Universitat de Lleida
Escola Politècnica Superior
Enginyeria en Informàtica.

Integració de la memòria virtual i altres funcionalitats en el simulador ESPPADA.

Treball Final de Carrera

Per
Damià Castellà Martínez

Director:
Francesc Giné
Fernando Guirado

Septembre de 2006

Índex

Índex de figures

Índex de taules

1 Introducció

La resolució de problemes des de el paradigma de la programació paral·lela no és trivial perquè existeixen múltiples factors que influeixen alhora d'afrontar el problema, desde la detecció del paral·lelisme existent al problema inicial, fins l'elecció de la millor arquitectura possible. Aquests factors es presenten com variables que poden tenir múltiples valors i que fan intractable l'intentar determinar la solució òptima d'entre tots els casos possibles. Sols mitjançant l'assignació d'alguns d'aquests paràmetres, com pot ser el número d'unitats de processament a l'arquitectura, es pot arribar a aconseguir un resultat en un temps de desenvolupament vàlid, i que no té perqué ser el millor.

Per aquest motiu s'utilitzen eines de simulació, aplicades com un pas previ a la implementació final del programa i que permeten predir resultats en un temps acceptables. Aquestes eines requereixen que els programadors siguin capaços de representar els algorismes que desitgen implementar i caracteritzar l'arquitectura. La gran avantatge de la utilització d'aquest tipus d'eines és la possibilitat d'escalar el problema tant a l'àmbit de la programació com el de l'arquitectura sense interferir directament a la implementació real.

És esencial en el desenvolupament de programes paral·lels predir el seu rendiment sobretot si s'executen sobre arquitectures amb la capacitat de ser escalades. La predicció analítica pot ser intractable en la majoria dels casos en que els programes són de gran complexitat i per aquest motiu s'utilitza la simulació que pot resoldre aquest problema.

El simulador ESPPADA (*Entorn de Simulació de Programes Paral·lels sobre Arquitectures Distribuïdes)* és una eina de l'autor Fernando Guirado Fernández[1] que simula el comportament d'un sistema amb arquitectura distribuida executant un o varis models de programes paral·lels. Aquesta eina permet realitzar l'avaluació de prestacions del sistema mitjançant un conjunt de resultats oferits per la màquina de simulació. Després mitjançant la interacció de l'usuari és possible ajustar, modificant paràmetres com el *mapping* realitzat, la quantitat i característiques propies de cada node de processament o la xarxa d'interconnexió. D'aquesta manera el programador es capaç d'extrapolar d'una manera simple els resultats obtinguts a la realitat sense tenir que diposar de la màquina paral·lela.

1.1 Objectius

Els objectius del treball es poden resumir en els següents punts:

- Conèixer les caracteristiques principals del simulador ESPPADA, el seu disseny, la seva implementació, els models d'entrada i els resultats de simulació oferits.

- Conèixer a nivell teoric la principal funcionalitat que s'ha afegit al simulador que és la integració de la memòria virtual i tot el que comporta com la gestió de les faltes de pàgina, l'algorisme de reemplaçament, el temps de sobrecarrega de les faltes, etc. i a nivell pràctic les especificacions d'entrada dels paràmetres de la memòria virtual en el simulador, la implementació i l'experimentació realitzada.
- Conèixer que és la multiprogramació a nivell de tasques locals, la seva definició, el seu propòsit al simulador, la seva implementació i l'experimentació realitzada.
- Conèixer que és la multiprogramació a nivell de tasques paral·leles, la seva definició, la seva finalitat, la seva implementació i l'experimentació realitzada.
- Analitzar i definir un nou algorisme de mapping que s'anomena algorisme de balanceig de memòria que redueix els intercanvis de memòria i la sobrecàrrega de faltes de pàgines. Comentar la seva implementació i els resultats obtinguts en l'experimentació.
- Definir un nou algorisme de mapping basant en el balanceig de CPU que té en compte les caràcteristiques de la CPU alhora d'assignar tasques. Descriure la seva implementació i analitzar els resultats obtinguts en l'experimentació.
- Proposar les conclusions obtingudes durant la fase d'elaboració d'aquesta memòria i exposar projectes de futur per la seva continuació.

1.2 Organització i planificació

Aquest document es divideix en les següents parts:

- **El simulador ESPPADA**: En aquest capítol introductori s'explica en la primera secció el conceptes bàsics de la simulació, fent incidència en la seva utilitat per l'avaluació de rendiment a partir de l'anàlisi dels resultats que ofereix. Després a la següent secció s'explica la part més important que és el simulador ESPPADA i el seu funcionament. Per explicar el seu funcionament es detalla en una primera subsecció els models d'entrada com per exemple el model de programa, el model d'arquitectura i representació del graf de tasques i el mapping associat. A continuació s'explica els resultats obtinguts com el temps d'execució, el speed up, l'utilització dels nodes de processament, la quota pel número de nodes de processament, paràmetres del model TTIG i assignació de tasques a nodes de processament. En la següent i última secció s'explica resumidament la implementació i disseny del simulador ESPPADA, com per exemple l'analisi i disseny del simulador, l'esquema general de la màquina de simulació, el disseny de les unitats funcionals i l'interacció d'aquestes unitats.
- **Integració de la memòria virtual en el simulador ESPPADA**: En aquest capítol s'explica detalladament com s'ha integrat la memòria virtual en el simulador ESPPADA a través de la programació. Inicialment en la primera secció s'explica

una introducció de qué és la memòria virtual i perquè s'ha implementat, després en la segona secció es comenta l'algorisme general de simulació de la memòria virtual utilitzat en el simulador ESPPADA. A continuació, en la tercera secció es comenta l'especificació dels paràmetres de la memòria a l'arquitectura utilitzant llenguatge XML, l'especificació dels paràmetres de memòria virtual al model de programa i el disseny de l'interfície del programa. Finalment en les dues últimes seccions s'explica la implementació detallada de la memòria virtual i l'experimentació realitzada amb la simulació d'un programa paral·lel que consumeix molta memòria.

- **Extensió del simulador cap a un entorn multiprogramat**: En aquest capítol es comenta dues funcionalitats afegides al simulador que és l'aparició de tasques locals per simular el comportament d'un entorn cluster no dedicat i la possibilitat d'afegir més d'un programa paral·lel a la simulació. En la primera secció s'explica una breu introducció sobre aquestes dues funcionalitats afegides. En la segona secció es detalla la multiprogramació a nivell de tasques locals i en la tercera secció la multiprogramació a nivell de tasques paral·leles. En la següent secció es comenta detalladament la implementació d'aquestes dues funcionalitats i finalment en la última secció es realitza l'experimentació que consta de dos apartats, el primer s'explica els resultats obtinguts de la simulació amb tasques locals i el segon els resultats obtinguts de la simulació amb varis programes paral·lels.

- **Algorismes de mapping adaptat a l'entorn multiprogramat**: En aquest capítol s'expliquen dos nous algorismes de mapping afegits al simulador que són l'algorisme de balanceig de memòria i l'algorisme de balanceig de CPU per millorar el rendiment dels programes paral·lels. En la primera secció s'explica una introducció sobre aquests dos algorismes. En la segona secció s'explica l'algorisme de balanceig de memòria de forma generalitzada i en la tercera secció s'explica l'algorisme de balanceig de CPU. La quarta i última secció consta de dos apartats, el primer explica la experimentació realitzada amb l'algorisme de balanceig de memòria fent una comparativa amb altres mappings i el segon explica els resultats obtinguts amb l'algorisme de balanceig de CPU fent també una comparativa amb altres mappings.

- **Conclusions i treball futur**: En aquest últim capítol es comenten les conclusions finals del document i es proposa diferents treballs de futur per la seva continuació.

2 El simulador ESPPADA

L'objectiu d'aquest capítol és entendre quin és el propòsit d'un simulador d'entorn paral·lel, els conceptes més rellevants de la programació paral·lela i descriure breument la implementació de l'eina ESPPADA.

2.1 Introducció

La resolució de problemes desde el paradigma de la programació paral·lela no es trivial. Existeixen múltiples factors que influeixen alhora d'afrontar el problema, desde la detecció del paral·lelisme existent al problema inicial per determinar l' algorisme adequat, fins l'elecció de la millor arquitectura possible. Aquests factors podent tenir infinits valors que fan que sigui intractable trobar la solució òptima.

Es per aquest motiu que en molts casos s'utilitzen eines de simulació que poden preveure resultats en temps acceptables. Per aconseguir-ho es necessari que els programadors siguin capaços de poder representar els algorismes que desitgin implementar o inclòs caracteritzar l'arquitectura.

Preveure el rendiment dels programes paral·lels és un pas essencial en el desenvolupament de programes que s'executen sobre arquitectures amb capacitat de ser escalades.

Les simulacions d'aquest tipus de programes poden fer-se molt lentes quan el sistema a simular és gran, per tant es planteja dos tècniques per reduir els temps d'execució: L' execució directa del codi que apareix com seqüencial a l'aplicació[4] [5] i la simulació utilitzant algorismes de simulació paral·lela. Una altra possibilitat passa per la definició de models que representin el comportament de l'aplicació i utilitzar-los com entrada per la simulació reduint el problema inicial a un problema molt més simple que permeti afrontar la simulació d'una manera més eficaç.

El simulador ESPPADA té la capacitat d'avaluar, mitjançant la simulació, la execució de programes paral·lels que utilitzen el paradigma de programació per pas de missatges. També ofereix la possibilitat de '*jugar*' amb les característiques de l'arquitectura, com pot ser l'escalabilitat o la capacitat de processament, per representar arquitectures que no es troben presents físicament en un principi. Un altre factor important es que permet disminuir el cost de desenvolupament de qualsevol programa paral·lel. A més permet avaluar les diferents fases existents dins de la programació paral·lela, com pot ser l'avaluació de les polítiques de mapping i inclòs la valoració de l'arquitectura existent.

El simulador ESPADDA incorpora una màquina de simulació, que permet extreure la informació rellevant al comportament de les tasques, així com el rendiment obtingut en els nodes de processament, de manera que el programador pugui tenir criteris de valoració.

2.2 Entorn de simulació: Models d'entrada i resultats

El simulador ESPPADA obté un entorn que es capaç d'avaluar, mitjançant la simulació, la execució de programes paral·lels creats amb el model de la programació per pas de missatges. El *programa* es divideix en un conjunt d'activitats anomenades *tasques*, les quals representen una part del problema inicial. Pot ser el programador qui les defineixi mitjançant l' anàlisi previ del problema o mitjançant mecanismes assistits.

L'*arquitectura* és un factor que intervé de forma decissiva, com la determinació del número de *nodes de processament* o inclòs la capacitat de comunicació entre aquests pot forçar a l' elecció del tipus d'algorisme. El número de nodes poden determinar la granularitat a la definició de les tasques, es a dir aquestes poden ser ajustades o no a l'arquitectura existent, llavors el programa és depenent a l'arquitectura. En canvi en un programa independent de l'arquitectura el número de tasques no coincideix amb el número de nodes i en conseqüència es necessari una assignació de tasques a nodes de processament anomenat *mapping*. L'objectiu del *mapping* és aconseguir solucionar el problema mínimitzant el temps d'execució.

El *mapping* es soluciona basant-se amb l' elecció d'un model de representació del programa mitjançant grafs de tasques, que després són tractats mitjançant diferents tècniques. Podem trobar tres maneres de modelar el programa; el TPG (*Temporal Task Graph)*[6][7], el TIG (*Task Interaction Graph*)[8] i el TTIG (*Temporal Task Interaction Graph*)[9] [10].

Els tres components que son capaços de representar tota la informació referent al sistema a simular són: el programa, l'arquitectura i el model de graf de tasques amb el mapping associat. El model de programa ha de ser suficient precís per representar el comportament del disseny original i suficientment abstracte perquè es pugui utilitzar en qualsevol tipus de llenguatge de programació. L'arquitectura ha de poder ser representada amb la garantia de que els resultats obtinguts siguin representatius en un entorn real.

Finalment s'ha de poder avaluar els resultats obtinguts, això serà possible si l' entorn genera les dades relatives a la simulació i la informació rellevant al programa i al seu mapping.

A la figura 2.1 pot observar-se l' esquema sobre els components que formen l'entorn i la seva interacció. Les entrades del simulador són la definició del model del programa i la definició del model d'arquitectura. Pel model del programa existeixen tres possibilitats; definició explícita de l'usuari; obtenció a partir d'un estudi del codi font mitjançant tècniques de compilació i parsing; i finalment a partir de traces d'execució. A la definició de l'arquitectura es representen els nodes de processament amb les seves característiques particulars i la xarxa d'interconnexió associada.

El programa sintètic es representa mitjançant el model TTIG, que serà utilitzat per la obtenció del mapping automàtic i per l' obtenció d'una representació prèvia al TTIG denominada TFG (*Temporal Flow Graph*). Per el procés de simulació s'agafen com entrades, l'arquitectura, el programa i el mapping, obtenint resultats que permeten preveure el rendiment del programa.

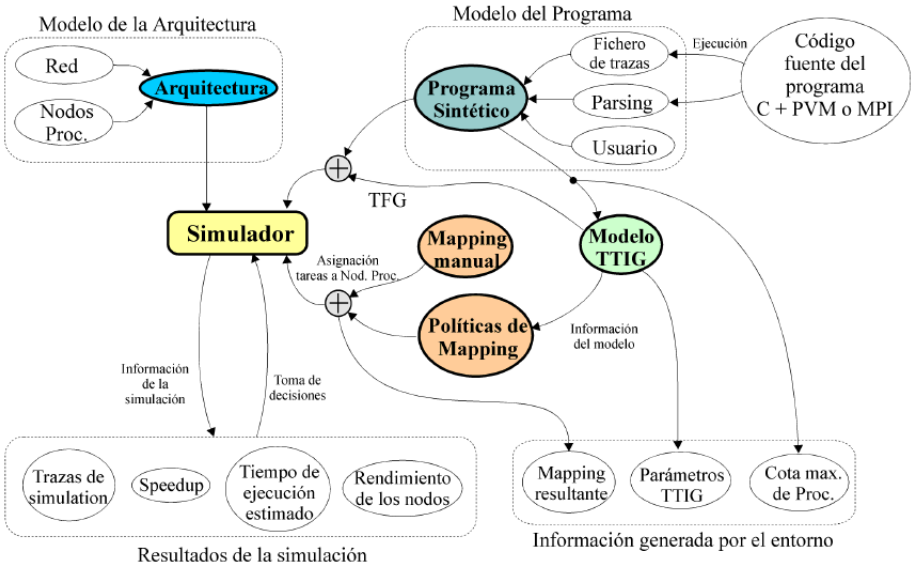

Figura 2.1: Esquema de l'entorn de simulació ESPPADA.

2.2.1 Entrades al simulador

Les entrades al simulador són aquelles que conformen totes les fases del disseny d'un programa paral·lel. En aquest apartat s'explicara les entrades dels paràmetres del simulador ESPPADA. A la subsecció 2.2.1.1 es parla sobre el model de programa sintètic com a entrada al simulador. A la subsecció 2.2.1.2 es comenta el paràmetre d'entrada l'arquitectura del sistema. I finalment a la subsecció 2.2.1.3 es parla sobre l'entrada dels diferents grafs de tasques i sobre el mapping associat.

2.2.1.1 Model de programa sintètic

Un programa paral·lel està compost per un conjunt de tasques que intercanvien missatges. Aquesta tècnica que forma part del paradigma de programació per pas de missatges es explotada per la programació paral·lela. Les tasques intercanvien informació, mitjançant missatges, que permeten realitzar la funció assignada, de manera que es van seqüenciant les fases d'execució i de comunicació. Quant la tasca té tota la informació necessària executa l'acció fins que necessita més dades provinents d'altres tasques. Existeixen primitives de comunicació *bloquejants* com les de recepció, que espera fins que hagi rebut alguna dada d'una altra tasca, o les *no bloquejants* com les d'enviament que permet continuar la seva execució. També existeixen primitives de sincronització, les qualsn estan en espera mútua fins hagin arribat totes les tasques al punt de sincronització.

Pot observar-se, que definides d'aquesta forma les tasques, el paral·lelisme s'explota en les fases d'execució quan les dades necessàries per aquestes són accessibles.

Les tasques dels programes són representades per un conjunt de primitives simples, com les de còmput, de control, de flux, les iteratives, les condicionals, de comunicació i

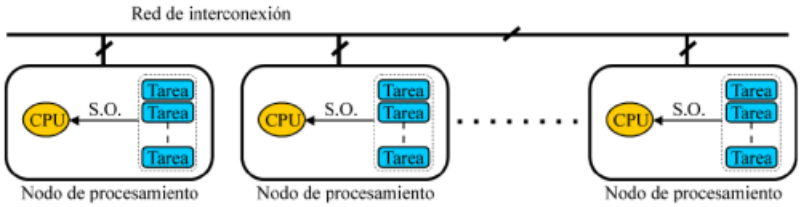

Figura 2.2: Esquema genèric per una arquitectura distribuïda per pas de missatges.

finalment de sincronització.

El simulador ESPPADA utilitza per representar les tasques el llenguatge XML[11]. [1]

2.2.1.2 Model d'arquitectura

L'arquitectura correspon a la part física del sistema informàtic que s'està modelant. Posseeix característiques pròpies que defineixen de forma directa quins resultats seran els obtinguts en l' execució del programa. Existeixen múltiples arquitectures paral·leles, com les de memòria compartida, o les distribuïdes, .

En un arquitectura distribuïda existeixen dos components diferenciats, els *nodes de processament*, i la *xarxa* d'*interconnexió*. El primer s'encarrega d'executar les tasques que defineixen el programa i el segon permet als *nodes* intercanviar informació entre ells, movent els missatges que generen les tasques.

En la figura 2.2 pot veure's una arquitectura distribuïda genèrica. En aquesta apareix tant els nodes de processament com la xarxa d'interconnexió. A cada node es mostra la seva CPU i les tasques planificades.

- **Nodes de processament**:
  Els nodes de processament s'encarreguen d'executar les tasques assignades mitjançant tècniques de mapping. A més són l'element principal que determina la heterogeneitat de l'arquitectura al poder tenir, per exemple, diferents capacitats de còmput a la CPU.
  Per determinar la capacitat de còmput dels nodes de processament s'ha definit en el simulador ESPPADA un valor anomenat CET (*Cycle Execution Time*)[1] que indica el cost per executar cada unitat de simulació. D'aquesta manera es pot diferenciar la capacitat de còmput de cada node. Fent servir la següent expressió s'assigna un valor a aquest paràmetre.

$$Cycle\_execution\_time = Base\_execution\_time \times Compensation\_Factor$$

  En aquesta expressió apareixen el valor *BET* (*Base Execution Time*)[1] i el valor

[1] Per saber com s'especifica una tasca i les seves primitives, llegir documentació del treball [1].

CF (*Compensation Factor*)[1]. El *BET* correspon a un valor de referència comú que estableix la base del temps en tots els nodes. Per altra banda, el valor *CF* determina la forma individual de cada node, com es comporta respecte el paràmetre *BET*.
Si volem per exemple, que un node sigui el doble de ràpid que un altre, llavors el valor *BET* ha de ser el mateix i el valor *CF* ha de ser 1 i 2 respectivament.
Les tasques s'assignen als nodes de processament mitjançant una estratègia de mapping existent, de manera que hi pot haver més d'una tasca assignada. Quan sols existeix una, no hi ha problema alhora de decidir quina tasca executar, però quan hi ha més d'una s'utilitzen tècniques existents d'apropiació de la CPU. Al simulador ESPPADA hi ha implementat la planificació *no apropiativa* i l'*apropiativa*.
Les polítiques no apropiatives són les que permeten que una tasca s'executi sense sense interrupció fins que de forma voluntària abandoni la CPU, sigui perquè ha finalitzat o perquè no pot continuar més la seva execució.
Les polítiques apropiatives permeten treure la tasca de la CPU encara que no hagin aconseguit el final de la seva execució per donar pas a una altra. Per aquest tipus de planificació existeixen algorismes, encara que el més utilitzat és el *Round-Robin*. Aquest està implementat en el simulador ESPPADA. El seu funcionament és el següent, cada tasca té assignat un temps d'execució anomenat *quantum*, que determina el màxim interval que pot estar dita tasca a la CPU, quan s'agota aquest temps l'abandona donant pas a una altra tasca i s'espera ena una cua circular, fins que li toqui el seu torn. Aquestes tècniques de planificació formen part del sistema operatiu.
Un altre aspecte a tenir en compte és que l'intercanvi de tasques mai és instantani ja que el sistema operatiu ha d'agafar el control i processar els canvis de context i com a conseqüencia provoca un retard en el temps final. Això ho té en compte el simulador ESPADDA que permet especificar el retard que es produeix en els canvis de context.

- **Xarxa d'interconnexió**:
  La xarxa d'interconnexió permet l' intercanvi de missatges originats per les tasques i que són transferits entre els nodes de processaments.
  El simulador ESPPADA implementa dos models de comportament. El primer d'ells està implementat de forma genèrica i permet adaptar-se a la gran majoria de xarxes existents. En aquesta implementació cal tenir en compte dos paràmetres, el *número de canals* i l'*ample de banda*.
  El *número de canals* es refereix a la quantitat de comunicacions simultàaies que es poden donar. Si el número de canals està limitat, els missatges tindran que esperar-se per accedir als canals i això provoca un retard en la comunicació. Pel contrari, si són suficientment elevats, aquest retard no es produïra permetent una transmissió més ràpida.
  L'*ample de banda* indica la velocitat de transferència en Bytesoctets per segon, que permeten els canals. El retard que es pot produir en la transferència és el següent:

$$Temps = \frac{M}{Ample\_de\_Banda}$$

En aquesta expressió M és la longitut del missatge en Bytes.
El simulador ESPPADA inclou apart d'aquest model, un altre model de xarxa molt més simple amb una conectivitat completa entre els nodes de processament, sense limitació en el número de canals de comunicació i en la que s'especifica un parametre que és el *factor de retard* respecte al volum de dades transmeses. La següent expressió indica el càlcul del temps necessari per la transmissió d'un missatge amb longitud M Bytes.

$$Temps = M \times Factor\_de\_retard$$

En aquest model l'anàlisis d' execució és molt més simple i transparent perquè el seu objectiu és aïllar el comportament de la xarxa, l'accés als canals de comunicació, en els resultats obtinguts per la màquina de simulació.
Un altre aspecte sobre el simulador que cal tenir en compte és que s'ha afegit als nodes de processament dos valors que afecten de forma individual a cada node. Un és el cost involucrat en les comunicacions que fan referència a tasques mapejades en el mateix node i un altre és el cost involucrat ena les comunicacions que afecten a nodes diferents.
En la figura 2.3 es mostra un exemple d'arquitectura amb format XML. Pot observarse que es defineix una arquitectura amb planificació CPU del tipus Round-Robin i quantum de temps 10 unitats. El canvi de context és de 0 unitats. La xarxa d'interconnexió està formada per 3 canals de comunicació amb un ample de banda de 10 MBytes/seg. Hi han dos nodes de processament, el primer node 2.2 vegades més ràpid que la CPU_Base i amb un overhead sobre la comunicació local de 1 unitat, externa de 2 unitats. En canvi el segon és igual de ràpid que la CPU_Base i sense sobrecàrrega a la comunicació.

```
<architecture name= "Ejemplo de
                     arquitectura">
   <cpu_base value="1"/>
   <planification method="round_robin"
                  quantum= "10"/>
   <context_change value= "0"/>
   <net type= "user_defined">
      <bandwith value= "10"
                dimension="Mbytes"/>
      <channels value= "3"/>
   </net>
 <processor id= "0">
    <cpu_period value= "2.2"/>
    <comm_overhead local= "1" out= "2"/>
 </processor>
 <processor id= "1">
    <cpu_period value= "1"/>
 </processor>
</architecture>
```

Figura 2.3: Exemple d'una arquitectura modelada mitjançant XML.

$$Temps = M \times Factor\_de\_retard$$

2.2.1.3 Representació del graf de tasques i el mapping associat

La solució d'un problema mitjançant un programa paral·lel requereix tres fases. Primer, les activitats que es puguin executar amb paral·lel han de ser detectades i també les seves interdependències. A partir d'aquí es realitza el graf de tasques, definint el conjunt de tasques i les accions que s'han d'executar en cadascuna d'elles, i finalment, s'ha de realitzar el mapping, assignant les tasques de l'aplicació als nodes de processament.

El graf de tasques i el mapping realitzant, han d'explotar el paral·lelisme potencial prèviament detectat, per aconseguir d'aquesta manera una implementació eficient sobre l'arquitectura.

El simulador ESPPADA té implementat tres models de representació de grafs: El TPG (*Task Precedence Graph*)[6][7], el TIG (*Task Interaction Graph*)[8] i el TTIG (*Temporal Task Interaction Graph*)[9][10].

El TPG és un graf dirigit en que els nodes i els arcs representen les tasques i les precedències entre aquestes respectivament. Les tasques sols poden interaccionar al principi i al final de la seva execució.

El TIG és un graf no dirigit en que dos tasques poden comunicar-se si existeix un arc d'unió entre els nodes que la representen. Aquest model permet una interacció arbitraria entre les tasques paral·leles, podent-se donar comunicació entre elles en qualsevol moment.

En aquests dos grafs hi han valors associats als nodes i als arcs d'unió, que representen el còmput associat i el volum de comunicació respectivament. El TPG és efectiu en

programes on les iteracions entre les tasques es donen al principi i al final de la seva execució. En canvi el TIG s'utilitza quan les comunicacions poden donar-se en qualsevol moment de l' execució de les tasques.

Finalment, el model TTIG, inclou un paràmetre més realista per representar la interacció existent entre les tasques dels programes paral·lels, és el *grau de paral·lelisme*. Aquest valor, dona informació sobre el màxim percentatge d'execució paral·lela que dos tasques que interaccionen entre si són capaces de realitzar. El grau de paral·lelisme està normalitzat en l'interval [0,1], on un valor 0 indica que no hi ha paral·lelisme, i un valor 1 indica que són totalment paral·leles. Aquest model integra els models TPG i TIG. En el TPG el grau de paral·lelisme és 0 i en el TIG el grau de paral·lelisme és 1. Qualsevol altre valor per aquest paràmetre ens indicaria situacions intermedies.

Existeixen una serie de passos per obtenir el graf TTIG. Si es vol obtenir més informació llegir documentació del treball [1].

2.2.2 Resultats obtinguts

Es important saber quins son els resultats obtinguts de la simulació per que tinguin una utilitat alhora de decidir accions posteriors per part de l'usuari. Els resultats que ofereix el simulador ESPPADA són: el conjunt de dades que denoten el comportament de la simulació del programa, el speedup obtingut, la cota màxima de nodes de processament necessaris per la execució del programa, el model TTIG d'aquest programa i el mapping obtingut.

- **Temps total d'execució**:
  El *temps total d'execució* calculat mitjançant la simulació ens informa d'una aproximació del temps que seria necessari per executar l'aplicació en un entorn paral·lel real.

- **Comportament temporal de les tasques**:
  Amb el *comportament temporal de les tasques* ens descriu com evolucionen aquestes dins dels nodes de processament. Ens poden ajudar a donar informació sobre els colls d'ampolla que han aparegut en la simulació i que es podrien donar en la execució real.

- **Utilització dels nodes de processament**:
  Aquesta mesura ens informa sobre el temps total d'execució de cada node, el benefici obtingut de l'augment o no del número d'aquests i el percentatge d'ocupació.

- **Speedup**:
  El *speedup* és un valor que informa sobre com és de bona la execució paral·lela respecte l'execució en un sol node de processament. La expressió que s'utilitza és la següent:

  $$Speedup = \frac{Temps\_execucio\_un\_node}{Temps\_execucio\_paral \cdot lel}$$

  Aquest paràmetre es troba en el rang [0, Num_Nodes_Processament] de manera

que un valor pròxim a zero, indica que el rendiment és pitjor que sobre la execució en un únic node de processament. Un valor a 1 indica que el temps es el mateix que amb un sòl node, i un valor major a la unitat indica millores de rendiment.

- **Quota pel número de nodes de processament**:
  Indica quina és la quantitat de nodes de processament necessaris per explotar el màxim el paral·lelisme i obtenir un millor resultat. Aquest valor es possible calcular-lo coneixent el comportament temporal del programa, el qual està present mitjançant el model introduït.

- **Paràmetres del model TTIG**:
  Aquest model obté informació sobre el *grau de paral·lelisme* de les tasques, de manera que pugui avaluar-lo l'usuari i també la granularitat aplicada en la resolució del problema, representat mitjançant el programa paral·lel.

- **Assignació de tasques a nodes de processament**:
  Aquesta informació és interessant per poder tenir constància de les condicions en les que s'ha realitzat la simulació i poder aplicar-la a l'entorn real.

2.3 Implementació de la màquina de simulació

Aquest apartat exposa la màquina de simulació. Primer es comentarà el problema que es planteja i la solució de disseny adoptada. A partir d'aquí es detallarà el disseny de cadascun dels components que formen la màquina de simulació.

2.3.1 Anàlisi del disseny de la màquina de simulació

La primera fase que s'ha realitzat pel desenvolupament del simulador és determinar quins components s'han de modelar. A partir d'aquí el segon pas correspon a especificar el model de comportament d'aquests components i la seva interacció per poder realitzar la implementació.

Per dur a terme la implementació, s'ha de definir l' organització i el disseny que es dura a terme. Per realitzar-lo existeixen tres dissenys possibles que són: disseny monolític, disseny modular centralitzat i disseny modular distribuït.

- En el disseny monolític, la idea és crear un mòdul tancat, que vagi simulant ordenadament el funcionament de cada unitat de processament i per cada unitat processar les tasques mapejades, i en conseqüència, les seves primitives. El problema és que és difícil fer ampliacions, modificacions, detectar errors, etc.

