveis i API. L'eina Globus (*Globus toolkit*) està formada per un conjunt de mòduls amb interfícies ben definides per a interactuar amb altres mòduls o serveis i la seva instal·lació és complexa i meticulosa, ja que requereix un conjunt de passos previs, noms i certificats que solament es justifiquen en cas d'una gran instal·lació. La funcionalitat d'aquests mòduls és la següent:

- **Localització i assignació de recursos**: permet comunicar a les aplicacions quins són els requeriments i situar els recursos que els satisfacin, ja que una aplicació no pot saber on es troben els recursos sobre els quals s'executarà.

- **Comunicacions**: subministra els mecanismes bàsics de comunicació, que representen un aspecte important del sistema, ja que han de permetre diversos mètodes perquè les aplicacions les puguin utilitzar eficientment. Entre ells s'inclouen el pas de missatges (*message passing*), les crides a procediments remots (RPC), la memòria compartida distribuïda i els fluxos de dades (*stream-based*) i de multidestinació.

- **Servei d'informació (Unified Resource Information Service)**: subministra un mecanisme uniforme per a obtenir informació en temps real sobre l'estat i l'estructura del metasistema on s'estan executant les aplicacions.

- **Interfície d'autenticació**: són els mecanismes bàsics d'autenticació per a validar la identitat dels usuaris i els recursos. El mòdul genera la capa superior que després utilitzaran els serveis locals per a accedir a les dades i els recursos del sistema.

- **Creació i execució de processos**: utilitzat per a iniciar l'execució de les tasques que han estat assignades als recursos, passant-los els paràmetres d'execució i controlant l'execució fins a la seva finalització.

- **Accés a dades**: és el responsable de facilitar un accés d'alta velocitat a les dades emmagatzemades en arxius. Per a DB, utilitza tecnologia d'accés distribuït o mitjançant CORBA i és capaç d'obtenir prestacions òptimes quan accedeix a sistemes d'arxius paral·lels o dispositius d'entrada/sortida per xarxa, tals com els HPSS (*High Performance Storage System*).

Enllaços d'interès

A http://www.globus.org/toolkit/about.html es pot observar l'estructura interna de Globus. El lloc web de The Globus Alliance és http://www.globus.org [9]. Aquí es poden trobar tant el codi font, com tota la documentació necessària per a transformar la nostra intranet com a part d'una graella (*grid*). Formar part d'una graella significa posar-se d'acord i adoptar les polítiques de totes les institucions i empreses que formen part d'ella. A

Espanya existeixen diferents iniciatives basades en Globus. Una d'elles és IrisGrid [13], a la qual és possible unir-se per a obtenir els avantatges d'aquesta tecnologia. Per a major informació, consulteu http://www.rediris.es/irisgrid.

Activitats

1. Instal·leu i configureu OpenMPI sobre un node; compileu i executeu el programa `cpi.c` i observeu-ne el comportament mitjançant xmpi.

2. Instal·leu i configureu OpenMP; compileu i executeu el programa de multiplicació de matrius (https://computing.llnl.gov/tutorials/openmp/exercise.html) en 2 nuclis i obtingueu proves de la millora en l'execució.

3. Utilitzant dos nodes (pot ser amb VirtualBox), instal·leu Ganglia i Cacti i monitoreu-ne l'ús.

4. Utilitzant Rocks i VirtualBox, instal·leu dues màquines per a simular un clúster.

5. Instal·leu una versió de Debian virtualitzat utilitzant Qemu.

Bibliografia

[1] **Barney, B.** *OpenMP*. Lawrence Livermore National Laboratory.
<https://computing.llnl.gov/tutorials/openMP/>

[2] *Beowulf Web Site*.
<http://www.beowulf.org>

[3] *Cacti*.
<http://www.cacti.net/>

[4] **Dietz, H.** (2002). *Linux Parallel Processing*.
<http://cobweb.ecn.purdue.edu/ pplinux/PPHOWTO/pphowto.html>

[5] *Extreme Linux Web Site*.
<http://www.extremelinux.info/>

[6] *FAQ OpenMPI*.
<http://www.open-mpi.org/faq/>

[7] **Foster, I.; Kesselmany, K.** (2003). *Globus: A Metacomputing Infrastructure Toolkit*.
<http://www.globus.org>

[8] *Ganglia Monitoring System*.
<http://ganglia.sourceforge.net/>

[9] *Globus5*.
<http://www.globus.org/toolkit/docs/5.0/5.0.2/>

[10] *GT5 Quick Start Guide*.
<http://www.globus.org/toolkit/docs/5.0/5.0.2/admin/quickstart/>

[11] *Guia d'instal·lació de Ganglia i Cacti*.
<http://debianclusters.org/index.php/Main_Page>

[12] *Guia ràpida d'instal·lació de Xen*.
<http://wiki.debian.org/Xen>

[13] **Martín Llorente, I.** *Estado de la Tecnología Grid y la Iniciativa IrisGrid*.
<http://www.irisgrid.es/>

[14] *MPI Examples*.
<http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples/main.htm>
(Download Section)

[15] *Mpich Project*.
<http://www.mcs.anl.gov:80/mpi/>

[16] *MPICH2 high-performance implementation of the MPI standard Mpich Project*.
<http://www.mcs.anl.gov/research/projects/mpich2/>

[17] *OpenMP Exercise*.
<https://computing.llnl.gov/tutorials/openMP/exercise.html>

[18] *OpenMPI*.
<http://www.open-mpi.org/>

[19] *Oracle (Sun) Grid Engine*.
<http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html>

[20] *Plataform LSF*.
<http://www.platform.com>

[21] *Qemu*.
<http://wiki.qemu.org/Main_Page>

[22] *QEMU. Guia ràpida d'instal·lació i integració amb KVM i KQemu*.
<http://wiki.debian.org/QEMU>

[23] **Radajewski, J.; Eadline, D.** *Beowulf: Installation and Administration*.

[24] *Rocks Cluster. Installation Guide*.
<http://www.rocksclusters.org/roll-documentation/base/5.4/>

[25] **Swendson, K**. *Beowulf HOWTO* (tlpd).

[26] *System-config-cluster (FC)*.
<http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual/cluster_Administration/index.html>

[27] *The Xen hypervisor*.
<http://www.xen.org/>

[28] **Wisconsin University**. *Condor Web Site*.
<http://www.cs.wisc.edu/condor>

[29] **Woodman, L**. *Setting up a Beowulf cluster Using Open MPI on Linux*.
<http://techtinkering.com/articles/?id=32>