Universitat de LLeida

Escola Politècnica Superior

Enginyeria Tècnica Informatica de Sistemes

Treball de final de carrera

Disseny i implementació d'un algorisme predictiu de coscheduling dins l'espai d'usuari en un cluster Linux

Autor: Bernat Loomans Guardia

Director: Francesc Giné de Solà

25 de Gener de 2007

Índex

1 Introducció **7**

1.1 Processament Paral·lel 7

1.2 Tècnica Coscheduling 9

1.3 Introducció al Sistema Operatiu Linux 10

1.4 Motivació i Objectius 11

1.5 Metodologia 12

2 Concepte Teòric Mètode Coscheduling **13**

2.1 CMC 13

2.1.1 Notació 14

2.1.2 Mètriques de rendiment 16

2.2 Planificador Local (LC) 17

2.3 Predictive Coscheduling 19

3 Implementació **21**

3.1 Captura de paquets 21

3.1.1 Comandes disponibles del S.O. 22

3.1.2 Sistema de Fitxers /proc 23

3.2 Gestors d'Aplicacions Distribuïdes PVM/MPI 25

3.2.1 PVM 25

3.2.2 MPI 26

3.3 Disseny de l'aplicació 27

3.3.1 Consideracions Prèvies 29

3.4 Busqueda dels fluxos de comunicació 30

3.4.1 Funció Bpids() 30

3.4.2 Funció Bsockets() 32

3.4.3 Funció isTCP() 34

3.5 Procés de lectura continuada [Funció funcioFil()] 35

3.5.1 Classe fil 39

3.6 Tractament del tràfic (Planificació de prioritats) 40

4 Experimentació **46**

4.1 Entorn Experimental 46

4.1.1 Execució 46

4.1.2 Aplicació Distribuïda/Local 47

4.2 Evolució de la implementació realitzada 48

4.2.1 Prioritat a assignar 49

4.2.2 Interval de lectures 49

4.3 Realització de les proves 51
4.3.1 Aplicació Distribuïda en Solitari 51
4.3.2 Aplicació Local en Solitari 53
4.3.3 Aplicació Distribuïda + Aplicació Local 54
4.3.4 Aplicació Distribuïda + Aplicació Local Parcialment 58
4.4 Conclusions de l'experimentació 61

5 Conclusió 63

A Implementació programa principal 64

B Funcions implementades 67

C Aplicacions externes 71
C.1 Aplicació Distribuïda *sintree_hc_hl.c* 71
C.2 Aplicació Local *carga.c* 74
C.3 Aplicació Local *calcul.cc* 76

D Bibliografia 77

Índex de figures

1	Arquitectures Computadors Paral·lels	7
2	Traffic sintree	48
3	Trafic Capturat vs Retard CPU	50
4	Aplicació Distribuïda de baixa intensitat	52
5	Aplicació Distribuïda d'alta intensitat	53
6	Aplicació Local 1	54
7	Aplicació Local 2	54
8	A. Distribuïda de baixa intensitat + A. Local de consum moderat	55
9	A. Distribuïda de baixa intensitat + A. Local d'alt consum	56
10	A. Distribuïda d'alta intensitat + A. Local de consum moderat	57
11	A. Distribuïda d'alta intensitat + A. Local d'alt consum	57
12	A. Distribuïda d'alta intensitat + 1/2 A.Local de consum moderat	58
13	A. Distribuïda d'alta intensitat + 1/2 A.Local d'alt consum	59
14	1/2 A. Distribuïda d'alta intensitat + 1/2 A. Local de consum moderat	60
15	1/2 A. Distribuïda d'alta intensitat + 1/2 A. Local d'alt consum	61

Lista de algoritmos

1 CMC Local Coscheduler (LC) 18
2 Algorisme Predictive Coscheduling (*PCA*). *S_C*$\equiv T[i].de < MCO$.C_C$\equiv h.freq > T[i].freq$ 19
3 Disseny de l'aplicació 27
4 procés fill capture(SOCK[]) 28
5 Disseny Bpids() 30
6 Implementació Bpids() 31
7 Disseny Bsocket() 33
8 Implementació Bsockets() 33
9 Implementació istcp() 35
10 Disseny funcioFil() 36
11 Implementació funcioFil() 37
12 Implementació Classe Intercanvi 39
13 Disseny Prioritats 41
14 Implementació assignació Prioritats 42
15 Implementació Re-lectura 44
16 Programa Principal 64
17 Assignació Prioritats 65
18 Implementació Re-lectura 66
19 Funció funcioFil() 67
20 Funció Bpids() 68
21 Funció Bsockets() 69
22 Funció istcp() 69
23 Classe Intercanvi 70
24 Inici de sintree_hc_hl.c 71
25 Continuació de sintree_hc_hl.c 72
26 Codi Fill sintree_hc_hl.c 73
27 carga.c 74
28 Funció carga_cpu() 75
29 Funció dream() 75
30 Calcul.cc 76

Resum

En aquest projecte s'utilitzen els recursos disponibles en un clúster no dedicat per realitzar processament paral·lel. La finalitat del projecte serà millorar el temps de comput del processament paral·lel de manera que el temps d'execució de les aplicacions paral·leles sigui el més ràpid possible.

S'ha desglossat el treball en cinc apartats: un primer apartat introductori;en el segon capítol es realitza una descripció teòrica dels algorismes usats;a continuació es descriu la implementació d'aquests algorismes i la plataforma on s'executa; en el quart capítol es realitza una descripció de les proves i resultats obtinguts per l'aplicació; i un últim capítol on s'exposen les conclusions que s'han pogut extreure del projecte.

En el primer apartat, per tant es descriu primerament que és un clúster i en que consisteix el processament paral·lel. Es fa una breu introducció sobre el sistema operatiu que s'ha emprat, i els motius i objectius que han dut a terme a la realització d'aquest treball.

Seguidament s'aprofunditza en la descripció teòrica de la tècnica de Coscheduling i concretament en la vesant "Predictive Coscheduling" i en els possibles algoritmes que conformen el model "Predictive Coscheduling" i paràmetres que s'han de tenir en compte alhora de dur a terme la implementació de l'aplicació.

En el tercer apartat es descriu la implementació de l'algorisme citat en l'anterior capítol, així com l'entorn que el sistema operatiu ens ofereix per controlar les comunicacions i poder assolir el nostre objectiu.

En el posterior apartat es detalla en els resultats obtinguts, s'estudien i es contrasten amb l'algorisme "Predictive Coscheduling" implementat al kernel en anteriors projectes.

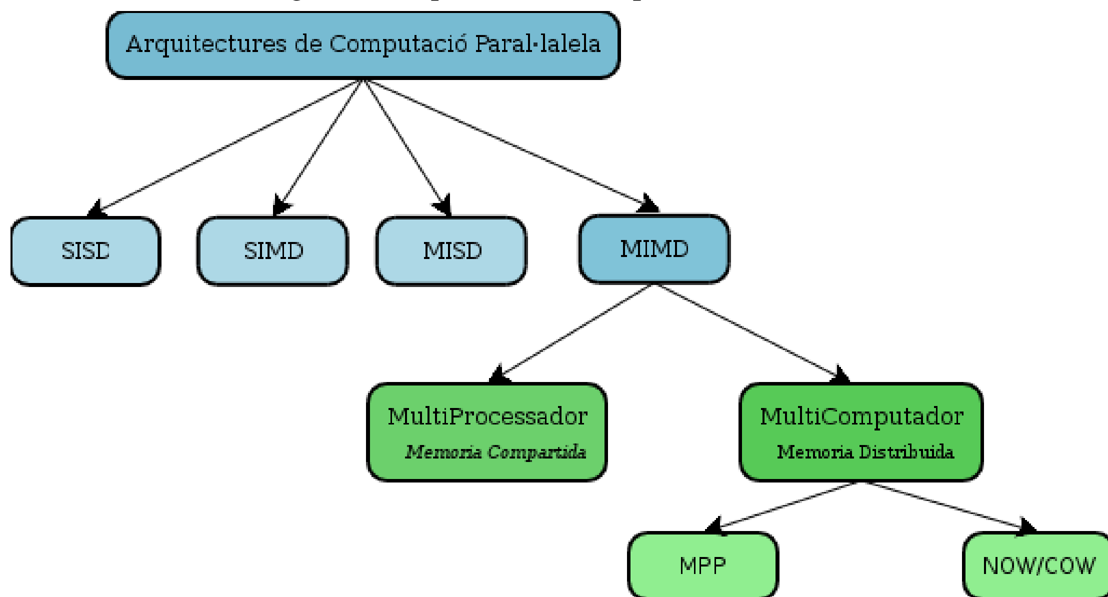
Per últim se'n extreu les conclusions que s'han obtingut amb la realització d'aquest projecte.

1 Introducció

1.1 Processament Paral·lel

Quan les necessitats de còmput són superiors a les que un ordenador convencional ofereix es pot recórrer al processament paral·lel per solventar la situació. El processament paral·lel es basa en la idea de la divisió de tasques, així una aplicació que realitza una serie de còmputs per a un resultat determinat es pot desglossar en diferents còmputs o tasques menors que unides obtinguin el mateix resultat, i cada una d'aquestes tasques o còmputs pot ésser realitzat de forma independent, sempre que es mantingui la cooperativitat, la interacció i la sincronització, en la mesura possible entre totes les tasques. Dins del processament paral·lel s'han anat descobrint vàries arquitectures de còmput possibles i diferents mètodes sota aquestes arquitectures per a poder realitzar les tasques. La taxonomia fixada per Tanenbaum a [13] és la següent:

Figura 1: Arquitectures Computadors Paral·lels



El primer nivell esta basat en la classificació de Flynn definida en [14] sobre les diferents arquitectures de computadors, usant els conceptes de fluxos d'instruccions i de Dades com a base per la classificació. Així per exemple SISD(*Single Instruction stream Single Data stream*) correspon a la clàssica maquina de Von Newman.

En el nostre cas ens centrem en les MIMD (*Massive Instruction stream Massive Data stream*), que implica la possibilitat de vàries instruccions executant-se simultaneament(paral·lelament) usant varis fluxos de dades.

Dins de l'arquitectura MIMD s'hi classifiquen dos sub-tipus, segons la implementació dels fluxos de memòria que disposen, els multiprocessadors i els multicomputadors. Els multiprocessadors es basen en un conjunt de processadors que disposen d'una memòria compartida implementada. L'altre grup son els multicomputadors, els quals no disposen d'una memòria compartida sinó distribuïda, es

a dir es comuniquen mitjançant el pas de missatges. En el nostre cas ens basem en aquest segon grup, els multicomputadors, alhora divisibles en dues arquitectures diferents.

Una opció son els MPPs (Massive Parallel Processor), que consisteix en una serie d'elements de processament (EP) que interconectats entre si formen un sistema homogeni i totalment dedicat a la computació paral·lela. L'altra opció son les anomenades NOWs (*Networks of Workstations*) o COW (*Cluster of Workstations*), que consisteix en aprofitar el rendiment d'un conjunt de maquines per un propòsit comú.

Els MPPs impliquen un important augment dels costos en la millora del equipament. Un MPP consta d'un gran nombre d'Elements de Processament (EP) interconectats per una xarxa pròpia d'alta velocitat, que assoleixen una alta capacitat de processament. La major part dels supercomputadors utilitzen aquest sistema per a resoldre grans problemes o còmputs complexes.

Els MPPs tenen dos grans inconvenients, el primer és el ja anomenat augment de costos. I l'altre gran inconvenient és que la maquinaria acostuma a quedar obsoleta en poc temps. I aquesta segona des-avantatja es la que pot convertir en avantatja el segon sistema de processament paral·lel.

Les NOWs en canvi intenten assolir un rendiment molt similar als MPPs però amb una reducció important de costos respecte a les anteriors i sense l'inconvenient de quedar obsoleta ja que són fàcilment ampliables.

Es tracta d'un conjunt de màquines heterogènies de baix cost que estan interconectades en xarxa i que pretenen treballar conjuntament per un mateix resultat. Cada una d'aquestes maquines rep una part del processament i es comunica amb la resta de maquines per estar coordinades en tot moment, així assoleixen un processament paral·lel de la aplicació/comput.

La dedicació de les NOWs pot ser exclusiva per a les aplicacions paral·leles (dedicats) o bé compartida amb els usuaris locals de cada node (no dedicat), a diferencia dels MPPs els quals sempre són dedicats. Una NOW no dedicada implica la possibilitat de compartir els recursos amb aplicacions locals quan els usuaris ho requereixen. Això fa que els algorismes de planificació entre sistemes dedicats i no dedicats siguin diferents.

Un dels majors inconvenients de les NOWs respecte als MPPs és la velocitat de comunicació entre nodes, molt més lenta que la interconnexió dels EPs d'un MPP. Tot i així es pot suplir incorporant nous nodes i augmentant així el rendiment mantenint encara un cost rentable, o la mateixa utilització d'algorismes de planificació, que pretenen minimitzar aquesta mancança reduint el temps de comunicació entre tasques ubicades en diferents nodes.

Per a treballar en aquests clústers s'utilitzen diferents llibreries de programació existents per facilitar-ne la programació, la crida i mantenir una certa estandardització. Els més comuns són els sistemes MPI[12] i PVM[11]. Ambdós estan estandarditzats i són de lliure distribució.

PVM es desenvolupa basant-se en el concepte de màquina virtual (Parallel Virtual Machine), es a dir, una col·lecció de recursos computacionals usats com a un sol computador paral·lel. MPI en canvi es centra en el pas de missatges (Message Passing Interface), tot i no tenir el concepte de màquina virtual, si fa una abstracció de tots els recursos en termes de topologia de pas de missatges.

Tot i que fins ara PVM havia estat la biblioteca estàndard per a computació distribuïda, cada

vegada mes, s'està implantant la llibreria MPI, convertint-se en el nou estàndard d'aquest tipus de programació.

1.2 Tècnica Coscheduling

Com ja s'ha avançat en l'apartat anterior, les NOWs comparteixen els recursos dels equips amb els usuaris locals que puguin estar presents i, per tant, amb els processos que puguin estar executant. Aquesta condició ens porta a crear una serie d'algorismes de planificació de recursos per a coordinar els recursos entre les tasques paral·leles i les locals i que alhora existeixi una certa coordinació entre els diferents nodes que conformen la NOW. Donat que cada equip disposarà d'una càrrega de comput local diferent s'ha d'intentar mantenir un progrés uniforme entre les tasques paral·leles, es a dir mantenir una sincronització entre les tasques que formen un treball paral·lel. Aquesta és la finalitat del Coscheduling.

Les tècniques Coscheduling pretenen minimitzar el temps d'espera dels events de comunicació i sincronització, entre les tasques d'una mateixa aplicació distribuïda. Si be aquestes tècniques es poden classificar en dos grans grups:

- **Tècnica Coscheduling de control explicit[9]:** Aquesta tècnica es basa en planificar simultaneament totes les tasques d'una mateixa aplicació paral·lela mitjançant un canvi global de context a tot el clúster. Tot i que aquesta tècnica assegura una alta velocitat en aplicacions distribuïdes amb una alta càrrega de transit de missatges, aporta un retard, a causa de l'overhead necessari per la sincronització, tant a les aplicacions locals com a les aplicacions distribuïdes sense altes necessitats de comunicació.

  Una derivació d'aquesta tècnica es el *Buffering Coscheduling*[10]. Diferenciada de la tècnica anterior pel fet d'emmagatzemar temporalment els missatges, per ser lliurats al destinatari posteriorment, en comptes de enviar-los sincronitzada-ment amb les aplicacions distribuïdes. Això redueix la necessitat de re-planificar les tasques que es comuniquen.

- **Tècnica Coscheduling de control implícit:** En aquest cas cada node planifica les taques distribuïdes, que pertoquen al propi node, d'acord amb les necessitats de comunicació i sincronització.

  Demand-based Coscheduling és un algorisme introduït per Solbalvarro en [8] que explota la comunicació entre processos per deduir quins processos han de ser planificats i aplicar-los-hi Coscheduling. Això es efectiu gràcies a que únicament els processos comunicatius han de ser planificats. Dues variants d'aquest algorisme son "Dynamic Coscheduling" i "Predictive Coscheduling".

  Dynamic Coscheduling[15] és un tipus de Coscheduling baixa demanda (demand-based)on la planificació es realitza directament amb l'arribada dels missatges de comunicació. Es podria definir com la variant aplicada de "demand-based Coscheduling".

Mentre que *"Predictive Coscheduling"[3, 4]* es basa en planificar al mateix temps els processos que més recentment s'han de comunicar. El qual implicaria conèixer per anticipat el conjunt de tasques per a poder-les planificar simultaneament. Donat que aquesta clàusula es molt costosa i difícil d'aconseguir, la selecció de processos a cada node es realitza prioritzant els processos amb més alta freqüència de comunicació.

Un altra característica d'aquest últim algoritme es que té en compte tant el tràfic de sortida com el de recepció. Ja que existeix la teoria que les tasques que tenen missatges a les cues de recepció i enviament tenen més necessitat de ser planificades que les que únicament tenen comunicació en un sentit. Al contrari que la tècnica Dynamic Coscheduling la qual únicament té en compte els missatges de recepció.

1.3 Introducció al Sistema Operatiu Linux

La computació paral·lela no esta limitada a un únic sistema operatiu, inclús es possible disposar d'una NOW on els nodes es trobin sota sistemes operatius diferents. Per exemple mitjançant PVM podem tenir una mateixa maquina virtual amb equips usant Windows i Linux al mateix temps.

Però nosaltres hem escollit el sistema operatiu Linux per a treballar-hi per les seves nombroses avantatges.

Linux és un sistema operatiu creat per Linus Torvalds, el qual parteix com a una adaptació del sistema operatiu Unix al seu PC. Va començar explotant únicament el xip Intel 386 però avui en dia esta adaptat a quasi totes les plataformes. El codi es obert, això vol dir que qualsevol pot utilitzar-lo i modificar-lo per tal de millorar i contribuir en el desenvolupament d'aquest sistema operatiu i, en la majoria de distribucions, de lliure distribució.

El fet de que Linux sigui GNU (General Public License) i de codi obert implica una facilitat addicional alhora de controlar el sistema operatiu o modificar-lo per al nostre propòsit concret, amb la possibilitat inclús, de distribuir posteriorment el nostre nou sistema operatiu modificat, sempre i quan mantinguem la llicencia GNU.

Linux es un sistema operatiu multi-usuari i multi-tasca.

Multi-usuari implica que varis usuaris diferents poden estar utilitzant simultàniament el sistema operatiu, conservant la privacitat de la seva informació. Cada usuari disposa de la seva identificació pròpia, espai de treball i privilegis diferenciats. Cada un dels usuaris està diferenciat amb un número d'identificació anomenat UID.

Multi-tasca significa que poden haver-hi varis programes en funcionament simultàniament i el propi sistema operatiu s'encarrega de distribuir els recursos de l'equip entre aquests programes. Aquesta distribució es realitza mitjançant una arquitectura de prioritats, que poden ésser consultades i modificades en qualsevol moment si es disposa dels permisos necessaris.

En un sistema operatiu multi-tasca com Linux necessitem identificar cada programa amb una identificació única, factor que, a priori, no es compleix ja que s'identifica mitjançant el nom del programa. Així doncs, els programes passen a ser processos quan son carregats a memòria per executar-se. A

cada procés li correspon un numero identificatiu anomenat PID (*Process Identifie*r) que l'identifica singularment, complint d'aquesta manera la condició d'identificació única.

Un de les avantatges importants d'aquest sistema operatiu es la informació que ofereix sobre tots els events que succeeixen en el transcurs de les execucions. Aquesta informació pot resultar summament útil alhora d'observar o controlar els anomenats PIDs u obtenir l'estat sobre aquests.

El sistema operatiu compta amb un seguit de comandes que faciliten el treball alhora de configurar, consultar o modificar aspectes u processos que composin el s.o. Així per exemple disposa de comandes per canviar la prioritat dels processos, comanda realment útil com es podrà veure més endavant, entre d'altres.

També disposa d'un sistema de fitxer dissenyat per proporcionar informació a les pròpies comandes anteriorment citades o a les aplicacions que ho desitgin, essent aquesta informació accessible per a qualsevol usuari en forma de directori, composat d'una serie de fitxers destinats a informar sobre un aspecte específic del s.o..

Així doncs es tracta d'un s.o. obert que proporciona un alt ventall de facilitats per al control d'aquest i ens ofereix tota la informació desitjada sobre els elements que composen la pròpia màquina.

1.4 Motivació i Objectius

El que ens ha dut a realitzar aquest projecte és la voluntat de crear un mòdul en l'espai d'usuari del s.o. Linux que pogués implantar o adaptar un algorisme de planificació "Predictive Coscheduling" en un clúster no dedicat.

L'algorisme de planificació "Predictive Coscheduling" busca,com ja hem dit en l'apartat anterior, la màxima priorització per a les aplicacions paral·leles quan aquestes requereixen comunicar-se amb la resta d'equips que conformen el clúster.

En anteriors projectes[3, 4] es va implementar l'algorisme de planificació "Predictive Coscheduling" en el kernel del sistema operatiu,mentre en aquest el que es pretén és implementar-lo en l'espai d'usuari. El Kernel és el nucli del Linux, es podria definir com el motor d'aquest sistema operatiu. Tot mòdul software que estigui present en el sistema estarà controlat i dirigit per el kernel, així que sembla obvi que sigui el lloc idoni per integrar-lo.

Tot i que aquesta implementació al kernel pot resultar ser la òptima a nivell de eficiència no ho és a nivell de portabilitat. Donat que quan es modifica un kernel es modifica el nucli del sistema operatiu i tal modificació implicarà reconstruir el nostre sistema, es a dir re-compilar el modul dins del nou el kernel. Aquesta última condició és costosa en termes de temps i requereix uns certs coneixements del sistema operatiu Linux.

Deixant de banda els costos que pot suposar el re-compilar el kernel, el problema ve quan la versió de kernel s'actualitza, cosa que acostuma a passar cada pocs mesos. Així doncs, cada vegada que hi hagués un canvi en la versió de kernel s'hauria d'adaptar la implementació al nou nucli, ja que és possible que es variï algun apartat que afectaria al nostre algorisme de planificació.

Optem doncs per la implementació a nivell d'aplicació o usuari. D'aquesta manera l'aplicació és

portable a totes les versions de kernel i independent a les diferents distribucions.

Un dels majors desavantatges de controlar i prioritzar les aplicacions paral·leles mitjançant una aplicació pròpia es l'ús de CPU que s'afegeix degut a l'execució d'aquesta, per tant, el retard que afegim tant a les aplicacions paral·leles com a les locals. Això comporta que una de les fites a assolir ha d'ésser minimitzar aquest retard.

Així doncs el que ens motiva a realitzar en aquest projecte és que l'implementació de l'algorisme Predictiu a nivell d'aplicació pugui assolir un rendiment similar o superior al que l'algorisme de planificació "Predictive Coscheduling" implementat en el kernel ha arribat, treballant sota el sistema operatiu Linux.

1.5 Metodologia

La finalitat d'aquest projecte és implementar i avaluar un algorisme de planificació "Predictive Coscheduling" implementat en l'espai d'usuari d'un sistema operatiu Linux.