- En el disseny modular centralitzat, es basa en dividir el problema inicial en problemes més simples encarregats del processament individual d'una part del problema de la simulació. Per exemple, es crearia un mòdul independent per les unitats de processament, un altre per la unitat de xarxa, etc. En aquest disseny es necessari un mòdul de control que determini l'avanç correcte de cadascun dels mòduls implementats.
  L'avantatge és que simplifica el problema inicial però en contra afegeix complexitat al mòdul de control.

- El disseny modular distribuït, és el mateix que l'anterior disseny però amb la diferència que el mòdul de control es troba repartit entre tots els mòduls implementats. Aquesta forma d'implementar permet que cada mòdul sigui independent de la resta de mòduls però per contra planteja el problema de la sincronització d'aquests.

Pel disseny del simulador ESPPADA s'ha utilitzat el disseny modular centralitzat perquè es coneix en tot moment la situació de la simulació i pot determinar que han de realitzar la resta de mòduls.

A partir d'aquí es determina el nombre de mòduls que haurien d'avaluar-se per la seva implementació: el mòdul que implementa la unitat de processament i un mòdul per la xarxa d'interconnexió.

- El mòdul de les unitats de processament s'encarregarà de representar el comportament d'aquests, la planificació de la CPU, la sobrecarrega del sistema operatiu, el processament de les primitives etc. Aquest interaccionarà amb el mòdul que implementa la xarxa d'interconnexió per aconseguir l'intercanvi de missatges entre tasques.

- El mòdul de la xarxa d'interconnexió que ha de controlar l' accés als canals de comunicació, l'enviament i l'arribada de missatges, el cost de comunicació, etc.

Una vegada s'ha determinat el conjunt de mòduls necessaris, cal determinar el seu comportament per que arribi finalment a la seva implementació. Per això s'avalua la forma en que el sistema real actua.

Un exemple de les unitats de processament es el que es mostra a la figura 2.4. Aquest exemple sols té en compte les accions que provoquen un canvi d'estat, com són l' execució, l' enviament, recepció i bloqueig a una barrera de sincronització.

Pel cas de la xarxa d'interconnexió basada en canals, el comportament és el que és mostra a la figura 2.5.

2.3.2 Esquema general de la màquina de simulació

En el simulador ESPPADA s'han definit dos unitats funcionals independents que implementen el comportament bàsic de cada un dels components identificats anteriorment.

Pel cas concret del node de processament, la unitat funcional associada s'anomena UP (*Unitat de Processament*), es repeteix tantes vegades com nodes existeixin a l'arquitectura, permeten de forma senzilla l' escalabilitat del sistema. Mentre que per la xarxa

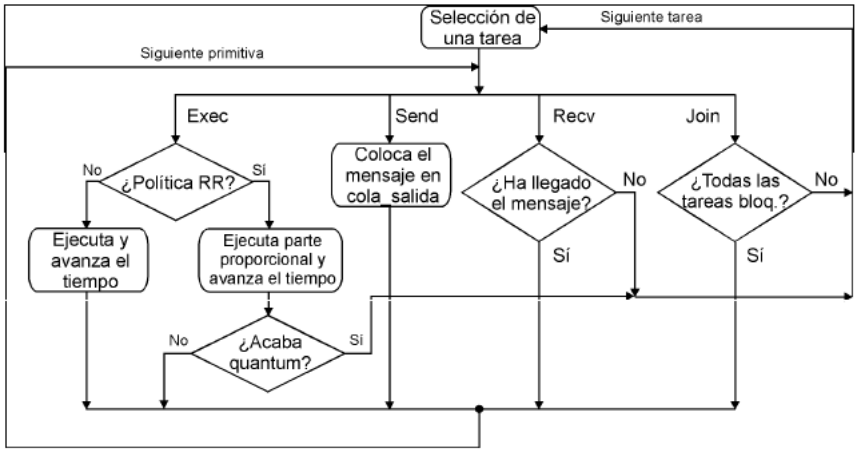

Figura 2.4: Comportament de les unitats de processament.

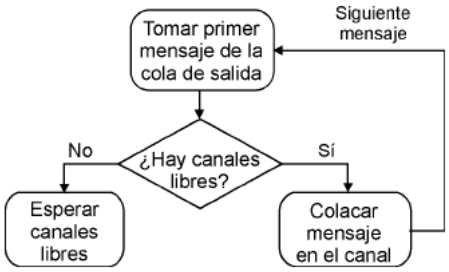

Figura 2.5: Comportament de la xarxa basada amb canals de comunicació.

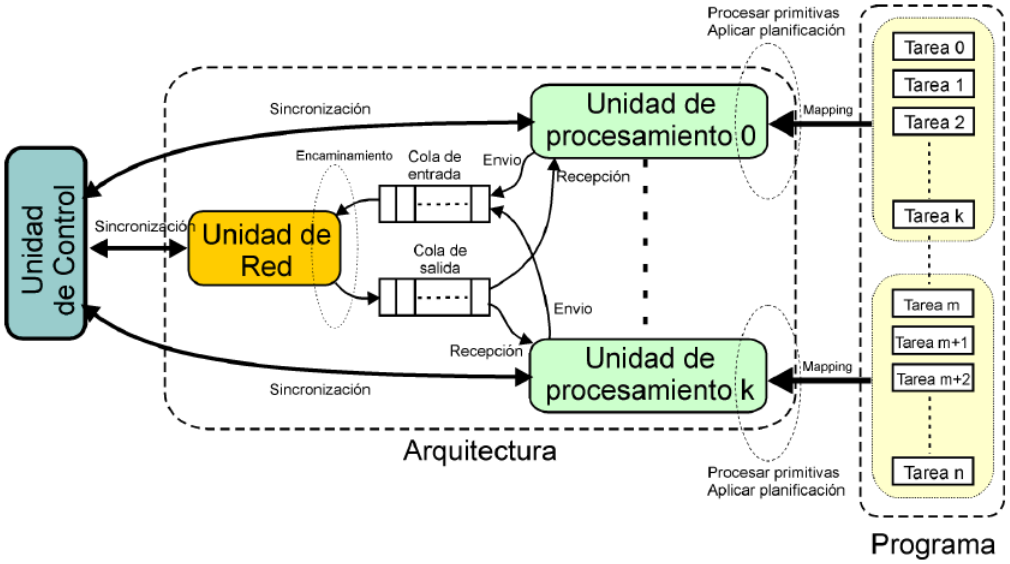

Figura 2.6: Esquema de les unitats funcionals que formen el simulador ESPPADA.

d'interconnexió sols apareix una única unitat funcional, que s'anomena UR (*Unitat de Xarxa*) i que interacciona amb la resta.

Per relacionar entre sí les unitats que representen finalment el sistema, existeix una unitat de control, UC (*Unitat de Control*), que determina en tot moment l' estat actual de la simulació i sincronitza les accions entre les UPs i la UR.

Aquest esquema està representat en la figura 2.6. Pot observar-se que existeix una correspondència amb la idea original, apareixent de forma centralitzada el control, mentre que les UPs agafen com a pròpies l' execució de les primitives de les tasques que gràcies al mapping té associades. La unitat de control, a la seva vegada, realitza la sincronització amb la unitat de xarxa, evitant que es produeixin incongruències en la simulació.

El simulador ESPPADA s'engloba dins dels simuladors d'events discrets[1]. Es a dir, en aquests simuladors l'evolució del sistema es basa amb el canvi d'estat, produït per un event, de les unitats funcionals, en temps determinats (moments discrets).

Com que aquest simulador és d'events discrets, aplica el criteri del *'següent event'* per l'avanç del temps, es a dir, no cal tenir en compte el temps d'inactivitat del model perquè el canvi es produeix en el moment de l'event.

El simulador ESPPADA adopta dos criteris, un és que els events venen donats per la propia evolució del simulador i l'altre és que formen part d'una entrada externa que dirigeixi el seu avanç. Uns són creats de forma dinàmica, a l' executar-se les primitives d'enviament a les unitats de processament, activant la unitat de xarxa. Altres pel contrari venen donats a partir de les primitives, que formen les tasques i que les UPs processen per simular l' execució d'aquestes.

2.3.3 Disseny de les unitats funcionals

A continuació es detalla el funcionament de les unitats que composen el simulador ESPPADA. Aquestes són: la UP (*Unitat de Processament*), la UR (*Unitat de Xarxa*) i la UC (*Unitat de Control*).

2.3.3.1 Unitat de Processament - UP

La unitat de processament implementa el comportament del node de processament. Els nodes de processament implementats en el simulador ESPPADA es fan càrrec de les següents accions:

- Acceptar l'assignació de les tasques que provenen de la fase de mapping. Aquestes tasques estan guardades en unes llistes internes.
- Processar i enregistrar els events produïts per les primitives d'execució de les tasques i crear els missatges adequats per la interacció entre tasques.
- Aplicar les polítiques de planificació de la CPU, de manera que puguin intercanviarse les tasques a mesura que succeïx la simulació. Cal tenir en compte la sobrecarrega produïda pels canvis de context.
- Controlar la sobrecàrrega de comunicacions per l'enviament de missatges a través de la xarxa. S'ha de tenir en compte aquesta sobrecàrrega, perquè a la realitat l'enviament d'un missatge passa per la pila de protocols de comunicació que pot ser diferent si es transmeteixen localment o remotament.
- Monitoritzar el comportament i rendiment de cada node que simula, per generar les dades pel seu anàlisi posterior.

La UP apareix repetida el mateix número de vegades que nodes de processament hi han presents a l'arquitectura. Cada UP posseeix una execució independent, avançant la simulació sense tenir en compte que està passant a la resta d'unitats. Cal destacar que el simulador ESPPADA no implementa les UPs totes sincronitzades per tant poden tenir temps diferents, al poder avançar una més ràpida que l'altra. Si les UPs no interaccionen entre si aquesta diferència no es rellevant. Pel contrari, provoca possibles problemes alhora de processar les primitives de recepció ja que no pot assegurar el moment d'arribada del missatge que s'espera, del que pot ser possible que inclòs no s'hagi processat la primitiva d'enviament associada. Aquesta situació pot provocar que el missatge pugui arribar al passat, present o inclòs al futur respecte el moment actual de la UP receptora.

Si el missatge arriba al passat o present la tasca podrà continuar la execució sense major dificultat, mentre que si encara ha d'arribar haurà d'abandonar la CPU, donant pas a la planificació de la següent tasca.

Aquesta situació es pot agreujar si el comptador de temps de la UP emisora, conté un valor que l'ubica en el passat respecte el comptador de temps de la UP receptora. Si a més, la tasca emisora no s'està executant en aquests moments, llavors no es pot determinar

l'arribada del missatge i en conseqüència no es té certesa de si s'ha de permetre que continuï la execució de la tasca receptora o pel contrari planificar una altra tasca.

La manera en que soluciona aquest problema el simulador ESPPADA és mitjançant un mètode d'avanç conservador per la simulació al no permetre avançar la UP receptora fins que no sapigui amb seguretat quina es la situació.

Al aplicar mètodes conservadors es pot produir un estat de *deadlock* al donar-se situacions en les que la tasca que bloqueja la UP no deixa avançar a altres tasques que poder ser les emissores d'aquells missatges que espera una altra UP bloquejada.

Per poder realitzar la simulació, la UP conté unes estructures utilitzades per la definició de les tasques denominades SPCB (*Simulation Process Control Block*)[1]. Aquesta estructura manté informació sobre el estat actual de cada tasca, indicant el seu identificador, el punt a on s'ha quedat bloquejada si ho està, la tasca emissora a la que espera en el bloqueig, el identificador del missatge que la bloqueja, etc. D'aquesta manera en el moment en que la tasca torni ha estar preparada per la execució, es té tota la informació necessària per fer-ho.

A més la UP posseeix quatre cues que determinen en tot moment la situació a la que es troben les tasques que té assignades. En aquestes cues es guarden els SPCB, i existeix la possibilitat de que aquests vagin canviant de cua a mesura que la simulació avanci. Aquestes cues son les següents:

- *Cua de Preparats*: Aquesta cua manté una llista de totes les tasques que estan preparades per processar les seves primitives. Segons la política de planificació seleccionada es manté un tipus d'ordenació diferent. Pel cas de la *planificació no apropiativa*, es troben ordenades pel moment d'activació de les tasques, de manera que la primera de la llista correspont a la primera tasca que es troba disponible per processar-se. A la *planificació Round-Robin* la ordenació sols indica l'ordre en el que poden accedir a la CPU, seguint una estructura circular.

- *Cua d'Espera*: En aquesta cua s'inclouen totes les tasques, que al haver executat una primitiva de recepció, han determinat que no provocaran un bloqueig de la UP, o bé perquè es coneix el moment de recepció del missatge i es al futur, o perquè el missatge ve d'una tasca mapejada a la mateixa UP. Els missatges que arribin al passat o al present son processats sense obligar a la tasca l'abandonament de la CPU.
  Les tasques que es troben en aquesta cua no es poden passar directament a la cua de preparats per no pot determinar-se que pot succeir fins al moment de recepció, perquè pot ser que s'hagi de planificar una altra tasca o s'hagi d'avançar el temps actual fins aquest moment.

- *Cua de Bloquejats*: Conté les tasques que al executar la primitiva de recepció provoquen el bloqueig de la UP. Aquest bloqueig es degut a que no existeix informació sobre el moment d'arribada del missatge. Les tasques que es troben en aquesta cua, provoquen que la UP no pugui avançar la simulació, i per tant necessita informació provinent de la unitat de control per determinar quina es la situació.
  Aquestes cues estan separades de la cua d'*Espera* degut a que pot donar-se el cas

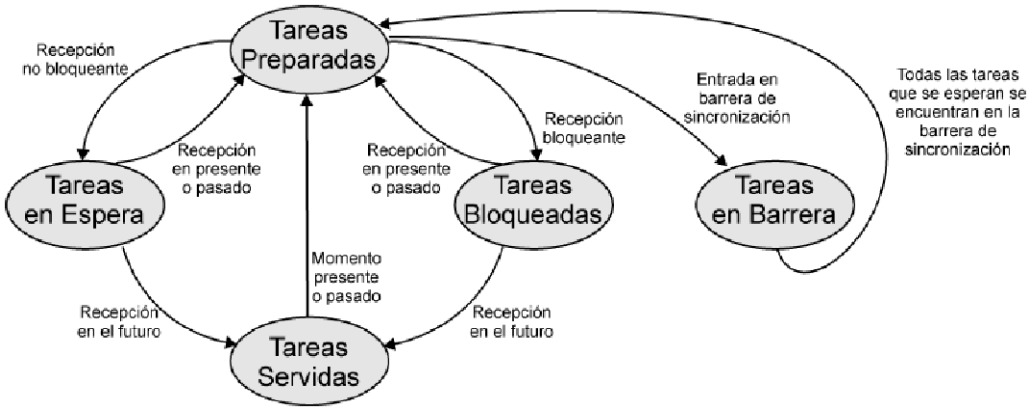

Figura 2.7: Diagrama de comportament de les tasques a la simulació.

que la UC permeti la planificació d'una altra tasca i sigui necessari, mes endavant mantenir un bloqueig existent prèviament.

- *Cua de Sincronització*: Les tasques poden executar primitives de sincronització que les detinguin fins que totes les tasques passin per aquest punt de sincronització. Aquesta situació no provoca un bloqueig de la UP si no que para la execució d'aquesta tasca en concret, així pot residir a en aquesta llista fins que la sincronització s'ha produït, llavors passa a la cua de *Preparats*.

Les tasques tenen un comportament dinàmic durant la simulació de manera que poden canviar d'una cua a mesura que es van processant les primitives, aquest moviment entre cues es produeix segons el diagrama mostrat a la figura 2.7.

El control de missatges que es generen a partir de les primitives d'enviament es realitza en dos nivells. El primer d'ells fa referència als missatges *interns* que son aquells es produeixen entre tasques que estan mapejades sobre la mateixa UP. El segon es pels missatges que van dirigits a UPs diferents i han de moure's a través de la xarxa d'interconnexió.

El simulador ESPPADA distingeix aquest dos nivells, perquè els missatges interns mai provocaran un bloqueig intern de la UP ja que si no està present el missatge en aquest moment, es pot assegurar que a través de la planificació de tasques, aquest arribarà al futur, mai al passat. Aquests missatges a causa de que no són arribin accedir a la xarxa, estan emmagatzemats a una llista interna, separada dels missatges que envien altres UPs, per tenir un accés més eficient.

Pel contrari, en el cas dels missatges externs si que poden provocar el bloqueig de la UP, degut a que no es possible que no es pugui determinar el moment d'arribada al estar relacionat amb el moment d'enviament de la UP emissora.

2.3.3.2 Unitat de Xarxa - UR

La unitat de xarxa s'encarrega de simular l' encaminament dels missatges que es generen a les unitats de processament. Per realitzar aquesta funció utilitza dos cues diferents,

una és on s'emmagatzemen els missatges que surten de les UPs i en l'altra on arriben a les UPs. D'aquesta manera quan les UPs processen els missatges d'enviament, deixen els missatges a la cua de sortida i llavors la UR s'encarrega de fer que arribi a la cua d'entrada.

En aquestes cues s'emmagatzemen unes estructures que contenen tota la informació necessària, com el moment d'enviament i el de recepció. A més estan ordenades segons el moment d'enviament, per la cua d'entrada, i segons el moment d'arribada, per la cua de sortida.

Quant una UP processa una primitiva de recepció, que és externa, es comprova si el missatge associat es troba a la cua de sortida. Si aquest és el cas es pot assegurar el moment d'arribada i en conseqüència correspon a una recepció no bloquejant. Pel contrari si no es troba dins d'aquesta cua, aquesta primitiva passa a ser bloquejant, llavors la UP actuarà en conseqüència.

La unitat UR ha d'assegurar que les precedències dels enviaments dels missatges es mantinguin de manera correcta, es a dir, que un missatge enviat en un moment determinat no s'avanci a altres que puguin arribar en el passat. Aquesta anomalia pot donar-se ja que les UPs avancen asíncronament i possiblement amb temps de simulació diferents.

Per la implementació de la xarxa, s'utilitzen dos algorismes que representen les dos xarxes que s'han comentat en la subsecció 2.2.1.2.

El primer algorisme fa referència a una xarxa en la que existeix una comunicació de totes les unitats de processament amb totes i sense límit pel nombre d'enviaments simultanis.

El segon algorisme es defineix a partir de la utilització d'un número de canals i un ample de banda, els dos definits per l'usuari. Aquest tipus de xarxa provoque restriccions majors a la transmissió de dades, ja que els missatges han de processar-se en estricte ordre d'arribada.

A continuació es comenta cada una de les implementacions.

- **Xarxa totalment connectada**
  Aquesta xarxa permet una transmissió sense restriccions en elal límit de missatges simultanis que es puguin donar i connecta tots els nodes de processament entre aquests de forma directa.
  L'única penalització existent és un factor de retard que, en funció del volum de dades a transmetre, produeix un major o menor temps de transmissió. En la següent expressió s'indica el càlcul del temps necessari per la transmissió d'un missatge de longitut M bytes.

$$Temps = M \times Factor\_de\_retard$$

- **Xarxa basada amb canals**
  Aquesta xarxa té la característica que el número de comunicacions simulatanies ve definit pel número de canals existents, de manera, que quan aquests estiguin ocupats no podran accedir a la xarxa fins que aquesta no estigui lliure. Això provoca un retard que s'ha de sumar al temps d'enviament.

En la següent expressió s'indica com es calcula el moment d'arribada del missatge:

$$Moment\_arribada = Moment\_entrada\_canal + \frac{Volum\_dades}{Ample\_de\_Banda}$$

Es pot observar que l'expressió del moment d'arribada d'un missatge no depèn del moment d'enviament sinó del moment d'entrada al canal. A més si tenim un ample de banda més gran el missatge tardarà menys en arribar.
En aquesta xarxa, el simulador, controla que cap missatge amb un moment d'enviament determinat no pugui entrar al canal si existeix un missatge amb moment d'enviament menor.
Aquesta restricció pot provocar que UPs bloquejades en espera de missatge que ja han estat enviats i que es troben a la cua d'entrada, no puguin accedir als canals perquè pot ser que arribin missatges en el passat. La solució és que la UR sols s'ha d'activar quan totes les UPs estiguin bloquejades, ja que assegura que ninguna d'elles podrà generar més missatges que puguin estar al passat.
Una situació que es pot donar, es que totes les UPs quedin bloquejades en espera de missatges que no poden accedir perquè els canals no acaben de ser alliberats i a més, els missatges dels canals no acaben de ser rebuts degut a que les UPs tenen un moment menor que moment d'arribada del missatge. Aquesta situació bloquejant s'anomena *deadlock* del sistema.
El simulador ESPPADA soluciona el *deadlock* de dues maneres, sense necessitat d'unitat de control. El primer, si sols hi ha una UP activa i el segon si sols hi ha un canal de comunicació.
En el primer cas, a l'haver una UP activa s'avança el moment de la UP al menor dels moments d'alliberació dels canals, permetent que aquest quedi lliure.
En el segon cas, a l'haver un sol canal de comunicació es comprova si la UP al que va dirigit té un moment menor que el de l'alliberació del canal, llavors s'avança el canal fins aquest instant de recepció.
En qualsevol altre cas, la UC serà l'encarregada de trobar la solució.

2.3.3.3 Unitat de control

El disseny del simulador ESPPADA té una similitut amb la simulació paral·lela d'events discrets PDES (*Parallel Discrete Events Simulation*)[12] ja que les unitats de processament poden activar-se concurrentment i avançar independentment.

La simulació PDES utilitza un conjunt de processos lògics anomenats LP (*Logical Process*) que avancen asíncronament encara que d'alguna forma coordinada a través d'un mecanisme de sincronització. Cada LP té unes cues d'entrada i sortida d'events que s'utilitzen per intercanviar informació entre diferents LPs.

El problema d'aquesta simulació es que els LPs s'executen de manera independent i en conseqüència poden avançar a diferents velocitats. Això pot provocar un error de

causalitat. Aquest error ve motivat pel fet que una LP pot processar events en el futur, mentre altres LPs situades en el passat, encara no han generat events anteriors als que està processant. Per solucionar aquest problema, els LPs s'han de sincronitzar i per això existeixin dos mecanismes de sincronització: el mètode conservatiu i el optimista.

El mètode conservatiu, ho és en el sentit que un event no es processa fins que no es pugui assegurar que no provocarà un error de causalitat. Això pot provocar un deadlock al bloquejar la LP en espera de les dades que assegurin el seu avanç. En el mètode optimista, el LP assumeix que tots els events poden ser processats, per tant avança, encara quant aquest fet no es pugui assegurar. Degut a que possiblement hi hagi situacions d' error de causalitat, es permet retrocedir al passat, fins un unt que es reconeix com a segur a la simulació.

Tornant al cas particular del simulador, podem veure com existeix una gran semblança entre els LPs i les UPs. Tant la UP com la UR poden descriure's com LPs. Per la sincronització de les unitats el simulador utilitza el mètode conservador, de manera que no permeti l'avanç de la simulació fins que no asseguri el event.

Per detectar el deadlock produït pel mètode conservador, s'utilitza la unitat de control que té una visió global de l'estat del sistema i determina les accions a realitzar. El simulador utilitza la tècnica de detecció del deadlock per després aplicar un mecanisme de recuperació.

El deadlock sols es produeix quant les UPs processen una primitiva de recepció, de la que no es coneix a priori el moment d'arribada del missatge associat. Això passa, o bé perquè el missatge no ha estat encara enviat, o perquè està sent processat en la UR i encara no ha arribat l'event que indica la seva recepció.

Per entendre quan es produeix deadlock es necessari formalitzar les possibles situacions i esquematitzar el procediment de recuperació. En aquest projecte no explicarem els mecanismes de detecció i recuperació que utilitza el simulador ESPADDA. Per a més informació llegir el llibre [1].

2.3.4 Interacció de les unitats funcionals

Les unitats funcionals interaccionen entre si per poder realitzar la simulació de l' execució del programa. La interacció entre les diferents UPs es realitza de forma seqüencial perquè tot i que existeix la possibilitat que les UPs avancin de manera concurrent no ho faran perquè sols s'utilitza una màquina.

En l'algorisme d'interacció de les unitats funcionals sols s'activa la UR quan totes les UPs es troben bloquejades. També permet l'avanç de les UPs fins que aquestes determinen de forma individual que no ho poden fer. Igualment, la UC s'activa quan totes les UPs es troben bloquejades, de fet, el número de vegades que aquesta s'activa depèn directament de les comunicacions entre tasques mapejades en UPs diferents.

3 Integració de la memòria virtual en el simulador ESPPADA

3.1 Introducció

A partir d'aquesta secció comença a explicar-se detalladament el que s'ha en aquest projecte de fi de carrera. Fins ara haviem parlat del simulador ESPPADA realitzat per l'autor Fernando Guirado[1]. L'objectiu d'aquest projecte és entendre el funcionament d'aquest i a partir d'aquí afegir nous mòduls o funcionalitats per estudiar altres aspectes que afecten el rendiment de l'execució de tasques, com la memòria, l'integració de tasques locals, etc.

En aquest capítol parlarem de la integració de la memòria virtual en el simulador que és important per estudiar la influència en el rendiment de les tasques paral·leles i com actua el sistema operatiu davant d'una sobrecarrega de memòria. En la realització d'aquest apartat ens hgem recolçat amb l'expèriencia adquirida en la realització del PFC de l'Enginyeria Tècnica de Informatica de Sistema sobre la memòria virtual del nucli del sistema operatiu Linux. Això ens permet aportar aquestes idees a la programació d'un entorn paral·lel per aproximar encara més a la realitat els resultats del rendiment d'un programa paral·lel executat en el simulador ESPPADA.

Per integrar la memòria virtual al simulador s'ha estudiat el seu codi font escrit amb Java versió 1.3.0.02, s'ha canviat la interfície gràfica, afegint nous formularis i paràmetres que determinen el comportament del simulador i s'ha integrat al codi original, nous algorismes i funcions que explicarem amb detall en les següents seccions.

En aquest apartat es necessari parlar de tots els aspectes que comporta la memòria virtual com les faltes de pàgina, l'algorisme de reemplaçament de pàgines, quant actua el sistema operatiu per càrregar pàgines, etc.

3.1.1 Breu descripció del subsistema de memòria virtual del sistema operatiu Linux

En un sistema que soporta memòria virtual, la memòria principal es divideix en blocs de mida fixa anomenats marcs de pàgina. Per executar un programa, inicialment es carrega part de la seva imatge del fitxer de memòria secundària a memòria principal en pàgines. Aquestes pàgines contenen segments de codi del programa, segments de dades, la pila, biblioteques específiques, etc. Durant l'execució del programa es produeïxen faltes de pàgina, que és controlat per la MMU (*Memory Management Unit*) de la màquina, i en conseqüència es càrreguen noves pàgines de memòria per part del sistema operatiu. Cada cop que es demana carregar una pàgina a memòria, el sistema operatiu comprova que

hi hagi suficient memòria disponible, si n'hi ha assigna un marc de pàgina i copia la pàgina de la memòria secundaria. Pel contrari, si no hi ha memòria disponible crida a l'algorisme de reemplaçament de pàgines per alliberar marcs de pàgina. Aquest algorisme utilitza un criteri per seleccionar les pàgines candidates a ser desallotjades o desplaçades a la memòria d'intercanvi. Per exemple, existeixen algorismes com el LRU (*Least Recently Used*)[2] que selecciona les pagines menys recentment utilitzades, o el LFU (*Least Frequently Used*)[2] que selecciona les pàgines menys freqüentment utilitzades, el MRU (*Most Recently Used*), el FIFO *(First Input First Output)* i finalment el NRU (*Not Recently Used*). Una vegada ha alliberat els marcs de pàgina, copia les pàgines de disc a memòria principal i retorna l'execució del programa.

Al finalitzar l'execució del programa, allibera totes les pàgines de memòria principal que pertanyen a aquest programa i destrueix el procés.

En la subsecció 3.2 s'explicarà i s'il·lustrarà el algorisme utilitzat per realitzar tot el procediment de la gestió de la memòria virtual durant la simulació d'un programa paral·lel a l'ESPPADA.

En la subsecció 3.3 es parlarà com s'especifica a l'arquitectura la memoria virtual utilitzant el llengüatge XML per recollir dades. També es parlarà sobre el disseny de l'interfície gràfica del simulador relacionat amb els paràmetres afegits de la memòria virtual.

En la subsecció 3.4 es detallarà la implementació de les funcions, algorismes, variables utilitzades per integrar el mòdul de la memòria virtual al simulador.

I finalment a la subsecció 3.5 s'exposa l'experimentació, es a dir, els resultats obtinguts de la simulació d'un programa paral·lel que utilitza memòria virtual.

3.2 Algorisme

En aquesta secció descriurem l'algorisme que dur a terme el simulador ESPPADA per la gestió de la memòria virtual durant la simulació de la primitiva Exec d'una tasca.

En la figura 3.1 es mostra l'algorisme de gestió de la memòria virtual durant la simulació de la primitiva Exec.

Aquest algorisme s'activa durant la fase de la simulació de la primitiva Exec perquè és quan es produeixen faltes de pàgina de la memòria virtual de la tasca, perquè la resta de primitives no són primitives que necessiten memòria pel seu processament.

Inicialment l'algorisme determina si el mòdul de memòria virtual està activat al simulador o no. Si està activat comença el procés de simulació de la memòria. Pel contrari si no està activat, realitza la simulació de la primitiva Exec sense la gestió de la memòria virtual.

Quan està activat[1], primer calcula els cicles de CPU de la primitiva Exec segons el tipus de planificació, que pot ser Round Robin o No Apropiativa. Si la planificació és Round Robin tindrà en compte que els cicles que es simularan d'aquesta primitiva és el *quantum* assignat a la tasca. Pel contrari, si és No Apropiativa agafarà com a cicles inicials de la simulació els cicles de la primitiva Exec.