Per implementar l'algorisme s'ha fet ús de la informació que ens dona el propi sistema operatiu Linux sobre el tràfic de xarxa que manté, sospesant les vàries opcions disponibles. S'ha intentat aconseguir un equilibri entre eficiència i temps de retard ja que a l'implementar-ho a nivell d'aplicació i no a un nivell inferior, com es el cas de la implementació en el kernel, afegim càrrega de comput a la CPU.

Una vegada s'ha controlat el tràfic de xarxa hem implementat l'algorisme de planificació adaptant-lo també a les nostres circumstancies i buscant sempre l'equilibri entre eficiència i retard o càrrega de comput afegida.

Un cop implementada l'aplicació s'ha avaluat, executant-lo dins d'un clúster amb una serie de nodes que utilitzen Linux i realitzant les pertinents proves, variant el nombre de missatges enviats per l'aplicació paral·lela, el nombre d'execucions locals que hi actuaven simultaneament, etc.

També s'ha comparat aquesta implementació amb la implementació ja existent en el kernel del algorisme "Predictive Coscheduling", realitzada en un projectes anteriors.

2 Concepte Teòric Mètode Coscheduling

Mitjançant la tècnica Coscheduling el que es pretén es planificar els recursos a favor de les tasques paral·leles que major comunicació han de realitzar, treballant, en el nostre cas, dins d'un Cluster no-dedicat o NOW.

Dins del model teòric Coscheduling hi han vàries variants, tal com s'han descrit en l'apartat 1.4. En el nostre cas ens hem centrat en el mètode "Predictive Coscheduling", molt semblant al Dynamic Coscheduling[15], ambdós variants del demanded-based Coscheduling de Solbalvarro[8]. Es tracta d'un mètode que s'engloba dins la tècnica de control implícit de Coscheduling, definida breument en l'apartat 1.4.

Per a poder realitzar aquesta tècnica de planificació ("Predictive Coscheduling") correctament el que hauria de succeir, tal com indica el seu nom, es que s'hauria de conèixer anticipadament les comunicacions que s'han de realitzar. Actualment en clústers les tasques de les que estan formades les aplicacions distribuïdes són executades com a tasques locals en els diferents nodes que conformen la NOW, per tant es molt complicat fer suposicions inicials. Donat que cada aplicació distribuïda es comporta de forma singular i cada node pot disposar d'un sistema diferent, tan pel que respecta a software com hardware, això fa pràcticament impossible determinar el comportament de l'aplicació paral·lela, dada la gran quantitat de factors a tenir en compte per a poder tenir una certa exactitud en la predicció. El model Coscheduling opta per una via alternativa. Planificar els recursos en funció dels events locals, que es podria traduir com la freqüència d'enviament i recepció de paquets, tenint en compte tant la comunicació actual com la acumulada fins al moment, per part de les pròpies tasques paral·leles (a diferencia de la tècnica Dynamic Coscheduling la qual només té en compta els missatges de recepció).

2.1 CMC

La tècnica Predictive esta formalitzada per la definició i disseny d'un model per a sistemes Clúster, anomenat CMC (*Coscheduling Model for non-dedicated Clústers*)[1]. És a dir que un algorisme "Predictive Coscheduling" PCA (*Predictive Coscheduling Alorithm*) està basat en el model CMC.

Aquest model contempla vàries assumpcions a tenir en compte:

- No es necessari que tots els nodes del clúster no-dedicat o NOW (*Network of Workstations*) es trobin sota el control centralitzat d'un sistema Coscheduling, a diferència del que trobem en un sistema explicit Coscheduling.

- Les aplicacions distribuïdes estan composades d'un seguit de tasques mapejades i executades en els diferents nodes del clúster.

- Actualment en Clústers les aplicacions distribuïdes son llençades en cada node com a simples tasques locals. Per aquesta raó no es realitzen assumpcions inicials sobre el comportament de les diferents tasques distribuïdes de cada node que composen l'aplicació distribuïda.

- Cada node del Clúster es suposa mono-processador i treballant amb un sistema operatiu de temps compartit que nomes conté una sola cua de preparats per a execució i una política de planificació del tipus *appropiative Round-Robin* (R-R) amb un TS (*Time Slice*), i una demanda de cicles d'execució de CPU variable per a cada tasca. Aquesta política de planificació coincideix amb la major part dels sistemes operatius de temps compartit real (p.e. Linux, Solaris,...).

El model CMC aporta una sèrie de característiques molt interessants:

Aplicacions simultànies: Suporta múltiples execucions d'aplicacions distribuïdes simultaneament.

Identificació dinàmica dels processos distributius: Els processos de que tenen necessitats de comunicació, i per tant coplanificació, són identificats en temps d'execució.

Planificació coordinada: Els processos mapejats en diferents nodes que es comuniquen entre sí són planificats quasi simultaneament al llarg dels diferents equips, complint el seu objectiu. La coordinació dels processos hauria de minimitzar el temps d'espera dels missatges.

Manteniment local: El sistema de Coscheduling no hauria degradar excessivament l'execució de les tasques locals o d'usuari del node. Tanmateix, el temps de resposta de les aplicacions locals interactives no descendeix dramàticament.

Autonomia: Cada node manté el control sobre sobre les seves pròpies accions, participant en el sistema de forma no-imposada.

Reconfiguració: Els equips poden incorporar-se i excloure's del Clúster dinàmicament sense necessitat de re-iniciar el sistema.

Fiabilitat: La fallida d'un dels nodes qualsevol no afecta a la permanència del sistema.

2.1.1 Notació

Definim l'algorisme general CMC utilitzant la notació adequada per entendre la implementació pseudo-llenguatge.

En tota màquina que es trobi sota un sistema operatiu de temps compartit es pot trobar una mateixa tasca en diferents estats: *ready*, executant-se; *ready to run*, preparada per ser executada; *blocked*, bloquejada; *etc*. Una tasca *ready* l, que es troba dins de RQ (*Ready Queue*), es denota com a T[l]. Per contra, si la tasca l no esta preparada (*ready*), serà referenciada com a l, sense el prefix T[]. El mètode usat per identificar singularment una tasca sense dependre del ser estat, es utilitzant l'identificador de tasca (*tid*). Així doncs, es pot afirmar que T[l] i h son la mateixa tasca si es compleix *tid*(T[l])=*tid*(h).

A continuació es defineixen algunes notacions bàsiques relacionades amb l (tenint en compte que el temps es compta en cicles):

- $T[l]$: Tasca l , tant local com distribuïda, de la cua RQ. Casos singulars dins de RQ són les tasques on l=0 (primera de la cua), tasca que està executant-se a la CPU, i $l=\infty$ (última o fons de la cua), l'última tasca que serà executada.

- $T[l].c$ ($T[l].tc$): -*c* cycles; -*tc* total cycles : Cicles executats per la tasca l des de l'últim cop que ha entrat a la cua RQ (*c*), o des de l'inici de la seva execució (*tc*).

- $T[l].pco$: -*pco* potential Coscheduling: Variable booleana que informa sobre el potencial Coscheduling de la tasca l. Quan una tasca entra a RQ i aquesta està comunicant en freqüència major a 0, el camp *pco* s'activa. Això indica que la tasca és una candidata pel Coscheduling, i per tant ha de ser planificada en quan es pugui.

- $T[l].co$ ($T[l].tco$): -*co* Coscheduling; -*tco* total Coscheduling: Nombre de cicles planificats (=cicles executats) de la tasca l a partir de que es trobat el potencial Coscheduling (el camp *pco* ha estat activat a l'inserció dins de RQ), des de l'última vegada que la tasca ha entrat a RQ (*co*), o des de l'inici de l'execució (*tco*).

- $T[l].th$ ($T[l].tth$): -*th* trashing; -*tth* total trashing: Nombre de cicles Coscheduling perduts (cicles no executats) de la tasca l des de que es trobat el potencial Coscheduling, des de l'última vegada que ha entrat a la cua RQ (*th*), o des de l'inici de l'execució (*tth*).

- $T[l].de$: -*de* delay: Nombre de vegades que la tasca ha estat sobrepassada dins la cua RQ per un altra tasca a causa del Coscheduling, des de l'últim cop que ha entrat dins la cua RQ.

- $T[l].d$ ($T[l].td$): -*d* delay; -*td* total delay: Nombre de cicles de la tasca l retardats a causa del Coscheduling, des de l'últim cop que ha entrat a la cua RQ (*d*), o des de l'inici de la seva execució (*td*).

- $T[l].ovt[MCO]$: Array (vector) de *MCO*(Maximum number of Coscheduling Overtakings) identificadors de tasca (*tid*). Quan una tasca h adelanta una tasca l (causat pel Coscheduling) dins la cua RQ, el corresponent identificador d'h ($tid(h)$) és guardat en un dels camps *MCO* de l.

Si una tasca no està preparada, nomes els camps *tc*, *td*, *tco* i *tth* són emprats, amb el propòsit de emmagatzemar la informació sobre tota la execució, no només des de l'última vegada que ha entra entrat dins la cua de preparats RQ.

En cas tenir un Clúster C, composat per n nodes ($C=\{N[k]\}$, k=1.. n); per referir-se al camps *tc* de la tasca l d'un node k, s'utilitza la notació $N[k].T[l].tc$ per una tasca $T[l]$ preparada (*ready*), i $N[k].l.tc$ en cas de que no ho estigui.

En aquest model CMC es suposa que els missatges d'entrada (i sortida) cap a un Clúster node $N[k]$, són emmagatzemats en una cua de missatges, RMQ (*Receiving Message Queue*) per als missatges d'entrada i SMQ (*Sending Message Queue*) per als de sortida. De fet tenim unes cues homònimes en Linux a nivell de kernel anomenades *receive_queue* i *write_queue* que emmagatzemen tots els

paquets d'entrada i sortida respectivament. L'opció d'escollir a quin nivell volem aplicar la tècnica Coscheduling dependrà de l'implementació i no del model en sí.

L'algorisme "Predictive Coscheduling" està basat en calcular les freqüències de comunicació de les tasques. Tenint en compte l'existència dels *buffers* (cues de missatges) anteriorment citats, la següent informació pot ésser obtinguda, i usada posteriorment en els algorismes, per a una tasca *l*:

- $T[l].cur_freq_r$: Actual freqüència de missatges de recepció.
- $T[l].cur_freq_s$: Actual freqüència de missatges d'enviament.
- $T[l].freq_r$: Anterior freqüència de missatges rebuts.
- $T[l].freq_s$: Anterior freqüència de missatges enviats.
- $T[l].freq$: Freqüència de missatges rebuts i enviats.

2.1.2 Mètriques de rendiment

Per definir les mètriques de rendiment en aquest apartat s'utilitza la notació CMC descrita en l'apartat immediatament anterior. Aquestes mètriques poden ser utilitzades per mesurar el rendiment d'un algorisme Coscheduling, o en particular d'un algorisme "Predictive Coscheduling" com es proposa més endavant.

- Task Delay ($TaskD(T[l])$): Informa del temps de retard introduït a les tasques a causa de la política Coscheduling.

  $$TaskD(T[l])=T[l].td$$

- Node Delay ($NodeD(N[k])$): Retard afegit al node *k*.

  $$NodeD(N[k])=\sum_{l} T[l].td$$

- System Delay (*SystemD*): Retard afegit a tot el sistema (tot el clúster).

  $$SystemD=\sum_{k}\sum_{l}(N[k].T[l].td)$$

- Task Coscheduling Degree ($TaskCoDe(T[l])$): Proveeix informació sobre el bon ús de la tècnica Coscheduling. Es defineix com la relació entre el total de cicles Coscheduling ($T[l].tco$) des de que s'activa el camp $T[l].pco$ i tots els cicles Coscheduling que podien haver estat possibles ($T[l].pco$ + $T[l].tth$) per la tasca $T[l]$.

  $$TaskCoDe(T[l])= \frac{T[l].tco}{T[l].tco+T[l].tth}$$

- Node Coscheduling Degree ($NodeCoDe(N[k])$): És la aproximació de la mètrica *TaskCoDe* de totes les tasques d'un node.

$$NodeCoDe(N[k]) = \frac{\sum lT[l].tco}{\sum l(T[l].tco+N[k].T[l].tth}$$

- System Coscheduling Degree (*SystemCoDe*): Relació entre els cicles executats en mode Coscheduling (*coscheduled*) i tots els que podrien haver executat en tot el sistema.

$$SystemCoDe=\frac{\sum k \sum lN[k].T[l].tth}{\sum k \sum l(N[k].T[l].tco+N[k].T[l].tth}$$

- Task Trashing (*TaskThDe*$(T[l])$): Informa sobre el mal ús de la tècnica Coscheduling, éssent la relació entre els cicles que no s'han executat en Coscheduling i tots els que podrien haver-ho estat per a una tasca *l*. Es per tant el contrari de *TaskCoDe*.

$$TaskThDe(T[l]) = 1 - TaskCoDe(T[l])$$

- Node Trashing (*TaskThDe*$(T[l])$): Relació entre els cicles que no han estat executats usant la tècnica Coscheduling i tots els que podrien haver-ho estat per a un node *k*. El definim doncs com el contrari de *NodeCoDe*.

$$NodeThDe(N[k]) = 1 - NodeCoDe(N[k])$$

- System Trashing (*TaskThDe*$(T[l])$): Informa sobre el pobre ús de la tècnica Coscheduling, essent aquesta vegada la relació entre els cicles que no s'han executat en Coscheduling i tots els que podrien haver-ho estat per a tot el sistema. Es doncs el contrari de *SystemCoDe*.

$$SystemThDe = 1 - SystemCoDe$$

Un apunt a fer referència és que aquestes mètriques no consideren el temps consumit per els processos en espera d'algun event, normalment en estat *bloqued* (bloquejat). El rendiment del Coscheduling depèn principalment d'una bona elecció de les tasques que han de ser tractades per la CPU en cada pas de la planificació. Aquells processos bloquejats com poden succeir en l'espera de missatges, I/O(entrada i Sortida) o sincronització d'events per exemple, són descartats ja que depenen de molts factors externs, principalment limitats pels recursos de hardware.

2.2 Planificador Local (LC)

Aquest algorisme s'aproxima molt al planificador Local real d'un sistema operatiu de temps compartit. Realitza les mateixes funcions que un planificador típic d'un *o.s.* de temps compartit, però amb capacitats de Coscheduling. Les funcions que no influencien el nostre model no han estat considerades.

El següent algorisme mostra en pseudo-codi el *Round-Robin Local Coscheduler* (*LC*) proposat per un sistema operatiu (*o.s.*) de temps compartit d'un Clúster node $N[k]$.

L'algorisme *LC* únicament funciona si la cua RQ no està buida ($T[0] \neq NULL$). Primerament assigna la primera tasca a la CPU (línia 3). Una vegada l'execució ha finalitzat, s'ha expirat el seu

Algorithm 1 CMC Local Coscheduler (LC)

```
1  do forever
2    if(T[0] ≠ NULL)
3      dispatch(T[0]);
4    ACCOUNTING;
5      T[0].{c, tc}+ = exe_time;
6      if(T[0].pco)T[0].{co, tco}+ = exe_time;endif;
7      for(i = 0; T[i + 1] ≠ NULL; i + +)
8        if(T[i + 1].pco)T[i + 1].{th, tth}+ = exe_time;endif;
9        if(T[i + 1].ovt[j] == tid(0)for any j = 0..MPO − 1)
10         T[i + 1].{d, dt}+ = exe_time;
11       endif;
12     endfor;
13     T[0].d = 0; T[0].c; T[0].co = 0; T[0].th = 0; T[0].de = 0;
14     for(j = 0; j < MPO; j + +)T[0].ovt[j] = NULL;endfor;
15     if(T[0].cur_freq == 0)T[0].pco = false;edit;
16     delete_RQ(T[0]); insert_RQ(T[0]);
17   endif;
18 enddo;
```

Time Slice, o es reemplaçada per un altra tasca, l'execució continua per la línia 4 (*ACCOUNTING*), on es realitza un recull de dades útils per al Coscheduling de la tasca (línies 5-15).

En la línia 5, el temps d'execució (*exe_time*) és afegit a la sortida de la CPU de la tasca en qüestió. Aquesta informació ja és generalment afegida per un planificador d'un *o.s.* de temps compartit. A partir de la línia 6, on es comptabilitza el Coscheduling, el bucle en el que entra (línies 7-12) realitza les modificacions necessàries als camps de retard o cicles perduts de la tasca.

En les següents tres línies s'inicialitzen els camps *d,c,co,th* i *de* per a aquells camps que requereixen comptabilitzar la informació "a partir de l'última inserció de tasca a la cua RQ".

Per últim a la línia 16 realitzem *delete_RQ*($T[0]$) i *insert_RQ*($T[0]$). Primerament s'esborra la tasca $T[0]$de la cua RQ, i posteriorment *insert_RQ*($T[0]$) indica que la primera tasca $T[0]$ es re-inserida a RQ en concordança amb l'algorisme PCA (*Predictive Coscheduling Algorithm*). D'aquesta manera, una petita variació de la planificació *Round-Robin* és implementat. Aquest moviment dins de RQ es realitza quan $T[0]$ no ha completat el temps d'execució requerit.

A continuació es mostra una taula amb les inicialitzacions que cal efectuar sobre les tasques que corresponen al CMC. Tots aquests camps s'han d'inicialitzar en el moment en que la tasca es crea.

Taula 1: Inicialitzacions dels camps de les variables CMC i global

$N[k].T[l]$												*glob. var.*
c	*tc*	*pco*	*co*	*tco*	*th*	*tth*	*de*	*d*	*td*	*ovt*[0]	*ovt*[1]	*MPO*
0	0	false	0	0	0	0	0	0	0	0	0	2

2.3 Predictive Coscheduling

En aquesta secció es proposa un Algorisme Predictive Coscheduling (*PCA*).

Algorithm 2 Algorisme Predictive Coscheduling (*PCA*). *S_C*≡ $T[i].de < MCO$.C_C≡ $h.freq > T[i].freq$

```
1  insert_RQ(task h)
2  INICIALIZATION
3  if(h.cur_freq ≠ NULL)h.pco = true;endif;
4  if(T[0] == NULL)insert(h, 0);endif;
5  else;
6     i = bottom;
7     while((S_C)and(C_C)and(i ≠ −1))
8        T[i].ovt[de] = tid(h);
9        T[i].de + +;
10       i − −;
11    endwhile;
12    if(i ≠ −1)insert(h, i);endif;
13    else context_switch(h, 0);endelse;
14 endelse;
```

L'algorisme PCA ha estat implementat dins de la rutina genèrica *insert_RQ*. Aquesta rutina fou escollida per implementar "Predictive Coscheduling" perquè totes les tasques han de passar-hi quan es troben en l'estat *ready to run* (preparat per a ser executat) abans d'ésser planificades.

La part *INICIALIZATION* és on s'efectuen les diferents inicialitzacions son efectuades. Es necessari recordar que la rutina original *insert_RQ* conté únicament una línia, *insert*(h, ∞), que vindria a ser la inserció de la tasca *h* al cap damunt de la cua RQ.

En les línies 5-14 s'observa com es modifica la posició de la tasca *h* dins la cua RQ tan amunt com ho permetin dues condicions, la *STARVATION* i *COSCHEDULING CONDITIONS* (*S_C* i *C_C*). En la línia 8, $tid(h)$ es guarda en un camp *ovt* de cada tasca adelantada, i en la línia 9, el camp de retard (de) de les tasques adelantades es incrementat.

*STARVATION CONDITION (*S_C*)*: Condició equivalent a $T[i].de < MCO$. MCO(*Maximum Number of Predictive Overtakes*) es defineix com al nombre màxim de tasques sobrepassades a causa de la política Coscheduling. L'objectiu d'aquesta condició es evitar la inanició (*STARVATION*) de les tasques locals. Per aquest motiu, la inserció de la tasca *h* sobrepassa aquelles on el seu camp *de* $(T[i].de)$ és més baix que la variable Coscheduling global *MCO*.

COSCHEDULING CONDITION (*C_C*): Condició equivalent a $h.freq > T[i].freq$, on $T[i].freq$ indica la freqüència de missatges adreçats a la tasca $T[i]$. L'objectiu és incrementar la prioritat de planificació de les tasques d'acord amb la seva freqüència de comunicació. És a dir, aquelles tasques que tinguin una major freqüència de missatges enviats/rebuts se'ls hi assignara una major prioritat de planificació. Tenint en compte això les freqüències d'enviament $(h.freq_s)$i de recepció $(h.freq_r)$ d'una tasca *h* estan definides de la següent forma:

$$h.freq = P \star h.freq + (1 - P) \star h.cur_freq$$

On *P*, valor el qual es conté entre 0 i 1, defineix la relació entre les freqüències de comunicació actual i anterior, *P* és el percentatge assignat a la freqüència d'enviament i recepció anterior i $(1 - P)$ és per tant l'actual. Tenint en compte la següent equivalència:

$$h.cur_freq = h.rm + h.sm \text{ si es compleix } ((h.rm \neq 0) \text{ i } (h.sm \neq 0))$$

On *h.rm* i *h.sm* son les freqüències de missatges rebuts i enviats per la tasca *h* respectivament.

Fins aquí la definició del mètode "Predictive Coscheduling", definició aplicada al nivell més baix del sistema operatiu, a nivell de kernel. Donat que el nostre projecte el que pretén és implementar el mètode "Predictive Coscheduling" a nivell d'aplicació,és a dir sense interferir en el kernel, això implicarà no poder accedir directament a la cua RQ. El que si es pot fer és influir indirectament en la decisió de quines tasques han escalar posicions dins la cua RQ, mitjançant la prioritat de la tasca. Es pot dir per tant que utilitzarem l'algorisme de Planificació Local però controlant la prioritat de cada tasca des de la nostra aplicació, que implementarà l'algorisme "Predictive Coscheduling".

En el pròxim apartat es pretén doncs esbrinar i definir com realitzar aquesta aplicació.

3 Implementació

Per a realitzar la implementació de l'aplicació s'ha de tenir en compte dos etapes diferenciades. La primera tasca que s'haurà d'implementar serà la de capturar els paquets que envien les tasques par-al·leles, és a dir, observar el tràfic de xarxa per posteriorment, en una segona etapa, tractar el tràfic capturat amb la finalitat de planificar les nostres aplicacions basant-nos amb el mètode "Predictive Coscheduling".

3.1 Captura de paquets

Abans de descobrir com capturar el tràfic de xarxa hem de tenir en compte una sèrie de limitacions i objectius que ajudaran a obtenir la millor solució. Per a controlar la quantitat de paquets que un procés envia i/o rep s'han de tenir en compte les següents premises:

- Una de les premises més importants d'aquest projecte és l'entorn en que ha d'estar ubicada la tasca. S'ha d'implementar en un entorn d'usuari, és a dir, a nivell d'aplicació, incloent dins d'aquest nivell la possibilitat de ser executat per l'usuari *root*[1]. Amb aquesta premisa neguem la possibilitat de modificar el kernel o integrar-lo en algun mòdul del kernel. Es busca la portabilitat, i incloure'l a un nivell mes baix que el d'aplicació implicaria haver d'adaptar el codi per cada màquina o kernel.
- S'ha de poder controlar i filtrar els paquets de comunicació segons el procés al que pertanyen, identificat per un número anomenat PID. No és suficient amb tenir constància del tràfic de xarxa de l'equip o de les interfícies.
- L'accés a aquesta informació haurà de ser el més ràpida possible per tal de poder actuar segons les circumstàncies amb el menor temps possible. Aquesta premisa implica alhora dues condicions:
  - Intentar, en la mesura possible, no usar altres aplicacions que afegeixin un retràs addicional. En cas de necessitar una aplicació externa per efectuar la captura, implementar-la, si existeix la possibilitat, dins de la nostra pròpia aplicació i amb les nostres pròpies premises, donat que aquesta manera sol ser la més eficient.
  - La aplicació o script que s'utilitza per calcular el tràfic ha de ser el més simple possible en referència als càlculs, per evitar que es consumeixi molta CPU i que el propi temps de càlcul afegeixi un retard addicional.