[1] Per activar l'algorisme llegir la subsecció 3.3.3

Després l'algorisme passa a inicialitzar o recollir els parametres de la tasca relacionats amb la memòria virtual, com el *rss minimum* i *pagefault interval*. El rss minimum és un paràmetre que indica la quantitat d'unitats de memòria que assignarà per a cada falta de pàgina. El pagefault interval indica cada quant de temps es produira una falta de pàgines.

Un cop obtingut aquests paràmetres específics, inicialitza els paramètres generals com el temps de gestió d'una falta, anomenat temps RSS, que indica l' overhead de cada falta de pàgina. Aquest paràmetre el recull de l'arquitectura del sistema.

A partir d'aquest punt comença el bucle principal del qual està composat l'algorisme. En cada iteració determina si es produeix falta de pàgina, simula la primitiva Exec, i avança el temps de la simulació amb els cicles de l'interval de faltes de pàgina. Tot aquest procés s'explicà més detalladament a continuació.

El bucle principal comprova en cada iteració els cicles que queden per finalitzar la simulació de la primitiva Exec. Dins del cos del bucle, primer genera un número aleatori mòdul 100. A continuació determina si es produïrà en aquest instant una falta de pàgina, a través de la següent expressió:

$$numero\_aleatori < 100 - \frac{task\_rss \cdot 100}{task\_vm}$$

El número aleatori generat es comparat amb l'expressió que calcula el percentatge que no hi ha memòria assignada a la tasca. Podem observar que si el valor de task_vm és relativament gran comparat amb el valor task_rss hi haurà més probabilitat de falta de pàgina. En canvi si els dos valors són iguals l'expressió del percentatge valdrà zero i per tant no es produïrà cap falta de pàgina.

Si es produeix falta de pàgina passa a assignar el pàrametre rss_minimum unitats de memòria a la tasca. Pel contrari, si no es produeix falta de pàgina llavors passa directament a simular la primitiva Exec.

Durant el procés de gestió de les faltes de pàgina, inicialment incrementa el comptador de faltes de pàgina que servirà per mostrar els resultats de la simulació un cop finalitzada aquesta. Després calcula el rss minimum, que pot ser aquest mateix valor o la resta d'unitats de memòria que falten per assignar a la tasca si aquesta es superior a rss minimum. A continuació crida a la funció que farà l'assignació de memòria i que al mateix temps comprova si ha suficient memòria disponible. Si no n'hi ha, l'algorisme principal crida a l'algorisme de reemplaçament de pàgines i aquest allibera les pàgines necessàries per poder assignar més memòria a la tasca. L'algorisme de reemplaçament de pàgines selecciona la tasca que té més memòria resident. Tot el proces d'alliberament de pàgines de memòria ho realitza dins d'un bucle. En cada iteració crida a la funció d'alliberar memòria i si ha alliberat suficient memòria acaba el bucle. En cada iteració mostra i enregistra l'event de simulació de l'alliberació de memòria principal.

Un cop finalitzat l'alliberació de memòria, assigna la memòria, mostra per la consola i enregistra l'event d'assignar memòria principal mostrant la sobrecàrrega produïda per la gestió de la falta de pàgina, etc. Després decrementa els cicles de la sobrecàrrega produït per la falta de pàgina, als cicles que queden per l'execució de la primitiva Exec que són els del bucle principal.

El següent pas és determinar si la iteració actual es l'última del bucle, i si es l'última actualitza el comptador de cicles que queden per tal d'assegurar que acabarà el bucle principal. A més calcula els cicles que és simularan a la primitiva Exec en la iteració actual.

A continuació passa a determinar el tipus de planificació i en funció de la planificació fa un tractament especial de la primitiva i després mostra per la cònsola i enregistra l'event d'execució de la primitiva Exec. Llavors actualitza el comptador de cicles del bucle principal decrementant-li els cicles que s'han simulat de la primitiva Exec a la iteració actual.

Finalment, determina un altre cop si es l'última iteració i en cas afirmatiu elimina la primitiva de la llista de primitives de la tasca i la guarda a la llista de repositori per després ser eliminada.

Llavors acaba el bucle i torna a començar una nova iteració si queden més cicles per simular aquesta primitiva o si és el contrari finalitza la simulació d'aquesta.

3.3 Especificació

La memòria virtual està definida tant en l'arquitectura del sistema com en els programes paral·lels. Les dues es troben definides en fitxers escrits amb llenguatge XML que és el llenguatge que utilitza el simulador, amb l'ajuda d'un parser escrit amb Java, per recollir les dades i els seus atributs en forma d'arbre i manejar-les fàcilment.

En la secció 3.3.1 s'explica detalladament la definició de la memòria en l'arquitectura de l'entorn paral·lel.

En la secció 3.3.2 s'explica detalladament com es defineix les característiques o comportament de les tasques d'un programa paral·lel relacionat amb la memòria.

3.3.1 Especificació de l'arquitectura del sistema

En la figura 3.2 es mostra una arquitectura definida amb XML amb els seus requisits de memòria.

Com es pot observar, el nom de l'arquitectura és "Arquitectura amb memoria virtual", amb un valor base de temps d'execució BET de 1.0, amb un overhead de 0 unitats de temps i a la següent linia hi ha definida l'etiqueta '**memory_enabled**' amb valor igual a cert que serveix per especificar que existeix a l'arquitectura del sistema la memòria virtual i pertant el simulador haurà de gestionar tot el que comporta aquest subsistema.

En la següent linia hi ha definit una etiqueta anomenada '**local_task**' que també forma part de l'estudi del rendiment de la memòria en un entorn paral·lel i que veurem en el capítol 4 per a què serveix.

Després defineix el tipus de xarxa que és "full connect" amb un factor de retard 1.0 unitats de temps. A continuació hi han definits els processadors del sistema que en total són 5. A cada processador hi ha definit l' identificador i el període de CPU. A més tenim tres etiquetes que són '**memory_size**', '**pagefault_overhead**' i '**local_task**'. La primera serveix per especificar el tamany de la memòria local d'aquesta CPU. Les unitats són abstractes ja que depén de l'ús que en vulgui fer l'usuari. La segona serveix

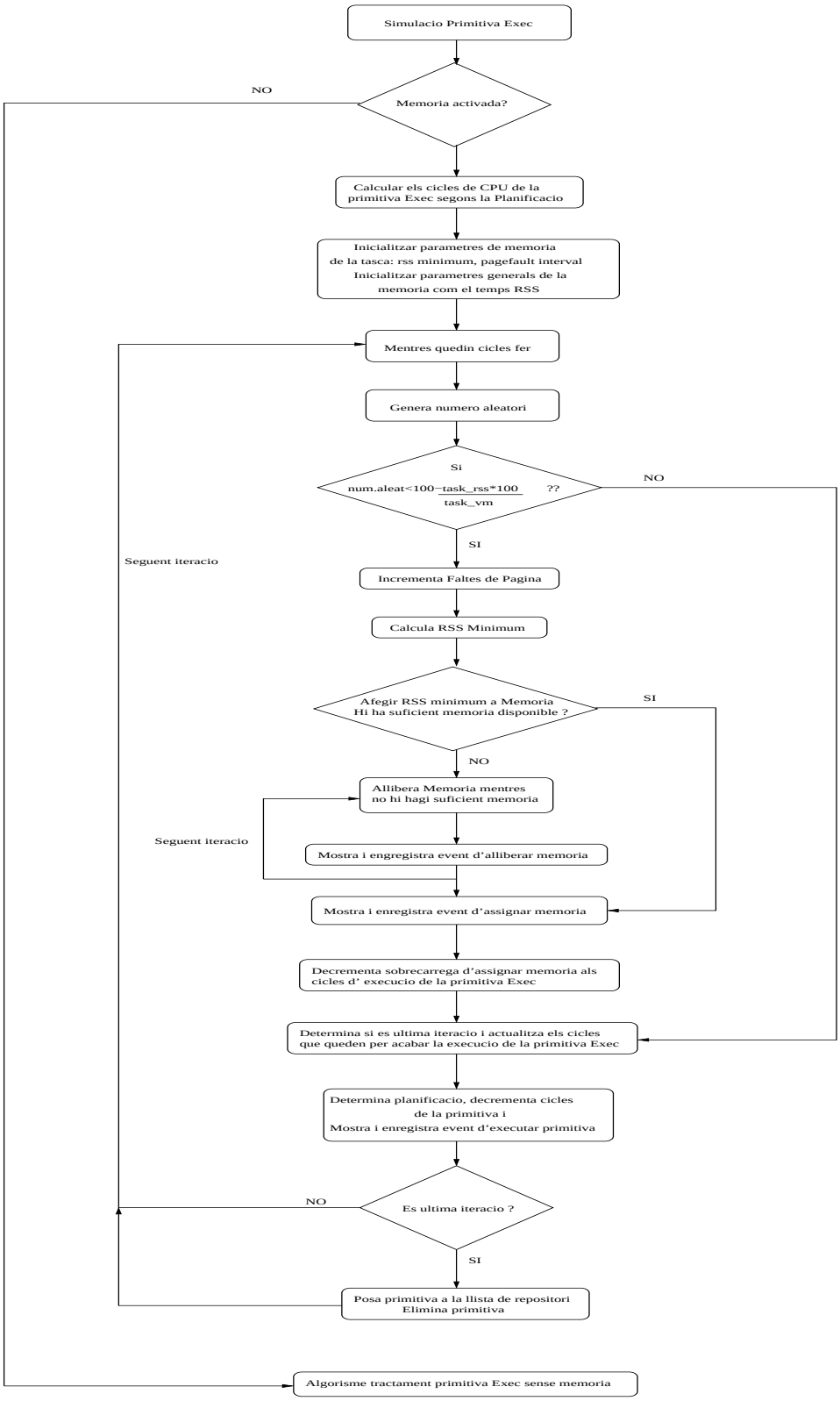

Figura 3.1: Algorisme de gestió de la memòria virtual

per especificar l' overhead de cada falta de pàgina que es produeïx en un instant de temps. El simulador té en compte aquesta sobrecàrrega durant la simulació ja que és important la influència que té en el rendiment global del sistema. I finalment la última que serveix per especificar si existeix tasca local en aquesta CPU. Aquesta última s'explicarà detalladament en el capítol següent.

Podem observar que la primera CPU té un identificador igual a 1, un període de CPU igual a 1.0, un tamany de memòria de 120 unitats, una sobrecàrrega de falta de pàgina de 10 unitats de temps i existeix una tasca local en el mapping d'aquesta CPU.

3.3.2 Especificació de la memòria virtual en els programes paral·lels

Els programes paral·lels també se'ls hi ha d'especificar quin comportament han de tenir respecte a la memòria virtual, com per exemple la memòria virtual que té cada tasca, quanta memòria s'ha d'assignar per cada falta de pàgina, etc. Els paràmetres necessaris per definir el comportament d'un programa relacionat amb la memòria virtual són els que es mostren a la figura 3.3.

En aquesta figura podem observar que el programa s'anomena "Exemple memoria virtual" i que té definides tres tasques. Recordem que un programa paral·lel està composat per tasques i cada tasca té la seva memòria virtual. Cada tasca es defineix amb l'etiqueta '**task**' i té els atributs '**id**', '**virmem**', '**rss_minimum**' i '**pagefault_interval**'. El primer serveix per definir l' identificador de tasca que pot ser qualsevol número no repetit. Es necessari aquest identificador perquè el simulador internament treballa amb un cert nombre de tasques que han de tenir identificació per saber de quina es tracta. El segon paràmetre serveix per especificar la quantitat de memòria que s'assignarà quan es produeixi una falta de pàgina. El tercer paràmetre serveix per especificar el tamany de la memòria virtual de la tasca. El simulador, per tenir en compte aquest tamany, el guarda en una llista d'estructures de tasques de memòria que conté tota la informació referent a la memòria de les tasques. En la secció 3.4 es parla detalladament sobre la implementació de la memòria virtual. El tercer paràmetre serveix per especificar cada quan es produïrà una falta de pàgina. Si recordem la secció 3.2, parla de l'algorisme general de la memòria on hi ha un bucle que avança en cada iteració aquest paràmetre de l'interval de fallides de pàgina. Cal destacar que aquest paràmetre ha de valer mes gran que zero si s'activa la memòria virtual en el simulador.

A l'exemple podem observar que la primera tasca té un identificador igual a 7, una memòria virtual de 50 unitats, un rss_minimum de 50 unitats i un interval de fallides de pàgina de 190 cicles de rellotge.

També cal destacar, que si no hi hagues definit la primitiva Exec a l'exemple, la memòria virtual no funcionaria ja que es l'unica primitiva que treballa amb memòria virtual.

3.3.3 Disseny de la interficie del simulador

Per integrar la funcionalitat de la memòria virtual s'ha modificat el disseny de la interfície gràfica del programa ESPPADA.

```
<?xml version="1.0"?>
<!-- Architecture definition created by ESPPADA tool v1.0 -->
  <architecture name="Arquitectura amb memoria virtual">
  <cpu_base value="1.0"/>
  <context_change value="0"/>
  <memory_enabled value="true"/>
  <local_task for="1" exec="300" wait="30" terminate="true" virtual_memory="128"
   rss_minimum="20" pagefault_interval="200"/>
  <net type="full_connect">
    <delay value="1.0"/>
  </net>
  <planification method="round_robin" quantum="200"/>
  <!-- Process units definition -->
  <processor id="1">
    <cpu_period value="1.0"/>
    <memory_size value="120"/>
    <pagefault_overhead value="10"/>
    <local_task value="true"/>
  </processor>
    <processor id="2">
    <cpu_period value="1.0"/>
    <memory_size value="120"/>
    <pagefault_overhead value="10"/>
    <local_task value="true"/>
  </processor>
  <processor id="3">
    <cpu_period value="1.0"/>
    <memory_size value="150"/>
    <pagefault_overhead value="10"/>
    <local_task value="false"/>
  </processor>
  <processor id="4">
    <cpu_period value="1.0"/>
    <memory_size value="100"/>
    <pagefault_overhead value="10"/>
    <local_task value="false"/>
  </processor>
  <processor id="5">
    <cpu_period value="1.0"/>
    <memory_size value="140"/>
    <pagefault_overhead value="10"/>
    <local_task value="false"/>
  </processor>
</architecture>
```

Figura 3.2: Definició de l'arquitectura del sistema escrit amb XML i els seus requisits de memòria.

```
<?xml version="1.0"?>
  <program name="Exemple memoria virtual">
  <task id="7" virmem="50" rss_minimum="50" pagefault_interval="190">
    <exec run="200"/>
    <exec run="333"/>
  </task>
  <task id="4" virmem="175" rss_minimum="10" pagefault_interval="160">
    <exec run="400"/>
  </task>
  <task id="6" virmem="100" rss_minimum="50" pagefault_interval="190">
    <exec run="200"/>
    <exec run="333"/>
  </task>
  </program>
```

Figura 3.3: Definició d'un programa paral·lel amb memòria virtual

En la figura 3.4 es mostra la pantalla general del formulari de la definició de l'arquitectura. En aquest formulari, dins de la pestanya '**General**' s'ha afegit un nou checkbox anomenat '**Memory enabled**' que serveix per activar o desactivar l'activitat de la memòria virtual del sistema durant la simulació. Els requeriments a l' activar la memòria són que a cada processador la seva memòria principal ha de ser més gran que zero perquè sinó no funcionaria correctament la simulació.

En la figura 3.5 es mostra la interfície de la definició dels processadors en el formulari de la definició de l'arquitectura. En aquest formulari, està seleccionat la pestanya '**Process units**' per la definició de les característiques dels processadors. A l'esquerra de la pantalla hi ha representat la llista de processadors que actualment estan definits. Per seleccionar un processador solament s'ha d'activar amb el ratolí el processador que es vol modificar els seus atributs i apretar el botó '**Modify**'. Els seus atributs apareixen a la dreta de la pantalla, a on hi posa '**Process unit parameters**', i si ens fixem una mica més avall veurem els parametres de la memòria principal. Aquests són els nous parametres que s'han afegit.

Existeixen tres paràmetres importants per definir la memòria principal i el seu comportament als processadors del sistema. Aquests són '**Memory size**', '**Pagefault overhead**' i '**Local task**'. El primer paràmetre serveix per especificar la mida de la memòria principal del processador. Com podem observar la memòria general del sistema és distribuïda, i cada processador té la seva memòria local. A aquesta figura podem veure que s'ha definit una memòria de 132 unitats. El segon paràmetre serveix per especificar la quantitat de temps de la sobrecarrega produïda per una falta de pàgina. A l'exemple podem obervar que es de 10 unitats de temps. L'altre i últim parametre serveix per especificar si aquest processador tindrà mapejada una tasca local. L'objectiu d'una tasca local es ocupar CPU, memòria i no comunicacions mentre es van executant les altres tasques per estudiar com afecta el rendiment del sistema en general. En el capítol 4 es

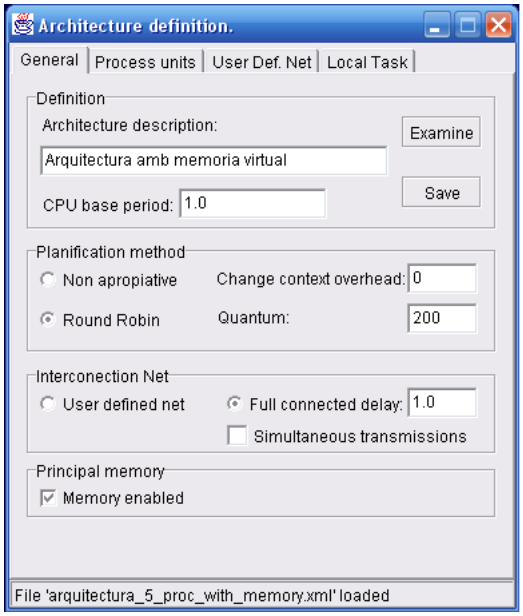

Figura 3.4: Interfície de la definició de l'arquitectura

parla detalladament sobre les tasques locals.

Cal destacar que aquesta especificació es pot fer tant amb XML com des de la interfície.

En la figura 3.6 es mostra la interfície de la definició d'un programa carregat al simulador.

Com es pot observar en l'apartat '**Program description**' es mostra l'estructura del programa en forma d'arbre. A l'arrel hi tenim en el nom del programa. Al següent nivell de l'arbre hi corresponen les tasques amb els seus identificadors i paràmetres de la memòria virtual. Els paràmetres que es poden visualitzar són **virmem**, **rss minimum** i **pagefault interval**. Aquests paràmetres sols apareixen a l'arbre si existeix algun valor específic per aquests atributs en el fitxer XML. A cada tasca podem observar al següent nivell de l'arbre les primitives dels quals està composada.

Cal destacar que es pot afegir més d'un programa paral·lel al simulador. Aquest tema s'anomena Multiprogramació que veurem en el capítol 4. També permet seleccionar el programa per veure la seva estructura en forma d'arbre i fer servir el programa en algunes utilitats del simulador com és el cas del graf TTIG.

3.4 Implementació

En aquesta secció comentarem detalladament sobre quines classes i funcions s'han implementat per que funcioni correctament la gestió de la memòria virtual en el simulador.

En la subsecció 3.4.1 és parla sobre la classe *Memory* i el seu comportament. En la subsecció 3.4.2 es comenta la classe *TaskMemoryAssigned* que serveix per guardar informació de la memòria d'una tasca. En la subsecció 3.4.3 es parla sobre els events

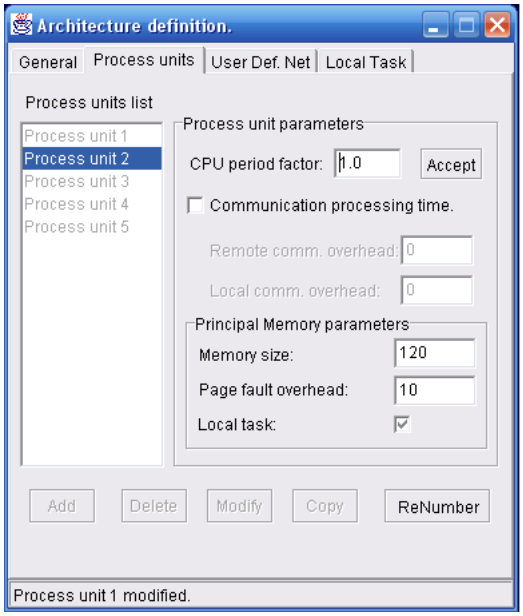

Figura 3.5: Interfície de la definició dels processadors de l'arquitectura

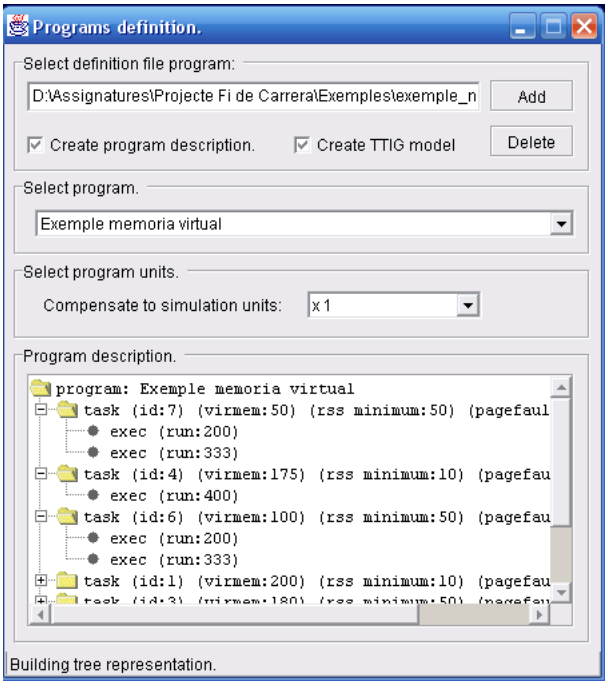

Figura 3.6: Interfície de la definició dels programes paral·lels

Tipus	Nom de la variable	Descripció
Lista	*list_tasks_memory*	Guarda una llista de classes de tipus *TaskMemoryAssigned* que contenen informació individual de la memòria de cada tasca.
int	*id_processor*	Conté el número identificador del processador.
long	*memory_size*	Conté el tamany de la memòria principal del processador.
long	*memory_used*	Conté la quantitat de memòria utilitzada en un instant concret durant la simulació.
int	*id_last_free_task*	Conté el identificador de l'última tasca que s'ha alliberat memòria.

Taula 3.1: Taula d'atributs de la classe *Memory*

que succeixen en l'activat de la memòria virtual. En la subsecció 3.4.4 s'explica com s'ha implementat el mecanisme d'obtenció dels paràmetres de la memòria amb XML. Finalment en la subsecció 3.4.5 es detalla la implementació de l'algorisme de memòria virtual.

3.4.1 La classe *Memory*

La primera classe que cal parlar s'anomena *Memory* i està definida al fitxer *Memory.java*. Aquesta classe representa la memoria principal de cada processador i conté atributs importants com la llista de tasques de memòria que serveix per guardar informació referent a la memòria virtual de cada tasca. Aquesta classe es troba declarada en la classe *SProcessor* definida en el fitxer *Simulator.java* que representa cada processador de la simulació.

Els atributs de la classe *Memory* es mostren en la taula 3.1 amb la seva explicació.

Cal destacar que la classe *TaskMemoryAssigned* definida en el fitxer *TaskMemoryAssigned.java* s'en paralarà amb més detall a la subsecció 3.4.2.

La variable *id_last_free_task* serveix per poder tenir constància de quina ha estat l' última tasca que l'algorisme de reemplaçament de pàgines ha alliberat de memoria per així informar-ho a l'event d'alliberació de memòria.

Les funcions principals de la classe *Memory* es troben descrites a la taula 3.2.

Les tres primeres funcions són els constructors de diferent tipus que s'utilitzen per construir i inicialitzar la classe *Memory* abans de començar la simulació.

Les funcions *set_memory_size()* i *get_memory_size()* es fan servir en qualsevol moment per assignar o obtenir la memòria principal del processador. La funció *get_memory_size()* es fa servir abans de cridar l'algorisme de reemplaçament de pàgines per calcular la memòria lliure que queda lliure juntament amb la funció *get_memory_used()*.

La funció *add_task_memory()* es crida amb un bucle abans de fer la simulació al constructor de la classe *SProcessor* per afegir la informació de la memòria de les tasques mapejades a aquest processador.

Tipus	Nom de la funció	Descripció
public	*Memory()*	Constructor de la classe Memory que inicialitza a zero els contadors i a -1 els identificadors.
public	*Memory(int idproc, long ms)*	Constructor de la classe Memory que se li passa per paràmetres el identificador de process i el tamany de la memòria.
public	*Memory(long ms)*	Constructor de la classe Memory que se li passa per paràmetres el tamany de la memòria.
public void	*set_ memory_ size(long ms)*	Assigna a l'atribut *memory_ size* el tamany de la memòria del paràmetre *ms*
public long	*get_ memory_ size()*	Retorna el tamany de la memòria principal.
public long	*get_ memory_ used()*	Retorna la memòria utilitzada.
public void	*add_ memory_ size(long add)*	Afegeix memòria principal al comptador de memòria corresponent.
public void	*substract_ memory_ size(long sub)*	Resta memòria principal al comptador de memòria corresponent.
public int	*add_ task_ memory(int idproc,int idtsk,long vm,long rss,long rss_ minimum, long pagefault_ interval)*	Afegeix una tasca de memòria amb els seus paràmetres i si existeix llavors la modifica amb els nous paràmetres. La memòria *RSS* es afegida, no substituida. Retorna -1 en cas no hi hagi suficient memòria. Altrament retorna 0.
public int	*set_ rss_ task_ memory (int idtsk,long rss)*	Assigna la memoria *RSS* a la tasca *idtsk*. Retorna 0 si ha pogut assignar la memòria. Altrament retorna -1.
public int	*add_ rss_ task_ memory (int idtsk,long rss)*	Afegeix la memoria rss a la actual resident de la tasca idtsk. Retorna 0 si ha pogut assignar la memòria. Altrament retorna -1.
public TaskMemoryAssigned	*get_ task_ memory (int idtsk)*	Retorna la classe *TaskMemoryAssigned* de la tasca corresponent a partir del seu identificador. Retorna null quan no existeix la tasca.
public long	*allibera_ memoria (long size)*	Allibera la memòria demanada utilitzant el algorisme de reemplaçament de pàgines. Cal destacar que selecciona la tasca amb més memòria resident. Retorna la memòria que queda per alliberar. Assigna el identificador de la tasca seleccionada a *id_ last_ free_ task*.
public int	*get_ id_ last_ free_ task()*	Retorna el identificador de la tasca del qual s'ha alliberat memòria.
public long	*allibera_ memoria_ tasca(int realtask)*	Allibera tota la memòria de la tasca *realtask*.

Taula 3.2: Taula de funcions de la classe *Memory*

Tipus	Atribut	Descripció
private int	*id_task*	El identificador de la tasca que fa servir la memoria definida
private long	*virtual_memory*	La memòria virtual definida per la tasca.
private long	*rss_memory*	La memòria resident actual durant la simulació
private long	*rss_minimum*	El conjunt de memòria que s'assigna en una falta de pàgina.
private long	*pagefault_interval*	L'interval de temps entre faltes de pàgina.
private int	*id_processor*	El Identificador del processador que conté la memoria

Taula 3.3: Taula d'atributs de la classe *TaskMemoryAssigned*

La funció *add_rss_task_memory* s'utilitza a l'algorisme de la memòria virtual per assignar memòria a la tasca el identificador *idtsk* i en cas que retorna -1 llavors es crida l'algorisme de reemplaçament de pàgines.

La funció *get_task_memory()* es crida a la fase inicial de la simulació de la primitiva *Exec* per obtenir informació del rss minimum, del pagefault interval i de la memòria virtual de la tasca que s'està simulant. Cal recordar que el paràmetre de la memòria virtual de la tasca es fa servir per calcular el percentatge de memòria no assignada a la tasca i així determinar a traves del número aleatori si es produïra falta de pàgina o no.

La funció *allibera_memoria()* és l'algorisme de reemplaçament de pàgines. Aquesta s'utilitza durant la simulació de la primitiva *Exec* de la tasca actual quan no hi ha suficient memòria principal.

La funció *get_id_last_free_task()* és crida després d'alliberar memòria principal per saber l'identificador de l'última tasca que s'ha alliberat memòria. Aquesta s'utilitza a l'event d'alliberació de memòria.

Finalment la funció *allibera_memoria_tasca()* s'utilitza al final de la simulació de la tasca actual per allibera tota la seva memòria assignada.

3.4.2 La classe *TaskMemoryAssigned*

La classe *TaskMemoryAssigned* definida en el fitxer *TaskMemoryAssigned.java* serveix per obtenir informació sobre la memòria virtual i resident de cada tasca durant la simulació. Com s'ha dit en la secció anterior, existeix per cada tasca existent una instància d'aquesta classe que es desada en la llista *list_tasks_memory*.

Els seus atributs es mostren a la taula 3.3 amb la seva descripció.

Els paràmetres *rss_minimum* i *pagefault_interval* s'utilitzen al principi de la simulació perquè el simulador sàpigui les característiques del comportament de la memòria virtual de la tasca actual. Els contadors *virtual_memory* i *rss_memory* s'utilitzen durant la simulació, per calcular el percentatge de memòria lliure del processador i d'aquesta manera determinar si es produirà falta de pàgina amb el número aleatori.

En la taula 3.4 es mostren les funcions utilitzades per la classe *TaskMemoryAssigned*.