Tenint en compte aquestes condicions s'han tingut en compte vàries opcions a estudiar per efectuar el nostre objectiu de lectura de paquets. S'han desglossat en dos apartats: les pròpies aplicacions o comandes que proporciona o que estan disponibles en el sistema operatiu en el qual es treballa,

[1] root: usuari que disposa de tots els permisos del s.o., també anomenat super-usuari.

Linux/Unix; i un sistema de fitxers usat també per el propi sistema operatiu per mostrar tota mena d'informació sobre el sistema operatiu, els diferents processos e inclús el hardware de la màquina on es troba.

3.1.1 Comandes disponibles del S.O.

El Sistema Operatiu Linux disposa d'una sèrie de comandes destinades al control de la xarxa. Tot i que hem d'evitar usar aplicacions que retardin el temps de còmput és adient fer-ne referència.

1. *ps*: Tot i que aporta molta informació sobre els processos, no ens informa sobre el tràfic de xarxa d'aquests

2. *vmstat*: Informa sobre processos, memòria, paginació, blocs d'entrada i sortida (I/O) i CPU. Veiem que no ens serà útil per al nostre projecte donat que de l'únic que ens facilita informació es dels blocs rebuts i enviats per la nostra màquina, sense especificar ni tan sols la interfície.

3. *strace*: Strace es una comanda la qual captura les crides a sistema que un procés rep i envia. Es una molt bona eina per controlar i filtrar els missatges a sistema d'un procés. El fet de poguer especificar el PID de l'arxiu (-p PID) a escoltar fa que sigui molt útil per al nostre cas. Altrament també es pot especificar el camp que ens interessa, podem especificar que ens imprimeixi només el relacionat amb la xarxa (-e trace=network), d'aquesta manera ens imprimirà per pantalla la connexió i desconnexió de la transmissió i cada un dels paquets enviats o rebuts. El gran inconvenient d'aquesta comanda és que aportaria un greu retard al calcul del nombre de paquets quan el que ens interessa és un temps real o immediat.

4. *netstat*: Netsat seria la eina més complerta alhora de controlar qualsevol paquet que es rep o s'envia, ja sigui dins de propi sistema o a la xarxa. Pot ser molt útil per calcular els paquets enviats i rebuts per els processos ja que podem especificar el protocol que interessi, en el nostre cas seria inet (–protocol=inet). Es pot imprimir el PID dels paquets també, de manera que es pot filtrar els paquets destinats a la xarxa d'un PID específic mitjançant la comanda AWK o qualsevol altre mètode de filtratge ja que no interessen els paquets externs a l'aplicació paral·la que es desitja planificar. Aquesta eina es troba en tot sistema operatiu Linux, independentment de la distribució, per tant no hi hauria problema de portabilitat. Una altra virtut d'aquesta aplicació és el temps de resposta, no es tracta d'un temps real però si relativament ràpid. Un exemple de filtratge simple podria ser:

   ```
   netstat -p --protocol=inet -c |grep PID |grep ESTABLISHED
   ,on PID seria l'identificador de la nostra aplicació paral·lela
   ```

   Executant aquesta instrucció s'obtenen els paquets enviats i rebuts a la xarxa (–protocol=inet) pel procés "PID" (-p: imprimir el PID del procés, grep PID: capturar únicament les línies on hi aparegui PID) establerts (grep ESTABLISHED: capturar únicament les línies on hi aparegui

ESTABLISHED), ja que no ens interessen els paquets de petició de connexió o desconnexió, de forma continuada (-c: captura de forma continuada), fins que aturéssim el netstat.

Indagant sobre aquestes comandes es pot esbrinar que treballen sempre sobre el sistema de fitxers /proc. Es a dir que el sistema de fitxers /proc és el que proporciona la informació pertinent que després l'aplicació mostra de forma més accessible a l'usuari. Així doncs, s'estudia aquest sistema de fitxers amb la finalitat de poder crear una aplicació pròpia que únicament capturi el necessari amb el menor retard possible.

3.1.2 Sistema de Fitxers /proc

Treballant en un entorn Linux tota la informació sobre els processos, i part del hardware, queda emmagatzemada en forma de sistema de fitxers de lectura dins de /proc. Aquests fitxers són una espècie d'interfície entre l'usuari o l'aplicació i el kernel, la qual permet visualitzar o controlar el que cada procés efectua i els diferents events del sistema operatiu.

Dins de /proc existeix un directori per a cada procés, essent l'identificador numèric PID del procés el nom del directori. Donat que probablement la informació que es desitja es pot trobar dins d'aquests, s'indaga sobre aquests directoris, quins camps el composen i quina informació poden oferir. A continuació s'ofereix una breu descripció del que es pot saber d'un procés dins de /proc/PID:

- cmdline: Conté la comanda que s'ha executat a l'arrancar el procés.
- CPU: Conté informació sobre l'ús de les CPU's.
- cwd: Enllaç simbòlic al directori on es troba actualment el procés.
- environ: Llista les variables d'entorn per al procés.
- exe: Enllaç simbòlic a l'executable del procés.
- fd: Directori que conté tots els descriptors de fitxer per al procés. Un aspecte a tenir molt en compte és el fet que cada socket[2] està identificat amb un únic descriptor, contingut en aquest directori.
- maps: Mapa de memòria per als arxius executables i llibreries associades al procés.
- mem: Memòria del procés.
- root: Enllaç al directori root del procés.
- stat: Ens informa de l'estat del fitxer i ens mostra informació sobre aquest. Com per exemple quan ha estat modificat, els ID de l'usuari propietari i el grup, el número d'inode, el tipus d'arxiu, etc. Cal remarcar però que no hi ha informació relacionada amb el tràfic de xarxa.

[2]Canal de comunicació per el qual es transfereix informació d'un únic emisor a un únic receptor. L'emisor és principalment el pròpi procés que ha creat el socket i el receptor pot ésser tant intern com extern a la mateixa màquina.

- statm: Estat de la memòria en us per el procés.
- status: Ens informa de certes propietats incloses dins de stat i statm però de forma més entenedora per l'usuari.

Veiem doncs que es pot obtenir certa informació interessant del procés PID mitjançant els descriptors de sockets, és a dir, cada connexió que efectuï el procés PID crearà un socket que el trobarem dins de /proc/PID/fd/Socket:[numero].

Dins del mateix sistema de fitxers /proc trobem un apartat /proc/net, que és precisament el que ens interessa controlar.

/proc/net proporciona paràmetres i estadístiques de xarxa. El arxius de més transcendència que podem trobar dins d'aquest directori són els següents:

- arp: Conté la taula de kernel ARP.
- atm: Conte configuracions i estadístiques ATM (Asynchronous Transfer Mode) i les targetes ASDL.
- dev: Llista els dispositius de xarxa configurats. Informant del nombre de paquets enviats i rebuts per cada interfície entre altres estadístiques. Aquesta informació ens interessaria si no fos perquè ens informa únicament de la interfície i no d'un procés en particular.
- igmp: Llista les direccions IP multicast a les que el sistema s'ha incorporat.
- ip_masquerade: Taula d'informació de l'emmascarament de xarxa.
- ip_mr_cache: Llista el cache de routing de múltiples destinataris.
- ip_mr_vif: Llista les interfícies virtuals multicast.
- netstat: Conté estadístiques de xarxa. Orientat principalment a TCP. Una vegada més, no contempla els sockets o els PIDs per a poder especificar i per tant, no podem saber d'on provenen les connexions.
- psched: Llista els paràmetres de planificació global del paquet.
- raw: Llista les estadístiques de dispositius bruts (raw).
- route: Visualitza la taula de ruta del kernel.
- snmp: Llista de les dades del protocol SNMP per als diferents protocols de xarxa en ús.
- sockstat: Proporciona estadístiques de socket. Únicament ens informa dels sockets relacionats amb el protocol, una vegada mes, no especifica el procés propietari.

- tcp: Conté informació més detallada sobre el socket TCP [16]. Aquí es pot veure el socket relacionat amb el procés que l'està usant mitjançant l'UID que ens mostra quin és l'usuari que ha creat el socket. Per tant, si es coneix el socket que ens interessa controlar es pot filtrar aquest fitxer de manera que ens mostri la informació del socket concret.
- udp: Conté informació més detallada sobre el socket UDP [17]. Igual que /proc/net/tcp però aplicat al protocol UDP
- unix: Llista els sockets de domini UNIX.
- wireless: Llista les dades de l'interfície de radio wireless.

Un altre directori que podríem estar interessats en estudiar seria el /proc/sys/net, però comprovem que principalment el que ens permet aquest sistema de fitxers es configurar paràmetres de les diferents capes de xarxa.

Després d'experimentar vàries de les opcions aquí exposades s'arriba a la conclusió que el sistema d'utilitzar una aplicació externa, com són les comandes del s.o., per al nostre objectiu ens aporta un retard innecessari i que la d'extreure tota la informació necessària a partir del sistema de fitxers /proc és més rapida i eficient.

Donat que es pot obtenir la informació d'un socket concret es necessita esbrinar quins són els sockets que interessa controlar. S'ha vist que és possible saber quins sockets ha creat un procés PID concret, per tant, únicament necessitem el PID de la tasca paral·lela per obtenir els seus respectius sockets creats, via /proc/PID/fd, i poder supervisar-los posteriorment mitjançant /proc/net.

3.2 Gestors d'Aplicacions Distribuïdes PVM/MPI

Abans de descobrir com capturar el tràfic de xarxa, és important conèixer quins sockets s'han de controlar i quins protocols seran transcendents. Donat que en aquest projecte es treballa amb els dos principals gestors d'aplicacions paral·leles, PVM[11] i MPI[12], es descriu el sistema d'intercomunicació que aquests utilitzen.

3.2.1 PVM

PVM[11] es va desenvolupar basant-se en el concepte de màquina virtual ("Paral·lel Virtual Machine") , es a dir, una col·lecció de recursos computacionals gestionats com un sol computador paral·lel.

La màquina virtual paral·lela és una màquina que no existeix, però una *API* adequada ens permet programar-la com si així fos. El model abstracte que ens permet usar la API de la PVM consisteix en una màquina multiprocessador escalable (es a dir, que podem augmentar i disminuir el nombre de processos en temps d'execució). Per dur-ho a terme, oculta la xarxa en que es troba treballant, així com les màquines connectades a la xarxa i les seves característiques.

L'arquitectura de la PVM es composa de dues parts.La primera és el daemon, anomenat *pvmd*. El daemon ha d'estar funcionant en totes les màquines que vagin a compartir els seus recursos computacionals amb la màquina paral·lela virtual. A diferencia d'altres daemons i programes del sistema, el daemon de la PVM pot ésser instal·lat per un usuari en el seu directori home. Això permet fer la supercomputació com a simples usuaris, sense tenir necessitat de ser super-usuari. Aquest daemon pvmd és el responsable de la màquina virtual en si, és a dir, de que s'executin els programes per la PVM i de generar els mecanismes de comunicació entre màquines, la conversió automàtica de dades i ocultar la xarxa al programador.

La segona són les pròpies tasques programades per la PVM i arrancades en la mateixa.

PVM realitza les comunicacions entre processos a través de sockets. Un socket és un canal de comunicació per un procés que permet emetre o rebre informació. Hi han dos tipus de socket que PVM utilitza, segons el tipus de protocol que es vol utilitzar en la comunicació, pot ésser TCP (*Transport Control Protocol*) o UDP (*User Datagram Protocol*) .

PVM pot realitzar tres tipus de connexions diferents, connexió entre dues tasques directament (tasca-tasca), connexió entre una tasca i el dimoni pvmd de la mateixa màquina (tasca-daemon) o viceversa daemon-tasca. El tipus de protocol citat anteriorment dependrà del tipus de connexió establerta. En una connexió dimoni-dimoni s'utilitza el protocol UDP donat que la utilització del TCP presenta greus inconvenients en cas de tenir una gran quantitat de màquines conformant la PVM.

Tot i que UDP és un protocol no fiable i no orientat a connexió que pot perdre, duplicar o reordenar paquets, és la solució al problema de tenir un nombre de descriptors de fitxers TCP mes gran del permès pel sistema operatiu, doncs un únic socket es pot comunicar amb qualsevol altre node PVM introduint en els missatges les adreces origen i destí. Per altra banda, la no fiabilitat del protocol UDP obliga a PVM a implementar una gestió de les comunicacions fiable.

Es el cas de les connexions tasca-tasca i tasca-dimoni, on es requereix un protocol fiable que permeti l'execució de les tasques sense interrupcions, ni overheads innecessaris, provocats per les comunicacions. En aquest cas s'utilitza el protocol TCP.

Així doncs, donat que el que realment interessa controlar en aquest projecte són les comunicacions entre les tasques per damunt de les comunicacions entre els propis dimonis de la PVM, ens centrarem en controlar els sockets que utilitzin el protocol TCP.

3.2.2 MPI

El pas de missatges és una tasca àmpliament usada en certes màquines paral·leles, especialment aquelles que compten amb memòria distribuïda. Encara que existeixen moltes variacions, el concepte bàsic en el procés de comunicació mitjançant missatges està ben entès. Recentment diferents sistemes han demostrat que un sistema de pas de missatges pot ésser implementat eficientment i amb un alt grau de portabilitat.

Al dissenyar-se MPI[12], es van tenir en compte les característiques més atractives dels sistemes existents per al pas de missatges, en comptes de seleccionar únicament un sol d'aquests sistemes i

adapta'l com a estàndard. Resultant així, en una forta influencia per MPI els treballs realitzats per IBM, INTEL, NX/", nCUBE, Vernex, p4 i PARMACS.

La meta de MPI (Message Passing Interface) és la de desenvolupar un estàndard per escriure programes que implementin el pas de missatges. Pel qual la interfaç intenta establir un estàndard pràctic, portable, eficient i flexible. En un ambient de comunicació amb memòria distribuïda en la qual les rutines de pas de missatges són de nivell baix, els beneficis de l'estandardització són notables. La principal avantatja al establir un estàndard pel pas de missatges és la portabilitat i la facilitat d'us.

El protocol utilitzat actualment en MPI es el protocol de xarxa TCP, implementat i generalitzat per LAM/MPI[22]. Tot i que cada vegada s'està extenent una implementació usant un protocol de xarxa anomenat SCTP similar a TCP.

Altrament veiem que el protocol a controlar en cas de monitoritzar aplicacions paral·leles distribuïdes per MPI es el protocol TCP.

3.3 Disseny de l'aplicació

Una vegada es disposa de les eines adequades per realitzar la nostra tasca, el poper pas es dissenyar la aplicació que portarà a terme la tècnica "Predictive Coscheduling". L'estructura de l'aplicació realitzada es mostra en l'algorisme 3.

Algorithm 3 Disseny de l'aplicació

```
1  if(apli ≠ NULL)
2    PID[]=Bpids(apli);
3    for(i=0;PID[i]≠ NULL;i++)
4    SOCK[]=Bsockets(PID[i]);
5    endfor
6    SOCK[]=isTCP(SOCK);
7    throw capture(SOCK);
8    do forever
9     read_traffic(cur_freq);
10     freq = P * freq + (1 − P) * cur_freq;
11     prior=new_priority(freq);
12     asign_priority(PID,prior);
13     read(apli);
14   enddo;
15 endif;
```

Aquest es el disseny del algorisme general de l'aplicació que ha de realitzar la tècnica Coscheduling sobre les tasques paral·leles que es desitgin.

Tenint en compte que cada execució d'aquesta aplicació treballa sobre una única aplicació distribuïda, la qual es identificada amb el nom de l'aplicació, el primer que es comprova en el nostre programa es l'existència d'aquesta aplicació distribuïda que es desitja planificar amb la tècnica "Predictive Coscheduling" (1).

Aquest algorisme general (algorisme 3) es pot desglossar l'aplicació en tres grans apartats, la busqueda dels canals de comunicació a capturar (2-6), la captura continuada d'aquests mateixos fluxos (7) i la planificació de l'aplicació paral·lela(8-15).

▷ Primerament es realitza el procés de busqueda dels canals de comunicació, identificats com a sockets, a controlar. Mitjançant la informació que proporciona el sistema operatiu, informació descrita en l'apartat 3.1, s'ha d'obtenir els sockets de l'aplicació paral·lela que es desitja planificar. Aquesta busqueda es realitzada per passos, ja que únicament es disposa del nom de l'aplicació, el primer pas consisteix en obtenir els PIDs o identificadors de procés de l'aplicació. Una vegada es disposa dels PIDs correctes obtenim els sockets que cada un d'aquests han creat i seleccionem únicament els que pertanyin al protocol TCP, ja que en l'apartat anterior s'ha observat que és el protocol usat per els gestors d'aplicacions paral·leles sota els quals es realitza la experimentació d'aquest projecte.

▷ El proper pas a realitzar és una captura continuada dels paquets que transmeten o reben els sockets ja obtinguts. Aquesta captura es realitza de manera independent al programa principal ja que interessa que la captura sigui el mes fiable possible, altrament es veuria retardada per la resta de funcions de l'aplicació. La funció capture, anomenada funcioFil en el codi mostrat en els annexos, ha de realitzar les següents tasques mostrades en l'algorisme 4.

Algorithm 4 procés fill capture(SOCK[])

```
1  file=open_file_read("/proc/net/tcp");
2  do forever
3    while(!file.end)
4      sock_aux=read_sock(file);
5      traf_aux=read_traf(file);
6      for(i=0;SOCK[i]≠ NULL;i++)
7        if(sock_aux==SOCK[i])
8          add_traf(traf_aux);
9        endif;
10     endfor;
11   endwhile;
12   file_start(file);
13 endo;
```

Com s'ha exposat en l'apartat 3.2 la forma més eficient d'obtenir informació sobre el tràfic de xarxa és mitjançant el sistema de fitxers /proc, i concretament el fitxer /proc/net/tcp, així doncs el primer pas consisteix en obrir aquest fitxer per poder llegir-lo (1). A continuació entrem en un bucle indefinit on primerament es llegeix el fitxer capturant-ne línia a línia el socket en el qual ens trobem (4) i la quantitat de paquets que es troben en la cua, tant de recepció com d'enviament (5), en aquell moment. Una vegada es disposa d'aquesta informació es compara amb els sockets que ens interessa capturar, els que pertanyen a l'aplicació paral·lela en qüestió, i en cas de coincidir s'envia el tràfic de xarxa al programa principal perquè actuï en conseqüència (6-10). Donat que aquest bucle és indefinit, una vegada efectuada la captura es retorna al inici del fitxer /proc/net/tcp per realitzar mes captures(12).

▷ Tornant al disseny de l'aplicació (Algorisme 3), l'últim conjunt d'instruccions consisteix en realitzar la planificació de l'aplicació, a partir de l'informació obtinguda en els processos anteriors. En aquest cas ens tornem a trobar dins un bucle indefinit. Obtenim el tràfic de xarxa que ens facilita la funció capture, descrita recentment, aquest tràfic es la freqüència actual de comunicació (9). A partir d'aquí realitzem l'algorisme formulat en l'apartat 2.3 per obtenir la freqüència corresponent a la tasca (10). En funció d'aquesta freqüència assignarem la prioritat a la nostra aplicació, per mitja dels seus PIDs (11,12). Finalment hem de tenir en compte que l'aplicació pot variar, parant-se, reinciant-se o modificant-se i això afectaria als sockets i PIDs que estem capturant, per tant s'intenta rellegir aquests cada cert interval de temps per evitar errors, intentant no sobrecarregar excessivament al sistema amb aquestes re-lectures.

S'ha de tenir present que aquest bucle indefinit no necessita ésser tant ràpid i persistent com el que es troba en el procés capture, ja que es sotmetria a una càrrega innecessària sobre la CPU. És per tant un motiu afegit per separar els dos bucles.

3.3.1 Consideracions Prèvies

Aquesta aplicació ha estat implementada en llenguatge C++ [7] dada la seva portabilitat i eficiència, mantenint una coherència amb anteriors projectes relacionats els quals han estat implementats en aquest llenguatge.

Abans de descriure cada una de les funcions que realitza la aplicació és important conèixer les variables globals disponibles, variables a les que es pot accedir tant des del programa principal com des del procés fill. D'aquesta manera es permet una comunicació entre el procés capturador i el planificador. S'han definit tres variables importants:

- **Llista<char*> *pids*:** Una variable tipus Llista on s'hi emmagatzemen els identificadors de fitxers PID de l'aplicació paral·lela que es desitja planificar. La classe llista no és més que un conjunt d'elements creats de forma dinàmica, evitant així la memòria estàtica que ens limita a un nombre finit d'elements i a un possible desaprofitament de la memòria. El tipus d'element que contindrà aquesta llista seran cadenes de caràcters, seguint amb la filosofia de la memòria dinàmica aquests també seran creats en el moment de l'execució dinàmicament.

- **Llista<char*> *socks*:** Una variable novament de tipus Llista on, en aquest cas, s'hi emmagatzemen els sockets que es desitgen processar. Es tracta de la mateixa classe que l'anterior també seguint la filosofia de la memòria dinàmica. En aquest cas es podria emmagatzemar els sockets com a enters però mes endavant es comprova que és més fàcil operar amb cadenes de caràcters donat que els fitxers amb els quals es comparen són llegit com a caràcters.

- **fil *intercanvi*:** Variable de classe fil que s'utilitza per efectuar la comunicació entre procés principal i procés fill (capture). Aquesta classe pròpia es descriu en els propers apartats, però bàsicament consta de dos funcionalitats: la de rebre el tràfic del procés fill i facilitar l'enviament d'aquest al procés principal; i informar al procés fill del estat en que es troba el programa.

Una vegada es coneixen el llenguatge en que s'ha implementat i certes variables que ens facilitaran la comunicació es disposa a la descripció de cada una de les funcions de l'aplicació, agrupades, com s'ha definit en el disseny representat en l'algorisme 3, en tres grups.

3.4 Busqueda dels fluxos de comunicació

El primer a realitzar en l'implementació del nostre programa serà la captura dels fluxos de comunicació de l'aplicació paral·lela per tal de poder monitoritzar-los. Tal com s'ha vist en el punt 3.1, la millor manera de capturar els paquets serà mitjançant el sistema de fitxers /proc.

S'ha dividit el programa en les funcions que realitzen parts especifiques del procés de captura. En les següents subseccions s'expliquen cadascuna d'aquestes funcions. Per cada funció, en primer lloc s'explicarà el disseny i a continuació es descriurà la corresponent implementació.