Tipus	Funció	Descripció
public	*TaskMemoryAssigned()*	Constructor sense paràmetres de la classe *TaskMemoryAssigned*
public	*TaskMemoryAssigned(long virtmem, long rssmem, long rss_min, long pagefault, int idtsk, int idproc)*	Constructor que inicialitza tots els atributs de la classe *TaskMemoryAssigned* amb els valors passats per paràmetres
public int	*set_virtual_memory(* long vm)	Assigna la memòria virtual a la instancia de la classe *TaskMemoryAssigned.*
public void	*add_virtual_memory(long vm)*	Suma la memòria virtual passada per parametres a la tasca
public long	*get_virtual_memory()*	Retorna la memòria virtual de la tasca
public int	*set_rss_memory(long rss)*	Assigna la memòria resident de la tasca. Retorna -1 en cas que no hagi pogut assignar la memòria resident perquè es més gran que la memòria virtual. Altrament retorna 0.
public long	*get_rss_memory()*	Retorna la memòria resident de la tasca
public int	*add_rss_memory(long add_rss)*	Suma a la memòria resident el valor passat per paràmetres.
public long	*substract_rss_memory(long sub_rss)*	Resta a la memòria resident el valor passat per paràmetres.
public void	*set_processor(int proc)*	Assigna el identificador de processador.
public int	*get_processor()*	Retorna el identificador de processador.
public void	*set_task(int idtsk)*	Assigna el identificador de tasca.
public int	*get_task()*	Retorna el identificador de tasca.
public void	*set_rss_minimum(long rss)*	Assigna el rss minimum de la tasca.
public long	*get_rss_minimum()*	Retorna el rss minimum de la tasca.
public void	*set_pagefault_interval(long pagefault)*	Assigna el interval de faltes de pàgina a la tasca.
public long	*get_pagefault_interval()*	Retorna l'interval de faltes de pàgina de la tasca.

Taula 3.4: Taula de funcions de la classe *TaskMemoryAssigned*

3.4.3 Els events de memòria

Durant la simulació d'un programa paral·lel succeixen events que aquests s'han de notificar a l'usuari o enregistrar-los a una llista per després desar-los en un fitxer *log* per tal de ser posteriorment processats per un programa extern al simulador i així poder veure els resultats. Els events estan representats per diferents classes on totes tenen la mateixa estructura i comportament. Els events de memòria succeixen durant la simulació de la primitiva Exec. En total són 3 events de memòria. El primer event està representat per la classe *Trace_Node_FREEMEM* definida al fitxer *Trace_Node_FREEMEM.java*. Aquest event succeix quan s'allibera memòria de la memòria principal del simulador a través de l'algorisme de reemplaçament de pàgines. El missatge que informa l'event és el següent:

```
"Operating system starts freeing of "+free_rss+" units resident memory
at processor "+procId+" to "+MProg.find_original_task_name(TaskId)+
" at moment "+eventTime
```

La variable *free_rss* conté el número d'unitats de memòria que s'han alliberat. El paràmetre *procId* indica el número de processador en que s'ha alliberat la memòria. La funció *MProg.find_original_task_name(TaskId)* és una crida a una funció de la classe *Multiprogram* per saber el nom de la tasca i el nom del programa a qui pertany la tasca. Aquesta funció s'en parlara amb més detall en el capítol 4 que tracta sobre la Multiprogramació. Finalment, la variable *eventTime* informa del temps actual en que ha succeït l'event.

El segon event està representat per la classe *Trace_Node_RSS* definida en el fitxer *Trace_Node_RSS.java*. Aquesta classe informa de l'event assignar memòria resident a una tasca durant la simulació quan s'ha produït una falta de pàgina per la necessitat de carregar memòria del programa a memòria principal. El missatge que informa és el següent:

```
"Operating system starts allocation of "+rss+" units resident memoryy
at processor "+procId+" to "+MProg.find_original_task_name(TaskId)+
" period at "+Start+" to "+End+" Length:"+Length
```

El parametre *rss* indica la memòria a assignar, la variable *procId* indica el número de processador, la funció *MProg.find_original_task_name(TaskId)* informa del nom de la tasca més el nom del programa, la variable *Start* indica el temps inicial que ha succeït aquest event, la variable *End* indica el temps final que ha finalitzat l'event i finalment la variable *Length* indica la duració de l'event que correspon a l'overhead de la falta de pàgina.

El tercer i últim event és representat per la classe *Trace_Node_RELEASEMEMTASK* definida en el fitxer *Trace_Node_RELEASEMEMTASK.java*. Aquest event informa

sobre l'alliberació de tota la memòria de la tasca i succeeix al final de la simulació de la tasca quan acaba aquesta la seva execució.

El missatge que informa és el següent:

```
"Operating system starts deallocating all resident memory ("+free_rss+
" units) to "+MProg.find_original_task_name(TaskId)+" at processor "
+procId+ " at moment "+eventTime;
```

En aquest missatge apareix el paràmetre *free_ rss* que indica la quantitat de memòria que s'ha alliberat de la tasca al finalitzar la execució d'aquesta. Un altre cop apareix la funció *MProg.find_ original_ task_ name(TaskId)* que retorna el nom de la tasca juntament amb el nom del programa per saber de quina tasca es tracta i a quin programa pertany. El número de processador es indicat per la variable *procId* i el temps en que ha succeït l'event es descrit per la variable *eventTime*.

3.4.4 Recull de dades amb XML

El simulador utilitza el llenguatge XML per obtenir les definicions de l'arquitectura del sistema i del programa a traves d'un *parser* escrit amb codi Java que llegeix i analitza el fitxer XML si existeix algun error en la definició i les guarda estructurades amb forma d'arbre per que puguin ser tractades fàcilment.

A continuació en les subseccions 3.4.4.1 i 3.4.4.2 s'explica la implementació de com es recullen les dades de l'arquitectura i del programa relacionat amb la memòria.

3.4.4.1 Recull de dades amb XML de l'arquitectura del simulador

L'arquitectura del sistema està definida amb el llenguatge XML i aquesta es recull a traves d'un *parser* escrit amb Java i guarda la informació del document en forma d'arbre perque puguin ser tractades pel programa.

La classe que s'encarrega de recullir les dades de l'arquitectura, s'anomena *XMLtoArchitecture,* definida en el fitxer *XMLtoArchitecture.java.* Aquesta classe conté tot el codi que analitza si la descripció de l'arquitectura amb XML és correcta.

Per recollir informació sobre els paràmetres de la memòria hem afegit en aquesta classe a la funció *toArchitecture()* el següent codi:

```
else if (Item_Node.getNodeName().toLowerCase().equals("memory_enabled"))
{                  convert_Item_Memory(Item_Node);              }
```

Aquest s'encarrega de examinar si a l'arrel del programa existeix una etiqueta anomenada *memory_ enabled* comprovant si en el node anomenat *Item_ Node* existeix un node amb el nom d'aquesta etiqueta. Recordem que aquesta etiqueta serveix per activar la memoria de l'arquitectura del sistema. Si existeix entra dins de la sentencia condicional i crida a la funció *convert_ Item_ Memory(Item_ Node).* A continuació es mostra el codi d'aquesta funció.

```
private void convert_Item_Memory(Node Item_Node)
{
 String NodeName = Item_Node.getAttributes().item(0).getNodeName().toLowerCase();
 String NodeValue = Item_Node.getAttributes().item(0).getNodeValue().toLowerCase();
 if (Item_Node.getAttributes().getLength()==1 && NodeName.equals("value"))
 {
    if (NodeValue.equals("true"))
      Archi.enable_memory(true);
    else Archi.enable_memory(false);
 }
 else { line = "Error in Memory definition.";
        result=Constants.ERROR_XML_ARCHITECTURE_ATTRIBUTE;
 }
}
```

Aquesta funció s'encarrega d'examinar l'atribut **value** de l'etiqueta **memory_enabled** examinant els atributs i els seus valors del node *Item_Node*. Si el número d'atributs és igual a 1 i l'atribut s'anomena **value** llavors entra dins del cas condicional i activa o desactiva la memòria a l'arquitectura anomenada *Archi* de tipus *Architecture* definida al fitxer *Architecture.java*. En cas que la llargada sigui diferent de 1 o l'atribut sigui diferent de **value** llavors entra al cas condicional que informa d'un error en la definició de l'arquitectura del simulador descrivint el tipus del missatge "Error in Memory definition." i el tipus de resultat *ERROR_XML_ARCHITECTURE_ATTRIBUTE*.

El següent codi que s'ha afegit és el que tracta amb els processadors del sistema. S'ha afegit a la funció ja definida *convert_Item_Processor(Node Item_Node)* el codi que tracta les etiquetes relacionades amb la memòria dels processadors.

A continuació es mostra el codi que s'ha implementat pel seu tractament:

```
else if (NodeName.equals("memory_size") &&
         Item_Node2.getAttributes().getLength()==1 &&
         .equals("value"))
         {
          memorysize = Long.parseLong(Item_Node2.getAttributes()
          .item(0).getNodeValue());
         }
else if (NodeName.equals("pagefault_overhead") &&
         Item_Node2.getAttributes().getLength()==1 &&
         Item_Node2.getAttributes().item(0).getNodeName().toLowerCase()
         .equals("value"))
         {
          overheadpagefault = Long.parseLong(Item_Node2.getAttributes()
          .item(0).getNodeValue());
         }
else if (NodeName.equals("local_task") &&
```

```
Item_Node2.getAttributes().getLength()==1 &&
Item_Node2.getAttributes().item(0).getNodeName().toLowerCase()
.equals("value"))
{
  if (Item_Node2.getAttributes ().item(0).getNodeValue()
  .equals ("true"))
    localtask=true;
  else
    localtask=false;
}
```

Aquest codi s'executa dins d'un bucle i examina en cada cas si el nom del node equival a una de les tres etiquetes que defineixen les característiques de la memòria del processador. Aquestes són: **memory_size**, **pagefault_overhead** i **local_task**. Per cada cas comprova si pertany a una d'aquestes etiquetes i en cas afirmatiu entra en la sentència condicional corresponent i recull en una variable el valor de l'atribut **value** que va lligat amb l'etiqueta.

Llavors més endavant crea una instància del processador de tipus *Processor* definida en el fitxer *Processor.java*, que li afegeix els paràmetres recollits i finalment l'afegeix a la llista que emmagatzema els processadors de l'arquitectura principal del sistema.

```
Processor proc = new Processor(ProcId,Cpu_Period,comm);
if (comm) {
  proc.set_Local_Comm_Overhead(local_com);
  proc.set_Out_Comm_Overhead(out_com);
}
proc.set_Memory_Size (memorysize);
proc.set_Overhead_PageFault (overheadpagefault);
proc.set_local_task (localtask);
Archi.add_processor(proc);
```

3.4.4.2 Recull de dades amb XML del programa paral·lel

La classe que conté el codi que controla el *parser* i tracta amb la definició del programa amb XML s'anomena *XML2Program* i està definida en el fitxer *XML2Program.java*.

En aquesta classe s'ha afegit a la funció principal *toProgram()* el codi que utilitza el *parser* i tracta amb les etiquetes relacionades amb la memòria virtual de la tasca.

A continuació es mostra el codi següent:

```
virmem=0;
rss_minimum=0;
pagefault_interval=0;
num_param=Task_Node.getAttributes().getLength();
for (int index=1;index<num_param;index++)
{
```

```
if (Task_Node.getAttributes().item(index).getNodeName ().
toLowerCase().equals("virmem"))
  virmem =Long.parseLong(Task_Node.getAttributes()
.item(index).getNodeValue());
else if (Task_Node.getAttributes().item(index).getNodeName().
        toLowerCase().equals("rss_minimum")
rss_minimum =Long.parseLong(Task_Node.getAttributes().item(index).
getNodeValue());
else if (Task_Node.getAttributes().item(index).getNodeName().
        toLowerCase().equals("pagefault_interval"))
pagefault_interval =Long.parseLong(Task_Node.getAttributes()
.item(index).getNodeValue());
}
```

La funció principal d'aquest codi és recollir informació dels atributs **virmem**, **rss_minimum** i **pagefault_interval** del node *Task_Node* i ser tractat posteriorment.

El que realitza el codi és semblant als codis comentats en la secció anterior sobre el recull de dades XML de l'arquitectura del sistema i l'estructura també és molt semblant. Existeix un bucle que recorre tots els atributs de l'etiqueta **task** que defineix una tasca del programa. Aquesta etiqueta equivaldria al node *Task_Node*. En cada iteració incrementa la variable *index* que correspon a l'index de l'atribut a ser tractat. Llavors comprova en cada sentencia *if* quin atribut es correspon i dins de la sentencia obté el valor de l'atribut que ho guarda en la variable corresponent.

Posteriorment aquestes variables són guardades a la classe *Task* definida en el fitxer *Task.java*. A continuació es mostra el codi que realitza aquest procediment:

```
tasca = new Task(taskId);
tasca.setMemVirt (virmem);
tasca.set_rss_minimum (rss_minimum);
tasca.set_pagefault_interval (pagefault_interval);
```

3.4.5 Implementació de l'algorisme de memòria virtual

En aquesta secció es comenta detalladament la implementació de l'algorisme de memòria virtual.

L'algorisme de memòria s'ubica en el fitxer *Simulator.java* dins de la classe *SProcessor* que representa els processadors de la simulació. Per cada processador existent hi ha declarat una instància d'aquesta classe. Al constructor se li passa els paràmetres *processor* de tipus *Processor*, *assignment* de tipus *Assig*, *MProg* de tipus *Multiprogram* i *SimulId* de tipus *int*. El primer conté tota la informació del processador, el segon conté la informació sobre l'assignació de tasques d'aquest processador, el tercer conté tota la informació dels diferents programes carregats al simulador[2] i finalment el quart conté l' identificador de la simulació.

[2]En el capítol 4 es detalla la implementació dels Multiprogrames.

Al constructor de la classe *SProcessor* hem afegit el següent codi:

```
memory_enabled=archi.get_enabled_memory ();
if (memory_enabled)
{
  num_aleat=new Random(System.currentTimeMillis()*ProcId);
  Memoria=new Memory(ProcId,processor.get_Memory_Size ());
  memorysize=processor.get_Memory_Size();
  pagefault_overhead=processor.get_Overhead_PageFault();
}
```

La funció principal d'aquest codi es recollir la informació de l'arquitectura del sistema i dels processadors. Com es pot observar primerament crida a una funció de la instància *archi* de tipus *Architecture* per saber en general si la memòria virtual està activa o no. Si està activa llavors entra dins de la sentencia condicional i crea una instància de número aleatori de tipus *Random* passant per paràmetres la llavor que és el temps actual de l'ordinador multiplicat pel número de processador. Després crea una instància de la classe *Memory*, passant per paràmetres l' identificador de processador i el tamany de la memòria del processador actual. A continuació inicialitza la variable *memorysize* amb el tamany de memòria del processador actual i la variable *pagefault_ overhead* amb la sobrecàrrega de faltes de pàgina del processador.

Després dins del constructor de la classe *SProcessor* passa a afegir les tasques assignades a la llista de Preparats i també afegeix la informació de la memòria de les tasques dins de la llista *list_ task_ memory* de la classe *Memory*. A continuació es mostra el seu codi font:

```
for (int i=0;i<numberTaskAssigned;i++)
{ // Identificador de la tarea asignada
  task_Id = (Integer) assignment.Task_Assig.elementAt(i);
  taskId = task_Id.intValue();
  //Afegim les tasques que tenen memoria virtual dins de la classe Memoria
  Task tasca=MultiProg.get_Task(taskId);
  if (tasca!=null) //Sempre s'hauria de complir.
  {
    if (memory_enabled && (tasca.getMemVirt ()>0 || tasca.get_rss_minimum ()>0
        || tasca.get_pagefault_interval ()>0))
      Memoria.add_task_memory (ProcId,taskId,tasca.getMemVirt(),0,
      tasca.get_rss_minimum (),tasca.get_pagefault_interval ());
      // Buscar la tarea en la lista de tareas del programa
      taskQueue = new Task_Queue(taskId, tasca);
      Preparados.insertBack(taskQueue, taskId);
      if (Internal_traces)
        PTrace.new_TaskTrace(taskId);
  }
}
```

La finalitat d'aquest codi és recorrer totes les tasques assignades a traves del vector *assignment.Task_ Assig* i obtenir l' identificador de la tasca assignada. Llavors a través de la funció *get_ Task(taskId)* de la instància *MultiProg* obté la tasca corresponent amb tota la seva informació. Llavors si la tasca és diferent de null comprova que la memòria estigui activa i a la vegada la memòria virtual de la tasca o el rss minimum de la tasca o el pagefault interval de la tasca sigui més gran que zero. En cas afirmatiu afegeix a la classe *Memory* una nova tasca de memòria amb els seus paràmetres de memòria. Després crea una instància de tipus *Task_ Queue* passant per paràmetres la tasca de tipus *Task* amb tota la informació sobre la simulació de la tasca i li insereix a la llista de *Preparats*.

Sobre la memòria virtual en el constructor de la classe *SProcessor* no hi ha res més a comentar. Tot seguit passem a detallar l'algorisme dins de la funció *advance()* de la classe *SProcessor*. El que realitza aquesta funció és avançar la simulació del processador fins al màxim permés. Cal recordar que el simulador es asíncron per tant avança els processadors a diferents temps fins trobar-se en un punt de bloqueig. Llavors intervé la unitat de control i prén una o varies desicions per desbloquejar la situació en que es troben els processadors.

A continuació es mostra el codi font de la primitiva Exec que tal com s'ha dit anteriorment és a on succeïx tots els events de la memòria:

```
do {
    //Para todas las primitivas que se puedan ejecutar de esta tarea
    switch (primitives_node.getId()) {

    case Constants.EXEC:
      pexec = (P_Exec) primitives_node.getNodo();
      //Aqui comena el codi del tractament de memoria
      if (memory_enabled)
      {
       cicles_EXEC= (long) (pexec.get_TCicle ()*CPU_Period);
       time_acumulat=0;
       time_resta=0;
       if (planification == Constants.NON_APROPIATIVE )
         time_resta=cicles_EXEC;
        //Planificacio round robin
       else if (planification == Constants.ROUND_ROBIN && N_Tasks_Assigned>1)
       //Sols hi ha una tasca pertant no es dur a terme la planificacio
         Round Robin
       else if ( planification == Constants.ROUND_ROBIN && N_Tasks_Assigned==1)
       TaskMemoryAssigned task_mem=Memoria.get_task_memory(realTask);
       if (task_mem!=null)
       {
          rssminimum=task_mem.get_rss_minimum ();
          pagefault_interval=task_mem.get_pagefault_interval ();
       }
```

```
else
{
  rssminimum=0;
  if (planification == Constants.NON_APROPIATIVE )
    pagefault_interval=cicles_EXEC;
  else if (planification == Constants.ROUND_ROBIN
           && N_Tasks_Assigned>1)
    pagefault_interval=Quantum2;
  else if ( planification == Constants.ROUND_ROBIN
           && N_Tasks_Assigned==1)
    pagefault_interval=cicles_EXEC;
}
timeRSS=pagefault_overhead;
```

El que es mostra en aquest codi és la fase inicial de la primitiva Exec, quan s'inicialitzen els paràmetres de la memòria. Tal com es pot observar, existeix un *switch* que comprova el tipus de primitiva i en cas que sigui de tipus *Constants.EXEC* entra en el codi de la simulació d'aquesta primitiva.

Primerament, recull la informació de la primitiva del node *primitives_ node* que està desat en la llista *tasca.Primitives*. Llavors si la memòria del simulador està activa passa a realitzar la simulació de la primitva juntament amb la gestió de la memòria entrant en la sentencia del *if*. Pel contrari, en cas negatiu, passaria a realitzar la simulació de la primitiva *Exec* sense la gestió de la memòria.

En l'algorisme de memòria primer calcula els cicles d'execució de la primitiva multiplicant la duració de la primitiva pel període de CPU. Llavors inicialitza les variables *time_ acumulat* i *time_ resta*. Aquestes són per saber el temps que porta acumulat durant la simulació d'aquesta primitiva i el temps que falta per acabar la simulació d'aquesta.

A continuació distingeix tres casos que *time_ resta* pendrà diferents valors. El primer cas si es planificació No Apropiativa el temps que queda per la simulació és igual als cicles de la primitiva *Exec*. En cas que sigui planificació Round Robin el temps que queda restant és igual al valor del *quantum* i en cas que sigui planificació Round Robin però només hi ha una tasca mapejada el temps passa a ser igual el valor dels cicles d'execució de la primitiva.

Llavors obté una instància de la classe *TaskMemoryAssigned* per saber la informació del comportament de la tasca relacionat amb la memòria.

Sols s'afegeix tasques a la llista de la classe *Memory* si aquestes tenen algun comportament referent a la memòria. Si aquesta és diferent de *null* obté els paràmetres **rss minimum** i **pagefault interval**. Pel contrari, en cas negatiu inicialitza els valors de **rss minimum** i **pagefault interval** amb els valors 0 pel primer paràmetre i *cicles_ EXEC* o *Quantum2* pel segon si la planificació és No Apropiativa o Round Robin respectivament. La intenció es anular les faltes de pàgina perquè aquesta tasca no té informació sobre les fallides i per tant s'ha de posar el valor de **pagefault interval** al màxim possible temps perquè no s'en produeixi cap.

Finalment inicialitza la variable *timeRSS* amb el valor del temps de sobrecarrega de

les faltes de pàgina.

A continuació es mostra la segona fase de l'algorisme que correspon al principi de la simulació de la primitiva Exec amb memòria quan entra en el bucle principal de l'algorisme i determina si es produeix falta de pagina o no.:

```
while (time_resta>0 && (pexec.get_TCicle ()*CPU_Period)>0)
{
   int pr = Math.abs(num_aleat.nextInt())%100;
   if (task_mem!=null)
   {
     if (task_mem.get_virtual_memory ()>0)
     {
       if (pr<(100-(task_mem.get_rss_memory ()*100/
           task_mem.get_virtual_memory ())))                {
         totalPageFault++;
```

Entra al bucle principal i itera mentre el temps que falta per acabar és més gran que zero o els cicles de la primitiva Exec és més gran que zero. Aquests es decrementen al final del bucle i quan un d'ells arriba a zero llavors para el bucle. Per cada iteració, tal com s'ha comentat en la secció 3.2 avança l'interval de faltes de pàgina, determinant si es produeix falta de pàgina i avançant la simulació de la primitiva.

Primer obté el valor absolut d'un nombre aleatori mòdul 100 utilitzant la llibreria matematica *Math* i el generador de nombres aleatori *Random* comentat anteriorment. Llavors passa a determinar si es produirà falta de pàgina si la tasca és diferent de null i la memòria virtual de la tasca és mes gran que zero. Per determinar-ho calcula el percentatge de memòria no assignada de la tasca i ho compara amb el valor aleatori. En cas afirmatiu es produeix una falta de pàgina, gestiona la falta i avança la simulació. En cas negatiu no es produeix falta i avança la simulació.

Finalment incrementa el contador de faltes de pàgina per mostrar els resultats al final de la simulació.

A continuació es mostra el codi de la tercera etapa de l'algorisme:

```
long rss_memory;
if ((task_mem.get_virtual_memory ()-task_mem.get_rss_memory())<=rssminimum)
else
  rss_memory=rssminimum;
if (Memoria.add_rss_task_memory (realTask,rss_memory)==-1)
{
  long free_rss=rss_memory-(Memoria.get_memory_size ()-
                Memoria.get_memory_used ());
           long free_rss2;
  while(free_rss>0)
  {
```

```
free_rss2=Memoria.allibera_memoria(free_rss);
                            (),ProcId,free_rss-free_rss2,MultiProg);
if (logSim) {
  file_log.writeBytes("Proc"+ProcId+"["+actualTime+"]: "+
  tnFREEMEM.toString()+"\n");
}
if (logCon) {
  System.out.println("Proc"+ProcId+"["+actualTime+"]: "+
  tnFREEMEM.toString());
}
if (External_traces)
{
  temporal=tnFREEMEM.toString()+"\n";
  output.writeBytes(temporal);
  temporal=tnFREEMEM.toString_XLS();
  output_XLS.writeBytes(temporal);
}
if (Internal_traces)
{
  PTrace.insert_TaskEventTrace(tnFREEMEM.get_TaskId
  (),tnFREEMEM.ID,tnFREEMEM,actualTime);
  PTrace.insert_EventTrace(tnFREEMEM.get_TaskId
  (),tnFREEMEM.ID,tnFREEMEM,actualTime);
}
else {
       tnFREEMEM=null;
     }
     free_rss=free_rss2;
}
}
```

Aquest codi és el que va a continuació de la sentencia condicional que determina si es produeix falta de pàgina o no. Primerament inicialitza la variable *rss_memory* amb la quantitat de memòria que ha d'assignar a memòria principal, és a dir el **rss minimum** si aquest és inferior a la quantitat de memòria no assignada a memòria principal de la tasca actual o sino la resta de memòria que falta per assignar. Després crida a la funció de la classe *Memory add_rss_task_memory (realTask,rss_memory)* per tal d'afegir a la tasca amb identificador *realTask* la nova memòria **rss memory**. Aquesta funció retorna un valor, si retorna -1 llavors no ha pogut assignar la **rss memory** a la memòria principal i pertant haurà de cridar a l'algorisme de reemplaçament de pàgines. A continuació entra dins de la sentencia condicional i inicialitza la variable *free_rss* amb el valor que ha d'alliberar que és *rss_memory-(Memoria.get_memory_size ()-Memoria.get_memory_used ())*. Llavors entra al bucle que no finalitzarà fins que no s'hagi alliberat tota la memòria necessària. Per alliberar memòria crida a la funció de la

classe *Memory allibera_ memoria(free_ rss)* que se li passa per paràmetres la quantitat de memòria a alliberar i retorna la quantitat de memòria que no ha pogut alliberar. Aquesta la fa servir per finalitzar el bucle. Després declara la classe *Trace_ Node_ FREEMEM* que representa l'event d'alliberació de memòria passant-li per paràmetre, el temps actual, l'identificador de l'ultima tasca que ha seleccionat l'algorisme de memòria, l' identificador de processador, la memòria que ha alliberat que és *free_ rss-free_ rss2* i l' instància de la classe *MultiProgram* que conté tots els programes carregats en el simulador, per saber l' identificador original de la tasca juntament amb el nom del programa. Existeixen dos booleans, *logSim* i *logCon* que determinen si es guardarà en un fitxer els events produïts durant la simulació o s'imprimiran per la cònsola. Al comprobar el primer booleà, en cas de verdader, guarda al fitxer lògic *file_ log* a través de la funció *writeBytes* el missatge de l'event d'alliberació de memòria. En el segon booleà, en cas de verdader, escriu per pantalla a través de la funció *System.out.println()* . Si les traces externes estan activades escriu a dos fitxers el missatge de l'event succeït i una codificació amb XLS del missatge per ser tractat posteriorment per un programa extern. Finalment si les traces internes estan activades enregistra els events succeïts en aquest processador a l'instància *PTrace* de la classe *Processor_ Trace* definida al fitxer *Processor_ Trace.java*.

A continuació es mostra el codi font de la següent part de l'algorisme:

```
Trace_Node_RSS tnRSS;
tnRSS = new Trace_Node_RSS(actualTime,actualTime+timeRSS,
                           timeRSS,realTask,ProcId,rss_memory,MultiProg);
if (logSim) {
              file_log.writeBytes("Proc"+ProcId+"["+actualTime+"]: "+
              tnRSS.toString()+"\n");
            }
if (logCon) {
              System.out.println("Proc"+ProcId+"["+actualTime+"]: "+
              tnRSS.toString());
            }
if (External_traces)
{
  temporal=tnRSS.toString()+"\n";
  temporal=tnRSS.toString_XLS();
  output_XLS.writeBytes(temporal);
}
if (Internal_traces)
{
   PTrace.insert_TaskEventTrace(realTask,tnRSS.ID,tnRSS,actualTime);
   PTrace.insert_EventTrace(realTask,tnRSS.ID,tnRSS,actualTime);
} else tnRSS=null;
  actualTime+=timeRSS;
  Usage+=timeRSS;
  totalRSS+=timeRSS;
```

```
if (planification==Constants.ROUND_ROBIN && N_Tasks_Assigned>1)
{
  time_resta-=timeRSS;
  time_acumulat+=timeRSS;
}
} //Fi del if (pr<...)
} //Fi del if tasca.get_virtual_memory()>0
} //fi del if (task_mem!=null)
```

En aquesta part de codi es mostra com declara l'event d'assignar memòria a la tasca actual i realitza el mateix procediment de guardar el missatge en un fitxer, imprimir a la còmsola i enregistrar les traces externes i internes. Més endavant ja no es mostrà el codi de registre de les traces perquè sempre és el mateix. A més es mostra com incrementa el temps actual de la simulació *actualTime* amb l' overhead de les faltes de pàgina, l'us de la CPU descrit per la variable *Usage* i el temps total de paginació descrit per la variable *totalRSS*.

Finalment hi ha una senténcia condicional que comprova si la planificació es Round Robin i el número de tasques assignades es més gran que 1 llavors actualitza la variable *time_resta* a traves de decrementar l' overhead de faltes de pagines i actualitza la variable *time_acumulat* a traves d'incrementar el temps d'overhead de faltes de pàgina. Nomes es té en compte l' overhead de faltes de pàgines si s'ha assignat un *quantum* de temps per la simulació de la tasca. Per tant se li ha de restar a aquest *quantum* que conté *time_resta* l' overhead. En canvi si és una planificació No Apropiativa o sols hi ha una tasca assignada el temps assignat a la tasca actual pel processador és infinit i per tant ha d'acabar tot el seu cicle i no s'ha de penalitzar amb l' overhead de faltes de pàgina.

```
if (time_resta>=pagefault_interval && (pexec.get_TCicle ()*
    CPU_Period)>=pagefault_interval)
  timeEXEC_RSS=pagefault_interval;
else
{
  if (planification == Constants.ROUND_ROBIN)
  {
    if ((Quantum2-time_acumulat)<=(pexec.get_TCicle()*
         CPU_Period) && N_Tasks_Assigned>1)
       time_resta=Quantum2-time_acumulat;
    else
       time_resta=pexec.get_TCicle()*CPU_Period;
  }

  timeEXEC_RSS=time_resta;
}
```

En aquest codi es mostra inicialment com actualitza els contadors de temps. Si el valor de *time_resta* és més gran o igual que l'interval de faltes de pàgina i els cicles de la

primitiva Exec són més grans o iguals que l'interval de faltes de pàgines, el contador *timeEXEC_RSS* que conté el número de cicles que a continuació es simularan de la primitiva equivaldrà a *pagefault_interval*. És a dir s'avançarà el temps de la simulació *pagefault_interval* cicles perquè durant aquest temps no es produirà cap event.