3.4.1 Funció Bpids()

La tasca d'aquesta funció consisteix en esbrinar quins són els identificadors PID de l'aplicació paral·lela que ens interessa controlar. Per poder dur-ho a terme és necessari saber únicament el nom de l'aplicació paral·lela, partint de la base de que hi ha una única aplicació paral·lela amb el mateix nom, o en cas d'haver-n'hi vàries, que ens interessa controlar totes les comunicacions d'aquestes de forma conjunta, es a dir planificar homogèniament tots els processos d'aquestes aplicacions.

Es mostra primer un disseny global (Algorisme 5) del que s'ha buscat alhora d'implementar en aquesta funció.

Algorithm 5 Disseny Bpids()

```
1  empty(pids[]);
2  empty(socks[]);
3  file=open_file_read("ps -ae|grep aplic");
4  while(!file.end)
5    pids[i]=read_pid(file);
6    if(!pids[])wait();
7  endwhile
```

Donat que aquesta funció *Bpids()* no es realitza únicament a l'iniciar l'aplicació, pot ésser cridada desprès d'un canvi en els sockets de l'aplicació distribuïda o, tal com es veu en posteriors apartats, cada cert interval de temps, on es rellegeixen per si hi ha hagut alguna modificació, el primer pas consisteix en esborrar tots els PIDs i sockets existents en les nostres llistes (1-2), en cas d'haver-n'hi. Un cop es disposa de les llistes buides ens disposem a esbrinar quins són els PIDs que corresponen a l'aplicació paral·lela en qüestió. La manera més simple i eficient trobada, basant-nos en els resultats de l'apartat 3.1, és mitjançant la comanda del sistema operatiu *ps* (3). El que retorna aquesta comand($"ps - ae"$) són tots els processos, que posteriorment filtrem per que únicament ens retorni els que pertanyen a la nostra aplicació ($" \mid grep\, apli"$), en aquest cas passada per paràmetre amb el nom d'*aplic*.

A partir d'aquí entrem en un bucle que realitzarà lectures fins que el fitxer, que conté el resultat de la comanda anteriorment descrita, finalitzi (4), capturant els PIDs corresponents a la nostra aplicació i emmagatzemant-los en la nostra llista (5) i, en cas de no existir cap PID corresponent a la nostra aplicació paral·lela, esperant fins que existeixi tal identificador (6), ja que aquesta situació indica que l'aplicació paral·lela encara no s'ha llançat.

Un cop explicat el disseny de la funció,l'algorisme 6 mostra la seva implementació detallada en C++.

Algorithm 6 Implementació Bpids()

```
1  bool Bpids(char* aplic){
2    INICIALITZACIÓ
3    IteradorLlista<char*> itPid(pids);
4    IteradorLlista<char*> itSock(socks);
5    itPid.situarInici();
6    while(itPid.fi()==false){
7      pids.eliminarFi();
8    }
9   itSock.situarInici();
10   while(itSock.fi()==false){
11     socks.eliminarFi();
12   }
13   strcpy(commanda,"ps -ae|grep ");
14   strcat(commanda,aplic);
15   while(err==0){
16     fps=popen(commanda,"r");
17     fscanf(fps,"%s %s %s %s",pid,tty,data,apli);
18     err=atoi(pid);
19     while(!feof(fps) && err!=0){
20       pidaux=new char[sizeof(pid)];
21       strcpy(pidaux,pid);
22       pids.inserirFi(pidaux);
23       fscanf(fps,"%s %s %s %s",pid,tty,data,apli);
24       i++;
25     }
26     pclose(fps);
27     if(err==0){
28       reset=true;
29       sleep(10);
30     }
31   }
32   return reset;
33 }
```

El primer que s'ha de saber és que es tracta d'una funció que retorna un booleà, que indica si l'aplicació ha hagut d'esperar per obtenir els PIDs corresponents a l'aplicació paral·lela, condició que indicarà que es tracta d'una nova execució d'aquesta aplicació distribuïda, i que per tant, com

més endavant s'indica, no es pot seguir monitoritzant els mateixos canals de comunicació en que es trobava.

Es passa per paràmetre el nom de l'aplicació a controlar, en forma de *char** (cadena de caràcters), creat dinàmicament a partir del pas de paràmetres de l'aplicació. Dins de la instrucció INICIALITZACIONS (2) hi englobem la declaració i inicialització de les diferents variables auxiliars pròpies de la funció: un fitxer auxiliar fps el qual s'utilitzarà per emmagatzemar la sortida de la crida del procés "comanda" que posteriorment realitzarem; les cadenes de caràcters pertinents, les destinades a emmagatzemar temporalment els diferents paràmetres de sortida de la crida "ps" i una altra cadena per crear la comanda ja citada; una variable booleana que facilita la informació citada en el paràgraf anterior; i per últim les variables comptador i error, que s'utilitzen com a comptador i comprovació d'errors.

Un altra de les possibles variables a incloure en aquest apartat d'inicialitzacions podria ser la creació dels iteradors de les llistes pids i socks (3-4). La funció d'aquests iteradors no és altra que la de permetre desplaçar-se per la llista associada a aquests.

Una vegada es disposa de totes les variables pertinents es buiden les llistes pids i socks pel motiu ja exposat (5-12). La raó per la qual es buidi la llista dels sockets en aquesta funció i no en la que realitza la busqueda de sockets es deu a que la segona funció (Bsockets) es crida vàries vegades. Això es degut a que aquesta funció *Bsockets* es cridada per a cada un dels PIDs existents, i per tant, si s'inclogues l'eliminació dels sockets existents en la funció Bsockets es tindria en compte només els sockets de l'últim PID, ignorant els demés a causa de l'eliminació.

Es crea la comanda amb la qual se'n extreu els pids de l'aplicació (13-14). Aquesta comanda està formada per la crida a la instrucció del sistema operatiu Linux "ps" amb les opcions pertinents per a poder llistar tots els processos de tots els usuaris de la màquina, i una vegada llistat es filtra la sortida mitjançant un "pipe" o "tuberia" seleccionant únicament els processos que inclouen el nom de l'aplicació (prèviament passada a la funció com a paràmetre "aplic").

S'executa la comanda descrita anteriorment i se'n emmagatzema la sortida en el fitxer *fps* (16). Es comprova que existeixi un procés que pertany a l'aplicació paral·lela, sinó és així (err=0) s'espera 10 segons abans de tornar-ho a intentar (27-30).

Si existeix un procés que pertany a l'aplicació que es busca s'emmagatzema a la llista *pids* i es continua llegint i emmagatzemant fins a arribar a la fi del fitxer (19-25).

Finalment la funció retorna la variable booleana *reset*, que ens indicarà si es tracta d'una aplicació paral·lela recent arrencada o ens trobem en una aplicació ja existent (32). En l'apartat 3.6 es veurà perquè es necessària aquesta informació.

3.4.2 Funció Bsockets()

El proper pas a realitzar és la busqueda dels sockets o fluxos de comunicació que utilitzen cada un dels processos de l'aplicació paral·lela, identificadors dels quals s'han capturat en l'anterior funció Bpids(). D'aquesta manera capturant els sockets i la informació que aquests transporten obtindrem els paquets de comunicació que comunica l'aplicació paral·lela.

Un possible disseny es mostra en l'algorisme 7.

Algorithm 7 Disseny Bsocket()

```
1  file=open_file_read("ls /proc/PID/fd -l");
2  while(!file.end)
3    socks[i]=read_sock(file);
4  endwhile
```

En aquest s'utilitza la comanda *ls* per llistar el contingut del directori "/proc/PID/fd", la comanda *ls* es utilitzada per llistar el contingut dels directoris, i l'opció *-l* mostra diferents característiques de cada un dels fitxers que hi han continguts, tals com els permisos, data de creació, tamany, o, el que es busca en aquest cas, el fitxer al qual esta associat un *link*[3]. Com s'ha descrit en el primer apartat d'aquesta secció aquest directori conté el llistat de descriptors de fitxer que ha obert el procés *PID* , entre els quals s'hi troben els sockets, els quals es troben en forma de *link* simbòlic. La sortida d'aquesta comanda queda emmagatzemada en un fitxer de lectura (1).

Una vegada es disposa del fitxer es llegeix línia a línia amb l'objectiu de cercar els descriptors de fitxers dels *sockets*, per a emmagatzemar-los en la llista de sockets.

L'algorisme 8 mostra la corresponent implementació en C++.

Algorithm 8 Implementació Bsockets()

```
1  void Bsockets(char* pid){
2    INICIALITZACIONS
3    strcpy(commanda,"ls /proc/");
4    strcat(commanda,pid);
5    strcat(commanda,"/fd -l");
6    ls=popen(commanda,"r");
7    while(!feof(ls)){
8    fscanf(ls,"%s",aux);
9      if(strncmp(aux,"socket",6)==0){
10       for(i=0,o=8;aux[o];i++,o++){
11         aux[i]=aux[o];
12       }
13       aux[i-1]='\0';
14       sock=new char[sizeof(aux)];
15       strcpy(sock,aux);
16       socks.inserirFi(sock);
17     }
18   }
19   pclose(ls);
20 }
```

[3]Un link no es més que un fitxer associat a un altre, pel qual la modificació d'un dels fitxers es veurà reflectida en la resta de fitxers. Aquesta associació pot ésser de dos tipus: hard-link, es tracta de varis fitxers que conten el mateix, es podria definir com un mateix fitxer amb diferents noms o ubicacions, en un llenguatge més tècnic es diu que comparteixen inode; l'altre tipus es symbolic-link, en aquest cas es tracta d'un fitxer que apunta a un altre, el primer no es pot definir exactament com a un fitxer en sí sinó únicament un apuntador al fitxer real, en aquest cas operacions com l'eliminació s'apliquen únicament al apuntador sense afectar al fitxer original.

La funció no retornarà res, raó per la qual es declarada com a void. L'únic paràmetre que és necessari és l'identificador de procés *PID* el qual estem capturant. Recordem que aquesta funció es crida per cada un dels *PID*s de l'aplicació paral·lela (1).

Iniciem la funció com és habitual amb la declaració de les variables (2): el fitxer auxiliar *ls* que emmagatzemarà la sortida de la crida a la comanda ls; les variables char[] que s'utilitzen per guardar temporalment la línia de sortida del fitxer ls, una variable que realitza una funció de pas intermig la qual s'emmagatzema en la llista de sockets, una variable per crear la cadena de caràcters que conformen la comanda que cridarà a la instrucció ls, i les variables comptador pertinents.

Es crea la comanda que s'utilitza per llistar el contingut de la carpeta /proc/*PID*/fd/ (3-5), on *PID* és el PID passat per paràmetre, és a dir l'identificador d'un dels processos que pertanyen a l'aplicació paral·lela. Dins d'aquesta carpeta hi ha un seguit d'entrades en forma *link* simbòlic que apunta a un descriptor per cada fitxer obert que te el procés, incloent com a fitxer de sortida un socket, entre d'altres. Així doncs dins d'aquesta carpeta es troben descriptors que apunten a fitxers anomenats "socket:[*Num*]", on Num és l'identificador del socket.

Es crida la comanda recentment creada emmagatzemant el resultat en un fitxer de lectura anomenat "ls" (6), per tal de poder tractar-ho posteriorment.

El que es fa a continuació és filtrar la sortida de la comanda ls per tal d'emmagatzemar únicament els identificadors de socket descrits anteriorment. Per a fer-ho es compara línia per línia la sortida de la comanda comparant la cadena fins a trobar la cadena "socket" (9-10), una vegada es coneix que es tracta d'un socket es captura el numero que l'identifica i s'emmagatzema en la variable auxiliar (10-13) per a ser introduït seguidament a la llista comuna de de sockets (socks) (14-16).

Per acabar es tanca correctament el fitxers obert a l'inici de la funció (19).

3.4.3 Funció isTCP()

En aquest punt es disposa dels tots els sockets utilitzats per l'aplicació paral·lela a planificar. Així doncs el proper pas és filtrar els sockets per obtenir únicament els que és troben sota el protocol TCP, donat que es el protocol de comunicació que ens interessa controlar, el motiu del qual es descriu en els apartats 3.1 i 3.2. Així doncs, disposant d'una llista de sockets, l'únic que s'haurà de comprovar és si es troben dins del fitxer "/proc/net/tcp". Aquest fitxer conté l'estat de tots els sockets que troben sota el protocol TCP, com ja s'ha descrit extensament en l'apartat 3.1.2.

La implementació d'aquesta funció es mostra en l'algorisme 9.

Algorithm 9 Implementació istcp()

```
1  bool istcp(char *socket){
2    bool res=false;
3    char temp[150];
4    FILE *fin;
5    fin=fopen("/proc/net/tcp","r");
6    while(!feof(fin)){
7      fscanf(fin,"%s",temp);
8      if(strcmp(socket,temp)==0){
9        res=true;
10       break;
11     }
12   }
13   fclose(fin);
14   return res;
15 }
```

Es tracta d'una funció booleana que ens retorna si el socket, transferit per paràmetre en la crida de la funció, pertany o no al protocol TCP.

Com a variables únicament es crea una variable booleana, anomenada *res*, que ens indicarà si pertany o no al protocol TCP (1), un fitxer de lectura *fin* on hi inclourem el contingut del fitxer "/proc/net/tcp" (4) i per últim una variable auxiliar *temp* que s'utilitza per emmagatzemar temporalment les línies del fitxer auxiliar (3).

Per comprovar si un socket utilitza o no al protocol TCP s'obre en mode lectura el fitxer /proc/net/tcp i es comprova línia a línia si el socket pertany al fitxer (5-12). Si un socket és TCP, encara que en el moment de lectura del fitxer /proc/net/TCP no estigui enviant ni rebent informació, es trobarà dins del fitxer fins al seu tancament.

Per últim es tanca correctament el fitxer auxiliar (13) i es retorna el resultat (14), és a dir si és o no TCP.

3.5 Procés de lectura continuada [Funció funcioFil()]

Una vegada s'ha realitzat el procés de busqueda i es disposa dels sockets que s'hauran d'observar per poder obtenir la freqüència de comunicació de l'aplicació paral·lela en qüestió, es disposa a realitzar una lectura continuada d'aquests sockets.

El disseny base sobre el qual s'ha realitzat posteriorment la implementació és mostra en l'algorisme 10.

Algorithm 10 Disseny funcioFil()

```
1  file=open_file_read("/proc/net/tcp");
2  do forever
3    sock_aux=read_sock(file)
4    traf_aux=read_traf(file)
5    for(i=0;sock[i]≠ NULL;i++)
6      if(sock_aux==sock[i])
7        save(traf)
8      endif
9    endfor
10   seek_start(file)
11 enddo
```

La lectura s'efectua sobre el fitxer "/proc/net/tcp". Així doncs el primer que s'haurà de fer és obrir en mode lectura aquest fitxer (1). S'ha de contemplar que en aquest punt existeix una llista de sockets els quals pertanyen als diferents processos de l'aplicació paral·lela, representat en aquest disseny com a sock[].

Disposant ja de la font de dades que facilitarà la comunicació que efectuen els diferents sockets esmentats, s'entra en un bucle indefinit en el qual es llegeix línia per línia el socket i els paquets que aquest comunica (3-4), tenint en compte que cada línia representa un únic socket i els seus atributs.

Es compara per cada socket capturat si coincideix amb algun dels sockets que interessa controlar (6), en cas afirmatiu s'emmagatzema el tràfic del socket en qüestió, és a dir els paquets d'enviament i recepció que apareixen en el fitxer (7), per a que posteriorment el programa principal pugui recollir aquesta informació i tractar-la.

Per últim, desprès de comprovar si els sockets concorden, es re-inicia el fitxer de forma que permeti rellegir des del principi el fitxer "/proc/net/tcp", actualitzant el seu contingut (10).

En un principi aquestes serien les úniques tasques que s'haurien de realitzar de forma il·limitada, sempre i quan no hi hagués cap modificació, la qual s'indicaria des del programa principal.

La principal funció del programa és la de capturar contínuament els paquets de l'aplicació desitjada i posteriorment actuar en conseqüència, ajustant la prioritat de tots els processos que composen l'aplicació paral·lela. La lectura de paquets comunicats pels sockets prèviament seleccionats s'efectua mitjançant un procés fill o $thread$[4], aquest s'encarrega de efectuar periòdicament el procés de lectura al fitxer "/proc/net/tcp" i passar les dades al procés principal per poder tractar-les i actuar en conseqüència.

En la implementació en C++ aquest procés s'ha anomenat funcioFil i és arrencat des del programa principal mitjançant la funció *pthread_create()* de la llibreria *pthread.h*. Aquest procés fill s'executa amb una menor prioritat per tal de no interferir en excés en la CPU i evitar el possible retard afegit.

La implementació realitzada doncs d'aquesta funció es mostra en l'algorisme 11.

Algorithm 11 Implementació funcioFil()

```
1  void *funcioFil(void *parametre){
2    IteradorLlista<char*> itSock(socks);
3    INICIALITZACIONS
4    fin=fopen("/proc/net/tcp","r");
5    while(intercanvi.getstat()<2){
6      err=fscanf(fin,"%s %s %s %s %s %s %s %s %s %s %s %s”,aux,
,aux,aux,aux,aux,aux,aux,aux,aux,aux,aux,aux);
7      while(!feof(fin) && err!=EOF){
8        err=fscanf(fin,"%s %s %s %s %s %s %s %s %s %s %s %s %s %s
%s %s %s",aux,aux,aux,aux,traf,aux,aux,aux,aux,sid,aux,aux,aux,aux,
,aux,aux,aux);
9        if(strcmp(aux,"-1")!=0)fseek(fin,-5*sizeof(aux),SEEK_CUR);
10       if(err!=EOF){
11         c_env=strtok(traf,":");
12         if(c_env==NULL)enviat=0X0;
13          else enviat=strtoul(c_env,NULL,16);
14         c_reb=strtok(NULL," ");
15         if(c_reb==NULL)rebut=0X0;
16          else rebut=strtoul(c_reb,NULL,16);
17         for(itSock.situarInici();itSock.fi()==false;itSock++){
18           if(strcmp(*itSock,sid)==0){
19             intercanvi.add(enviat,rebut);
20           }
21         }
22       }
23     }
24     usleep(50000);
25     rewind(fin);
26     }
27   pclose(fin);
28 }
```

Tot i que la funció conté un paràmetre en la declaració aquest és nul i únicament es present en el codi dada la seva declaració estàndard.

La primera acció que és realitza es l'associació d'un punter *itSock* amb la llista de sockets *socks* (2), que conté els sockets que es desitgen controlar per dur a terme la planificació.

A continuació es realitzen les declaracions i inicialitzacions de les variables auxiliars pertinents (3): primerament es disposa de dues variables numèriques hexadecimals, concretament *unsigned int* (enter sense signe), que contindran el tràfic (quantitat de paquets) que es capturin durant la lectura; es creen tres cadenes de caràcters auxiliars utilitzades per emmagatzemar l'identificador de socket (sid), el tràfic que hi ha en aquell moment sobre el socket en particular (traf), i una variable auxiliar que s'utilitza per emmagatzemar la resta de paràmetres del fitxer; una variable *err* per indicar l'estat dels *scanf* que s'executen; i un fitxer auxiliar que conté el fitxer "/proc/net/tcp" per efectuar les diferents lectures.

S'obre el fitxer anomenat en mode lectura associant-lo al descriptor *fin* (4), una vegada associat

veiem que únicament es necessari retornar al principi amb la funció rewind(FILE*) i al mateix temps es refresca la lectura actualitzant el seu contingut al actual.

A partir d'aquí entrem en un bucle indefinit, que s'interromp des del programa principal si es necessari.

El fitxer "/proc/net/tcp" està dividit en camps representats en forma de columnes, són camps predefinits i fixes, tinguin o no un valor. Per tant per avançar posicions dins del fitxer ho fem mitjançant la lectura de camps i, donat que és un nombre fix, s'emmagatzema únicament els camps que interessen pels posteriors càlculs i comprovacions. S'avança doncs la primera fila (6), la qual ens indica els noms dels camps, per entrar a un segon bucle (7), que es durà a terme mentre no s'arribi al final del fitxer.

Dins d'aquest bucle es va llegint línia per línia emmagatzemant els camps *sid* (Socket ID) i *traf* (quantitat de paquets presents en la qua d'enviament i sortida del socket en aquell moment) i ignorant la resta (8).

S'ha observat que existeixen uns determinats tipus de sockets interns, que es comuniquen dins de la pròpia maquina, que prescindeixen dels últims atributs dada la seva redundància. Es per aquest motiu que hem realitzat una comprovació prèvia per re-situar el punter del fitxer a la posició correcta en cas de trobar-se en aquest cas (9).

Sempre i quan no ens trobem al final del fitxer (10), es converteix el tràfic, o paquets presents en les cues d'enviament i recepció, a un nombre enter hexadecimal (11-16). El format en que trobem aquest tràfic és *"XXXXXXXX:YYYYYYYY"* on *X* és el nombre de paquets que es troben en la cua d'enviament i *Y* els paquets de recepció. Donat que es captura aquest camp com a una cadena de caràcters es divideix la cadena mitjançant la funció *strtok(char*,char*)*, on el primer paràmetre representa la cadena que es vol dividir i el segon el separador o delimitador en el qual realitzarà la separació, en cas d'error retorna *NULL* i per tant es defineix un trafic nul. Posteriorment es converteix la sortida d'aquesta funció a un format enter hexadecimal mitjançant la funció *strtoul(char*,char*,int)* on el primer paràmetre es la cadena de caràcters a convertir (en aquest cas la sortida de la funció strtok()), el segon ens emmagatzema l'última posició escanejada (en aquest cas no ens interessa, *NULL*) i l'últim paràmetre indica el format numèric al qual es convertirà.

Es compara cada un dels sockets que conformen la nostra llista (sock[]) amb el que estem en aquest moment capturant (sid), i en cas de coincidir s'envia el tràfic obtingut a una variable global que realitza la funció de comunicadora entre *thread* i programa principal (17-19).

Així successivament fins arribar al final del fitxer, moment en que el re-iniciem amb la funció rewind(FILE*) (25), per tal d'actualitzar-ne el seu contingut i iniciar una nova lectura. Com es pot comprovar més endavant en l'apartat d'experimentació s'ha hagut d'introduir un petit retard (24) alhora d'efectuar una nova lectura ja que en cas contrari el consum de CPU per part del programa era massa elevat, perjudicant tant a les aplicacions locals com paral·leles i impossibilitant l'objectiu del Coscheduling. La funció usleep(int) introdueix retards de l'ordre de microsegons. Els motius que han portat definir el valor d'aquest parametre estan exposats extensament en l'apartat 4.2.2.

3.5.1 Classe fil

Es pot observar que en l'enviament o emmagatzemament del tràfic s'utilitza la variable *intercanvi*, es tracta d'una variable global de classe fil. La classe fil és una classe dissenyada també en aquest projecte la qual únicament ens servirà per transferir les dades des del *thread* o procés fill al programa principal.El seu objectiu es emmagatzemar el tràfic capturat pel procés fill i facilitar-lo al programa principal quan aquest ho desitgi.

En aquest cas el disseny és extremadament simple, per tant és adient passar directament a la implementació.