Si és inferior a *pagefault_interval,* els dos contadors utilitzats per finalitzar el bucle principal de l'algorisme llavors es distingeix dos casos un si és planificació Round Robin i l'altre si és planificació No Apropiativa. A la primera planificació comprova dins d'una sentencia condicional si el temps real que queda per finalitzar la simulació de la tasca que és *Quantum2-time_acumulat* és més petit que els cicles de la primitiva Exec que queden per simular *(pexec.get_TCicle()*CPU_Period)* i el número de tasques mapejades és més gran que 1 llavors el valor de *time_resta* valdrà el valor mínim que serà el que finalitzarà el bucle. En canvi si és superior, el valor de *time_resta* valdrà els cicles restants de la primitiva Exec. Si hi ha una sola tasca el valor de *time_resta* sempre ha de ser els cicles restants de la primitiva Exec. En la planificació No apropiativa *time_resta* valdrà els cicles totals de la primitiva menys el temps acumulat de les faltes de pàgina. En aquest cas restem el valor de time_acumulat perquè al codi de simulació de la primitiva *Exec* de la planificació No Apropiativa no conté els cicles que falten per la simulació sinó els totals. Finalment assigna a *timeEXEC_RSS* el valor de *time_resta* que són els cicles de l'ultima iteració que es simularan.

A continuació es mostra el codi de la simulació de la primitva Exec en la planificació No Apropiativa.

```
 if (planification == Constants.NON_APROPIATIVE) {
  Trace_Node_EXEC tnExec = new Trace_Node_EXEC(actualTime,
                           timeEXEC_RSS,
                           realTask,ProcId,MultiProg);
                         (.....)
actualTime+=timeEXEC_RSS;
Usage+=timeEXEC_RSS;
ProcTimes[simulId] = actualTime;
time_resta-=timeEXEC_RSS;
time_acumulat+=timeEXEC_RSS;
if (time_resta<=0 || (pexec.get_TCicle ()*CPU_Period)<=0) //Ultima iteracio
  Repository.insertBack(primitives_node);
  tasca.Primitives.removeFirst();
  slots++;
}
```

En la simulació de la primitiva *Exec* es crea una instància de la classe *Trace_Node_EXEC* per notificar a l'event de simulació d'aquesta primitiva i realitza el tractament d'aquest event de la mateixa forma que els altres events.

Llavors passa a actualitzar els contadors de la simulació, el valor de *actualTime* avança *timeEXEC*_RSS cicles; el valor *Usage* que també li suma *timeEXEC_RSS* cicles; la variable *ProcTimes[simulId]* que guarda els diferents temps en que han succeït events

l'inicialitza amb el valor de *actualTime*; actualitza el valor de *time_ resta* decrementant el valor de *timeEXEC_ RSS* cicles i el valor de *time_ acumulat* li suma els mateixos cicles.

Llavors finalment determina si és l'última iteració comprovant si *time_ resta* és igual a zero o inferior o els cicles restants de la primitiva *Exec* són iguals o inferiors a zero. En cas afirmatiu insereix la primitiva a la llista de repositoris *Repository*, elimina la primitiva de la tasca i incrementa el número de slots de la simulació que serveixen per fer avançar la barra de progrés de la simulació.

A continuació es mostra el codi de la simulació d'aquesta primitiva a la planificació Round Robin:

```
else {
 temp_exec = pexec.Run_X((long)timeEXEC_RSS);
 if (temp_exec>=0) {
  Trace_Node_EXEC tnExec = new Trace_Node_EXEC(actualTime,
                                actualTime+timeEXEC_RSS,
                                timeEXEC_RSS,
                                realTask,ProcId,MultiProg);
                   (....)
  actualTime+=timeEXEC_RSS;
  Usage+=timeEXEC_RSS;
  ProcTimes[simulId] = actualTime;
  time_resta-=timeEXEC_RSS;
  time_acumulat+=timeEXEC_RSS;
  if (temp_exec==0)
  {
    if (time_resta<=0 || (pexec.get_TCicle ()*
    CPU_Period)<=0) //Ultima iteracio
    {
      Repository.insertBack(primitives_node);
      tasca.Primitives.removeFirst();
      slots++;
    }
  }
  if (time_resta<=0 || (pexec.get_TCicle ()*
  CPU_Period)<=0) //Ultima iteracio
  {
    Preparados.removeFirst();
    Preparados.insertBack(tasca,realTask);
    fin = true;
  }
} else {
  Trace_Node_EXEC tnExec = new Trace_Node_EXEC(actualTime,
(....)
  actualTime+=timeEXEC_RSS;
```

```
  Usage+=timeEXEC_RSS;
  time_resta-=timeEXEC_RSS;
  time_acumulat+=timeEXEC_RSS;
  ProcTimes[simulId] = actualTime;
  if (time_resta<=0 || (pexec.get_TCicle ()*CPU_Period)<=0)
  {
   Repository.insertBack(primitives_node);
   tasca.Primitives.removeFirst();
   slots++;
  }
 }
}
 if (time_resta<=0 || (pexec.get_TCicle ()*CPU_Period)<=0)
 if (actualTime>maxTime)
 {  finSimul = true; }
} //End del bucle tractament de memoria
break;
```

El codi de simulació de la primitiva Exec en la planificació Round Robin és bastant semblant al de la planificació No Apropiativa. Primer crida a la funció *Run_X()* de la primitiva per avançar els cicles *timeEXEC_RSS* a la primitiva. Llavors si *temp_exec* és més gran o igual que zero vol dir que no acaba l' execució al quantum i entra al primer cas condicional. En aquest cas primerament declara una instància de la classe *Trace_Node_EXEC*, passant-li els paràmetres corresponents. Després realitza el mateix procediment que en els anteriors events per enregistrar i mostrar per pantalla el missatge de l'event. Després actualitza els contadors de la mateixa manera que en el codi anterior. Llavors entra a un cas condicional que comprova si el valor de la variable *temp_exec* és igual a zero, llavors si és l'última iteració insereix la primitiva a la llista de repositoris i incrementa els slots d'avanç de la simulació.

Després fora de la sentencia condicional comprova de nou si és l'última iteració per tal d'eliminar-la de la primera posició de la cua de *Preparats* i posa el booleà *fin* a cert que informa que la tasca ha a acabat la simulació.

Altrament si la sentencia condicional que compara els cicles retornats per la funció *Run_X()* són inferiors a zero vol dir que acaba en el quantum i pertant entra en el cas *else*. En aquest cas enregistra l' event d'execució de la primitiva, actualitza contadors, i si és l' última iteració elimina la primitiva i la insereix a la llista de repositoris.

A la part final de l'algorisme comprova si el temps actual de la simulació és més gran que el temps màxim de la simulació llavors posa el booleà *finSimul* a cert per indicar que finalitza la simulació.

3.5 Experimentació

En aquesta secció es comenta detalladament els resultats de la simulació d'un programa paral·lel amb el simulador ESPPADA. L'objectiu d'aquesta Experimentació es veure i

```
<?xml version="1.0"?>
<!-- Architecture definition created by ESPPADA tool v1.0 -->
<architecture name="Arquitectura 1 processadors i memoria">
 <cpu_base value="1.0"/>
 <context_change value="0"/>
 <memory_enabled value="true"/>
 <local_task for="0" exec="0" wait="0" terminate="false"
 virtual_memory="0" rss_minimum="0"   pagefault_interval="0"/>
 <net type="full_connect">
  <delay value="1.0"/>
 </net>
 <planification method="round_robin" quantum="200"/>
 <!-- Process units definition -->
 <processor id="1">
  <cpu_period value="1.0"/>
  <memory_size value="256"/>
  <pagefault_overhead value="50"/>
  <local_task value="false"/>
 </processor>
</architecture>
```

Figura 3.7: Definició d'arquitectura amb suport de la memòria virtual per l' Experimentació.

entendre els resultats de la simulació d'un programa paral·lel amb memòria virtual.

3.5.1 Simulació d'un programa paral·lel amb memòria virtual

L'arquitectura que s'ha definit per a la simulació d'aquesta Experimentació és la que es mostra en la figura 3.7 amb format XML. Si l'observem podem veure que el paràmetre de la memòria virtual està actiu, que la planificació utilitzada és Round Robin amb un quantum de 200 i que sols hi ha definit un sol processador amb identificador igual a 1, amb una memòria principal de 256 unitats de memòria, una sobrecàrrega de faltes de pàgina de 50 unitats de temps i no existeix mapejada cap tasca local.

El programa paral·lel que s'ha simulat és el que es mostra en la figura 3.8 amb format XML.

Com es pot observar el programa s'anomena "Programa amb memoria virtual" hi té 3 tasques definides. La primera tasca té identificador igual a 1 amb una memòria virtual de 650 unitats de memòria, un **rss minimum** de 10 unitats de memòria i un **pagefault interval** de 20 unitats de temps. A més està composada per una sola primitiva de tipus *Exec* de 300 cicles de rellotge.

La segona tasca té identificador igual a 2, una memòria virtual de 250 unitats, un **rss minimum** de 20 unitats, un **pagefault interval** de 150 unitats de temps i està

```
<?xml version="1.0"?>
<program name="Programa amb memoria virtual">
 <task id="1" virmem="650" rss_minimum="10" pagefault_interval="20">
  <exec run="300"/>
 </task>
 <task id="2" virmem="250" rss_minimum="20" pagefault_interval="150">
  <exec run="250"/>
 </task>
 <task id="3" virmem="500" rss_minimum="35" pagefault_interval="50">
  <exec run="200"/>
 </task>
</program>
```

Figura 3.8: Definició del programa paral·lel amb suport a la memòria virtual per la Experimentació.

composada per una sola primitiva de tipus *Exec* de 250 cicles de rellotge.

Finalment la tercera i última tasca té identificador igual a 3, una memòria virtual de 500, un **rss minimum** de 35 unitats, un **pagefault interval** de 50 i una sola primitiva de tipus *Exec* de 200 cicles.

Per realitzar la simulació s'ha fet un mapping manual assignant les tres tasques a l'únic processador definit en l'arquitectura. Llavors iniciem la simulació.

El resultat de les traces de la simulació es el que es mostra en la figura 3.9.

Tal com es pot observar primer es produeix una falta de pàgina en la tasca 1 i el sistema operatiu assigna 10 unitats de memòria resident a aquesta tasca tal com haviem especificat en la definició de la tasca concretament en el **rss minimum**. La sobrecàrrega de temps que es produeix és de 50 unitats de temps tal com s'havia especificat en l'arquitectura. Llavors la tasca empren l' execució durant 20 cicles (**pagefault interval**) fins que es torna a produïr una falta de pàgina. I aixins va fent fins que s'agota el quantum assignat a aquesta tasca. Aquesta part de la simulació correspon al bucle principal de l'algorisme de memòria virtual que en cada iteració va determinant si es produeix una falta de pàgina i va simulant la primitiva *Exec* fins que no es produeixi de nou una nova falta de pàgina. En total s'assignen 30 unitats de memòria per aquesta tasca 1 durant aquest *Quantum* de temps.

Llavors, el simulador assigna a la tasca 2 la CPU i torna a mostrar-se la sèrie d'events produïts per les faltes de pàgines i l'execució de la primitiva *Exec*. En aquest cas a cada falta de pàgina assigna 20 unitats de memòria i una sobrecarrega de 50 unitats de temps durant el periode de 200 fins 250, el mateix que l'anterior tasca perquè és el mateix processador. Com que l'interval entre faltes de pàgina en aquesta tasca es de 150 unitats de temps llavors a la primera execució només es produeïx una sola falta de pàgina perquè s'agota el *Quantum* a la primera iteració del bucle principal de l'algorisme ja que a més el consumeix la sobrecarrega de la falta de pàgina. En total s'assignen 20 unitats de memòria per aquesta tasca 2 durant aquest *Quantum* de temps.

Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 0.0 to 50.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 50.0 to 70.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 70.0 to 120.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 120.0 to 140.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 140.0 to 190.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 190.0 to 200.0 Length:10.0
Operating system starts allocation of 20 units resident memory at processor 1 to Task: 2 Program: Programa amb memoria virtual period at 200.0 to 250.0 Length:50.0
Task: 2 Program: Programa amb memoria virtual starts execution period at 250.0 to 400.0 Length:150.0
Operating system starts allocation of 35 units resident memory at processor 1 to Task: 3 Program: Programa amb memoria virtual period at 400.0 to 450.0 Length:50.0
Task: 3 Program: Programa amb memoria virtual starts execution period at 450.0 to 500.0 Length:50.0
Operating system starts allocation of 35 units resident memory at processor 1 to Task: 3 Program: Programa amb memoria virtual period at 500.0 to 550.0 Length:50.0
Task: 3 Program: Programa amb memoria virtual starts execution period at 550.0 to 600.0 Length:50.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 600.0 to 650.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 650.0 to 670.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 670.0 to 720.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 720.0 to 740.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 740.0 to 790.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 790.0 to 800.0 Length:10.0
Operating system starts allocation of 20 units resident memory at processor 1 to Task: 2 Program: Programa amb memoria virtual period at 800.0 to 850.0 Length:50.0
Task: 2 Program: Programa amb memoria virtual starts execution period at 850.0 to 950.0 Length:100.0
Task: 2 Program: Programa amb memoria virtual finished at moment 950.0
Operating system starts deallocating all resident memory (40 units) to Task: 2 Program: Programa amb memoria virtual at processor 1 at moment 950.0
Operating system starts allocation of 35 units resident memory at processor 1 to Task: 3 Program: Programa amb memoria virtual period at 950.0 to 1000.0 Length:50.0
Task: 3 Program: Programa amb memoria virtual starts execution period at 1000.0 to 1050.0 Length:50.0
Operating system starts allocation of 35 units resident memory at processor 1 to Task: 3 Program: Programa amb memoria virtual period at 1050.0 to 1100.0 Length:50.0
Task: 3 Program: Programa amb memoria virtual starts execution period at 1100.0 to 1150.0 Length:50.0
Task: 3 Program: Programa amb memoria virtual finished at moment 1150.0
Operating system starts deallocating all resident memory (140 units) to Task: 3 Program: Programa amb memoria virtual at processor 1 at moment 1150.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 1150.0 to 1170.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 1170.0 to 1220.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 1220.0 to 1240.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 1240.0 to 1290.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 1290.0 to 1310.0 Length:20.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 1310.0 to 1330.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 1330.0 to 1380.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 1380.0 to 1400.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 1400.0 to 1450.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 1450.0 to 1470.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 1470.0 to 1520.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 1520.0 to 1540.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 1540.0 to 1590.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 1590.0 to 1610.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 1610.0 to 1660.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 1660.0 to 1680.0 Length:20.0
Operating system starts allocation of 10 units resident memory at processor 1 to Task: 1 Program: Programa amb memoria virtual period at 1680.0 to 1730.0 Length:50.0
Task: 1 Program: Programa amb memoria virtual starts execution period at 1730.0 to 1750.0 Length:20.0
Task: 1 Program: Programa amb memoria virtual finished at moment 1750.0
Operating system starts deallocating all resident memory (140 units) to Task: 1 Program: Programa amb memoria virtual at processor 1 at moment 1750.0

Figura 3.9: Resultat dels missatges del events de la simulació per la Experimentació.

Llavors entra a la CPU la tasca 3 i torna a realitzar el mateix procediment. Inicialment el sistema operatiu assigna 35 unitats de memòria resident a la tasca durant el periode de 500 a 550 unitats de temps. En total es produeïxen dos faltes de pàgina durant el *Quantum* assignat a aquesta tasca. En total s'assignen 70 unitats de memòria per aquesta tasca 3 durant aquest *Quantum* de temps.

Llavors torna a pendre la CPU la tasca 1 al moment 600, en total es produeixen 3 faltes de pàgina i a conseqüència assigna 30 unitats de memòria més. En total hi han assignats 60 unitats de memòria per la tasca 1.

A continuació es planifica la tasca 2 a la CPU i aquesta es la primera en finalitzar la seva execució al moment 950 perquè en general es produeixen poques faltes de pàgina degut a que té el paràmetre **pagefault interval** alt i no agoten la sobrecàrrega el *Quantum* assignat a aquesta tasca. Podem observar com es mostra l'event d'alliberació de tota la memòria de la tasca (40 unitats). En total es produeixen 2 faltes de pàgina.

La següent tasca a finalitzar es la número 3 al moment 1150. En total s'han assignat 140 unitats de memòria que són desallotjades al acabar la seva execució. En aquesta tasca es produeixen un total de 4 faltes de pàgina.

Llavors només queda la tasca 1 al processador que finalitza al moment 1750 desallotjant tota la seva memòria que són 140 unitats. En aquesta tasca es produeixen un total de 14 faltes de pàgina.

En aquesta simulació no ha intervengut l'algorisme de reemplaçament de pàgines perque hi havia suficient memòria per les tres tasques mapejades a l'únic processador del sistema.

A la figura 3.10 es mostra la gràfica dels resultats de la simulació.

El temps total per la simulació del programa al processador 1 és de 1750 unitats de temps que es el que es mostra en la primera barra. La segona barra que és la del *Workload Mapped* indica la carga de treball mapejada a la CPU que en total és de 750 unitats perquè no conta la carrega del sistema operatiu per gestionar les faltes de pàgina. La tercera barra, la del *Working Time* indica el temps d'ús de CPU que és el mateix que el de la primera barra perquè les tasques nomes tenien la primitiva Exec que utilitza al complet la CPU i el sistema operatiu utilitza també al complet la CPU per carregar la pàgina a memòria. L'última barra és la del *Waiting Time* que indica el temps d'espera per les comunicacions que és nul perquè no hi han comunicacions en la simulació d'aquest programa.

Per últim a la figura 3.11 és mostra els resultats de la simulació. En el primer apartat *Simulation Data* tenim el nom de l'arquitectura, la llista de programes que han intervengut en la simulació, el *mapping* on es mostra totes les tasques mapejades en el processador 1, el *speed up* que indica la quantitat de paral·lelisme del programa, el temps de simulació que és de 1750 unitats i finalment el temp real que ha necessitat la CPU real per simular el programa que és de 0 mil·lisegons.

Llavors al segon apartat mostra els resultats de la simulació del Processador. Els quatre primeres línes ja les hem comentat anteriorment i les dues últimes indiquen l' overhead total produït per les faltes de pàgines que és de 1000 unitats i el total de faltes de pàgines que són 20 en aquest processador.

En aquesta apartat de l'experimentació hem pogut veure la simulació d'un programa

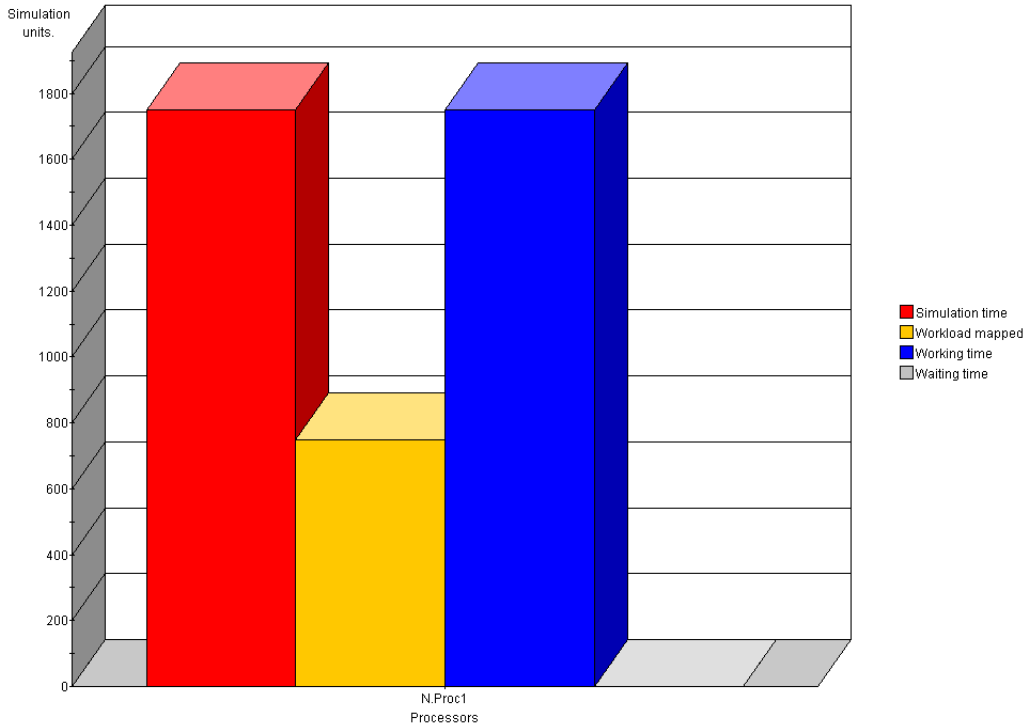

Figura 3.10: Gràfica dels resultats de la simulació per l'Experimentació.

```
---------------------------
 Simulation Data.
---------------------------

        Architecture: Arquitectura 1 processadors i memoria

        Programs List:
                Programa amb memoria virtual

        Mapping:

                Program Name: Programa amb memoria virtual:
                        T1: P1
                        T2: P1
                        T3: P1

        SpeedUp: 0.42857143

        Time: 1750.0

        Time need for simulation process: 0 ms.

---------------------------------
Processor 1:
        Simulation time: 1750.0
        Processing time: 1750.0
        Workload mapped: 750.0
        Waiting time: 0.0

        Overhead page fault time: 1000.0
        Total page faults: 20
```

Figura 3.11: Resultats de la simulació del programa paral·lel per l'Experimentació.

```
<?xml version="1.0"?>
<program name="Programa amb memoria virtual">
 <task id="1" virmem="650" rss_minimum="10" pagefault_interval="50">
  <exec run="7300"/>
 </task>
 <task id="2" virmem="550" rss_minimum="3" pagefault_interval="35">
  <exec run="6250"/>
 </task>
 <task id="3" virmem="700" rss_minimum="5" pagefault_interval="25">
  <exec run="5200"/>
 </task>
</program>
```

Figura 3.12: Programa paral·lel utilitzat pel segon apartat de l'experimentació amb memòria virtual.

paral·lel sense que intervingues l'algorisme de reemplaçament de pàgines perquè l'únic processador del sistema tenia suficient memòria principal per gestionar la memòria de les tres tasques. En el següent apartat es mostra un exemple de simulació diferent on l' activitat de la memòria virtual és més agressiva i es produeix reemplaçaments de pàgines molt freqüents.

3.5.2 Simulació de l'algorisme de reemplaçament de pàgines

Per fer la simulació de l'algorisme de reemplaçament de pàgines ha estat necessari limitar la memòria principal de l'únic processador present en l'arquitectura a 64 unitats de memòria. A més aquest té un overhead de faltes de pàgina de 15 cicles. La resta de pàrametres de l'arquitectura és el mateix que a l'anterior apartat de l'experimentació.

El programa utilitzat per la simulació és el que es mostra en la figura 3.12.

Aquest programa paral·lel està compost de 3 tasques que estan formades per una sola primitiva d'execució de bastants cicles, 7300 per la primera, 6250 per la segona i 5200 per la tercera. A més els requeriments de memòria de les tres tasques són molt alts ja que cada tasca sobrepassa molt més les 64 unitats de memòria princial. El rss minimum és bastant alt a les tres tasques i l'interval de faltes de pàgines es bastant baix, cosa que vol dir que la freqüencia d'aparició de faltes de pàgina serà molt alt.

S'ha procedit a la simulació del programa paral·lel i el resultat gràfic és el que es mostra en la figura 3.13.

El temps de simulació que ens mostra la gràfica en la primera barra es bastant alt, supera els 25000 cicles. I si comparem aquest resultat amb el Workload Mapped per tal de saber la diferència de temps de l'execució del programa amb i sense faltes de pàgina podem veure que també és bastant elevada, una diferencia de més 7000 unitats. Això vol dir que es destina més del 28% del temps de CPU a gestionar faltes de pàgina.

El Working Time és el mateix que el Simulation Time perquè en aquesta simulació es

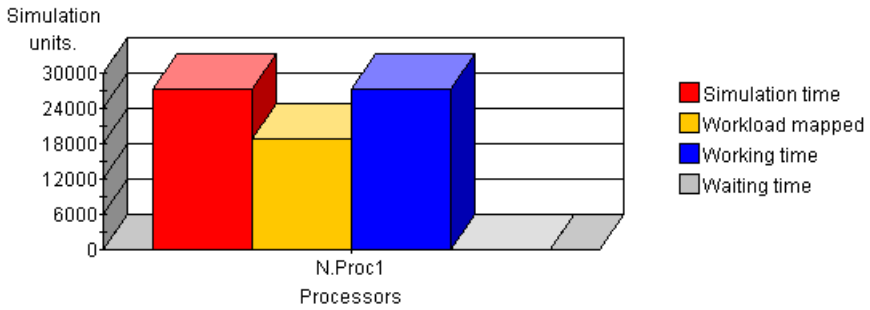

Figura 3.13: Gràfica dels resultats de la simulació de la segona part de l'experimentació amb memòria virtual.

destina el 100% de la càrrega a la CPU i no a les comunicacions.

Per veure-ho més detalladament els resultats en la figura 3.14 es mostren els resultats textuals de la simulació.

Tal com es pot observar, el speedup és baix, sols 0,688. Això vol dir que la simulació d'aquest programa paral·lel té un rendiment baix degut als requeriments de memòria del programa i l'arquitectura del simulador. El temps exacte per la simulació del programa és de 27240 cicles, un valor molt alt si ho comparem amb el workload mapped que tant sols és de 18750 cicles. Això vol dir que el 31,1% dels cicles es destinen a faltes de pàgina. El número de faltes produides és de 566, una mitja d'una falta de pàgina cada 33,12 cicles. Aquest valor significa que l'aparició de faltes de pàgina és molt freqüent si tenim en compte que de cada 33,12 cicles 15 cicles van destinats a la gestió d'una falta de pàgina. Per aquest motiu, el temps de sobrecarrega de faltes de pàgines és de 8490 cicles.

L'algorisme de reemplaçament de pàgines ha tingut d'intervindre moltes vegades per intercanviar o reemplaçar pàgines perquè aquestes tasques tenen uns requeriments de memòria molt elevats i el processador té una memòria molt baixa. Per cada tasca que s'apropiava la CPU tenia que intervindre l'algorisme de reemplaçament de pàgines per alliberar memòria perquè fos assignada per aquesta mateixa tasca. Hi han haguts bastants moments seguits que per cada falta de pàgina que es produia l'algorisme ha tingut que actuar.

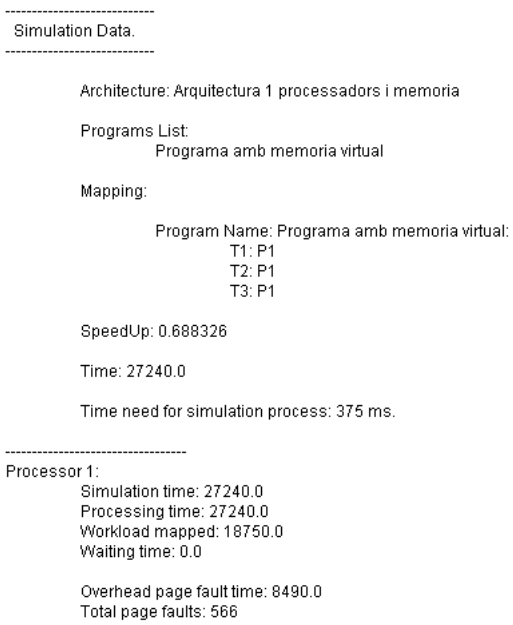

Figura 3.14: Resultats textuals de la simulació de la segona part de l'exeprimentació amb memòria virtual.

4 Extensió del simulador cap a un entorn Multiprogramat

4.1 Introducció

L'objectiu d'aquest capítol és extendre el simulador cap a un entorn multiprogramat a nivell de tasques locals i paral·leles.

Les tasques locals són tasques sense comunicacions que executen un codi preestablert. Basicament serveixen per simular la resposta del comportament d'un usuari local dins d'un entorn paral·lel. S'ha de tenir en compte que l'execució de les tasques loacvls provocarà una disminució en el rendiment de les tasques paral·leles donat que les tasques locals poden consumir una quantitat considerable tant de memòria com de CPU. Aquest tipus d'entorns compartits per aplicacions paral·leles i locals s'anomenen arquitectures no dedicades.[3]

Basicament s'ha realitzat per saber com respon la conducta de la memòria durant la simulació de tasques paral·les i locals.

El suport a la multiprogramació a nivell de tasques paral·leles es basa en donar la possibilitat al simulador de carregar més d'un programa paral·lel perquè d'aquesta manera es pugui mesurar el rendiment de varis programes paral·lels mapejats en els processadors de la màquina paral·lela.

En resum, les dues implementacions serveixen per dotar al simulador ESPPADA d'un entorn més real comparat amb el funcionament del sistemes paral·lels actuals.

En la secció 4.2 s'explica tot el que comporta la Multiprogramació a nivell de tasques locals.

En la secció 4.3 s'explica detalladament tot el que comporta la Multiprogramació a nivell de tasques paral·leles.

En la secció 4.4 s'explica la Implementació de la Multiprogramació de les tasques locals i paral·leles.

Finalment en la secció 4.5 s'explica la Experimentacio realitzada amb el simulador utilitzant aquestes dos implementacions.

4.2 Multiprogramació a nivell de tasques locals

Les tasques locals són simples tasques sense comunicacions que executen un codi preestablert per part d'un usuari. El seu objectiu és dotar d'una simulació més real al sotmetre l'entorn paral·lel a carregues de treball de memòria per part dels usuaris locals que puguin treballar en aquest entorn.

```
for i=0 to COUNT
begin
exec I cicles
wait J cicles
end
```

Figura 4.1: Primitives de les tasques locals

Totes les tasques locals estan composades de tres primitives i són de la mateixa forma. El codi d'una tasca local està format per les primitives mostrades en la figura 4.1.

El parametre **COUNT** indica el nombre d'iteracions de la tasca local, el paràmetre **I** indica els cicles de la primitiva Exec que s'executarà en cada iteració i el paràmetre **J** indica els cicles que estarà en espera la tasca en cada iteració. Aquests tres paràmetres són modificables en el simulador ESPPADA a partir de la seva interfície.

La primitiva *Exec* serveix per simular el comportament d'una tasca local real quan utilitza la CPU per oferir algun servei a l'usuari. La primitiva *Wait* serveix per simular l'espera que realitza una tasca local a la demanda de peticions de l'usuari. D'aquesta manera es manté bloquejada i va ocupant tots els recursos de la memòria per tal de obligar el sistema a l'intercanvi de pàgines de memòria mentres s'executa les tasques paral·leles.

Aquestes dues primitives estan dins d'un bucle que s'executa en cada iteració. D'aquesta manera es simula el comportament interactiu tipic d'un usuari local.

La primtiva *Wait* és una nova primitiva que s'ha implementat utilitzants els propis recursos del simulador. Internament és a la vegada una primitiva *Receive* i una *Send* que espera un missatge local enviat anteriorment en la primera crida de la primitiva. S'ha realitzat d'aquesta manera perquè el planificador del simulador pugui enviar la tasca local a la cua d'*Espera* i planifiqui una altra tasca mentres aquesta està en espera de rebre el missatge local.

La tasca local també utilitza la memòria virtual. Això implica que el simulador permet canviar els paràmetres del **rss minimum**, la quantitat de memòria virtual que utilitza la tasca i l'interval de fallides de pàgina. El valor d'aquests paràmetres són comuns a totes les tasques locals mapejades en els diferents processadors del sistema.

També s'ofereix la possibilitat de pendre la desició de si una tasca local estarà mapejada en un determinat processador o no. D'aquesta manera es simula un entorn cluster no dedicat, caracteritzat perquè solament una determinada quantitat de nodes disposen de tasques locals.

En resum la implementació no es gaire costosa, simplement es crea internament abans de començar la simulació una nova nova tasca que tingui aquestes primitives i un identificador reservat per la tasca local per saber en tot moment quan s'executa la tasca local. Després es realitza una expansió de les primitives de la tasca local perquè apareix la primitiva complexa *For* i exten el codi repetint les primitives que hi ha dins del bucle **COUNT** vegades.

Tots els events de la tasca local són reportats en el fitxer *log* de la simulació, a la

consola i en els resultats de la simulació.

4.3 Multiprogramació a nivell de tasques paral·leles

L'objectiu principal de la Multiprogramació a nivell de tasques paral·leles és dotar al simulador d'un entorn més real per tal de mesurar el rendiment d'un entorn paral·lel simulant un o varis programes paral·lels, que s'executen simultaniament.

S'ha modificat la interfície del programa per carregar més d'un programa paral·lel o eliminar-lo si aquest ja està carregat. A més s'ha modificat també la interfície del *manual mapping* i *Round Robin mapping* per poder seleccionar tasques de diferents programes. El maping manual realitza un mapeig tant individual com col·lectiu, es a dir, si vols pots mapejar les tasques d'un sol programa o de varis. El maping *Round Robin* realitza un mapeig individual del programa seleccionat.

La idea principal és bastant simple, crear una estructura de classes, en aquest cas una nova classe anomenada *Multiprogram* que conté la llista de programes carregats en el simulador. La implementació es una mica més costosa perquè ara tenim moltes més tasques que poden tenir identificadors de tasca repetit, pertant s'haurà de generar un nou identificador intern no repetit per cada tasca i primitiva i guardar la informació de la relació entre l'identificador original i el nou generat. La millor estructura de dades per guardar aquesta informació és una taula *hash*.

La simulació dels multiprogrames es realitza de la mateixa manera que en un sòl programa. Simplement, són més tasques amb un identificador nou no repetit. No s'ha tingut de modificar gaire el codi per realitzar la simulació.

El requeriment de la simulació és que totes les tasques d'un programa estiguin mapejades i no requereix haver de mapejar tots els programes carregats en el simulador.

Al principi de la simulació se li passa a la classe *Simulator* definida en el fitxer *Simulator.java* la classe *MultiProgram* per tal d'obtenir totes les tasques del programa a simular. Durant la simulació, alhora de mostrar els events per la consola, en un fitxer o en el formulari de resultats, es crida a les funcions de la classe *MultiProgram* per tal d'obtenir l'identificador original de la tasca i mostrar aquest identificador més el nom del programa per saber de quin programa es tracta. També ho realitza al final de la simulació al seleccionar les tasques per mostrar les seves traces.

Cal esmentar, que anteriorment sols simulava un programa i per tant hi havia una classe general de tipus *Program*. Ara s'ha tingut de canviar el codi per tal de que totes les parts del simulador disposin de la classe *Multiprogram* bàsicament per obtenir les tasques del programa seleccionat i l'identificador original de la tasca.

Cal destacar que algunes parts de simulació del programa ESPPADA, com per exemple la creació del graf TTIG sols treballen amb un sol programa perquè treballar amb més d'un no tendria sentit i pertant utilitza el programa actual seleccionat, en concret el que es mostra en forma d'estructura d'arbre al formulari *Program definition*.

4.4 Implementació

Aquesta secció està composada per dues subseccions. La primera subsecció, 4.4.1, s'explica detalladament com s'ha implementat la multiprogramació de les tasques locals a nivell de codi font i el disseny de la interfície per fer ús de les tasques locals.

En la segona subsecció, 4.4.2, es comenta amb detall la implementació de la multiprogramació de les tasques paral·leles a nivell de codi font i el disseny de la interfíce per carregar més d'un programa paral·lel en el simulador.

4.4.1 Implementació de la Multiprogramació de les tasques locals

4.4.1.1 Especificació de l'arquitectura amb tasques locals

En la figura 4.2 es mostra un exemple de definició d'arquitectura amb llenguatge XML que conté la definició de les tasques locals. A l'arrel de la definició de l'arquitectura apareix una etiqueta anomenada **local_task** amb els següents paràmetres. El primer és el paràmetre **for** que especifica 3 iteracions per aquesta primitiva, el següent és el parametre **exec** que especifica 200 cicles d'execució, el tercer és el paràmetre **wait** que especifica el número de cicles que estarà en espera la tasca local en cada iteració. Llavors apareix el paràmetre **terminate** que especifica si finalitzarà la tasca local juntament quan acabi el programa si el booleà és igual a cert. Després apareixen els paràmetres de memòria virtual de la tasca local que són **virtual_memory** que indica la quantitat de memòria virtual que utilitzarà la tasca local, el **rss_minimum** que indica tal com diu la paraula el **rss_minimum** de la tasca local i el **pagefault_interval** que indica l'interval de temps entre faltes de pàgina. Aquests 7 paràmetres són comuns a totes les tasques locals mapejades en els processadors. L'altra etiqueta que reconeix el simulador és l'etiqueta **local_task** definida com atribut a l'especificació dels processadors. És un booleà que indica si el processador definit contindrà una tasca local mappejada. Serveix per especificar quins processadors tindran tasques locals assignades.

4.4.1.2 Creació de les tasques locals

Les tasques locals es creen abans de començar la simulació quan ja estan definits els mapings dels processadors que contindran les tasques locals.

El codi que crea les tasques locals és el que es mostra a continuació:

```
public Task create_local_task()
{ Task tasca_local;
  P_For pfor;
  P_Wait pwait=new P_Wait(archi.get_local_task_wait ());
  P_Exec pexec=new P_Exec(archi.get_local_task_exec ());
  Lista forLista = new Lista();
  forLista.insertBack (pexec,Constants.EXEC);
  forLista.insertBack (pwait,Constants.WAIT);
  pfor = new P_For((int)archi.get_local_task_for (),forLista);
```

```
                                                    <?xml version="1.0"?>

<!-- Architecture definition created by ESPPADA tool v1.0 -->
<architecture name="Arquitectura 4 processadors i memoria">
 <cpu_base value="1.0"/>
 <context_change value="0"/>
 <memory_enabled value="true"/>
 <local_task for="3" exec="200" wait="400" terminate="false"
virtual_memory="256"  rss_minimum="20" pagefault_interval="100"/>
 <net type="full_connect">
 <delay value="1.0"/>
 </net>
 <planification method="round_robin" quantum="200"/>
 <!-- Process units definition -->
 <processor id="1">
  <cpu_period value="1.0"/>
  <memory_size value="128"/>
  <pagefault_overhead value="10"/>
  <local_task value="true"/>
 </processor>
 <processor id="2">
  <cpu_period value="1.0"/>
  <memory_size value="128"/>
  <pagefault_overhead value="12"/>
  <local_task value="true"/>
 </processor>
 <processor id="3">
  <cpu_period value="1.0"/>
  <memory_size value="128"/>
  <pagefault_overhead value="12"/>
  <local_task value="true"/>
 </processor>
 <processor id="4">
  <cpu_period value="1.0"/>
  <memory_size value="128"/>
  <pagefault_overhead value="10"/>
  <local_task value="true"/>
 </processor>
</architecture>
```

Figura 4.2: Definició d'arquitectura amb tasques locals

```
    tasca_local=new Task(Constants.ID_LOCAL_TASK);
    tasca_local.Primitives.insertBack (pfor,Constants.FOR);
    tasca_local.set_rss_minimum (archi.get_local_task_rss_minimum ());
    tasca_local.set_pagefault_interval(
      archi.get_local_task_pagefault_interval ());
    tasca_local.setMemVirt (
      archi.get_local_task_virtualmemory ());
    pfor=null;
    pexec=null;
    pwait=null;
    forLista=null;
    return tasca_local;
  }
```

La funció que mostra aquest codi crea una tasca local amb les seves primitives i la retorna. Està declarada dins de la classe *SProcessor* definit en el fitxer *Simulator.java*. Primerament declara una tasca local i les primitives *pfor*, *pwait*[1] i *pexec* de tipus *P_For*, *P_Wait* i *P_Exec* respectivament.

Després declara i crida al constructor d'una llista de tipus *Lista* anomenada *forLista*. Dins aquesta llista insereix la primitiva *pexec* i *pwait* amb els seus identificadors *EXEC* i *WAIT* respectivament perquè sapiga quin tipus d'objecte retorna la llista quan es faci una consulta. Despres crida al constructor de la primitiva *pfor* on insereix per paràmteres el número d'iteracions extretes de l'arquitectura del sistema (*archi.get_local_task_for()*) i la llista *forLista* que conté les primitives que aniran dins el bucle.

A continuació crea una tasca local amb l'identificador reservat *ID_LOCAL_TASK*. Llavors insereix la primitiva *pfor* dins la llista de primitives de la tasca amb el seu identificador *FOR*. Llavors assigna el **rss minimum** de la tasca local a través de la funció *set_rss_minimum()* on li passa per paràmetre el **rss minimum** de la tasca local que conté l'arquitectura del sistema (*archi.get_local_task_rss_minimum()*). Després assigna el **pagefault interval** a través de la funció *set_pagefault_interval()* que se li passa per paràmetre el **pagefault interval** de la tasca local que està en l'arquitectura del sistema (*archi.get_local_task_pagefault_interval()*) i finalment assigna la memòria virtual de la tasca local a través de la funció *setMemVirt()* que se li passa per paràmetres la memòria virtual de la tasca local que conté també l'arquitectura del sistema (*archi.get_local_task_virtualmemory()*).

L'ultima acció que realitza és destruïr les primitives declarades posant-les a null perquè ja no s'en faran ús i retornar la tasca local.

A continuació es mostra el codi del constructor de la classe *SProcessor* que crida la funció *create_local_task()* per crear la tasca local:

```
if (processor.get_local_task())
{
  Task local_task=create_local_task();
```

[1] La implementació de la primitiva P_Wait s'explicarà despres de comentar la creació de tasques locals.

```
    if (memory_enabled && (local_task.getMemVirt ()>0
    || local_task.get_rss_minimum ()>0 ||
    local_task.get_pagefault_interval ()>0))
      Memoria.add_task_memory(ProcId,local_task.getId(),
    local_task.getMemVirt(),0,local_task.get_rss_minimum (),
    local_task.get_pagefault_interval ());
    taskQueue = new Task_Queue(local_task.getId(),local_task);
    Preparados.insertBack(taskQueue,local_task.getId ());
    if (Internal_traces)
      PTrace.new_TaskTrace(local_task.getId());
    N_Tasks_Assigned++;
  }
```

Les accions que realitza aquest codi són les següents: Primerament crida a la funció *create_local_task()* per crear, com hem dit anteriorment la tasca local i aquesta ho retorna a la variable *local_task* de tipus *Task* definida en el fitxer *Task.java*.

Llavors l'afegeix a la llista de tasques de memòria de la classe *Memory* a traves de la funció *add_task_memory* passant-li per paràmetres l'identificador del processador, l'identificador de la tasca local, la quantitat de memòria virtual, el conjunt de pàgines residents, el **rss minimum** i l'interval entre faltes de pàgina si la memòria està activa i a més la memòria virtual de la tasca local és més gran que zero i el **rss minimum** és també superior a zero.

Després crea una instància de tipus *Task_Queue* definida en el fitxer *Simulator.java* passant-li al constructor per paràmetre l'identificador de tasca local i la tasca local. Aquesta classe com hem dit anteriorment serveix per implementar la cua de tasques i per expandir la llista de primitives de la tasca. Llavors la insereix a la cua de Preparats.

Finalment si el booleà *Internal_traces* està actiu que determina si es faran durant la simulació traces internes llavors afegeix una nova traça de tasques passant-li per paràmetre l'identificador de la tasca local. Després incrementa el número de tasques assignades en aquesta CPU.

4.4.1.3 La primitiva Wait

La primitiva *Wait* està implementada per una classe anomenada *P_Wait* i definida en el fitxer *P_Wait.java*. Els seus atributs són l'identificador de la primitiva Wait de tipus *static* per saber quin tipus d'objecte retorna la llista de primitives quan es fa una consulta i una variable anomenada *cycles* que conté el número de cycles que ha d'estar en espera. Les funcions que conté aquesta classe són el constructor *P_Wait(long c)* que se li passa per paràmetre el número de cicles, la funció *long getCycles()* que retorna el número de cicles, la funció *void setCycles(long c)* que assigna el número de cicles de la primitiva en qualsevol instant després de la seva declaració i la funció *public String toXML()* que retorna un missatge codificat amb XML de la primitiva *Wait*.

La primitiva *Wait* funciona enviant un missatge local a ella mateixa i mentre no arribi planifica una altra tasca.

La primera fase del codi font de tractament de la primitiva *Wait* és el que es mostra a continuació:

```
case Constants.WAIT:
 Trace_Node_RECEIVE tnReceive_Local_Task;
 pwait = (P_Wait) primitives_node.getNodo();
 cnode = InnerMessages.firstnodo;
 while (cnode!=null)
 {
  Messages mis = (Messages) cnode.getNodo();
  if  (mis.get_Id()==Constants.ID_MSG_LOCAL_TASK  &&
  mis.get_EmissorTaskId()==realTask &&
  realTask == mis.get_ReceptorTaskId())
  {
    if (mis.get_receiveTime()<=actualTime)
    {
      Repository.insertBack(primitives_node);
      tnReceive_Local_Task = new Trace_Node_RECEIVE(
      mis.get_receiveTime(), mis.get_EmissorTaskId(),
      mis.get_ReceptorTaskId(),mis.get_Id(),
      mis.get_vol(),realTask,ProcId,MultiProg);

      (....)
      tasca.Primitives.removeFirst();
      removeMessageFromList(InnerMessages,mis.get_EmissorTaskId(),
                            realTask,mis.get_Id(),true);
      slots++;
    }
    else
    {
      fin = true;
      if (logSim)
      { file_log.writeBytes("Proc"+ProcId+"["+actualTime+"]: Local
        Task waits a local task message (Internal send)\n");
      }
      if (logCon)
      { System.out.println("Proc"+ProcId+"["+actualTime+"]: Local
        Task waits a local task message (Internal send)");
      }
      task2wait = realTask;
      message2wait = Constants.ID_MSG_LOCAL_TASK;
      tasca.set_ActualMoment(actualTime);
      tasca.set_Task2Wait(task2wait);
      tasca.set_MessageId(message2wait);
```

```
        tasca.set_Processor(ProcId);
        Espera.insertPos(tasca,realTask,actualTime);
        Preparados.SacaNodo(realTask);
      }
      break;
    }
    cnode = cnode.next;
  } //end del while
```

El codi de la primitiva *Wait* està dins d'un switch igual que la primitiva *Exec*. Primerament obté la primitiva pwait del node *primitives_node* de la llista de primitives de la tasca actual. Llavors obté el primer node, *cnode*, de la llista de missatges locals del processador. Després recorre tota la llista de missatges locals del processador dins d'un bucle fins que no exiteixi cap node més. A cada iteració primerament obté l'objecte que apunta el node *cnode* que és de tipus *Messages* definit en el fitxer *Messages.java* que representa un missatge d'una tasca. Si l'identificador del missatge és igual a l'identificador de missatge de tasca local i l' identificador del receptor i emissor és igual al de la tasca local llavors fa el tractament del missatge rebut. Aquest tractament consisteix en comprovar si el temps de rebuda del missatge és inferior al temps actual de la simulació, llavors vol dir que el pot aténdre i pertant col·loca a la llista de repositoris la primitiva per ser després eliminada, enregistra i mostra per la cònsola un missatge d'event de recepcio del missatge local indicant que ha rebut aquest missatge la tasca local i després elimina la primitiva de la llista de primitives d'aquesta tasca, elimina el missatge de comunicacions rebut i incrementa el número de slots de simulació. Altrament si el temps actual és inferior al temps de rebuda del missatge llavors actua posant el booleà *fin* igual a true per indicar al simulador que ha de planificar una altra tasca, ja que aquest no és el moment en que ha de rebre el missatge. Llavors escriu al fitxer *log* i a la cònsola un missatge d'un event que espera la tasca local el seu propi missatge enviat, assigna el moment actual a la tasca, l' identificador de la tasca que espera el missatge que és ella mateixa, l' identificador del missatge que espera i el processador que conté el missatge i l'insereix a la cua d'*Espera* eliminant-lo de la cua de *Preparats*. Llavors surt de la primitiva *Wait* per ser planificada una altra tasca.

L'explicació d'aquest codi font equivaldria a la recepció del missatge de la tasca local, i pertant la següent fase es enviar el missatge perquè en un futur pròxim el pugui atendre. El codi següent mostra aquesta fase d'aquesta primitiva:

```
if (cnode==null)
{
  Trace_Node_SEND tnSend_Local_Task;
  Messages ms = new Messages(realTask,realTask,0,
                             Constants.ID_MSG_LOCAL_TASK,
                             map.get_ProchasTask(realTask));
  if (actualTime<timeNextSend)
  {
```

```
      actualTime = timeNextSend;
      timeNextSend += gapTimeNextSend;
    } else
    {
      timeNextSend = actualTime+gapTimeNextSend;
    }
    ms.set_sendTime(actualTime);
    latency=Local_Comm_Overhead;
    Usage+=Local_Comm_Overhead;
    ms.set_receiveTime(actualTime+latency+pwait.getCycles ());
    InnerMessages.insertPos(ms,realTask,actualTime+latency+pwait.getCycles());
    tnSend_Local_Task = new Trace_Node_SEND(
                        actualTime+latency,ms.get_EmissorTaskId(),
                        ms.get_ReceptorTaskId(),ms.get_Id(),
                        ms.get_vol(),realTask,ProcId,MultiProg);
             (....)
    slots++;
    fin = true;
    blocked = false;
    if (logSim)
    {
      file_log.writeBytes("Proc"+ProcId+"["+actualTime+"]:
                           Local Task"+" blocked waiting local task
                           message at moment"+actualTime+" (Internal send)\n");
    if (logCon)
    {
      System.out.println("Proc"+ProcId+"["+actualTime+"]: Local Task"
                        +" blocked waiting local task message at moment "
                        +actualTime+" (Internal send)");
    }
    task2wait = realTask;
    message2wait = Constants.ID_MSG_LOCAL_TASK;
    tasca.set_Processor(ProcId);
    tasca.set_Task2Wait(task2wait);
    tasca.set_MessageId(message2wait);
    tasca.set_ActualMoment(actualTime);
    Espera.insertPos(tasca,realTask,actualTime);
    Preparados.SacaNodo(realTask);
  } // end del if
  break;
```

Si *cnode* és igual a null significa que ha recorregut tota la llista de missatges interns de la CPU i no ha trobat cap missatge de la tasca local. Per tant ha d'enviar el missatge perquè sigui atés en un futur pròxim. Primerament, declara un nou missatge on se li

passa al constructor l'identificador de la tasca emisora que és la tasca local, l'identificador de la tasca receptora que també és la tasca local, el volum de dades que és nul perquè no interessa enviar cap contingut de dades sols el missatge, l' identificador del missatge que és l' identificador del missatge de la tasca local i per últim l' identificador del processador que conté la tasca local.

En segon lloc comprova mitjançant una sèntencia condicional si el temps actual de simulació és inferior al temps que pot tornar a enviar un missatge. En cas afirmatiu avança el temps actual de simulació (*actualTime*) al temps absolut del pròxim enviament (*timeNextSend*), i aquest últim temps l'incrementa amb el temps mínim de tornar a enviar un missatge (*gapTimeNextSend*). Pel contrari en cas negatiu, actualitza el *timeNextSend* amb el temps actual de simulació més el temps mínim que ha d'esperar per tornar a enviar un missatge (*gapTimeNextSend*).

Llavors fora de la sentència condicional, assigna el temps d'enviament del missatge que és *actualTime,* assigna la latència de comunicacions i el temps d'utilització de CPU amb la sobrecàrrega de comunicacions. Després assigna el temps de rebuda del missatge amb el temps actual de la simulació, més la latencia de comunicacions, més els cicles d'espera de la primitiva *Wait*.

A continuació insereix el missatge creat en la llista de missatges interns de la CPU actual, i crea un event nou d'enviament del missatge local. Llavors l'enregistra, l'escriu al fitxer *log* i a la consola. Després incrementa el número de slots de la simulació, posa el booleà *fin* igual a cert per indicar que ha d'assignar una altra tasca a la CPU i el booleà *blocked* igual a fals per indicar que no està bloquejada la CPU actual. Llavors escriu per pantalla i al fitxer *log* el missatge que la primitiva *Wait* està esperant el missatge local que ella mateixa s'ha enviat.

Finalment prepara la tasca local de tipus *TaskQueue* per ser insertada en la cua d'Espera, la insereix i elimina de la cua de *Preparats* la tasca local.

Totes aquestes accions són les que realitza el codi de la primitiva *Wait* implementada per les tasques locals.

4.4.1.4 Disseny de l'interfície de les tasques locals

En la figura 4.3 es mostra el disseny de l'interfície de l'apartat "Local Task" dins del formulari de l'arquitectura. En aquesta figura es pot observar que hi han dos apartats dins de la pestanya "Local Task". El primer s'anomena "Local Task Execution Parameters" i conté els parametres d'execució de la tasca local. El primer paràmetre és el contador de la primitiva *For*, el segon paràmetre són els cicles d'execució de la primitiva *Exec* i el tercer paràmetre indica els cicles d'espera de la primitiva *Wait*. Llavors apareix un checkbox que indica si es vol finalitzar les tasques locals de la simulació, quan la execució del programa paral·lel acabi. En cas d'estar activat, la simulació acabarà quan el programa hagi finalitzat sense tenir en compte si les tasques locals han acabat. En cas d'estar desactivat la simulació acabarà quan totes les tasques incloses les tasques locals finalitzin.

El segon apartat s'anomena "Local Task Memory Parameters" i indica els paràmetres de memòria de la tasca local. El primer paràmetre és la quantitat de memòria virtual

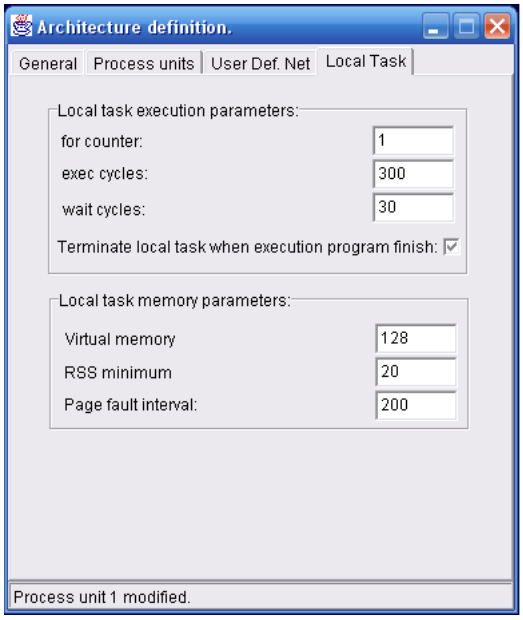

Figura 4.3: Interficie de les tasques locals en el formulari de l'arquitectura

de la tasca local, el segon paràmetre és el **rss minimum** de la tasca local i el tercer paràmetre és el **pagefault interval** d'aquesta tasca.

Si es vol realitzar la simulació amb tasques locals i la memòria virtual està activada és necessari que el **pagefault interval** sigui més gran que zero, ja que sinó crearia un bucle infinit que no avançaria mai la simulació.

L'altra interfície que s'ha modificat es la que es mostra a la figura 4.4. A cada processador definit s'ha afegit un *checkbox* dins de l'apartat "Principal Memory Parameters" que permet indicar si existirà mappejada una tasca local en el processador actual.

4.4.2 Implementació de la Multiprogramació de les tasques paral·leles

La implementació de la multiprogramació de les tasques paral·leles ofereix la possibilitat de tenir més d'un programa paral·lel carregat en el simulador.

En la subsecció 4.4.2.1 es comenta amb detall la implementació de la classe *MultiProgram* i les seves funcions.

En la subsecció 4.4.2.2 s'explica la implementació del *mapping* adaptat a la multiprogramació i quines modificacions s'han realitzat a l'interficie.

Finalment en la subsecció 4.4.2.3 es comenta la implementació de la multiprogramació en la simulació.

4.4.2.1 Implementació de la classe *MultiProgram*

Per tal de comentar la implementació primer explicarem la nova classe anomenada *MultiProgram* definida en el fitxer *MultiProgram.java*. Aquesta classe conté la llista de progra-

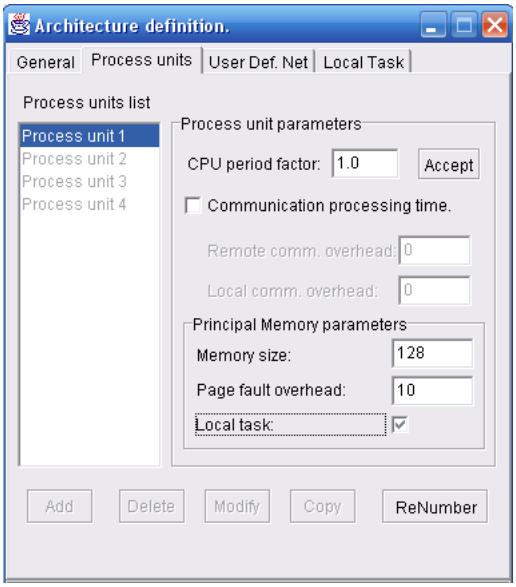

Figura 4.4: Interfície dels processadors del sistema amb tasques locals

Tipus	Nom de l'atribut	Descripció
public Lista	list_programs	Llista de programes carregats al simulador
private Hashtable	ID_conversion_table	Taula hash on guardem els identificadors originals a partir del seu corresponent modificat.
private Hashtable	ID_conversion_table_reverse	Taula hash on guardem els identificadors modificats a partir del nom del programa i del nom de la tasca

Taula 4.1: Taula d'atributs de la classe *MultiProgram*.

mes carregats al simulador i guarda la relació de l'identificador creat amb l'identificador original de la tasca i viceversa mitjançant dues taules *hash*. Cal recordar que per cada tasca, d'un nou programa inserit en la llista, genera un identificador intern no repetit per saber de quina tasca és tracta i a quin programa pertany.

En el simulador sols hi ha declarada una instància de la classe *MultiProgram* en la classe *Marco_Principal* definida en el fitxer *Marco_Principal.java*. Aquesta classe conté instàncies globals de les classes utilitzades en totes les parts del simulador, com l'arquitectura del sistema, el *mapping*, el programa TTIG etc.

Els seus atributs són els que es mostren en la taula 4.1.

L'atribut *list_programs* s'utilitza per guardar la llista de programes carregats en el simulador i per consultar la informació d'algun programa en qualsevol moment.

L'atribut *ID_conversion_table* és una taula hash que guarda objectes classe simples anomenats *ID_entry* definit en el fitxer *MultiProgram.java* que conté els atributs *pro-*

gram_name de tipus *String*, *original_id_task* de tipus *int* i *new_id_task* de tipus *int* també. El primer guarda el nom del programa corresponent a la tasca que conté l'identificador que volem inserir, el segon guarda l' identificador original de la tasca i el tercer guarda el nou identificador de tasca generat. La clau de la taula hash és l' identificador original de la tasca.