Algorithm 12 Implementació Classe Intercanvi

```
1  class fil{
2    unsigned long resACUM;
3    int stat;
4    int i;
5     public:
6    fil(){
7     resACUM=0X0;
8     stat=0;
9     i=1;
10   }
11   void add(unsigned long resE,unsigned long  resR){
12     resACUM=resACUM+resE+resR;
13     i++;
14   }
15   int getstat(){
16     return stat;
17   }
18   void setstat(int estat){
19     stat=estat;
20   }
21   unsigned long calcular(){
22     unsigned long resultat;
23     resultat=resACUM/i;
24     i=1;
25     resACUM=0;
26     return resultat;
27   }
28 };
```

En aquest classe s'hi troben únicament tres variables, una variable entera sense signe, la qual anirem emmagatzemant el nombre de paquets enviats i rebuts, també es crea una variable per indicar l'estat en que es troba el programa i una variable comptador.

En el que a funcions es refereix s'han creat 4 funcions, més la declarativa. En la declaració de la classe realitzem la inicialització de les variables (6-10), donant a conèixer al programa que la variable *resACUM* hexadecimal, i atorgant el valor inicial a les variables *stat* i al comptador *i*.

La funció *add(unsigned long,unsigned long)* (11-13)emmagatzema els dos enters sense signe a la variable *resACUM* , afegint-los al valor d'aquesta. Aquesta funció és la que es crida des del *thread* funcioFil cada vegada que trobem tràfic pertanyent a un dels sockets de la llista. S'incrementa un comptador que indica la quantitat de lectures efectuades, aquest comptador s'utilitza posteriorment per retornar la pendent del interval de temps, no la suma total de paquets.

La classe consta d'una funció per assabentar-se de l'estat del programa principal *getstat()* (15-17)i una funció per definir aquest estat *setstat(int)* (18-20). L'estat per defecte es "0", en cas de que l'estat sigui "2" el procés fill interpretarà que ha de finalitzar la lectura de paquets, i en cas de ser "3" el programa finalitzarà.

Per últim la funció *calcular(int)*, la qual és cridada des del programa principal alhora de realitzar el calcul de prioritats. L'objectiu d'aquesta funció no és altre que retornar els paquets enviats i rebuts durant un cert interval de temps, donat que durant aquest interval de temps les lectures efectuades poden variar depenen de la càrrega de treball de la CPU o de la mateixa aplicació, retornem la pendent del l'ultim interval de temps, dividint la freqüència total de comunicació per les lectures efectuades, que podem obtenir gracies a la variable comptador que incrementem a cada addició de dades. Una vegada hem definit el resultat a tornar re-inicialitzem les variables acumulatories i els comptadors pertinents per a entrar en un nou período de captura.

3.6 Tractament del tràfic (Planificació de prioritats)

L'últim pas a realitzar, i en definitiva el que ha de fer complir l'objectiu del projecte, és doncs el d'assignar les prioritats corresponents als processos paral·lels en funció de la informació obtinguda mitjançant les funcions anteriorment descrites, i realitzant així la planificació.

Tal com s'indica en l'apartat teòric la formula que defineix la freqüència de comunicació continua d'una tasca h en l'algorisme "Predictive Coscheduling" es:

```
h.freq = P*h.freq + (1-P) * h.curr_freq
```

On *P* és el percentatge assignat a l'anterior freqüència (*h.freq*) i (1-*P*) és el percentatge assignat a la freqüència actual (*h.curr_freq*).

Tenint sempre en compte tant els paquets enviats com els rebuts.

Donat que el que interessa en aquest cas no es el valor absolut de la freqüència de comunicació sinó un valor relatiu per poder comparar objectivament les diferents freqüències donant enfasis en les variacions que pot tenir les comunicacions entre les màquines que conformen la NOW, es modifica la formula per tal de poder obtenir aquest objectiu.

Així doncs la relació que s'utilitzarà per definir la prioritat a assignar s'obté mitjançant el següent algorisme:

```
h.rel = h.freq / h.curr_freq
```

Essent al igual que en la formula anterior, freq la freqüència acumulada fins al moment, i curr_freq la freqüència instantanea de comunicació.

La relació *rel* ens indicara doncs quina és la diferència relativa entre el tràfic actual i el transcorregut, i en funció d'aquesta relació es modificarà la prioritat dels processos.

El disseny d'aquest apartat és relativament simple i es mostra en l'algorisme 13.

Algorithm 13 Disseny Prioritats

```
1  do forever
2    curr_freq=get_traf();
3    freq=(P*freq)+((1-P)*curr_freq);
4    rel=freq/curr_freq;
5    if(rel > 1)new_priority++;
6    if(rel < 1)new_priority--;
7    for(i=0;PID[i]!= NULL;i++)
8      assign_priority(PID[i],new_priority);
9    endfor
10    wait();
11 enddo
```

La idea bàsica d'aquest algorisme és donar prioritat quan la freqüència de comunicació actual és superior a la obtinguda fins al moment, condició la qual indica que en aquests moments l'aplicació paral·lela està comunicant, i restar-ni quan aquesta deixa de comunicar. Sempre tenint en compte uns marges tant de màxim com de mínim d'aquestes prioritats.

Així doncs el que realitzem és l'obtenció de la freqüència de comunicació de l'últim interval de temps (2), freqüència que ve donada per la variable global definida en l'apartat 3.5.1 en forma de la pendent entre els paquets transferits i les lectures efectuades.

Un cop disposem d'aquesta freqüència *curr_freq* calculem la freqüència continua (*curr_freq*) o acumulada (*freq*) mitjançant la formula que ens permet complir la condició Coscheduling (3), com es descriu en l'apartat 2.3. I es calcula la relació entre aquestes dues freqüències (*rel*).

Posteriorment actuem en conseqüència dels resultats obtinguts (4-6), tenint en compte que en Linux una major prioritat implica un valor menor en aquesta i viceversa. I assignem aquesta nova prioritat a cada un dels processos identificats per als PIDs continguts en la nostra llista (7-9).

Per últim s'espera un temps prudencial abans de realitzar una nova planificació per no agreujar la major des-avantatja d'aquest sistema d'implementació del algorisme "Predictive Coscheduling", la sobrecarrega de CPU i el conseqüent retard.

Com es veu a continuació, en l'algorisme 14, la implementació es complica lleugerament donat que es té en compte diferents situacions sorgides durant l'experimentació.

Algorithm 14 Implementació assignació Prioritats

```
1   INICIALITZACIONS
2   while(intercanvi.getstat()<3){
3     curr_freq=intercanvi.calcular();
4     freq=(P*freq)+((1-P)*curr_freq);
5     if(curr_freq < 0.000001)curr_freq=0.000002;
6     rel=freq/curr_freq;
7     if(rel<0.0000001)rel=1.0;
8       if(rel > 1.25 && curr_freq > 0){
9       priority=priority+1;
10     }
11     else if (rel < 0.80 && curr_freq > 0){
12       if (rel < 0.5) priority=priority-2;
13       if (rel < 0.25) priority=priority-2;
14       priority=priority-2;
15     }
16      else priority=priority;
17     if(priority < MAX_PRI && priority > MIN_PRI);
18     else  if(priority <= MAX_PRI)priority=MAX_PRI;
19      else if(priority >= MIN_PRI)priority=MIN_PRI;
20     itPid.situarInici();
21     while(itPid.fi()==false && priority_ant!=priority){
22       curr_pid=strtoul(*itPid,NULL,10);
23       if(setpriority(PRIO_PROCESS,curr_pid,priority)!=0){
24         i=88;
25         rest=true;
26         break;
27       }
28       itPid++;
29     }
30     priority_ant=priority;
31     itPid.situarInici();
32     i++;
/****Re-lectura dels sockets de la aplicació****/
33     sleep(2);
34   }
```

Com en cada funció el primer pas es realitzar la declaració i inicialització de les variables. Es disposa de les corresponents variables a *curr_freq* i *freq*, que posteriorment es calculen, per a exercir la finalitat anteriorment descrita, la variable double *rel* per emmagatzemar-hi la relació (es necessari disposar de decimals ja que en cas de ser una diferencia negativa rel es trobara entre 0 i 1. La variable *curr_pid* contindrà el pid del procés al qual se l'hi modificarà la prioritat, i per últim es compta amb la variable auxiliar *i* que actuarà com a comptador.

S'inicia un bucle el qual no s'atura fins que no s'indiqui explícitament mitjançant la variable global *intercanvi* que es desitja finalitzar l'aplicació (2).

El primer pas és transferir el tràfic capturat en l'últim interval de temps (3), mitjançant la funció calcular de la variable *intercanvi* de classe *fil*. Una vegada es disposa de la freqüència de comunicació actual s'aplica l'algorisme "Predictive Coscheduling" per a obtenir la nova freqüència acumulada *freq* (4). I una vegada es disposa de les dues freqüències es calcula la relació entre ambdues, obtenint *rel* (6). En cas de haver-hi comunicació en l'últim interval de temps assignem un valor realment baix per evitar l'error que pot causar la divisió per 0 (5).

En funció de la diferencia existent entre *freq* i *curr_freq* es modifica la prioritat (7-19). Després d'un petit estudi realitzat s'han definit diferents increments i decrements de prioritat depenent de la diferencia entre freqüències. En el moment de realitzar aquestes diferències s'ha tingut en compte que no influís en excés les petites diferències en les comunicacions. Com es pot observar s'atorga una major importància alhora d'augmentar la prioritat, mentre que la disminució d'aquesta es suavitza, aquest fet es realitza amb aquest ordre per tal de facilitar una possible nova comunicació en un interval temps proper a l'ultima comunicació.

Una vegada realitzada la pertinent modificació de les prioritats, i comprovar que no excedeixin els límits definits (18-19) , s'assigna a cada un dels PIDs de l'aplicació paral·lela, la nova prioritat (20-29). En cas d'error en l'assignació de prioritats s'augmenta el comptador per tal de realitzar una re-lectura en breus moments.

D'acord amb les proves realitzades, s'espera una temps prudencial de 2 segons (33), intentant que sigui el menor possible però sense sobrecarregar el sistema i donar un interval de temps suficient per capturar una quantitat de paquets transcendental.

Dins d'aquest bucle s'inclou un comptador que realitza la funció de rellotge intern, de manera que augmente a cada iteració (2segons aproximadament). Aquest comptador s'inclou per realitzar re-lectures dels sockets. Aquestes re-lectures son efectuades per comprovar que no hagin variat els sockets o PIDs de l'aplicació paral·lela, puguen ésser que la aplicació paral·lela hagi afegit o eliminat sockets sense que el planificador ho tingués en compte, o s'hagués aturat l'aplicació en si.

Aquesta re-lectura es implementada en l'algorisme 15, coincidint en gran part amb el codi inicial del programa.

Algorithm 15 Implementació Re-lectura

```
/****Re-lectura dels sockets de la aplicació****/
1   if(i%90==0){
2      intercanvi.setstat(2);
3      rest=Bpids(argv[1]);
4      if(rest==true){
5        freq=1;
6        i=85;
7        priority=0;
8        priority_ant=0;
9      }
10      for(itPid.situarInici();itPid.fi()==false;itPid++){
11        Bsockets(*itPid,argv[1]);
12      }
13      itSock.situarInici();
14      while(itSock.fi()==false){
15        if(istcp(*itSock)==false){
16          socks.eliminar(itSock);
17        }
18        else{
19          itSock++;
20        }
21      }
22      intercanvi.setstat(0);
23      pthread_attr_init(&attr_param);
24      schd_param.sched_priority=19;
25      pthread_attr_setschedparam(&attr_param,&schd_param);
26      pthread_setschedprio(idFil, 19);
27      error=pthread_create(&idFil,&attr_param,funcioFil,NULL);
28      if (error != 0){
29        perror("No s'ha pogut crear el thread");
30        exit(-1);
31      }
32    }
```

En aquest algoritme es realitza tot el procés de busqueda, tant de PIDs com de Sockets de l'aplicació distribuïda, realitzant les mateixes funcions executades al iniciar el programa. Aquesta re-lectura es realitza com ja s'ha dit, a mode de refresc de l'aplicació paral·lela, ja que aquesta pot haver variat els seus processos o, donat que es tracta de forma conjunta totes les aplicacions paral·leles del mateix nom, s'hagi afegit una nova aplicació paral·lela.

Es comprova com s'ha dit que el comptador sigui un múltiple de 90 (valor definit a conseqüència de les proves realitzades) i en cas de ser-ho realitzem una re-lectura dels sockets que s'han de capturar (1). S'avisa, canviant l'estat de la variable *intercanvi* a 2 (2), al procés fill *funcioFil* que ha de finalitzar, donat que rellançarem un nou *thread* o procés fill amb els sockets corresponents.

Es realitzen totes les operacions pertinents per obtenir aquests sockets, funciones definides totes al llarg d'aquesta secció.

Primerament es busca els PIDs que corresponen a l'aplicació paral·lela (3), en cas de variar la variable *rest* ens ho indicara, i si es dona el cas implicarà que l'aplicació s'ha re-iniciat (4-9). En cas de que es re-iniciï l'aplicació, re-inicialitzem les prioritats i la freqüència acumulada, i assignem un valor proper al 90 per realitzar una nova re-lectura de sockets breument, això es deu a que durant el procés de d'arrancada l'aplicació va establint comunicació amb altres maquines i a mesura que les estableix va assignant sockets, aquest sistema implica un cert període de temps i molt probablement el nostre programa capturarà els sockets abans que aquest hagi finalitzat.

Una vegada definida la llista de sockets es disposa a recollir els sockets que cada un d'aquests PIDs ha creat (10-12), per a posteriorment filtrar de forma que únicament constin els sockets que pertanyen al protocol TCP (13-21).

Per últim rellancem el *thread* o procés fill amb la prioritat baixa i els nous sockets a capturar definits (22-32).

I continua en el punt on es trobava dins del bucle d'assignació de prioritats (algoritme 14).

4 Experimentació

En aquest capítol es mostren els resultats obtinguts en les proves realitzades en un clúster no-dedicat. Amb aquestes proves s'estudia el comportament de la tècnica "Predictive Coscheduling", implementada dins l'espai d'usuari en l'aplicació redactada en aquest projecte, davant les diferents possibles situacions que es poden a l'hora de posar en pràctica la planificació. Altrament les proves es realitzen sota diferents tipus de planificació per tal de poder comprar quina es la més eficient i en quins casos.

4.1 Entorn Experimental

Totes les proves han estat efectuades en un clúster no-dedicat el qual es composa de 8 equips de les següents característiques:

- Intel(R) Pentium(R) 4 CPU a 3.00GHz.
- 1MB de memòria cache.
- 1GB de memòria RAM.
- Sistema operatiu Red Hat Linux 3.2.2-5 amb kernel versió 2.4.22-ac4.
- Portable MPI Model Implementation Version 1.2.7p1

Les proves realitzades han estat realitzades sota tres tècniques de planificació diferents: la planificació original utilitzada per defecte en el nostre sistema operatiu Linux[20]; una segona tècnica "Predictive Coscheduling" implementada en Kernel, anul·lant així el planificador local; i una ultima tècnica que pertoca a la implementada en aquest projecte, novament usant la tècnica "Predictive Coscheduling" però en aquesta ocasió implementada en l'espai d'usuari i per tant no anul·lant directament el planificador local.

L'experimentació s'ha dut a terme utilitzant una aplicació distribuïda i dos aplicacions locals, que posteriorment es descriuen (apartat 4.1.2) i que es parametritzen per tal de configurar les diferents situacions.

A continuació es mostra com es realitza l'execució de l'aplicació implementada en aquest projecte i es descriu breument les citades aplicacions utilitzades per realitzar les proves.

4.1.1 Execució

En primer lloc s'ha de llençar l'aplicació que implementa el Coscheduling en cada una de les maquines que conformen la nostra NOW o clúster. En aquest cas, l'aplicació s'ha anomenat amb el nom de *CoschedCapt*.

Per a realitzar l'execució s'arrenca des d'usuari root el programa amb un sol paràmetre, el qual serà l'aplicació paral·lela a planificar.

root@node:$nohup nice -19 CoschedulingCapt Aplic, on *Aplic* es l'aplicació paral·lela que es desitja planificar.

El programa s'arrenca mitjançant la comanda *nohup* que s'utilitza per llençar aplicacions en segon pla i independitzar-les del *shell* o terminal en el qual estem treballant. D'aquesta manera evitem que ocupi un terminal o que al finalitzar la sessió s'aturi el programa, i no interfereix en l'usuari local. La comanda nice s'utilitza per donar la mínima prioritat al programa, i d'aquesta manera de interferir el mínim possible en els processos en execució de la màquina, tant els locals com els distribuïts, evitant així un retard innecessari en el sistema. Posteriorment es mostrarà la importància d'aquesta condició.

Donat que aquesta tasca pot resultar incomoda de realitzar s'han implementat una sèrie de *scripts*[4]que realitzen el llançament del planificador a cada un dels nodes, mitjançant el programa ssh[19] que ens permet accedir als diferents nodes mitjançant un canal segur. Com a paràmetres els *scripts* reben el nom de l'aplicació paral·lela que es desitja planificar, així ens evita la necessitat de variar el codi per a cada aplicació a controlar, altrament disposa d'unes variables fàcilment modificables que permet configurar el llançament d'aquests planificadors. Aquests *scripts* es troben amb el seu codi complert dins del CD adjuntat amb aquest projecte.

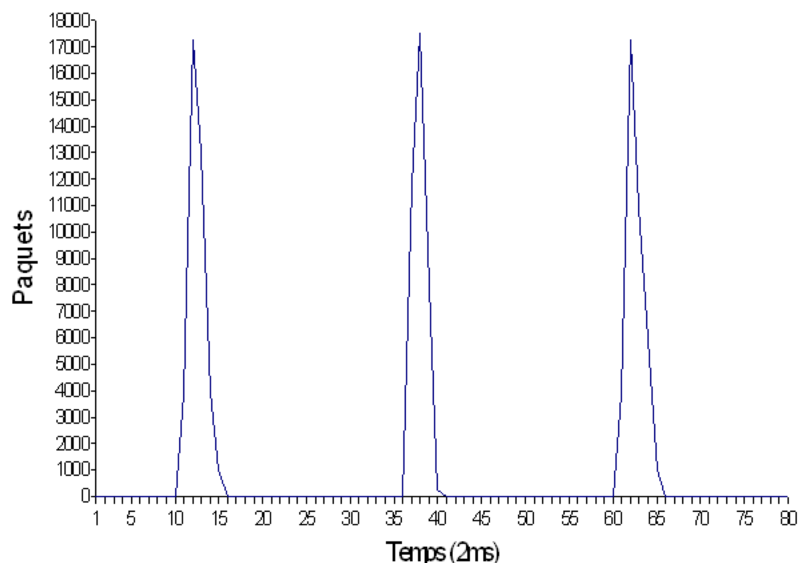
4.1.2 Aplicació Distribuïda/Local

L'aplicació distribuïda usada per efectuar les proves que es realitzen en aquesta experimentació és una aplicació anomenada sintree. Aquesta aplicació principalment el que fa és realitzar una sèrie de comunicacions iterativament (nombre d'iteracions les quals estan definides en una variable global) amb tots els nodes que conformen el clúster i aquests realitzen uns càlculs per posteriorment realitzar una comunicació en sentit invers. Així obtenim una rafega de transferències entre els nodes, controlada des dels paràmetres dins del codi, el qual es mostra en l'apèndix C.1.

Per a veure gràficament aquestes transferències es mostra una gràfica amb el tràfic capturat per la nostra aplicació. L'aplicació distribuïda has estat llençada passant-li com a paràmetres que efectues 3 *loops* (parametre *LOOP*) amb una comunicació de 6200 *doubles* (parametre N1*) en* 10000 missatges (parametre *COMH*). El funcionament d'aquests paràmetres es pot comprovar en el codi exposat en l'annex C.1.

[4]script: archiu que processa un conjunt d'instruccions proporciones, en aquest cas, per el s.o. Linux i en concret pel lleguantge Bash-Scripting.

Figura 2: Traffic sintree



En les proves presents en aquesta figura es poden observar clarament els 3 *loops* dels quals constava l'aplicació en aquell cas, bucles que poden ésser definits dins del codi de l'aplicació distribuïda, on efectuen les comunicacions.

Per altra banda pel que respecta a la simulació de recursos locals s'utilitzen dos programes diferents, un d'ells simula un usuari local mentre que l'altre consumeix el màxim de CPU possible durant un cert interval de temps fix.

En la simulació d'un usuari local es realitza un calcul constant de la càrrega mitja en funció d'uns paràmetres d'entrada durant un cert interval de temps, també definit per paràmetre, dins d'aquest període hi han intervals de calcul intensiu alternats amb temps d'espera però sempre mantenint un consum constant de la memòria. Per a simular la càrrega intensiva de CPU es calculen els n nombre primers basant-se en l'algorisme de Sieve Eratosthenes[21].

En el segon programa s'implementa una serie de càlculs que consumeixen el màxim de CPU que el planificador li permeti de forma continuada durant un període de temps fix, d'aquesta manera es força a l'aplicació distribuïda a un segon pla simulant un treball intensiu local sobre els nodes que conformen el clúster.

La implementació d'aquestes dues aplicacions locals es troba en els apendixos C.2 i C.3 respectivament.

4.2 Evolució de la implementació realitzada

A continuació es descriu breument l'evolució que ha sofert el programa de Coscheduling per solventar els inconvenients apareguts i millor el rendiment d'aquest.

- Primerament es va plantejar un programa en C++ que únicament llances diferents scripts implementats en Bash-Scripting[18] que realitzaven la busqueda, la lectura dels sockets i la planificació dels processos, de manera que el programa en C era únicament una interfície per arrencar

o aturar aquests scripts. Aquesta opció va ser ràpidament descartada donat que les funcions que realitzaven els scripts eren molt costoses a nivell de CPU i la comunicació entre ells implicava la manipulació de fitxers temporals que incrementaven el temps d'execució. També es va detectar certes diferencies de llenguatge entre les diferents distribucions de Linux i corresponents versions de *bash*, la qual cosa reduïa la compatibilitat del programa.

- Posteriorment es va dissenyar el programa descrit en l'apartat 3, tot i que en un principi s'utilitzaven fitxers per efectuar la comunicació entre el programa principal i el procés fill *thread* . Aquestes lectures i escriptures a fitxers també retardaven significativament la resta d'aplicacions del node, consumint CPU en excés. Així s'opta per les variables globals en forma de llista per facilitar l'implementació i paral·lelament disminuir drasticament el consum de CPU, tot i que augmentant lleugerament el consum de memòria.

- Una vegada s'ha implementat el disseny existeixen certs valors que s'han de sotmetre a una serie de proves per tal de calibrar-los, i buscar el major rendiment de la tècnica Coscheduling però alhora el menor consum de CPU, i evitar així el retard afegit. Les proves realitzades per atribuir un valor a aquests paràmetres son resumides en els següents apartats.

4.2.1 Prioritat a assignar

Un dels aspectes més importants d'aquesta aplicació creada en aquest projecte es la assignació de prioritats a les tasques paral·leles.

Com s'ha explicat en l'apartat 3.6, segons el tràfic capturat cada cert interval de temps, s'assigna una nova prioritat a cada un dels processos de l'aplicació distribuïda la qual estem planificant. Aquesta assignació ha de realitzar-se d'una forma relativament independent al nombre absolut de paquets enviats o rebuts, donat que cada aplicació realitza transferències de diferents longituds. És per aquesta raó, tal com es veu en la implementació (apartat 3.6), que s'utilitza la relació entre traffic anterior (*freq*) i actual (*curr_freq*) per realitzar la modificació de prioritat.

En base a les proves efectuades, s'han definit les relacions entre prioritat i freqüències de comunicació que es veuen implementades en l'apartat 3.6. Donant una major importar al augment de la prioritat alhora de comunicar i frenant la disminució d'aquesta al deixar de fer-ho, per una possible nova comunicació a curt termini.

4.2.2 Interval de lectures

Un paràmetre important a tenir present és l'interval de temps entre lectura i lectura al fitxer "/proc/net/tcp". Aquest interval ha de ser el menor possible per tal d'assegurar la màxima precisió i fidelitat. Per altra banda quan menor és l'interval d'espera més consum de CPU produirà l'aplicació. Així doncs és busca l'equilibri entre la fiabilitat i el retard afegit.

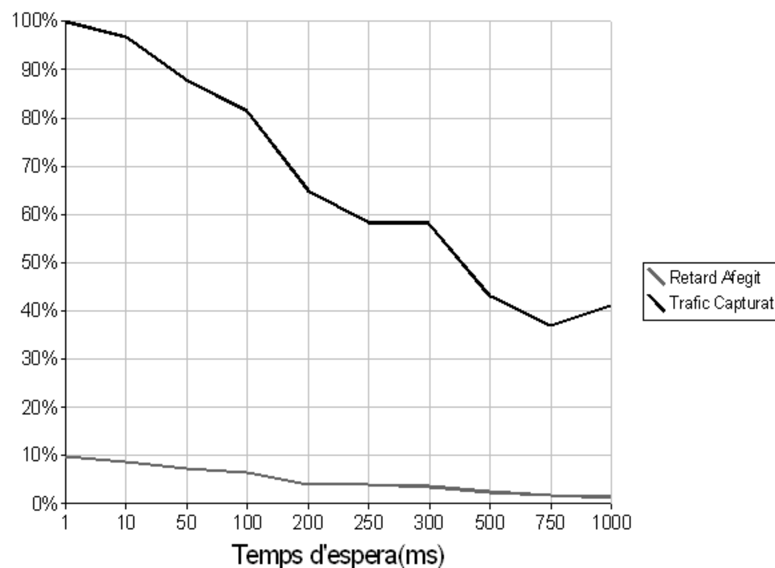
Per a dur a terme aquesta comprovació s'han realitzat proves variant l'interval de temps entre lectures i se'n ha extret el trafic capturat per l'aplicació que realitza la tècnica "Predictive Coscheduling"

i el retard que aquesta afegeix al sistema. Aquest trafic capturat s'ha extret imprimint els valors que anava retornant el programa al recaulcular la prioritat, es a dir, els valors de *curr_freq*. Aquests valors han estat sumats i les relacions entre aquestes sumes dels diferents valors de l'interval es el que es pot veure en la propera gràfica.

Per altra banda el retard afegit al sistema es calcula observant el temps que tarda l'aplicació distribuïda en finalitzar en cada una de les situacions. Al temps que tardaven se'ls hi ha extret el temps original, el temps que tarda en finalitzar l'aplicació distribuïda amb el planificador local original, i la resta d'aquest calcul s'ha dividit per aquest mateix temps original per extreure'n la relació, en percentatge.

Realitzades aquestes proves representem les dades obtingudes en percentatges, de manera que amb un interval de valor mínim (1ms), es comptabilitza com el 100% de tràfic capturat. De la mateixa manera és necessari representar el retard que la variació d'aquest paràmetre implica, en aquest cas el 0% serà el mínim retard afegit, es a dir el temps de comput original.

Figura 3: Trafic Capturat vs Retard CPU



Amb aquest gràfic s'observa la freqüència de comunicació, representada en percentatge, així com el retard afegit pel mateix interval d'espera. Pel que respecta al tràfic capturat per la nostra aplicació es pot observar el descens més dràstic es dona a partir dels 300ms, on la freqüència de comunicació capturada descendeix del 50% respecte al inicial. Altrament s'observa que el retard afegit segueix una pendent bastant uniforme.

Donat que el retard afegit significa un major consum de CPU per part de l'aplicació que realitza la planificació, implicant un retard en la resta de processos del node, tan tasques locals com distribuïdes, se l'hi ha atorgat, en aquest treball, una major prioritat al fet de reduir el retard afegit al sistema. Així doncs establint una fiabilitat relativa i procurant no descendir del 50% de tràfic capturat respecte al inicial, i donat que es desitja el mínim retard possible, el valor definit per aquest paràmetre és de 300ms.

Així doncs l'interval d'espera entre lectura i lectura al fitxer "/proc/net/tcp" és de 300ms.

Desprès de realitzar una sèrie de proves, execucions de tasques paral·leles i locals simultaneament, s'opta per augmentar encara més aquest interval de temps. Així és degut a que segueix aportant un retard massa elevat en els nodes i aporta un decrement del rendiment inclús en comparació amb l'execució d'aquestes mateixes tasques paral·leles i locals sense cap planificador addicional.

S'opta doncs per l'interval que comporta el menor consum de CPU, i donat que el tràfic capturat entre els últims intervals comprovats varia mínimament el nou valor adjudicat al interval de temps d'espera entre lectura i lectura és de 1000ms. Així es sacrifica la fiabilitat per obtenir un millor rendiment, però cal recordar que en les proves anteriors el tràfic capturat amb aquest interval d'espera no descendeix del 40% del total, i donat que la funció d'aquest projecte no es la de comptar o obtenir tot el tràfic d'una aplicació distribuïda sinó millorar el seu rendiment i temps de resposta, es tracta d'una fiabilitat suficient si així s'assegura la finalitat del planificador.

4.3 Realització de les proves

A continuació es realitzen una sèrie de proves per obtenir les diferències entre el rendiment d'un seguit d'aplicacions paral·leles executades en diferents situacions.

Per veure el comportament del Coscheduling i com afecta en el rendiment de les tasques del clúster, s'han definit estats que intenten simular les diferents situacions en que es pot trobar un clúster. Aquests estats es basen en dues variables, la intensitat del tràfic i el consum de recursos. Així doncs es realitzen les proves jugant amb el valor d'aquestes variables, havent-hi per exemple unes proves amb una alta intensitat de tràfic i un baix consum de recursos per part de les aplicacions locals o viceversa, formant així un entramat de situacions que ens mostren els diferents comportament dels planificadors.

Per tal d'assegurar una màxima fiabilitat s'han realitzat vàries proves en cada una de les situacions i se'n ha extret la mitjana d'aquests valors, els quals són els mostrats els següents sub-apartats.

A continuació es mostren els resultats obtinguts en les diferents proves interpretant breument les dades. Aquests resultats es mostren en forma de temps; el temps que s'ha tardat en realitzar tota l'execució de les diferents aplicacions. Per oferir una visió més objectiva i restar importància als valors concrets es representa aquest interval de temps en percentatges, essent sempre el 100% l'interval de temps que tarda la prova realitzada sense cap planificador addicional, usant el planificador local del sistema operatiu.

4.3.1 Aplicació Distribuïda en Solitari

En aquestes proves es calcula el temps que tarda en executar-se l'aplicació distribuïda, definida en l'apartat 4.1.1, en els seus dos possibles estats, el primer amb una baixa intensitat de tràfic de xarxa i el segon augmentant aquesta intensitat mitjançant l'execució simultània de tres d'aquestes aplicacions distribuïdes. En aquestes proves no s'ha executat cap aplicació local.

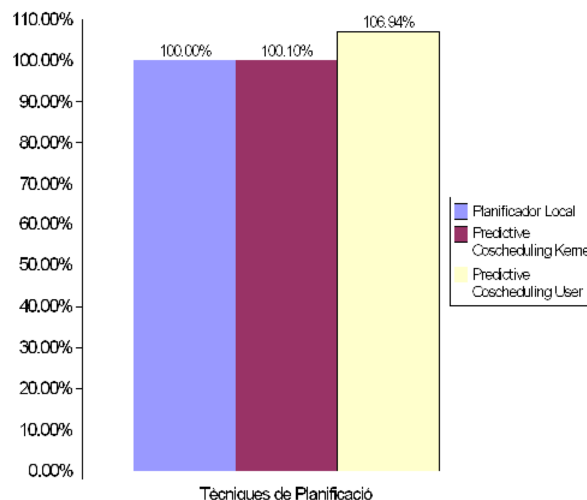
Els paràmetres que s'han definit en l'execució d'aquesta aplicació distribuïda son els següents:

- Nombre de *doubles* a enviar (per missatge): 6200

- Nombre de missatges a enviar (per *loop*): 10000
- Nombre de *loops* o iteracions: 5

Baixa intensitat de comunicació

Figura 4: Aplicació Distribuïda de baixa intensitat

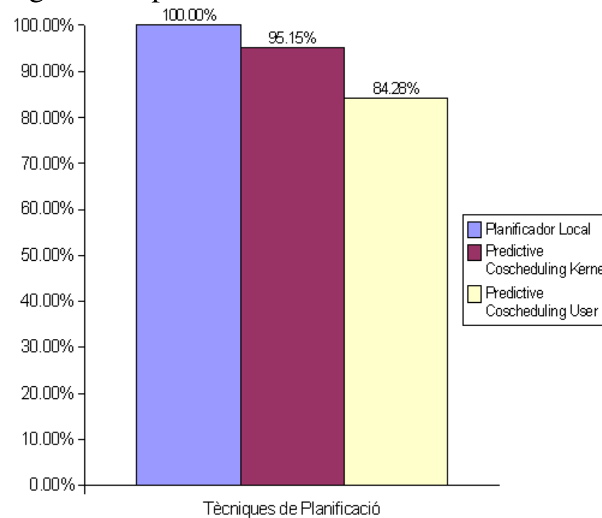


La gràfica mostrada en la Fig.4 representa el temps que tarda l'aplicació paral·lela en finalitzar la seva execució. En aquest cas no hi han tasques locals que puguin interrompre o afegir un retard sobre l'aplicació distribuïda.

En aquesta figura es pot apreciar com l'aplicació llençada en la tècnica "Predictive Coscheduling" en el kernel i el planificador local original, no hi ha pràcticament diferència de temps. Mentre que al llençar-la amb la tècnica "Predictive Coscheduling" realitzada en l'espai d'usuari per la nostra aplicació, es pot observar un lleuger increment de temps. Això es degut a que el Coscheduling en l'espai d'usuari consumeix, en certa mesura, recursos del sistema, mentre que les altres dues tècniques, al estar directament implementades al nucli del sistema (Kernel), no requereixen de recursos addicionals.

Alta intensitat de comunicació

Figura 5: Aplicació Distribuïda d'alta intensitat



En aquesta ocasió la prova s'ha realitzat llençant simultaneament tres aplicacions paral·leles iguals per tal de simular una major comunicació i demanda de recursos per part de l'aplicació paral·la. En aquest cas s'inverteixen les situacions respecte a l'anterior prova, i es la tècnica "Predictive Coscheduling" aplicada en l'espai d'usuari la que mostra un millor rendiment.

Una possible causa d'aquesta diferencia entre l'implementació en l'espai d'usuari i la del Kernel pot ser degut a que l'implementació en el Kernel tracta cada aplicació paral·lela de forma independent, mentre que l'implementació realitzada en aquest projecte les tracta totes com una sola aplicació distribuïda, mantenint una uniformitat que pot afavorir la comunicació de xarxa.

4.3.2 Aplicació Local en Solitari

En les següents dues proves s'executen les diferents aplicacions locals, sense cap aplicació paral·lela, i es comprova si existeix un retard afegit per part d'alguna tècnica. Com es veu a continuació les variacions són pràcticament imperceptibles, tal com correspon ja que al no haver-hi aplicació paral·lela la tècnica "Predictive Coscheduling" no ha d'actuar i, en el cas de la nostra aplicació, es manté a l'espera d'una tasca distribuïda.

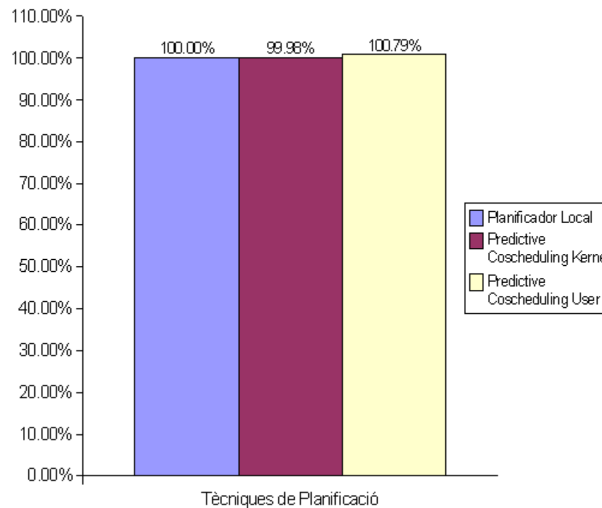
Tant en el primer cas, de consum moderat, com en el segon les aplicacions son llençades en cada un dels nodes que conformen el clúster, en aquest cas els 8 nodes.

En la primera de les aplicacions locals els paràmetres que s'han utilitzat son els següents:

- Temps d'execució: 300s
- Percentatge de CPU usada (durant els períodes d'alta intensitat): 45%
- Memòria consumida (constant): 30%

Consum moderat de recursos

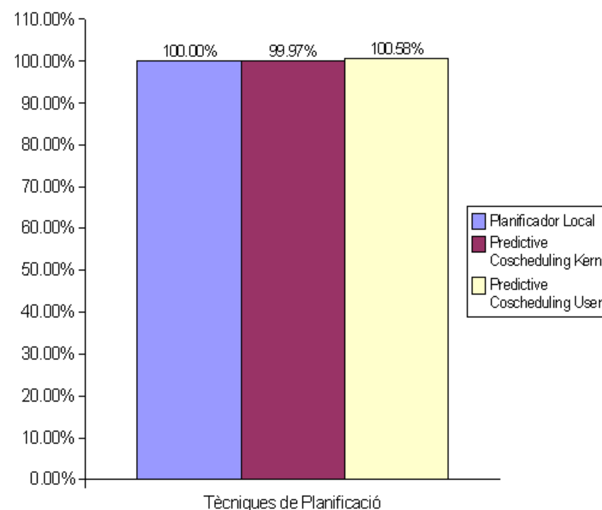
Figura 6: Aplicació Local 1



Alt consum de CPU

En aquesta segona aplicació no hi ha paràmetres definits ja que l'aplicació es de durada fixa i consum de CPU màxim.

Figura 7: Aplicació Local 2

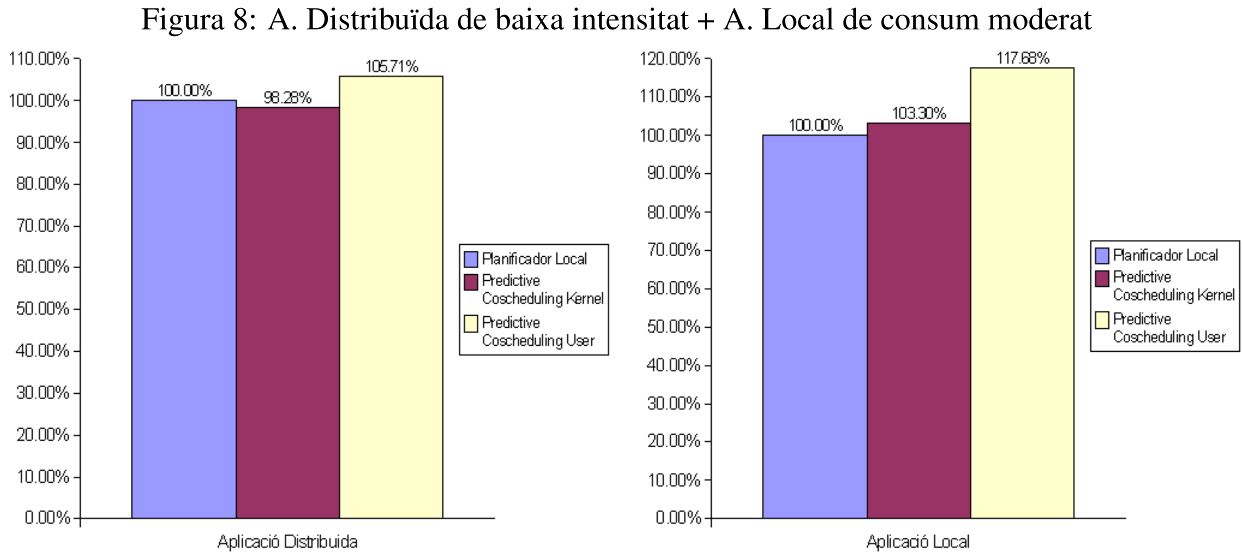


4.3.3 Aplicació Distribuïda + Aplicació Local

En les següents proves realitzades es llancen simultaneament aplicacions paral·leles i locals en tots els nodes que formen el clúster, variant el grau d'intensitat tant de les tasques paral·leles com de les locals. Així es comprova el comportament que tindran les diferents tècniques donat un consum local en el clúster, que consumeix part dels recursos de comput que pertocarien a l'aplicació distribuïda en cas de ser l'única execució del s.o..

Apl. Distribuïda de baixa intensitat de comunicació + Apl. Local de consum moderat de recursos

Figura 8: A. Distribuïda de baixa intensitat + A. Local de consum moderat



En aquesta primera prova es simula el que seria la situació més comuna en un clúster on hi intervenen usuaris locals. Es tracta d'una aplicació paral·lela compartint els recursos amb unes aplicacions locals, executades en cada un dels nodes, que pretenen simular a l'usuari local comú, amb intervals d'alt consum de CPU alternats amb estats d'espera i ús de memòria; aplicació local descrita en l'apartat 4.1.1.

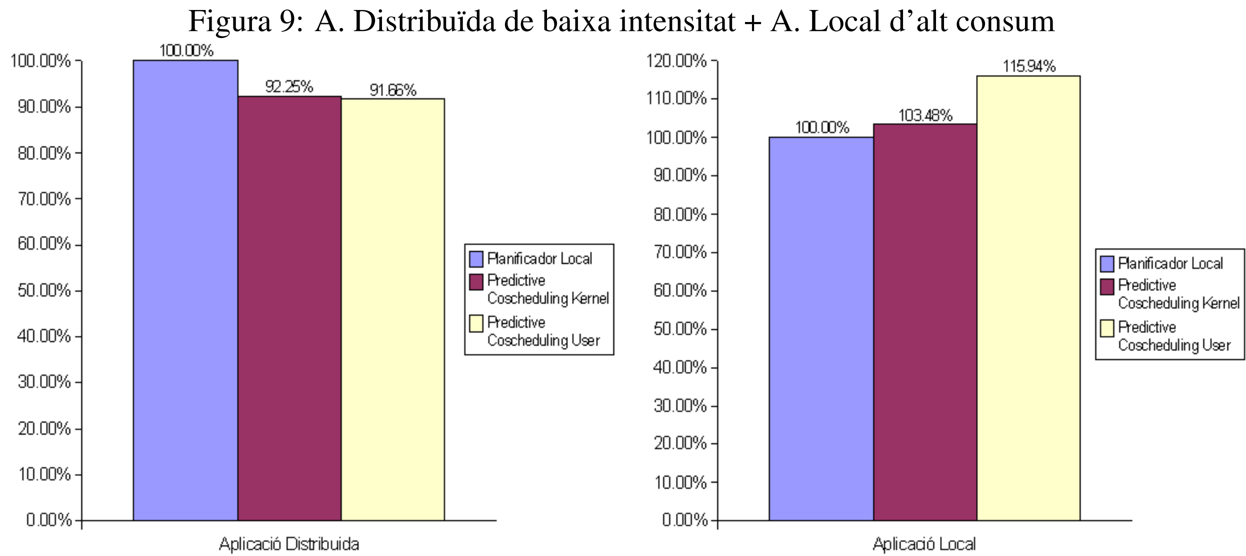
Es pot observar que el comportament de la tècnica "Predictive Coscheduling" implementada en el Kernel realitza la seva funció, millorant el rendiment de l'aplicació distribuïda i convertint aquesta millora en pèrdua per part de les aplicacions locals. Per contra l'aplicació que realitza la planificació en l'espai d'usuari no compleix el seu objectiu, i endarrereix tant l'aplicació distribuïda com les aplicacions locals. Aquesta situació es deu a que el retard afegit per la nostra aplicació es superior al retard afegit per les aplicacions locals i per tant no és rendible realitzar la planificació "Predictive Coscheduling", en aquest cas, en l'espai d'usuari ja que inclús sense cap planificació addicional, únicament amb el planificador local original, s'obté un millor rendiment.

Aquestes proves mostren també un fort augment del temps d'execució de les aplicacions locals sota la planificació en l'espai d'usuari. Com es veurà a continuació aquesta característica es constant en totes les proves, i és degut principalment a que la planificació "Predictive Coscheduling" consumeix, en la mínima quantitat possible, una sèrie de recursos que la implementada en el kernel. Un altre factor que afavoreix aquesta major diferència en el comportament de les aplicacions locals és el factor que la planificació implementada en Kernel actua sobre un usuari del s.o. mentre que la nostra aplicació ho fa únicament sobre l'aplicació paral·lela, independentment de l'usuari que estigui executant-la. Aquest fet afavoreix, en el cas de l'implementació en Kernel, a totes les aplicacions executades per l'usuari, tot i que en major mesura a les aplicacions paral·leles, i en cas de la implementació en l'espai d'usuari, la planificació de l'aplicació paral·lela de forma independent no

comporta cap avantatja cap a les aplicacions locals.

Apl. Distribuïda de baixa intensitat de comunicació + Apl. Local d'alt consum de CPU

Figura 9: A. Distribuïda de baixa intensitat + A. Local d'alt consum



En aquests gràfics es mostra les relacions entre els temps d'execució d'una aplicació paral·lela amb un seguit d'aplicacions locals, una a cada un dels nodes, que consumien el màxim de CPU que el s.o. li permetia.