L'atribut *ID_conversion_table_reverse* és una taula hash que guarda l'identificador generat de la tasca a partir de la clau que és un *string* que conté el nom de la tasca més el nom del programa. És de la següent forma: "Task: "+*original_id_task*+" Program: "+*program_name*.

En la taula 4.2 es mostra el conjunt de funcions implementades en la classe *MultiProgram*.

A continuació es comenten les funcions més importants:

- La funció *add_program()* afegeix un programa a la llista de programes de la classe *MultiProgram* i s'utilitza principalment en la classe *Marco_Prog* definida en el fitxer *Marco_Prog.java* que implementa el formulari *Programs Definition*. En aquest formulari existeix un boto amb l'etiqueta "Add" que permet afegeir nous programes definits amb XML en el simulador.

- La funció *delete_program()* també s'utilitza dins de la classe *Marco_Prog*, concretament quan es pulsa el boto "Delete" que esborra el programa seleccionat i el seu *maping* associat.

- La funció *find_original_task_name()* s'utilitza en moltes parts del simulador quan es vol saber el nom original de la tasca més el nom del programa per tal d'estar informat a quin programa pertany. En la simulació s'utilitza moltissim en els events que retornen els missatges per saber de quina tasca original es tracta. Al final de la simulació dins dels resultats de la simulació també s'utilitza per informar a l'usuari de quina tasca es tracta ja que internament s'ha reemplaçat el seu identificador i és necessari informar a l'usuari sobre l'identificador original de la tasca més el nom del programa. Aquesta funció utilitza la taula hash *ID_conversion_table* per fer la consulta.

- La funció *find_new_task_id()* retorna l'identificador nou generat a partir de l'identificador original. Aquesta funció utilitza la taula hash *ID_conversion_table_reverse* per fer la consulta.

- La funció *replace_program()* substitueix un programa a la posició indicada pel paràmetre *index*. Aquesta també s'utilitza en la classe *Marco_Prog* per substituir el programa seleccionat per un nou programa que conté els atributs de programa que s'han canviat a través de la interfície.

- Per últim, la funció *remove_all()* s'utilitza per eliminar tots els programes de la llista de programes i les entrades de les dues taules hash concretament en la classe *Marco_Prog* quan ja no queda cap programa carregat al simulador.

Tipus	Nom de la funció	Descripció
public void	*add_program(Program Prog)*	Afegeix un programa a la llista de programes i actualitza els identificadors de les tasques i primitives pel nou generat si està repetit el original.
private void	*update_tasks_identifier(Program newprog)*	Actualitza l'identificador de les tasques i primitives del programa passat per paràmetres si l'identificador original està repetit.
private void	*update_primitive_identifier_program(Program prog,int old_idtsk,int new_idtsk)*	Actualitza els identificadors de les primitives del programa passat per paràmetres. També se li passa l'identificador vell i el nou identificador.
private void	*update_primitive_identifier(Lista primitive_list,int old_idtsk,int new_idtsk)*	Actualitza els identificadors de les primitives de la llista de primitives passada per paràmetres. També se li passa l'identificador vell i el nou identificador.
public Program	*get_program_by_index(int index)*	Retorna el programa segons la posició indicada.
public Program	*get_program_by_task_id(int id_task)*	Retorna el programa si conté la tasca amb l'identificador passat per paràmetres.
public Program	*get_program_by_name(String name)*	Retorna el programa que correspon amb el nom de programa passat per paràmetres.
public void	*delete_program(int index)*	Elimina el programa segons la posició indicada.
public boolean	*is_empty()*	Retorna cert si està buït de tasques el programa.
public int	*get_N_Tasks()*	Retorna el número de tasques de tots els programes.
public int	*get_N_Programs()*	Retorna el número de programes de la llista.
public String	*find_program_name(int new_id_tsk)*	Retorna el nom de programa a partir del nou identificador de tasca generat.
public int	*find_original_task_id(int new_id_tsk)*	Retorna l'identificador original de la tasca a partir del nou identificador generat.
public String	*find_original_task_name(int new_id_tsk)*	Retorna el nom de la tasca més el nom del programa a partir del nou identificador generat.
public int	*find_new_task_id(String original_task)*	Retorna l'identificador nou generat a partir del nom de la tasca més el nom del programa.
public int	*find_new_task_id(int id_original,String prog_name)*	Retorna l'identificador nou generat a partir del id. original i el nom del programa passat per paràmetres.
public void	*remove_all()*	Elimina les entrades de les taules hash i la llista.
public void	*replace_program(int index,Program newprog1)*	Substitueix un programa pel programa nou passat per parametres a la posició indicada.
public Task	*get_Task(int idTask)*	Retorna una tasca segons el seu identificador nou generat.

Taula 4.2: Taula de funcions de la classe *MultiProgram.*

4.4.2.2 Implementació del mapping multiprogramat

Existeixen dos tipus de mappings seleccionables al simulador ESPPADA. El *manual mapping* que tal com diu la paraula serveix per definir un mapping de forma manual, seleccionant les tasques que vols que siguin assignades a un processador concret.

L'altre mapping és el *Round Robin* i és automatic, ja que realitza l'assignació automaticament distribuïnt les tasques per tots els processadors.

El primer mapping, el manual és individual i col·lectiu que vol dir que tant es pot fer l'assignació de tasques a processadors d'un sol programa com de varis. Pel contrari, el mapping *Round Robin* és individual, sols permet fer l'assignació d'un sol programa però si té algun mapping definit anteriorment no el borra sino que l'afegeix amb el del programa seleccionat.

El mapping manual està definit en la classe *Manual_Assignment* definit al fitxer *Manual_Assignment.java*. Aquesta classe representa un formulari que permet a través de la seva interfice canviar l'assignació de les tasques als processadors.

A més s'ha afegit un *listbox* que permet seleccionar el programa del qual es vol mapejar les seves tasques. En la figura 4.5 és mostra el formulari del mapping manual. Quan es selecciona el programa es veu la llista de tasques no mapejades en l'apartat *Task List* i en l'apartat *Tasks Mapped* es veu la llista de tasques mapejades en el processador seleccionat per l'altre *listbox*.

El botçpp *New* del centre del formulari serveix per crear un nou mapping; el botó *Next Proc.* serveix per seleccionar el següent processador; el botó *Add* serveix per assignar la tasca seleccionada en el processador seleccionat; el botó *Add All* serveix per afegir totes les tasques en el processador seleccionat; el botó *Remove* serveix per eliminar la tasca seleccionada de l'apartat *Tasks Mapped* i finalment el botó *Remove All* elimina tots les tasques assignades en el processador seleccionat.

El mapping Round Robin anteriorment no tenia cap interfície, simplement feia l'assignació de tasques automaticament. Ara s'ha afegit una nova classe *Marco_IndividualPlanification*

RoundRobin definida en el fitxer *Marco_IndividualPlanificationRoundRobin.java* que representa un formulari amb un *listbox* per seleccionar el programa que es vol fer l'assignació *Round Robin*. L'algorisme de planificació *Round Robin* està implementat dins d'aquesta classe.

En la figura 4.6 es mostra el formulari de la planificació *Round Robin*.

Aquest algorisme no elimina el mapping anterior sinó que el conserva i distribueïx les noves tasques del programa seleccionat a tots els processadors del sistema, tenint en compte si la tasca estava mappejada anteriorment o no.

Aquesta classe després de seleccionar el programa s'apreta el botó "Accept" per tal de començar l'assignació automàtica *Round Robin*. Un cop finalitzada retorna al simulador ESPPADA.

Existeix dos mappings més sofisticats que en parlarem amb detall al capítol 5.

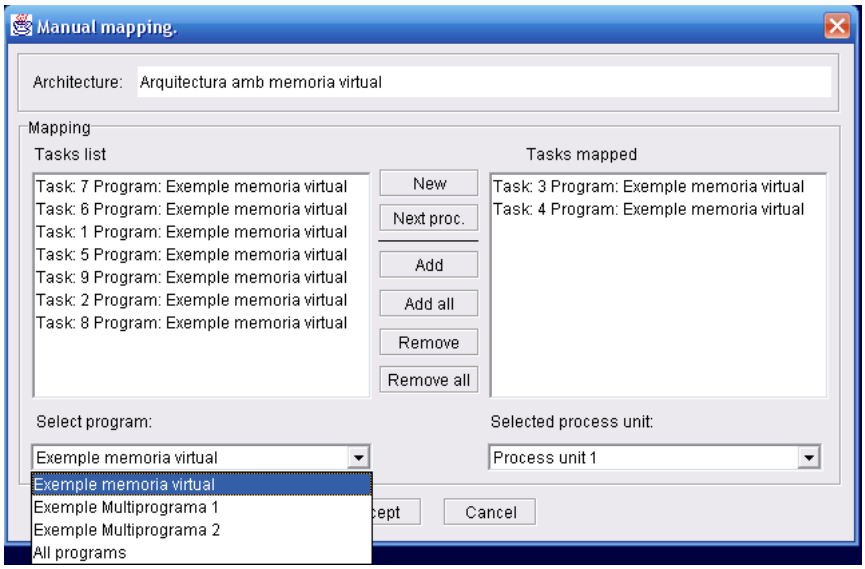

Figura 4.5: Interfície del mapping manual.

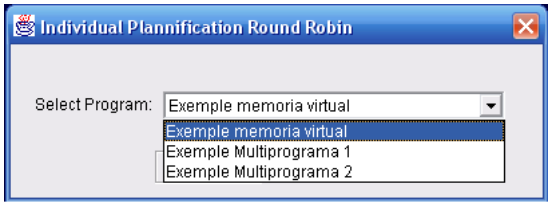

Figura 4.6: Interfície de la planificació Round Robin.

4.4.2.3 Implementació de la multiprogramació en la simulació

No s'ha tingut de canviar gaire codi per implementar la multiprogramació en la simulació. Simplement als constructors de les classes que representen els events s'ha afegit el paràmetre *MultiProgram* per tal de que aquesta classe tingui constància de l'identificador original de la tasca quan mostra els missatges dels events. A continuació es mostra un exemple de constructor d'una classe que representa l'event d'execució.

```
public Trace_Node_EXEC(float start, float end, float length,
int taskId, int ProcId,MultiProgram MPr)
```

En la funció *toString(),* de totes les classes que representen events, s'ha afegit la crida a la funció *find_original_task_id()* per tal de que retorni el missatge de l'event amb l'identificador original de la tasca més el nom del programa. Un exemple de missatge retornat per l'event d'execució és el següent:

```
MProg.find_original_task_name(TaskId)+" starts execution
period at "+Start+" to "+End+" Length:"+Length
```

Com que les tasques no tenen cap identificador repetit, ja que s'ha generat un nou identificador per cada tasca del programa paral·lel, i a les primitives per mantenir la coherència del programa, la simulació entén que quan afegim les tasques d'un programa paral·lel formen part tot el conjunt d'un sol programa general i pertant s'han de simular totes les tasques. A més les tasques de diferents programes estan mapejades totes juntes per tots els processadors del sistema, sense tenir en compte a quin programa pertany. Si es vol saber a quin programa pertany s'ha de consultar la classe *MultiProgram* que per això manté unes taules hash de consulta ràpida per saber-ho en qualsevol moment. S'ha fet d'aquesta manera per la reutilització de codi ja que sinó seria molt costós canviar tota la implementació del simulador.

El codi canviat en la classe *SProcessor* és al principi del constructor que se li passa per paràmetre la classe *MultiProgram* i durant la simulació concretament en la funció *advance()* a cada constructor de les classes que representen events que comencen amb el nom *Trace_Node* sel's hi passa per pàrametre la instància d'aquesta classe *MultiProgram*.

Al finaltitzar la simulació mostra les traces en un quadre de diàleg anomenat *Simulation Results*. Al seleccionar la tasca de la que es vol observar les seves traces, mostra l'identificador original de la tasca més el nom del programa. Cal destacar que a la pestanya *Result Text* on es mostren els resultats de la simulació, es mostra també el *mapping* de cada programa amb les tasques associades a cada processador.

4.5 Experimentació

Aquest capítol l'Experimentació consta de dos apartats. El primer, el 4.5.1 es comenta la Experimentació realitzada simulant un programa paral·lel amb tasques locals i el mateix sense tasques locals per comparar-los i observar com afecta en el rendiment del programa paral·lel l'execució de tasques locals.

El segon apartat, el 4.5.2 es comenta els resultats de la simulació de varis programes paral·lels executats simultaniament.

4.5.1 Experimentació de la multiprogramació amb tasques locals

En aquest apartat s'explicarà l'Experimentació realitzada fent la simulació d'un programa paral·lel amb l'assignació de tasques locals a tots els processadors del sistema. Després veurem la simulació del mateix programa paral·lel sense tasques locals assignades als processadors per tal de comparar el rendiment obtingut.

En la figura 4.7 es mostra el programa paral·lel definit amb llenguatge XML utilitzat per realitzar la simulació.

Com es pot observar el programa paral·lel consta de 9 tasques, totes formades per primitives *Exec* de diferents temps i amb els paràmetres de memòria virtual especificats, per que hi hagi una gran activitat de memòria principal.

Els paràmetres generals de la tasca local són els que es mostren a la figura 4.8.

Com es pot observar dins de l'apartat de paràmetres d'execució de la tasca local el paràmetre "for counter" és de 3 iteracions, el paràmetre "exec cycles" és de 200 cicles d'execució a cada iteració i el paràmetre "wait cycles" és de 400 cicles d'espera a cada iteració.

Els paràmetres de memòria virtual de les tasques locals tenen 256 unitats de memòria virtual, a cada falta de pàgina es carreguen 20 unitats de memòria i l'interval entre faltes de pàgina es de 120 cicles.

L'arquitectura és la que es mostra en la figura 4.9.

Tal com es pot observar, l'arquitectura conté 4 processadors definits amb 128 unitats de memòria principal cadascun. Es diferencien amb la sobrecarrega de faltes de pàgines que el primer i el quart són de 10 cicles i el segon i el tercer de 12 cicles. A més tots 4 processadors tenen mappejades una tasca local.

La planificació que s'ha utilitzat és la Round Robin que distribueix les tasques a tots els processadors de la següent forma:

El processador 1 conté assignades les tasques 7, 3 i 1. El processador 2 conté assignades les tasques 4 i 5. El processador 3 conté les tasques 6 i 9 i el processador 4 les tasques 1 i 2.

Llavors s'ha realitzat la simulació i els seus resultats gràfics són els que es mostren a la figura 4.10.

Podem observar que el processador 2 és el primer en acabar sobrepassant els 1000 cicles i a més sense produïr-se gaires faltes de pàgina ja que la diferència d'altura entre la barra "Simulation Time" i la barra "Workload Mapped" és molt poca. Cal recordar que la barra "Workload Mapped" indica els cicles d'execució sense tenir en compte la sobrecàrrega de faltes de pàgina. El segon processador en acabar és el processador 3 arribant casi als 1500 cicles d'execució. En aquest processador també es produeixen poques faltes de pàgina. El tercer processador en finalitzar és el processadror 1 sobrepassant els 2200 cicles. Tampoc es produeixen gaires faltes de pàgina. El processador 4 és el que ha necessitat més temps d'execució per finalitzar les tasques 1, 2 i la tasca local arribant casi als 4000 cicles. La

```
<?xml version="1.0"?>
<program name="Programa amb memoria virtual">
<task id="7" virmem="50" rss_minimum="50" pagefault_interval="190">
 <exec run="200"/>
 <exec run="333"/>
</task>
<task id="4" virmem="175" rss_minimum="10" pagefault_interval="160">
 <exec run="400"/>
</task>
<task id="6" virmem="100" rss_minimum="35" pagefault_interval="180">
 <exec run="150"/>
 <exec run="333"/>
</task>
<task id="1" virmem="90" rss_minimum="15" pagefault_interval="40">
 <exec run="350"/>
</task>
<task id="3" virmem="180" rss_minimum="60" pagefault_interval="200">
 <exec run="500"/>
 <exec run="320"/>
</task>
<task id="5" virmem="200" rss_minimum="100" pagefault_interval="200">
 <exec run="400"/>
</task>
<task id="9" virmem="125" rss_minimum="10" pagefault_interval="20">
 <exec run="200"/>
 <exec run="333"/>
</task>
<task id="2" virmem="500" rss_minimum="25" pagefault_interval="5">
 <exec run="1000"/>
</task>
<task id="8" virmem="350" rss_minimum="50" pagefault_interval="100">
 <exec run="250"/>
 <exec run="310"/>
</task>
</program>
```

Figura 4.7: Definició del programa paral·lel per la simulació amb tasques locals.

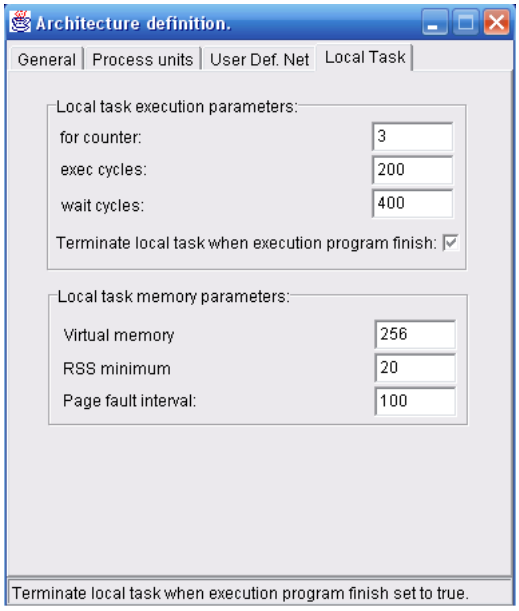

Figura 4.8: Paràmetres de la tasca local.

major part són de faltes de pàgina basicament perquè s'en produeixen moltes degut a l'interval de faltes pàgina baix de la tasca 2.

Per la simulació sense tasques locals s'ha utilitzat la mateixa arquitectura, el mateix programa paral·lel i el mateix *mapping* pero desactivant el *checkbox* "Local Task" als processadors. A la figura 4.11 es mostra els resultats gràfics de la simulació sense tasques locals.

Tal com es pot observar l'ordre en que acaben els processadors és el mateix amb la diferencia que els temps d'execució són inferiors. Finalitza més rapidament perquè els processadors no tenen assignades tasques locals que degraden el rendiment de l'execució del programa paral·lel. Per veure els temps d'execució exactes a les figures 4.12, 4.13 es mostren els resultats amb text de la simulació amb tasques locals i sense respectivament.

Si comparem els temps de finalització de la simulació podem observar que per la primera simulació amb tasques locals és de 3850 cicles i per la segona simulació sense tasques locals és de 2900 cicles amb una diferencia de 950 cicles, el 24,67 % més respecte la primera simulació.

El primer processador en acabar és el número 2 en les dues simulacions. A la primera finalitza amb 1120 cicles i a la segona amb 884 cicles, el 21,07% més respecte el temps de finalització del segon processador de la primera simulació. El número de faltes de pàgina que és produeixen són poques en les dues simulacions. La diferència de faltes de pàgina és igual a 3. El temps de sobrecarrega de faltes de pàgina del processador 2 a les dues simulacions, és de 120 cicles i 84 cicles respectivament, un 10,71% de sobrecarrega en la primera simulació i un 9,50% en la segona simulació respecte el temps de finalització d'aquest processador. Influeix bastant al rendiment general d'aquest processador perquè

```
<?xml version="1.0"?>
<!-- Architecture definition created by ESPPADA tool v1.0 -->
<architecture name="Arquitectura 4 processadors i memoria">
  <cpu_base value="1.0"/>
  <context_change value="0"/>
  <memory_enabled value="true"/>
  <local_task for="3" exec="200" wait="400" terminate="false"
virtual_memory="256" rss_minimum="20" pagefault_interval="100"/>
  <net type="full_connect">
    <delay value="1.0"/>
  </net>
  <planification method="round_robin" quantum="200"/>
<!-- Process units definition -->
  <processor id="1">
    <cpu_period value="1.0"/>
    <memory_size value="128"/>
    <pagefault_overhead value="10"/>
    <local_task value="true"/>
  </processor>
  <processor id="2">
    <cpu_period value="1.0"/>
    <memory_size value="128"/>
    <pagefault_overhead value="12"/>
    <local_task value="true"/>
  </processor>
  <processor id="3">
    <cpu_period value="1.0"/>
    <memory_size value="128"/>
    <pagefault_overhead value="12"/>
    <local_task value="true"/>
  </processor>
  <processor id="4">
    <cpu_period value="1.0"/>
    <memory_size value="128"/>
    <pagefault_overhead value="10"/>
    <local_task value="true"/>
  </processor>
</architecture>
```

Figura 4.9: Arquitectura amb tasques locals.

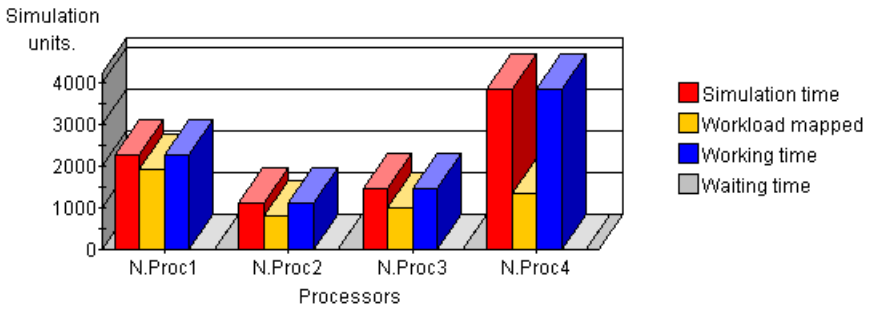

Figura 4.10: Resultats gràfics de la simulació amb tasques locals.

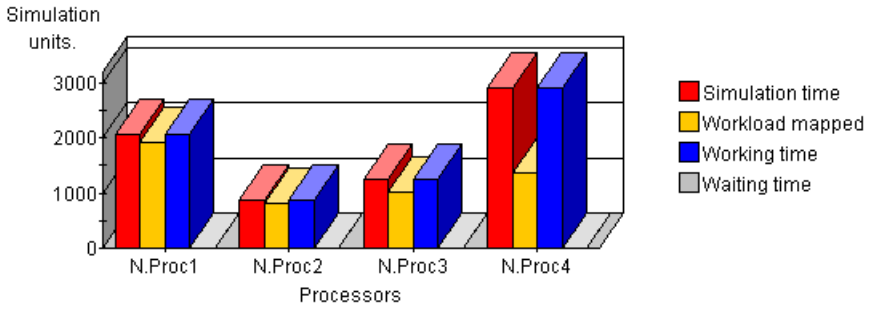

Figura 4.11: Resultats gràfics de la simulació sense tasques locals.

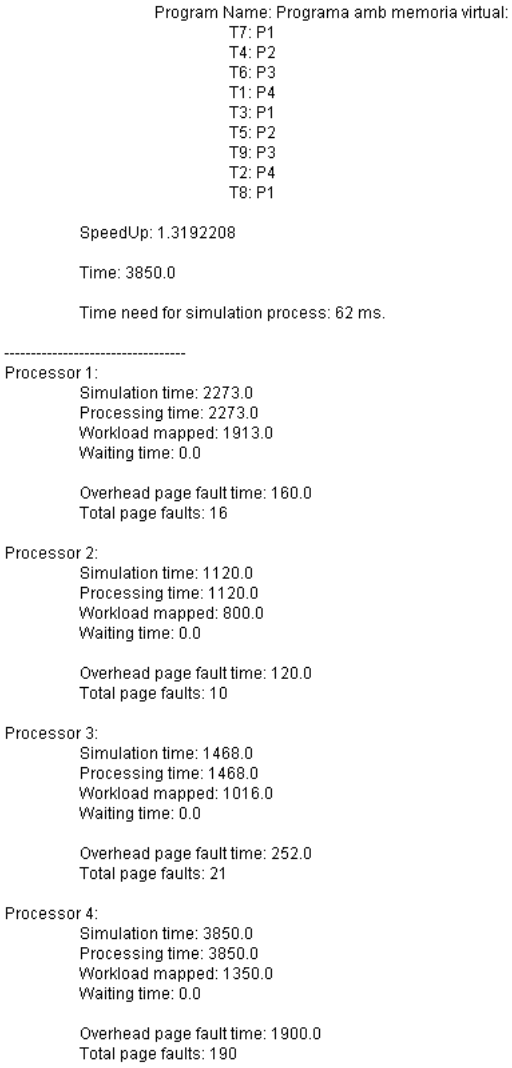

Figura 4.12: Resultats amb text de la simulació amb tasques locals.

```
Mapping:

        Program Name: Programa amb memoria virtual:
                T7: P1
                T4: P2
                T6: P3
                T1: P4
                T3: P1
                T5: P2
                T9: P3
                T2: P4
                T8: P1

        SpeedUp: 1.7513793

        Time: 2900.0

        Time need for simulation process: 32 ms.

---------------------------------
Processor 1:
        Simulation time: 2053.0
        Processing time: 2053.0
        Workload mapped: 1913.0
        Waiting time: 0.0

        Overhead page fault time: 140.0
        Total page faults: 14

Processor 2:
        Simulation time: 884.0
        Processing time: 884.0
        Workload mapped: 800.0
        Waiting time: 0.0

        Overhead page fault time: 84.0
        Total page faults: 7

Processor 3:
        Simulation time: 1256.0
        Processing time: 1256.0
        Workload mapped: 1016.0
        Waiting time: 0.0

        Overhead page fault time: 240.0
        Total page faults: 20

Processor 4:
        Simulation time: 2900.0
        Processing time: 2900.0
        Workload mapped: 1350.0
        Waiting time: 0.0

        Overhead page fault time: 1550.0
        Total page faults: 155
```

Figura 4.13: Resultats amb text de la simulació sense tasques locals.

la sobrecarrega de faltes de pàgina és relativament bastant alta (12 cicles) comparat amb els cicles d'execució de les tasques assignades al processador. Si s'hagués posat menys cicles de sobrecarrega llavors no hagues afectat tant al rendiment del processador a les dues simulacions.

El segon processador en finalitzar és el número 3 en les dues simulacions. En la primera simulació finalitza amb 1468 cicles i en la segona amb 1256 cicles, el 14,44% més. El número de faltes de pàgina és una mica més alt que en el processador 2. La diferencia no es nota gaire entre les dues simulacions perquè es produeixen poques faltes de pàgina de la tasca local. El temps de sobrecarrega de faltes de pàgina és de 252 a la primera i 240 a la segona, un 17,16% respecte el temps de finalització d'aquest processador a la primera simulació i un 19,10% respecte el temps de la segona. Les faltes de pàgina afecten més al rendiment d'aquest processador que el número 2.

El tercer processador en acabar és el número 1 en les dues simulacions. En la primera finalitza al cap de 2273 cicles i en la segona al cap de 2053 cicles, el 9,67% més. El número de faltes de pàgina a la simulació amb tasques locals és de 16 i a la segona simulació és de 14. Tampoc influeix gaire al rendiment del programa paral·lel les faltes de pàgina de la tasca local. El temps de sobrecàrrega de faltes de pàgina és de 160 cicles a la primera i 140 cicles a la segona, un 7,03% més respecte al rendiment general del processador amb tasca local i un 6,81% respecte la segona sense tasca local. Aquests valors són relativament baixos comparat amb els dos anteriors processadors perquè l'interval de faltes de pàgines de les tasques assignades que són la 7 i la 3 són alts comparats amb el de les altres tasques i pertant la freqüència de faltes de pàgina és mes baixa.

L'últim processador en finalitzar és el número 4. En la primera simulació finalitza al cap de 3850 cicles i en la segona al cap de 2900 cicles, el 24,67% més. En aquest processador és on afecta més el rendiment de les tasques paral·leles a causa de la tasca local perqué les tasques número 1 i 2 que són les que estan assignades a aquest processador contenen un total de cicles d'execució de 1350 cicles, un valor alt comparat amb les restants tasques mapejades i a conseqüència influeix més la tasca local. Les faltes de pàgina que es produeixen són 190 en la primera i 155 en la segona, el 18,42% més. Pels resultats és pot veure que les faltes de pàgina de la tasca local també afecten al rendiment. El temps de sobrecarrega de faltes de pàgina també es elevat en aquest processador a les dues simulacions, 1900 cicles i 1550 cicles respectivament, el 49,35% respecte al temps d'aquest processador de la primera i el 53,44% respecte la segona. Aquests valors són els més alts de les dues simulacions, perquè es produeixen moltes faltes de pàgina degut a l'interval de faltes de pàgina baixíssim de la tasca 2.

4.5.2 Experimentació de la multiprogramació amb tasques paral·leles

En aquest apartat s'exposa els resultats de la simulació de quatre programes paral·leles carregats en el simulador ESPPADA. Cal destacar que aquests programes contenen primitives de comunicacions per tal de demostrar que funciona correctament la coherència del programa canviant els identificadors originals de les tasques i primitives per uns identificadors generats no repetits.

La definició dels quatre programes paral·lels carregats es mostren en les figures 4.14,

4.15, 4.16, 4.17.

El primer programa conté quatre tasques amb els paràmetres de memòria virtual especificats. La primera tasca primerament s'executa la major part dels seus cicles (180 cicles) i envia tres missatges a les restants tasques. Després espera rebre un missatge de la segona tasca i torna a executar-se pocs cicles (20 cicles). La segona tasca realitza el mateix però en diferent ordre. La tercera tasca reb els missatges de les dues tasques anteriors intercaladament per dos primitives d'execució de 100 cicles. La quarta tasca reb primer el missatge de la primera tasca, s'executa 200 cicles i després reb el missatge de la segona tasca. Com es pot observar les quatre tasques tenen primitives de comunicació per demostrar que funciona les comunicacions del simulador carregant varis programes paral·lels.