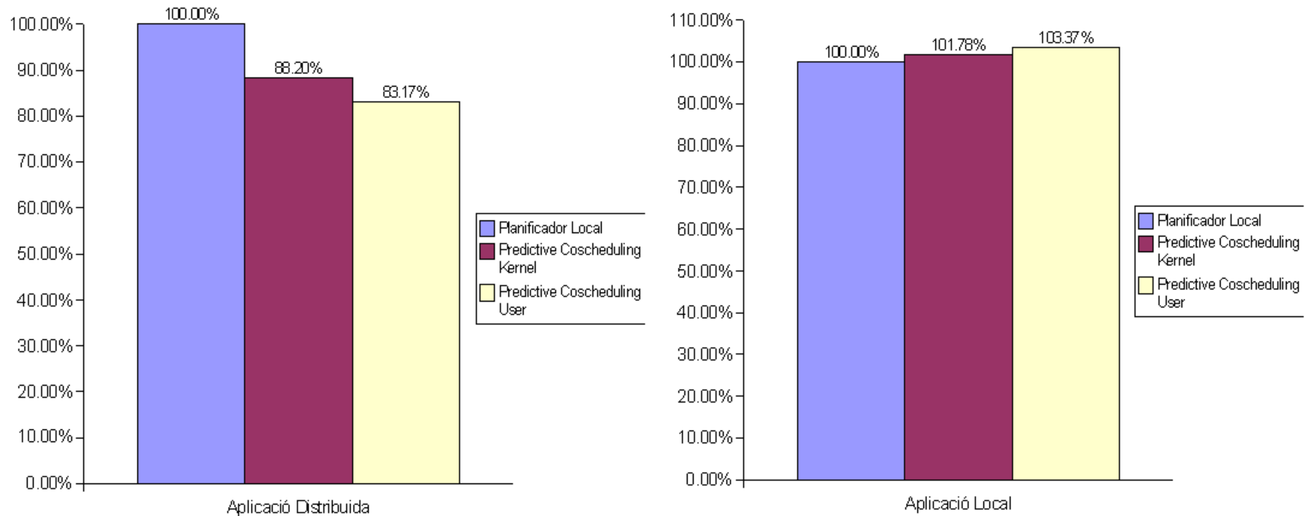
L'aplicació en l'espai d'usuari que planifica la tasca distribuïda ja compleix el seu objectiu, millorar el rendiment de l'aplicació distribuïda, superant inclús el rendiment obtingut utilitzant la tècnica "Predictive Coscheduling" implementada en el Kernel. Ambdues tècniques milloren considerablement el temps d'execució respecte a l'original, notant doncs que amb una major demanda de recursos per part de l'usuari local major és la millora de rendiment usant les tècniques "Predictive Coscheduling" respecte a la planificació original.

Altrament es torna a comprovar que la nostra aplicació aporta un retard important en l'aplicació local, molt superior al que comporta l'implementat en Kernel.

Apl. Distribuïda d'alta intensitat de comunicació + Apl. Local de consum moderat de recursos

En les següents proves es llencen vàries aplicacions paral·leles per tal de provocar una major comunicació en la xarxa i comprovar com es comporta en aquests casos. Com s'ha vist en l'apartat 4.3.1 el programa realitzat en aquest projecte, implementat en l'espai d'usuari, ja es comporta de forma més optima que la tècnica "Predictive Coscheduling" implementada dins el Kernel quan ens trobem sota aquesta situació concreta (aplicació distribuïda d'alta intensitat) , així doncs el coherent seria que aquesta relació continues.

Figura 10: A. Distribuïda d'alta intensitat + A. Local de consum moderat

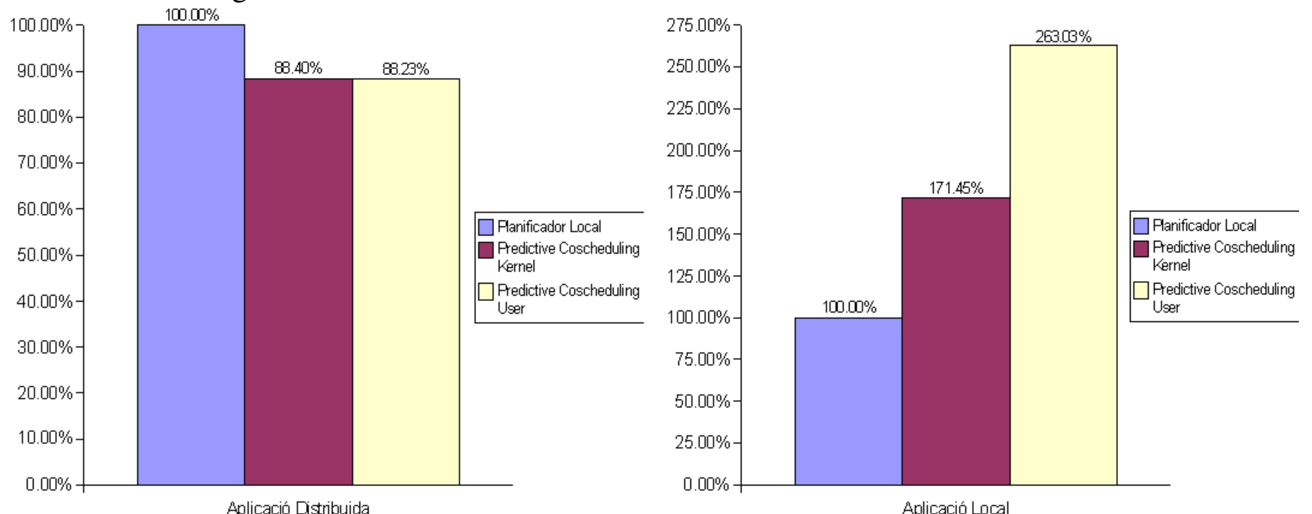


Les gràfiques ens tornen a mostrar el temps d'execució de les aplicacions distribuïdes i locals respectivament.

La millora de rendiment pel que respecta a l'execució paral·lela segueix millorant, i es manté la relació citada anteriorment, on la implementació realitzada en aquest projecte millora la implementada en el Kernel, a l'igual que es manté la relació, negativa en aquest cas, respecte al rendiment més pobre de l'aplicació local en cas d'executar-se sota la planificació "Predictive Coscheduling" implementada en l'espai d'usuari que respecte la implementada en el Kernel.

Apl. Distribuïda d'alta intensitat de comunicació + Apl. Local d'alt consum de CPU

Figura 11: A. Distribuïda d'alta intensitat + A. Local d'alt consum



Seguim amb una alta intensitat de tràfic de xarxa, mitjançant vàries aplicacions paral·leles executades simultàniament, però en aquest cas s'aplica també una alta demanda de recursos de CPU per part de l'usuari local, mitjançant la citada aplicació local (apartat 4.1.1).

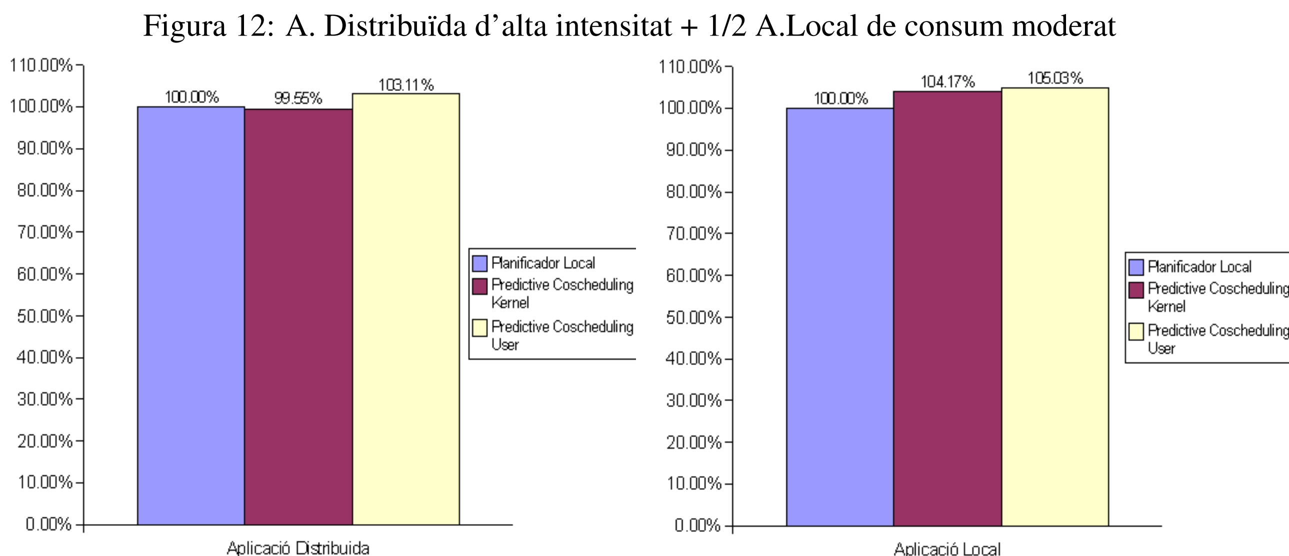
Es manté una millora de rendiment en l'aplicació distribuïda per part de les dues implementacions de la tècnica "Predictive Coscheduling" pràcticament idèntica. Però en aquest cas crida l'atenció la desmesura en que es retarda a les aplicacions locals. Degut a la alta freqüència de comunicació l'aplicació distribuïda rep una prioritat favorable durant pràcticament tota la seva execució i això comporta una disposició de recursos que al mateix temps reclama l'aplicació local. Donat que la demanda d'aquests recursos locals és molt més agreujada en aquest cas que en l'anterior, on les demandes són puntuals, és reflecteix amb el mateix agreujament el retard que aporta la tècnica "Predictive Coscheduling" aplicada en l'aplicació paral·lela, tant l'implementada en el Kernel com en l'espai d'usuari, tot i que en aquesta última és alhora substancialment mes apreciable aquest retard.

4.3.4 Aplicació Distribuïda + Aplicació Local Parcialment

Rarament en un clúster es troben tots els nodes ocupats per usuaris locals, es per això que es realitzen algunes proves, similars a les realitzades anteriorment, on únicament es troben ocupats localment la meitat dels nodes que conformen el clúster.

Apl. Distribuïda de baixa intensitat de comunicació + 1/2 Apl. Local de consum moderat de recursos

Figura 12: A. Distribuïda d'alta intensitat + 1/2 A.Local de consum moderat



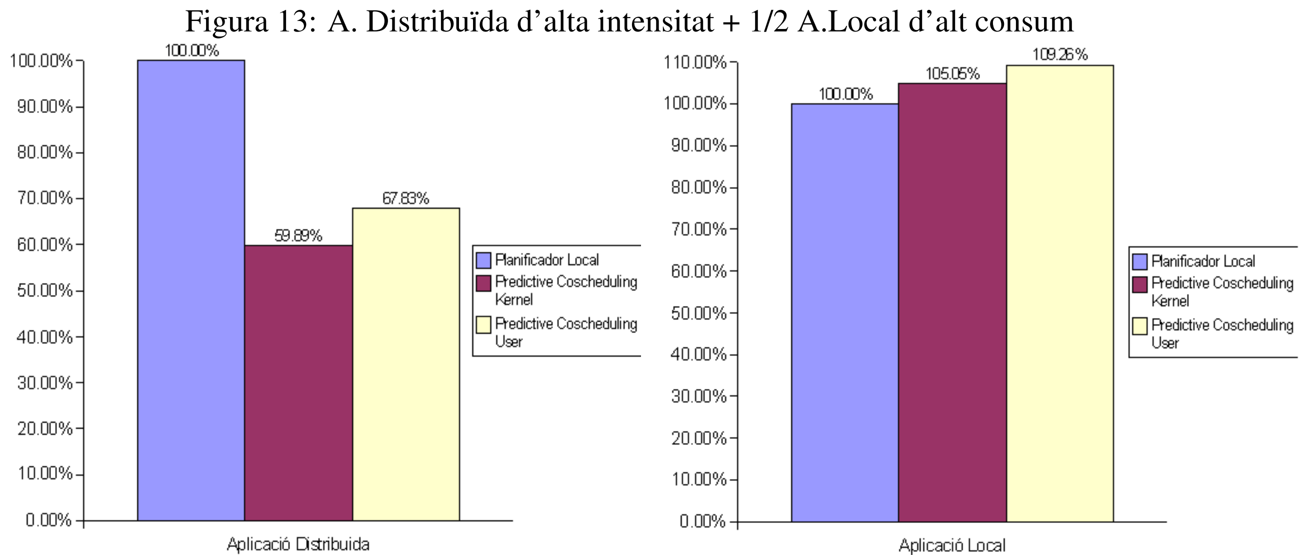
En aquesta prova es llença una aplicació paral·lela per tot el clúster i un seguit d'aplicacions locals executades únicament en la meitat d'aquests nodes.

Els resultats obtinguts són molt semblants als obtinguts en la mateixa prova però amb demanda local en tots els clústers. Apuntar que en aquest cas l'implementació en l'espai d'usuari millora lleugerament, tant en el que respecta a l'aplicació distribuïda com en la local, respecte a la prova amb aplicacions locals en tots els nodes, tot i no arribar al rendiment ni tan sols del planificador original. Al contrari que l'implementació de la tècnica "Predictive Coscheduling" implementada en l'espai

d'usuari, la implementada en Kernel disminueix lleugerament el seu rendiment, novament en ambdós entorns, distribuït i local, tot i que es pràcticament imperceptible.

Apl. Distribuïda de baixa intensitat de comunicació + 1/2 Apl. Local d'alt consum de CPU

Figura 13: A. Distribuïda d'alta intensitat + 1/2 A.Local d'alt consum



Continuant amb la mateixa aplicació distribuïda i amb una demanda local, tot i que en aquest cas es tracta de l'aplicació local que aporta un alt consum de CPU, en la meitat del clúster s'obtenen els resultats mostrats ens la figura 13.

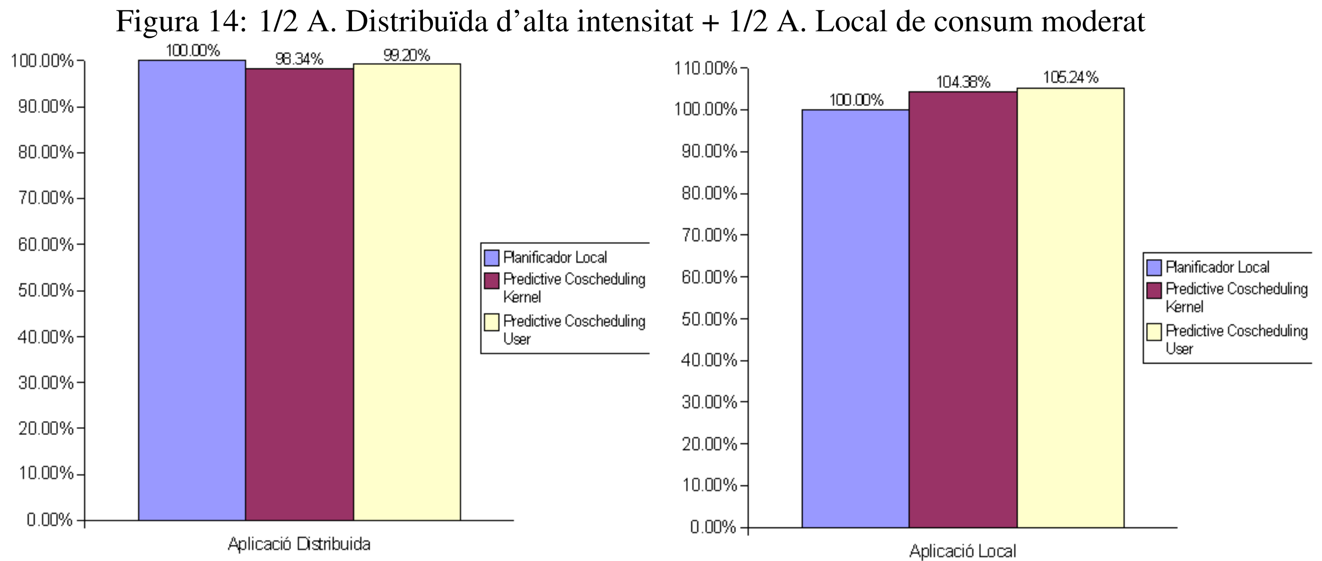
Es pot apreciar una millora del rendiment de quasi el 50% en cas de l'implementació en el Kernel i no menys notable en el cas de l'implementació d'aquest projecte. Una gran diferencia comparada amb la moderada millorada que es troba en la mateixa prova realitzada anteriorment, amb la diferencia de disposar d'aplicacions locals en tots els nodes, millora que inclús s'aprecia en l'aplicació local en el cas de l'implementació en l'espai d'usuari, tot i que novament és la que aporta un major retard.

Com ja s'ha explicat un dels objectius de la tècnica "Predictive Coscheduling" és mantenir una certa sincronització entre els nodes d'una mateixa tasca distribuïda.. En les anteriors proves tot el clúster tenia una demanda uniforme tant de recursos locals com distribuïts, així doncs la sincronització ja es complia en certa manera sense necessitat d'un planificador extraordinari. En aquesta prova l'aplicació paral·lela es troba amb nodes que li atorguen tots els recursos disponibles mentre que d'altres es disputen amb l'usuari local, és aquí on l'aplicació distribuïda s'endarrereix si no disposa d'una bona tècnica de planificació. I aquesta funció es la que duu a terme la tècnica "Predictive Coscheduling", a l'assignar una major prioritat a l'aplicació distribuïda en aquells casos en hi ha comunicació; d'aquesta manera es millora la sincronització entre els nodes que conformen el clúster.

1/2 Apl. Distribuïda de baixa intensitat de comunicació+ 1/2 Apl. Local de consum moderat de recursos

Per provar una de les assumpcions citades en l'apartat 2.1 sobre el model per a sistemes Clúster (*CMC*), es realitzen les dues últimes proves però aplicant la tècnica "Predictive Coscheduling" únicament en els nodes on hi ha una demanda de recursos locals.

Figura 14: 1/2 A. Distribuïda d'alta intensitat + 1/2 A. Local de consum moderat



Per primera vegada en aquesta situació es pot apreciar una lleugera millora del rendiment de l'aplicació distribuïda per part de la tècnica "Predictive Coscheduling" implementada en l'espai d'usuari, tot i no arribar a superar el rendiment obtingut amb la implementació en Kernel. Millora la qual es deu a l'exclusió del consum de recursos innecessari per part de l'aplicació implementada en aquest projecte, que tot i que ser mínima pot reflectir-se en aquest 2 o 3% en el temps d'execució total de l'aplicació paral·lela. La resta es manté amb unes diferències de rendiment imperceptibles.

1/2 A. Distribuïda de baixa intensitat + 1/2 A. Local d'alt consum de CPU

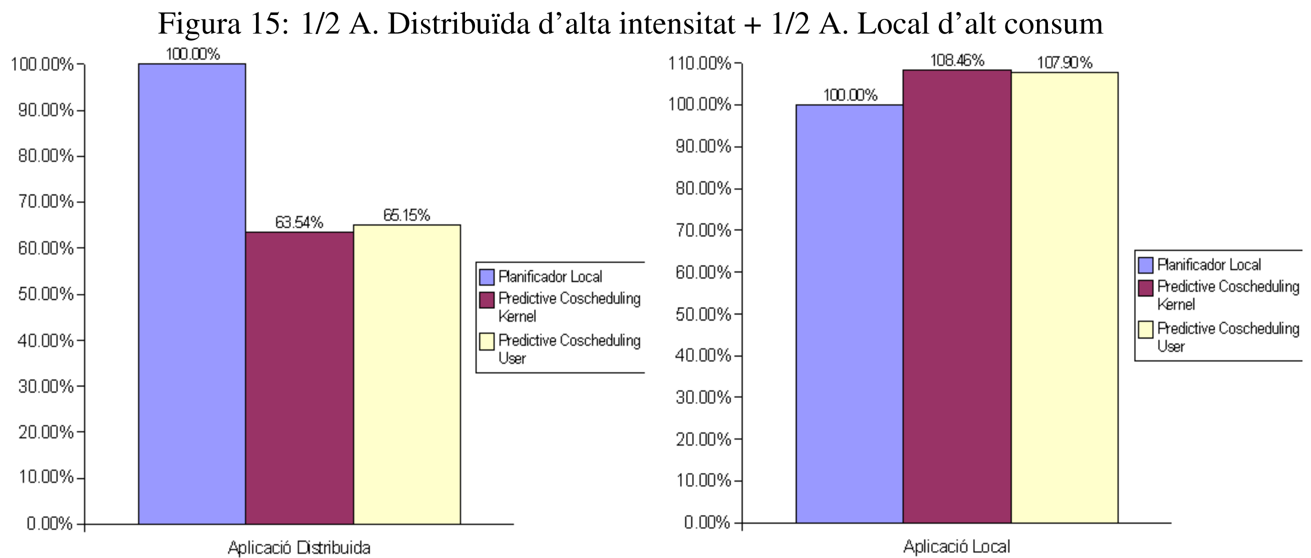


Figura 15: 1/2 A. Distribuïda d'alta intensitat + 1/2 A. Local d'alt consum

En aquest cas es torna a donar un augment del rendiment molt alt tant per la tècnica "Predictive Coscheduling" implementada en Kernel com la implementada en l'espai d'usuari. Tot i que, novament per primera vegada, l'implementació realitzada en aquest projecte iguala, inclús millora ínfimament, el rendiment de les aplicacions locals respecte a l'implementació de la tècnica "Predictive Coscheduling" en Kernel. Degut novament a la mateixa raó que provoca la millora del rendiment d'aquesta implementació en la prova immediatament anterior.

4.4 Conclusions de l'experimentació

Un cop analitzats els resultats obtinguts de l'execució de les proves anteriors, se'n dedueixen les següents conclusions:

- La conclusió més evident, donat que s'ha complert en totes les proves a excepció de l'ultima, és que l'implementació de la tècnica "Predictive Coscheduling" en el nostre programa comportarà sempre un increment en el temps d'execució de les tasques locals, superior en tots els casos al que implica l'implementació d'aquesta tècnica de planificació en el Kernel, essent aquest temps alhora superior al que s'obté amb el planificador local. Així doncs abans d'incorporar aquesta aplicació per a que realitzi la tècnica "Predictive Coscheduling" es necessari preguntar-se quina importància tenen els usuaris locals, i les seves respectives tasques locals, dins d'aquest clúster.

- Per altra banda en el que respecta a l'entorn de les tasques distribuïdes es pot comprovar que millora sempre que ens trobem sota altes demandes de recursos, tant per part de les tasques locals com distribuïdes. És a dir, si es disposa d'una alta freqüència de comunicació per part de les aplicacions distribuïdes o un alt consum de recursos per part dels usuaris locals, la tècnica "Predictive Coscheduling" millora clarament el rendiment de les aplicacions distribuïdes. En

la majoria d'aquests casos on es troba una alta demanda de recursos, tant en l'entorn local com distribuït, l'implementació de la tècnica "Predictive Coscheduling" en l'espai d'usuari millora lleugerament les aplicacions distribuïdes respecte a la implementada en el Kernel, ambdós millorant sensiblement respecte al planificador local original del s.o.. Contràriament si es disposa d'una freqüència de comunicació relativament baixa i una demanda local igualment poc elevada, la tècnica de planificació "Predictive Coscheduling" únicament es eficient en l'implementació al Kernel, l'implementada en aquest projecte aporta un retard superior a la millora que pot aportar sobre les aplicacions distribuïdes.

- S'aprecia com en cas de tenir un consum local no uniforme, es a dir, no mantenir la mateixa demanda de recursos en tots els nodes del clúster, el planificador "Predictive Coscheduling" és molt més efectiu donada la major sincronització de que disposen cada un dels nodes que conformen l'aplicació distribuïda, gràcies a la tècnica de planificació.

- En cas de no tenir càrrega local i executar únicament un aplicació distribuïda no és eficient incorporar la tècnica "Predictive Coscheduling" mitjançant la implementació en l'espai d'usuari, donat que, encara que sempre buscant el mínim possible, afegeix un retard degut al consum d'aquesta aplicació. De la mateixa manera si en el clúster únicament hi ha càrrega local i no s'ha d'executar cap aplicació distribuïda no aporta cap benefici l'implementació d'aquest projecte de la tècnica "Predictive Coscheduling", tal qual era d'esperar. Tanmateix en cas de no tenir càrrega local i disposar de varies aplicacions distribuïdes executant-se el disposar de la tècnica "Predictive Coscheduling" ens aportarà un millor rendiment en aquestes aplicacions, sigui la tècnica implementada en Kernel o, la realitzada en aquest projecte, en l'espai d'usuari, essent més notable la millora en aquesta segona implementació.

5 Conclusió

L'objectiu d'aquest projecte era el d'implementar una aplicació en l'espai d'usuari que fos capaç de planificar tasques distribuïdes utilitzant la tècnica "Predictive Coscheduling" i comparar-la amb la implementació d'aquesta mateixa tècnica en el Kernel. Després d'observar els resultats obtinguts en l'experimentació es pot afirmar que s'ha complert, i no només s'ha pogut realitzar aquesta planificació sinó que s'arriba a superar, depenent de la situació, a l'implementació realitzada en el Kernel.

El major problema plantejat en el disseny i implementació d'aquesta aplicació en l'espai d'usuari ha estat el control del consum de recursos. Al ser una aplicació independent requereix els seus propis recursos del sistema, demanda la qual no es tan greu en altres entorns com en el Kernel, on s'integra amb la tècnica de planificació existent, i aquest fet ha condicionat molt l'implementació del programa. Alhora d'implementar la planificació en si, l'assignació de prioritats, s'ha hagut d'anar adaptant a les reaccions que sorgien durant les diferents proves, sempre intentant mantenir una objectivitat i independència respecte als casos concrets, donat que cada aplicació paral·lela pot tenir un comportament diferent i s'han de procurar tractar els casos de forma generalitzada.

Es podria concloure l'experimentació afirmant que en cas de tenir una alta freqüència de comunicació i/o demanda de recursos locals el planificador implementat en aquest treball millora sensiblement el rendiment de l'aplicació distribuïda, alhora que afegeix un retard igualment significatiu en les tasques locals. Aquesta millora de rendiment es en molts casos superior a l'aportada per l'implementació en anteriors projectes en el Kernel, el que ens porta a afirmar un gran èxit en aquestes situacions.

Per contra en cas de no tenir una demanda de recursos locals o aplicacions distribuïdes, o inclús si existeixen ambdós però amb una baixa intensitat, l'aplicació implementada aporta més retard que el benefici que pot aportar a l'aplicació distribuïda, essent inclús més ineficient que la tècnica de planificació original que integra el Kernel per defecte. Aquesta situació és deguda al consum de CPU que realitza l'aplicació implementada quan es troba en execució i, com ja hem dit, el seu major desavantja respecte a l'implementació en Kernel.

Així doncs, abans de incorporar una d'aquestes implementacions de la tècnica "Predictive Coscheduling" convindria realitzar un petit estudi sobre el tipus de clúster sobre el qual es treballarà, quina importància se li atorga a l'usuari local i quines són les situacions comunes en que es troba el clúster en general. Estudi per altra part relativament simple de fer tenint a disposició les proves i experimentacions tant d'aquesta implementació com de l'implementada en el Kernel en projectes anteriors [3].

Per últim obrir la possibilitat de realitzar un treball futur on implementar aquesta aplicació en un entorn encara més portable. On no només sigui compatible amb les diferents distribucions i kernels del s.o. Linux, sinó ser capaç d'implementar l'aplicació portable a Windows, òbviament canviarien llibreries, però el major problema crec que seria la falta d'informació que aquest s.o. proporciona a l'usuari per a poder actuar sobre el sistema. Altrament es podria realitzar en el futur l'optimització encara major del consum de CPU, reduint el retard i millorant el rendiment d'aquesta implementació.

A Implementació programa principal

Algorithm 16 Programa Principal

```
#define MAX_PRI -15
#define MIN_PRI 0
#define P 0.5
fil intercanvi;
Llista<char*> pids;
Llista<char*> socks;
main(int argc, char* argv[]){
  INICIALITZACIONS
  Bpids(argv[1]);
  for(itPid.situarInici();itPid.fi()==false;itPid++){
    Bsockets(*itPid,argv[1]);
  }
  itSock.situarInici();
  while(itSock.fi()==false){
    if(istcp(*itSock)==false){
      socks.eliminar(itSock);
    }
    else{
      itSock++;
    }
  }
  pthread_attr_init(&attr_param);
  schd_param.sched_priority=19;
  pthread_attr_setschedparam(&attr_param,&schd_param);
  error = pthread_create (&idFil,&attr_param, funcioFil, NULL);
  if (error != 0){
    perror("No s'ha pogut crear el thread");
    exit(-1);
  }
  ASSIGNACIO_PRIORITATS (algoritme 17)
 }
}
```

Algorithm 17 Assignació Prioritats

```
1   INICIALITZACIONS
2   while(intercanvi.getstat()<3){
3     curr_freq=intercanvi.calcular();
4     freq=(P*freq)+((1-P)*curr_freq);
5     if(curr_freq < 0.000001)curr_freq=0.000002;
6     rel=freq/curr_freq;
7     if(rel<0.0000001)rel=1.0;
8       if(rel > 1.25 && curr_freq > 0){
9       priority=priority+1;
10     }
11     else if (rel < 0.80 && curr_freq > 0){
12       if (rel < 0.5) priority=priority-2;
13       if (rel < 0.25) priority=priority-2;
14       priority=priority-2;
15     }
16      else priority=priority;
17     if(priority < MAX_PRI && priority > MIN_PRI);
18     else  if(priority <= MAX_PRI)priority=MAX_PRI;
19      else if(priority >= MIN_PRI)priority=MIN_PRI;
20     itPid.situarInici();
21     while(itPid.fi()==false && priority_ant!=priority){
22       curr_pid=strtoul(*itPid,NULL,10);
23       if(setpriority(PRIO_PROCESS,curr_pid,priority)!=0){
24         i=88;
25         rest=true;
26         break;
27       }
28       itPid++;
29     }
30     priority_ant=priority;
31     itPid.situarInici();
32     i++;
/****Re-lectura dels sockets de la aplicació****/
33     sleep(2);
34   }
```

Algorithm 18 Implementació Re-lectura

```
/****Re-lectura dels sockets de la aplicació****/
1   if(i%90==0){
2      intercanvi.setstat(2);
3      rest=Bpids(argv[1]);
4      if(rest==true){
5        freq=1;
6        i=85;
7        priority=0;
8        priority_ant=0;
9      }
10      for(itPid.situarInici();itPid.fi()==false;itPid++){
11        Bsockets(*itPid,argv[1]);
12      }
13      itSock.situarInici();
14      while(itSock.fi()==false){
15        if(istcp(*itSock)==false){
16          socks.eliminar(itSock);
17        }
18        else{
19          itSock++;
20        }
21      }
22      intercanvi.setstat(0);
23      pthread_attr_init(&attr_param);
24      schd_param.sched_priority=19;
25      pthread_attr_setschedparam(&attr_param,&schd_param);
26      pthread_setschedprio (idFil, 19);
27      error = pthread_create (&idFil,&attr_param, funcioFil, NULL);
28      if (error != 0){
29        perror("No s'ha pogut crear el thread");
30        exit(-1);
31      }
32    }
```

B Funcions implementades

Algorithm 19 Funció funcioFil()

```
1  void *funcioFil(void *parametre){
2    IteradorLlista<char*> itSock(socks);
3    INICIALITZACIONS
4    fin=fopen("/proc/net/tcp","r");
5    while(intercanvi.getstat()<2){
6      err=fscanf(fin,"%s %s %s %s %s %s %s %s %s %s %s %s”,aux,aux,aux,aux,
, aux,aux,aux,aux,aux,aux,aux,aux);
7      while(!feof(fin) && err!=EOF){
8        err=fscanf(fin,"%s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s",aux,
, aux,aux,aux,traf,aux,aux,aux,aux,sid,aux,aux,aux,aux,aux,aux,aux);
9        if(strcmp(aux,"-1")!=0)fseek(fin,-5*sizeof(aux),SEEK_CUR);
10       if(err!=EOF){
11         c_env=strtok(traf,":");
12         if(c_env==NULL)enviat=0X0;
13          else enviat=strtoul(c_env,NULL,16);
14         c_reb=strtok(NULL," ");
15         if(c_reb==NULL)rebut=0X0;
16          else rebut=strtoul(c_reb,NULL,16);
17         for(itSock.situarInici();itSock.fi()==false;itSock++){
18           if(strcmp(*itSock,sid)==0){
19             intercanvi.add(enviat,rebut);
20           }
21         }
22       }
23     }
24     usleep(50000);
25     rewind(fin);
26     }
27   pclose(fin);
28 }
```

Algorithm 20 Funció Bpids()

```
1  bool Bpids(char* aplic){
2    INICIALITZACIÓ
3    IteradorLlista<char*> itPid(pids);
4    IteradorLlista<char*> itSock(socks);
5    itPid.situarInici();
6    while(itPid.fi()==false){
7      pids.eliminarFi();
8    }
9   itSock.situarInici();
10   while(itSock.fi()==false){
11     socks.eliminarFi();
12   }
13   strcpy(commanda,"ps -ae|grep ");
14   strcat(commanda,aplic);
15   while(err==0){
16     fps=popen(commanda,"r");
17     fscanf(fps,"%s %s %s %s",pid,tty,data,apli);
18     err=atoi(pid);
19     while(!feof(fps) && err!=0){
20       pidaux=new char[sizeof(pid)];
21       strcpy(pidaux,pid);
22       pids.inserirFi(pidaux);
23       fscanf(fps,"%s %s %s %s",pid,tty,data,apli);
24       i++;
25     }
26     pclose(fps);
27     if(err==0){
28       reset=true;
29       sleep(10);
30     }
31   }
32   return reset;
33 }
```

Algorithm 21 Funció Bsockets()

```
1  void Bsockets(char* pid){
2    INICIALITZACIONS
3    strcpy(commanda,"ls /proc/");
4    strcat(commanda,pid);
5    strcat(commanda,"/fd -l");
6    ls=popen(commanda,"r");
7    while(!feof(ls)){
8    fscanf(ls,"%s",aux);
9      if(strncmp(aux,"socket",6)==0){
10       for(i=0,o=8;aux[o];i++,o++){
11         aux[i]=aux[o];
12       }
13       aux[i-1]='\0';
14       sock=new char[sizeof(aux)];
15       strcpy(sock,aux);
16       socks.inserirFi(sock);
17     }
18   }
19   pclose(ls);
20 }
```

Algorithm 22 Funció istcp()

```
1  bool istcp(char *socket){
2    bool res=false;
3    char temp[150];
4    FILE *fin;
5    fin=fopen("/proc/net/tcp","r");
6    while(!feof(fin)){
7      fscanf(fin,"%s",temp);
8      if(strcmp(socket,temp)==0){
9        res=true;
10       break;
11     }
12   }
13   fclose(fin);
14   return res;
15 }
```

Algorithm 23 Classe Intercanvi

```
1  class fil{
2    unsigned long resACUM;
3    int stat;
4    int i;
5     public:
6    fil(){
7     resACUM=0X0;
8     stat=0;
9     i=1;
10   }
11   void add(unsigned long resE,unsigned long  resR){
12     resACUM=resACUM+resE+resR;
13     i++;
14   }
15   int getstat(){
16     return stat;
17   }
18   void setstat(int estat){
19     stat=estat;
20   }
21   unsigned long calcular(){
22     unsigned long resultat;
23     resultat=resACUM/i;
24     i=1;
25     resACUM=0;
26     return resultat;
27   }
28 };
```

C Aplicacions externes

Els algoritmes exposats en aquesta secció son merament informatius, han estat reduïts per tal de mostrar l'essencia de l'aplicació i poder observar el seu comportament.

C.1 Aplicació Distribuïda *sintree_hc_hl.c*

Algorithm 24 Inici de sintree_hc_hl.c

```
#define N1 620 /*Number of doubles to send. N1*8(BYTES) */
#define COML 300/*Low Comunication*/
#define COMH 10000 /*HIGH Comunication*/
#define LOOPS 1/*Execution loops. 1 loop equal 5sec.*/
#define INITIAL_TASK 0
int main (int argc, char **argv){
  if(strcmp(argv[1],"--help")==0){...}
  TIPUS=atoi(argv[2]);
  pf=fopen(argv[1],"w");
  /* OPF: number of FLOP operations OPFxOPF*/
  OPF=25000;
  /*Tipus d'aplicaciÃ³
  1:    Comput baix - ComunicaciÃ³ baixa
  2:    Comput baix - comunicaciÃ³ ALTA
  3:    Comput ALT  - ComunicaciÃ³ ALTA
  4:    Comput ALT - ComunicaciÃ³ baixa*/
  /*Tamany de la variable que envia cada tasca definit per la constant N1*/
  ENV=(double*)malloc(N1*sizeof(double));
  for(i=0;i<N1;i++) ENV[i]=0.1E-7;
    //InicialitzaciÃ³ de MPI
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&mytid);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    TASQUES=nproc;
    if((TIPUS==1 || TIPUS==4)) COM=COML;
    if((TIPUS==2 || TIPUS==3)) COM=COMH;
    if ( mytid==0 ){
      tinicial=MPI_Wtime();
      for (l=0;l<LOOPS;l++){
      if (ARQ==0){
        for(k=1;k<TASQUES;k++){
          til=MPI_Wtime();
          id_task=k;
          for(i=0;i<COM;i++){
           id_message=0;
           MPI_Isend(ENV,N1,MPI_DOUBLE,id_task,id_message, MPI_COMM_WORLD,&request);
          }
          tfl=MPI_Wtime();
          total_send=total_send+(tfl-til);
          bw=((N1*8*COM)/(tfl-til))/1E+6;
        }
      }// COMPUTING (Algorisme 25)
```

Algorithm 25 Continuació de sintree_hc_hl.c

```
        // COMPUTING
        /*INICI COMPUT*/
         tinici_cpu=MPI_Wtime();
         //Comput Intensiu
         if(TIPUS==3 || TIPUS==4){
           for(i=0;i<OPF;i++){
             for(k=0;k<OPF;k++){
               a3=(a1*a2)/a1;
               a3=a3+sqrt(a3);
             }
           }
         }
         //Get finish Task Time
         tfinal_cpu=MPI_Wtime();
         //Computational Time of Task mytid
         totaltime_cpu=totaltime_cpu+(tfinal_cpu-tinici_cpu);
         // The reception depends on the ARQ
         // IF ARQ=0 The Master Receives all the comunications
         // else only receive data of the TASQUES-1
         if (ARQ==0){
           total_recv=0;
           for(k=1;k<TASQUES;k++){
             id_source=k;
             id_message=k;
             til=MPI_Wtime();
             printf("Start reception from task %d -",id_source);
             for(i=0;i<COM;i++){
               MPI_Recv(ENV,N1,MPI_DOUBLE,id_source,id_message,MPI_COMM_WORLD,
   ,&status);
             }
             tfl=MPI_Wtime();
             total_recv=total_recv+(tfl-til);
             bw=((N1*8*COM)/(tfl-til))/1E+6;
             printf("End Reception to task %d, %fs %fMB/s\n",k,(tfl-til),bw);
           }
         printf("\nTotal Sending %f\nTotal Receive %f\nTotal LAN %f\n",total_send,
   ,total_recv,total_send+total_recv);
       }
       /*FINAL COMUNICACIO*/
     } //FINAL LOOPS
     tfinal=MPI_Wtime();
     totaltime=(tfinal-tinicial);                  //Execution Time of Task mytid
     totaltime_lan=totaltime-totaltime_cpu;  //Comunication Time*/
     //CODI_FILL (Algoritme 27)
```

Algorithm 26 Codi Fill sintree_hc_hl.c

```
/****************INICI CODI FILL***************/
  else{
    //Capturem l'instant de temps inicial
    tinicial=MPI_Wtime();
    for(l=0;l<LOOPS;l++){
      /*INICI COMUNICACIO*/
      /* Receiving data from task mytid-1 */
      if (ARQ==0){
        id_source=0;
        id_message=0;
        total_recv=0;
        til=MPI_Wtime();
        for(k=0;k<COM;k++){
          MPI_Recv(ENV,N1,MPI_DOUBLE,id_source,id_message,MPI_COMM_WORLD,&status);
        }
        tfl=MPI_Wtime();
        total_recv=total_recv+(tfl-til);
        bw=((N1*8*COM)/(tfl-til))/1E+6;
        }
        // COMPUTING
        /*INICI COMPUT*/
        tinici_cpu=MPI_Wtime();
        if(TIPUS==3 || TIPUS==4){        //Comput Intensiu
          for(i=0;i<OPF;i++){
            for(k=0;k<OPF;k++){
              a3=(a1*a2)/a1;
              a3=a3+sqrt(a3);
            }
          }
        }
        else{                            //Comput normal
          for(i=0;i<OPF;i++){
            for(k=0;k<OPF;k++){
              a3=(a1+a2);
              a3=a3+(a3-a1);
            }
          }
        }
        /*FINAL COMPUT*/
        /*FINAL COMUNICACIO*/
        tfinal=MPI_Wtime();
        totaltime=(tfinal-tinicial); //Execution Time of Task mytid
        totaltime_lan=totaltime-totaltime_cpu;  //Comunication Time
        id_task=INITIAL_TASK;
        id_message=mytid;
        MPI_Isend(&totaltime, 1, MPI_DOUBLE, id_task, id_message,
,MPI_COMM_WORLD, &request);
        MPI_Isend(&totaltime_cpu, 1, MPI_DOUBLE, id_task, id_message,
,MPI_COMM_WORLD, &request);
        PI_Isend(&total_send, 1, MPI_DOUBLE, id_task, id_message,
,MPI_COMM_WORLD, &request);
      }
      /****************FINAL CODI FILL***************/
      MPI_Finalize();
}
```

C.2 Aplicació Local *carga.c*

Algorithm 27 carga.c

```
main(int argc,char *argv[]){
 if (argc!=4)help(argv[0]);
 segundos=atoi(argv[1]);
 carga_deseada=atof(argv[2]);
 procesos=atoi(argv[2]);
 porcentaje=atoi(argv[3]);
 memoria=info_memoria();
 bogomips=info();
 MAX=bogomips*bogomips;
 memoria=(memoria)-(3*(MAX/1024));
 memoria_cargada=((memoria*porcentaje/100)*1024);
 memoria_cargada=memoria_cargada/(procesos+1);
 Es_primo=calloc(MAX,sizeof(int));
 carga=calloc(memoria_cargada,sizeof(char));
 if (carga==NULL){
   printf("\nMemoria insuficiente\n");
   exit(0);
 }
 for(x=0;x<memoria_cargada;x++)carga[x]=0;
  for(i=0;i<=procesos;i++){
    if ((id[i]=fork()) != 0) {
      carga_cpu(MAX,carga_deseada,segundos);
    }
  }
  free(Es_primo);
  free(carga);
}
```

Algorithm 28 Funció carga_cpu()

```
void carga_cpu(unsigned long int MAX,float carga_deseada,int segundos){
  int numero,auxiliar,iteracion;
  struct timeval t_inicio,t_control,t_final;
  float carga_actual=1;
  (void) gettimeofday(&t_inicio,(struct timezone *) 0);
  do{
    for(iteracion=0;iteracion<veces;iteracion++){
      for(numero=1;numero<=MAX;numero++)Es_primo[numero]=1;
      numero=2;
      while((numero*numero)<MAX){
        if (Es_primo[numero]){
          auxiliar=numero+numero;
          while(auxiliar<MAX){
            Es_primo[auxiliar]=0;
            auxiliar+=numero;
          }
        }
        numero++;
        carga_actual=load_avg();
        while (carga_actual>carga_deseada){
          carga_actual=load_avg();
          dream(100*(100/carga_deseada));
        }
      }
    }
  (void) gettimeofday(&t_control,(struct timezone *) 0);
  tvsub(&t_final,&t_control,&t_inicio);
  }while (t_final.tv_sec<segundos);
}
```

Algorithm 29 Funció dream()

```
void dream(unsigned long usec){
  struct timeval timeout;
  if(usec < 1000000L){
    timeout.tv_sec  = 0L;
    timeout.tv_usec = usec;
  }
  else{
    timeout.tv_sec  = usec / 1000000L;
    timeout.tv_usec = usec - (1000000L * timeout.tv_sec);
  }
  select(1, NULL, NULL, NULL, &timeout);
}
```

C.3 Aplicació Local *calcul.cc*

Algorithm 30 Calcul.cc

```
int main(){
  double a=1,b=0,c=2;
  float k=0.1,v=9.4;
  int i,o;
  for(i=0;i<99876543;i++){
    a=b*i+(a*c);
    k=((a/i*v)*v)*k;
    b=a*a*b*c/k*a*v-c;
    c=(((i*i*v)/k+a)*b)*c;
    for(o=0;o<a*b*c;o++){
      a=b*o+(a*c);
      k=((a/o*v)*v)*k;
      b=a*a*b*c/k*a*v-c;
      c=(((o*o*v)/k+a)*b)*c;
    }
  }
}
```

D Bibliografia

Referències

[1] Solsona,F., Ginè, F., Hernández, P., Luque, E. CMC: A Coscheduling Model for Non-Dedicated Cluster Computing.Universitat de Lleida. Universitat Autònoma de Barcelona. 2000.

[2] Solsona, F. Coscheduling Techniques for Non-Dedicated Cluster Computing. Universitat Autonoma de Barcelona. Març 2002.

[3] Pallarès, J., "Disseny, implementació i anàlisis d'un algorisme predictiu de Coscheduling en un clúster Linux". TFC, EPS, Universitat de Lleida. Setembre 2002

[4] Solsona, F., Ginè, F., Hernández, P., Luque, E. "Predictive Coscheduling Implementation in a non-dedicated Linux Cluster" 7^{th}International EuroPar Conference, (Europar 2001), August 2001.

[5] Linux Homepage. http://www.Linux.org.

[6] The Linux Documentation Project. http://tldp.org.

[7] The C++ Resource Project. http://cplusplus.com.

[8] Solbavarro, P.G. and Wheil, W.E.: Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessor. IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing. 1995.

[9] Corvella, M. et al.: Multiprograming on Multiprocessors. 3rd IEEE Symposium on Parallel and Distributed Processing. 1994.

[10] Petrini, F. and Feng, W.: Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems. International Parallel and Distributed Processing Symposium. 2000.

[11] Official Parallel Virtual Machine site. http://www.epm.ornl.gov/pvm/

[12] The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/

[13] Tanenbaum, A.S.: Structured Computer Organitzation. Prentice Hall.1999

[14] M.J.Flynn: Some Computer Organization and their effectiveness. *In IEEE Transactions on Computers*, volum C-21. 1972.

[15] Sendrós, D.: Implementació de Dynàmic Coscheduling en un entorn PVM-Linux. TFC. Universitat de Lleida. 2000.

[16] Information Science Institute: Transmission Control Protocol, Darpa Internet Program, Protocol Specification: University of Southern California. Setembre 1981.

[17] J.Postel: User Datagram Protocol: ISI. Agost 1980

[18] Advanced Bash Scripting Guide: http://tldp.org/abs/html/index.html

[19] OpenSSH: http://openssh.com

[20] Daniel P.Bovet & Marco Cesati: Understanding the Linux Kernel: October 2000

[21] Sieve of Eratosthenes: http://en.wikipedia.org/Sieve_of_Eratosthenes

[22] LAM/MPI Parallel Computing: http://www.lam-mpi.org