El segon programa paral·lel realitza el mateix que el primer programa pero en diferent ordre. També conté 4 tasques paral·leles. La primera tasca espera rebre dos missatges de dos altres tasques separades per primitives d'execució. La segona tasca també espera rebre dos missatges de les dos ultimes tasques separades per primitives d'execució. La tercera i quarta tasca són les que envien els missatges a les restants tasques i també esperen rebre un missatge de la tasca quarta i tercera en diferent ordre.

El tercer programa paral·lel esta format per tres tasques. Totes les tres tasques tenen primitives d'execució i a més les dues primeres tenen una primitiva *Send* que envia un missatge a la tercera tasca. La tercera tasca té dues primitives de recepció per rebre els missatges de les dues tasques anteriors.

El últim i quart programa està format per dues tasques que solament con tenen primitives d'execució per fer anar la gestió de la memòria del sistema juntament amb els altres programes paral·lels.

L'arquitectura de l'entorn no s'exposa amb llenguatge XML perquè és bastant semblant a la de l'apartat anterior. També s'han definit 4 processadors amb 128 unitats de memòria. A més contenen tasques locals però no a tots els processadors, sols al número 1 i 3 que són els que tenen una sobrecàrrega de faltes de pàgina de 5 cicles, mentre que els altres de 10 cicles. La planificació de les tasques és *Round Robin* amb un quantum de 75 cicles. El mapping assignat és el Round Robin que reparteix les tasques de cada programa entre els 4 processadors.

Després s'ha realitzat la simulació i els resultats gràfics i amb text són el que es mostra a les figures 4.18, 4.19 i 4.20.

En la gràfica es pot observar que el primer processador en acabar és el número 1, el segon el número 2, el tercer el número 4 i l'últim el número 3. El processador que ha necessitat més temps d'espera per les comunicacions és el número 4, el segon el número 3, el tercer el número 1 i el quart el número 1. Això és degut a com estan col·locades les tasques entre els diferents processadors.

Tal com es pot observar a la segona figura el *speed up* general dels 4 programes és bastant baix si tenim en compte que el màxim possible és de 4 unitats. El temps de simulació és de 1818 cicles, un valor força comú tenint en compte la quantitat de tasques assignades i que no hi han moltes comunicacions.

En la tercera figura es pot observar que el primer processador en acabar és el número 1 amb 1294 cicles. Aquest és caracteritza perquè és un dels processadors amb més tasques,

```
<?xml version="1.0"?>
<program name="Exemple 1 Multiprograma">
  <task id="1" virmem="50" rss_minimum="15" pagefault_interval="20">
    <exec run="180"/>
    <send id="0" task="5" cost="42"/>
    <send id="1" task="6" cost="42"/>
    <send id="2" task="4" cost="42"/>
    <receive id="0" task="5"/>
    <exec run="20"/>
  </task>
  <task id="5" virmem="150" rss_minimum="50" pagefault_interval="30">
    <receive id="0" task="1"/>
    <exec run="100"/>
    <send id="0" task="1" cost="42"/>
    <send id="1" task="4" cost="42"/>
    <exec run="100"/>
    <send id="2" task="6" cost="42"/>
  </task>
  <task id="6" virmem="200" rss_minimum="60" pagefault_interval="45">
    <receive id="1" task="1"/>
    <exec run="100"/>
    <receive id="2" task="5"/>
    <exec run="100"/>
  </task>
  <task id="4" virmem="75" rss_minimum="32" pagefault_interval="37">
    <receive id="2" task="1"/>
    <exec run="180"/>
    <exec run="20"/>
    <receive id="1" task="5"/>
  </task>
</program>
```

Figura 4.14: Definició amb XML del primer programa paral·lel.

```
<?xml version="1.0"?>
<program name="Exemple 2 Multiprograma">
  <task id="5" virmem="65" rss_minimum="35" pagefault_interval="100">
    <exec run="180"/>
    <receive id="0" task="7"/>
    <exec run="20"/>
    <receive id="0" task="8"/>
  </task>
  <task id="6" virmem="128" rss_minimum="50" pagefault_interval="75">
    <receive id="1" task="7"/>
    <exec run="100"/>
    <receive id="1" task="8"/>
    <exec run="100"/>
  </task>
  <task id="7" virmem="132" rss_minimum="40" pagefault_interval="50">
    <send id="0" task="5" cost="42"/>
    <send id="1" task="6" cost="42"/>
    <send id="2" task="8" cost="42"/>
    <exec run="100"/>
    <receive id="2" task="8"/>
    <exec run="100"/>
  </task>
  <task id="8" virmem="50" rss_minimum="50" pagefault_interval="190">
    <receive id="2" task="7"/>
    <exec run="180"/>
    <send id="0" task="5" cost="42"/>
    <send id="1" task="6" cost="42"/>
    <send id="2" task="7" cost="42"/>
    <exec run="20"/>
  </task>
</program>
```

Figura 4.15: Definició amb XML del segon programa paral·lel.

```
<?xml version="1.0"?>
<program name="Exemple 3 Multiprograma">
  <task id="1" virmem="150" rss_minimum="15" pagefault_interval="20">
    <exec run="180"/>
    <send id="0" task="3" cost="42"/>
    <exec run="20"/>
  </task>
  <task id="2" virmem="160" rss_minimum="20" pagefault_interval="30">
    <exec run="150"/>
    <send id="1" task="3" cost="42"/>
    <exec run="50"/>
  </task>
  <task id="3" virmem="170" rss_minimum="35" pagefault_interval="45">
    <receive id="0" task="1"/>
    <exec run="100"/>
    <receive id="1" task="2"/>
    <exec run="100"/>
  </task>
</program>
```

Figura 4.16: Definició amb XML del tercer programa paral·lel.

```
<?xml version="1.0"?>
<program name="Exemple 4 Multiprograma">
  <task id="1" virmem="200" rss_minimum="35" pagefault_interval="15">
    <exec run="200"/>
  </task>
  <task id="2" virmem="250" rss_minimum="10" pagefault_interval="10">
    <exec run="250"/>
  </task>
</program>
```

Figura 4.17: Definició amb XML del quart programa paral·lel.

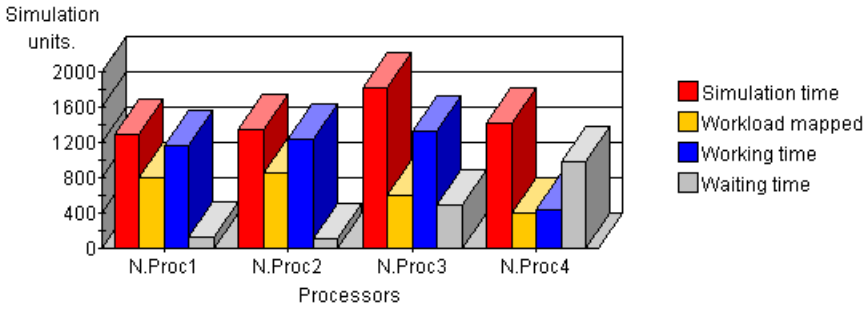

Figura 4.18: Resultats gràfics de la simulació amb multiprogrames.

```
----------------------------
 Simulation Data.
----------------------------

        Architecture: Arquitectura 4 processadors amb  memoria i tasques locals

        Programs List:
                Exemple 1 Multiprograma
                Exemple 2 Multiprograma
                Exemple 3 Multiprograma
                Exemple 4 Multiprograma

        Mapping:

                Program Name: Exemple 1 Multiprograma:
                        T1: P1
                        T5: P2
                        T6: P3
                        T4: P4
                Program Name: Exemple 2 Multiprograma:
                        T5: P1
                        T6: P2
                        T7: P3
                        T8: P4
                Program Name: Exemple 3 Multiprograma:
                        T1: P1
                        T2: P2
                        T3: P3
                Program Name: Exemple 4 Multiprograma:
                        T1: P1
                        T2: P2

        SpeedUp: 1.4568444

        Time: 1819.0

        Time need for simulation process: 157 ms.
```

Figura 4.19: Primera part dels resultats amb text de la simulació amb multiprogrames.

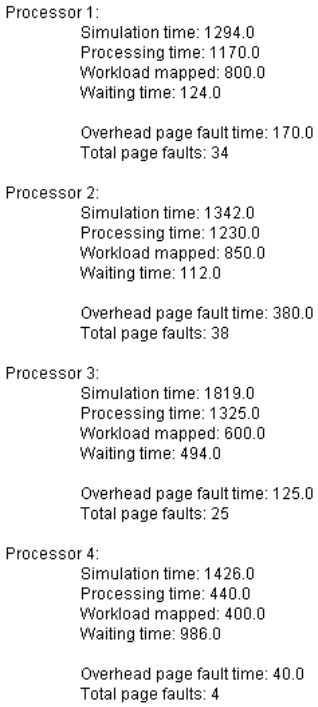

Figura 4.20: Segona part dels resultats amb text de la simulació amb multiprogrames.

té un workload mapped de 800 cicles i s'han produït bastantes faltes de pàgina, 34. La causa que ha fet rendir aquest processador és que per les comunicacions tant sols ha tingut d'esperar-se 124 cicles, el segon valor més baix dels 4 processadors.

El processador número 2 és el segon en acabar amb un temps de 1342 cicles tot i que és el processador amb més faltes de pàgina i té el workload mapped més alt però la comunicació és molt ràpida.

El processador 4 és el tercer en finalitzar amb 1426 cicles. En aquest cas el rendiment es degrada no per les faltes de pàgina que tant sols es produeixen 4 ni pel workload mapped que és de 400 cicles sinó per l'espera en les comunicacions que té un total de 986 cicles.

L'últim processador en acabar és el número 3 amb 1819 cicles. En aquest cas el que perjudica el seu rendiment són les faltes de pàgina que s'en produeixen 25 i el temps d'espera de les comunicacions que és de 494 cicles i a més la tasca local.

Pels resultats obtinguts es pot veure que les tasques locals que estaven assignades en els processadors 1 i 3 han afectat molt en el processador 3 i molt poc en el processador 1.

D'aquesta part de l'experimentació amb multiprogrames podem concloure que el rendiment individual de cada programa baixa perquè les tasques comparteixen CPU i a més tenen els seus requisits de memòria que afecten també al rendiment i si afegim tasques dels usuaris locals encara provoca més baix rendiment. Per aquest motiu el speedup dels 4 programes és baix.

5 Algorismes de mapping adaptats a l'entorn Multiprogramat

5.1 Introducció

Un factor important per millorar el rendiment d'un entorn multiprogramat disposant de les mateixes característiques de hardware és seleccionar el *mapping* més adecuat. Per això, en aquest capítol es presenten dos algorismes de mapeig que tenen en compte les característiques dels processadors.

El primer algorisme de *mapping* realitza una assignació de tasques segons el tamany de la memòria principal de cada processador. Com que es pot saber d'antemà les necessitats de memòria virtual de cada tasca del programa paral·lel doncs selecciona aquell processador amb més memòria principal disponible per fer l'assignació. En la secció 5.2 s'explica detalladament aquest algorisme.

El segon algorisme de *mapping* realitza una assignació de tasques segons el cost de CPU de cada tasca que es calcula sumant els cicles de totes les primitives d'execució, llavors selecciona aquell processador que té menys cost de CPU assignat. El cost de CPU és multiplica pel període de CPU per tenir en compte la velocitat del processador. En la secció 5.3 es comenta el disseny d'aquest algorisme.

Aquests algorismes són ideals quan es fa complexa la assignació de tasques als processadors degut a la multitut de tasques que poden tenir varis programes paral·lels carregats i la quantitat de processadors amb diferentes característiques hardware.

En la secció 5.4 s'explica detalladament la implementació duta a terme en el simulador ESPPADA.

Per tal de mostrar els resultats obtinguts en la secció 5.5 es comenta l'Experimentació realitzada amb aquests dos algorismes.

5.2 Algorisme de balanceig de memòria

L'heurística de l'algorisme de balanceig de memòria és buscar la CPU amb el màxim de memòria disponible i si dos o més processadors tenen la mateixa memòria disponible llavors seleccionar el primer. En la figura 5.1 s'exposa aquest algorisme.

Aquest algorisme accepta com a entrada dues llistes, una de tasques i l'altra de processadors que estiguin ordenades de més a menys per la memòria virtual que necessiten i per la memòria principal disponible respectivament.

La idea principal de l'algorisme és recorrer tota la llista ordenades de tasques seqüencialment i per cada tasca inserir-la en el primer processador de la llista de processadors que

Entrada:
$L_{tasques}$: Llista ordenada de tasques segons la memòria virtual requerida.
$L_{processadors}$: Llista ordenada de processadors segons la memòria disponible.

Algorisme Mapping_Memoria és:
{
  T_{actual} :=Primer_element($L_{tasques}$)
  Mentre ($T_{actual} \neq NULL$) fer
  {
    P_{actual}:=Primer_element($L_{processadors}$)
    Assigna_Mapping(P_{actual}, T_{actual})
    Elimina_primer_element($L_{processadors}$)
    Elimina_primer_element($L_{tasques}$)
    Inserta_ordenadament(Memoria_disponible(P_{actual}),P_{actual},$L_{processadors}$)
    T_{actual} :=Primer_element($L_{tasques}$)
  }
}
Retorna Mapping

Figura 5.1: Algorisme de balanceig de memòria

és el que té més memoria disponible. Com que les llistes estan ordenades ens estalviem de recorrer-les varis cops en cada iteració. L'acció que realitza l'algorisme és inicialment obtenir la primera tasca de la llista que és la que necessita més memòria virtual i dins el bucle el primer processador que és el que conté més memòria principal, i llavors assignar el *mapping* d'aquesta tasca. A continuació elimina el primer processador de la llista de processadors que és el que s'ha utilitzat per fer el *mapping*. Després elimina la primera tasca de la llista de tasques ordenada perque ja no es farà servir més. A continuació insereix ordenadament a la llista de processadors el processador que s'ha fet servir per l'assignació segons la memòria disponible actual. Recordem que els processadors estaven ordenats de més a menys memòria principal. Probablement el processador ja no caigui en les primeres posicions de la llista perquè s'ha reduït la memòria disponible. I finalment obté el primer processador amb més memòria disponible per la següent iteració. Aixins va fent el bucle a cada iteració fins haver assignat totes les tasques a un processador. Al finalitzar retorna el *mapping* calculat.

5.3 Algorisme de balanceig de CPU

L'algorisme de balanceig de CPU no té en compte els principals paràmetres de la memòria que és la part més important realitzada per aquest projecte de fi de carrera sinó que té en compte el cost de CPU de les tasques, el cost de CPU assignat a cada processador i el paràmetre de periode de CPU que ens indica el temps absolut que tardarà a computar les tasques.

Entrada:
$L_{tasques}$: Llista ordenada de tasques segons el cost de CPU assignat a cada tasca.
$L_{processadors}$: Llista ordenada de processadors segons el cost de CPU multiplicat pel periode.

Algorisme Mapping_CPU és:
{
  T_{actual} :=Primer_element($L_{tasques}$)
  Mentre ($T_{actual} \neq NULL$) fer
  {
    P_{actual}:=Primer_element($L_{processadors}$)
    Assigna_Mapping(P_{actual}, T_{actual})
    Elimina_primer_element($L_{processadors}$)
    Elimina_primer_element($L_{tasques}$)
    Inserta_ordenadament(Cost_CPU(P_{actual}) / Factor_Periode(P_{actual}),P_{actual},$L_{processadors}$)
    T_{actual} :=Primer_element($L_{tasqies}$)
  }
}
Retorna Mapping

Figura 5.2: Algorisme de balanceig de CPU

L'heurística principal d'aquest algorisme és buscar el processador amb menys cost de CPU assignat multiplicat pel seu periode entre cicle i cicle i si existeix varis processadors amb el mateix cost llavors seleccionar el primer. En la figura 5.2 es mostra l'algorisme de balanceig de CPU. Cal destacar que l'algorisme és molt semblant a l'anterior algorisme simplement canviant el paràmetre de la memòria pel cost de CPU de cada tasca.

Aquest algorisme també accepta com a entrada dues llistes ordenades de tasques i processadors segons el cost de CPU de cada tasca i el cost de CPU assignat a cada processador dividit pel factor de periode de CPU que es el paràmetre *Compensation Factor* comentat al capítol 2 .

L'algorisme està composat per un bucle on a cada iteració obte la primera tasca de la llista de tasques que és la tasca amb més cost de CPU i el primer processador que és el que té menys cost de CPU assignat dividit pel seu factor de periode. Llavors fa el *mapping* de la tasca actual amb el processador actual. Després elimina el primer element de la llista de processadors per ser insertada posteriorment en la posició que li pertocarà i elimina el primer element de la llista de tasques que és la tasca actual perquè ja no es necessitarà més. A continuacio insereix el processador ordenadament segons el cost de CPU actual que té assignat dividit pel seu factor de periode en la llista de processadors. D'aquesta manera tenim sempre la llista ordenada i no es necessari recorre tota la llista de processadors a cada iteració per buscar quin és el processador amb menys cost. Després obté el primer element de la llista de tasques per la següent iteració. Finalment quan acaba el bucle retorna el *mapping* calculat.

Tipus	Nom de la funció	Descripció de la funció
public	*BalancedMemoryMap(Mapping Map1, Architecture Archi1, Program Prog1, MultiProgram MProg1)*	Constructor de la classe *BalancedMemoryMap*. Passa a la classe els paràmetres de mapping, l'arquitectura, el programa seleccionat per fer el mapping balancejat i la classe *MultiProgram* per saber la memoria virtual de les tasques.
private void	*CreateSortedLists()*	Crea les llistes ordenades de processadors i tasques segons la seva memòria.
private long	*FindMemoryUsed(Processor Proc)*	Donat un processador busca la memoria utilitzada.
private void	*InsertProcessorList(long memfree, Processor Proc)*	Insertada ordenadament un processador a la llista de processadors segons la memòria disponible.
private void	*InsertTaskList(Task task)*	Insertada ordenadament una tasca a la llista de tasques segons la quantitat de memòria virtual que necessita.
private boolean	*CheckErrors()*	Testeja si hi ha algun error abans de fer el mapping.
public void	*CalculateMap()*	Realitza el mapping de balanceig de memòria.

Taula 5.1: Taula de funcions de la classe *BalancedMemoryMap*

5.4 Implementació

Aquesta secció es composa de dues parts: La implementació de l'algorisme de balanceig de memòria i la implementació de l'algorisme de balanceig de CPU. En la subsecció 5.4.1 es mostra la primera implementació i en la subsecció 5.4.2 es mostra la segona implementació.

5.4.1 Implementació de l'algorisme de balanceig de memòria

Tot l'algorisme està definit en la classe *BalancedMemoryMap* escrita en el fitxer *BalancedMemoryMap.java*. En la taula 5.1 es mostren les funcions d'aquesta classe. Solament es mostren les funcions d'aquesta classe perquè els atributs es mostren a continuació dins del codi font de les funcions.

La funció *CreateSortedLists()* crea les llistes ordenades de processadors i tasques necessàries per l'algorisme. A continuació es mostra el seu codi font:

```
private void CreateSortedLists()   {
```

```
    Nodo NodeProc=Archi.Processors_List.firstnodo;
    long TotalMem,MemUsed;
    while (NodeProc!=null)
    { Processor Proc = (Processor) NodeProc.getNodo();
      TotalMem = Proc.get_Memory_Size();
      MemUsed = FindMemoryUsed(Proc);
      InsertProcessorList(TotalMem-MemUsed,Proc);
      NodeProc=NodeProc.next;
    }
    Nodo NodeTask=Prog.Tasks.firstnodo;
    long virtmem;
    while (NodeTask!=null)
    { Task task = (Task) NodeTask.getNodo();
      InsertTaskList(task);
      NodeTask=NodeTask.next;
    }
  }
```

Primerament recull de l'arquitectura el primer node que conté el primer processador de la llista de processadors i entra dins del bucle si és diferent de *null*. Llavors obte la memòria principal del processador i la memòria utilitzada que s'ha de calcular a través de la funció *FindMemoryUsed()* . Després l'inserta ordenadament dins de la llista de processadors a través de la funció *InsertProcessorList()* passant-li per parametres la memòria disponible i el processador. A continuació va al següent processador per ser insertat dins la llista a la següent iteració i així va fent seqüencialment fins insertar tots els processadors.

A continuació realitza el mateix però per les tasques. Recorre totes les tasques del programa seleccionat i les va insertant ordenadament a través de la funció *InsertTaskList()*.

A continuació es mostra el codi font de la funció *InsertProcessorList()*.

```
  private void InsertProcessorList(long memfree,Processor Proc)
  { ProcessorFreeMemory procfreemem=new ProcessorFreeMemory();
    procfreemem.Proc=Proc;
    procfreemem.memfree=memfree;
    if (ProcList.vacio()) {
      ProcList.firstnodo=ProcList.lastnodo=new Nodo  (procfreemem,null);
    } else
    { Nodo posi,posi1;
      posi = ProcList.firstnodo;
      posi1 = ProcList.firstnodo;
      while (posi!=null)
      { ProcessorFreeMemory procfreemem1 = (ProcessorFreeMemory) posi.getNodo();
        if (procfreemem1.memfree<memfree)
          break;
        posi1=posi;
```

```
        posi=posi.next;
      }
      if (posi!=null)
      {
        if (posi!=posi1)
        { posi1.next = new Nodo (procfreemem, posi);
        }
        else {
          ProcList.firstnodo=new Nodo ( procfreemem, posi);
        }
      }
      else
      { ProcList.lastnodo=posi1.next=new Nodo ( procfreemem, null);
      }
    }
  }
```

Les accions que realitza aquesta funció és crear una instància de la classe *ProcessorFreeMemory* que conté solament dos atributs, *memfree* que indica la memòria lliure del processador i *Proc* que conté el processador. Serveix per tenir constància de quina és la memòria lliure del processador que és vol consultar a la llista ordenada de processadors. Després comprova si la llista està buida, en cas afirmatiu crea un nou node i l'insereix en la llista de processadors. En cas negatiu vol dir que la llista no està buida i pertant comproba dins un bucle tots els processadors fins trobar el primer processador que té la memòria disponible inferior al processador actual. Llavors insereix un nou node a la posició apuntada pel node *posi1* tenint en compte si és el primer node de la llista, està entremig o és l'últim. La funció *InsertTaskList()* és molt semblant a la funció comentada, simplement canvia els processadors per tasques, no insereix una nova classe que contingui la memòria virtual de la tasca sinó que de la mateixa tasca es pot obtenir directament la seva memòria virtual.

A continuació es mostra el codi font de la funció *CalculateMemoryMap()* que calcula el mapping balancejat de memòria:

```
public void CalculateMap()
{
  if (CheckErrors())
   return ;
  CreateSortedLists();
  Nodo nodetask=TaskList.firstnodo;
  while(nodetask!=null)
  { Task task=(Task)nodetask.getNodo();
    Nodo nodeproc=ProcList.firstnodo;
    ProcessorFreeMemory procfreemem=(ProcessorFreeMemory)nodeproc.getNodo();
    Assig assig=Map.BuscaAssig(procfreemem.Proc.get_Id());
```

```
      if (assig!=null)
      {
        if (!Map.isTaskAssigned(task.getId()))
        {
         assig.addAssign(task.getId());
         assig.add_cost(task.get_cost());
        }
      }
      else
      { if (!Map.isTaskAssigned(task.getId()))
        {
          Map.addNewProcessor(procfreemem.Proc.get_Id());
          assig = Map.BuscaAssig(procfreemem.Proc.get_Id());
          assig.addAssign(task.getId());
          assig.add_cost(task.get_cost());
        }
      }
      TaskList.removeFirst();
      ProcList.removeFirst();
      InsertProcessorList(procfreemem.memfree-task.getMemVirt(),procfreemem.Proc);
      nodetask=nodetask.next;
    }
  }
```

En primer lloc comprova si hi ha algun possible error, examinant que cada classe necessitada per l'algorisme no sigui *null*. Després crea les llistes ordenades de processadors i tasques a través de la funció ja comentada *CreateSortedLists()*. Llavors recorre cadascuna de les tasques per ser insertada dins el *mapping* amb el processador indicat. Al començament del bucle obté la primera instància de la classe *ProcessorFreeMemory* de la llista de processadors i comprova si existeix una assignació feta per aquest processador. En cas afirmatiu comprova que la tasca no estigui assignada anteriorment i en aquest cas la insereix. Com es pot observar l'algorisme té en compte el *mapping* anterior que no el modifica si ja existeix l'assignació feta d'aquesta tasca amb el processador. Altrament, en cas que no existeixi una assignacio feta d'aquest processador i la tasca no estigui assignada a un altre processador, crea una nova assignació amb l'identificador del nou processador i a continuació obté la nova assignació per inserir la tasca associada i el seu cost associat. Despres elimina el primer element de les llistes *TaskList* i *ProcList* que són les llistes ordenades. A continuació insereix ordenadament en la llista de processadors el processador actual amb la memòria disponible actual. Finalment obté el següent node de la llista i va realitzant seqüencialment el mateix procediment per totes les tasques fins que ja no existeix cap a la llista.

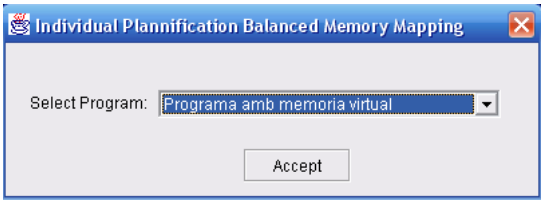

Figura 5.3: Selecció individual del programa per fer el mapping de balanceig de memòria.

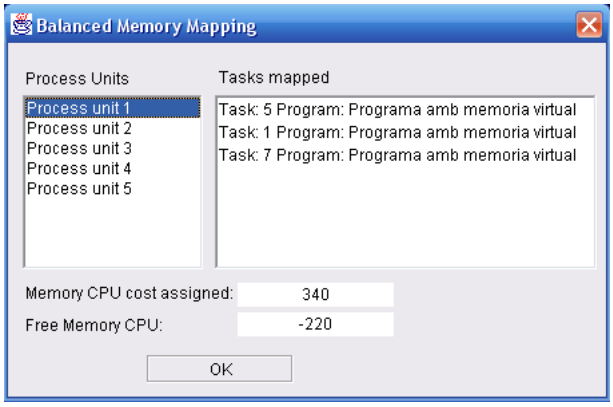

Figura 5.4: Formulari del mapping de balanceig de memòria

5.4.1.1 Disseny de l'interficie del mapping de balanceig de memòria

Quan es selecciona fer el mapping de balanceig de memòria al menú principal després d'haver creat una arquitectura i haver carregat un o varis programes paral·lels es mostra un formulari per seleccionar el programa que es vol fer el mapping. El formulari es mostra a la figura 5.3. El mapping es realitza individualment per programa però com s'ha dit anteriorment manté el mapping anterior per tal de ser mesclat amb varis programes.

La classe que representa a aquest formulari s'anomena *Marco_ IndividualPlanification* definida en el fitxer *Marco_ IndividualPlanification.java*. Aquesta classe s'utilitza tant en el mapping de balanceig de memòria com en el de balanceig de CPU.

Les accions que realitza es llistar tots els programes carregats en el simulador, permetre la selecció d'un programa i retornar el programa seleccionat al codi que l'ha cridat després d'apretar el botó acceptar. Després es crida la classe *BalancedMemoryMap* que conté l'algorisme i realitza el *mapping*.

A continuació es mostra un altre formulari per veure els resultats del mapping amb el cost de memòria assignat en cada processador, la memòria lliure de cada processador i les tasques assignades en cada processador. En la figura 5.4 es mostra aquest formulari. A la part de sota apareix indicat el cost de memòria assignat en el processador actual i la seva memòria lliure.

5.4.2 Implementació de l'algorisme de balanceig de CPU

L'algorisme es troba definit en la classe *BalancedCPUMap* escrita en el fitxer *BalancedCPUMap.java*. La implementació de l'algorisme és molt semblant a la de l'anterior algorisme. Aquesta classe conté els mateixos atributs i funcions excepte la funció *FindCostCPUUsed(Processor Proc)* que substitueix la funció *FindMemoryUsed(Processor Proc)* i calcula el cost de CPU assignat al processador passat per paràmetres i el retorna. Aquesta última l'utilitza la funció *CreateSortedLists()* per passar-li aquesta informació a la llista.

Les dues llistes ordenades contenen dues classes diferents a cada node amb la finalitat de guardar el cost de les tasques en la llista de tasques ja que sinó s'hauria de calcular a cada moment i el cost de CPU assignat en els processadors en la llista de processadors. La primera classe s'anomena *TaskCostProcessor* i conté els atributs *Proc* de tipus *Processor* que guarda el processador i un *float* anomenat *Cost* que guarda el cost associat. Les instàncies d'aquesta classe són les que s'emmagatzemen a la llista de processadors *ProcList*. L'altra segona classe s'anomena *TaskCostTask* i conté els atributs *Task* de tipus *Task* per guardar la tasca i el *float Cost* per guarda el cost de CPU d'aquesta tasca. Aquesta classe es guarda a la llista *TaskList* de tasques ordenades de menys a més cost de CPU.

El codi de la funció *InsertProcessorList(float cost,Processor Proc)* que se li passa per paràmetre el cost assignat en el processador i el processador és molt similar al codi d'aquesta funció de l'altre algorisme. Simplement canvia la classe que s'emmagatzema els nodes de la llista i l'ordre en que es guarden els processadors que és de menys a més cost de CPU assignat. L'altra funció d'accions similars *InsertTaskList(Task task)* obté inicialment el cost de la tasca a través de la funció *get_ cost()* de la classe *Task*. El cost de CPU s'obte sumant els cicles de totes les primitives *Exec* i si estan dins un bucle *for* o una sentencia *if* multiplicar-lo pel contador d'iteracions i per la probabilitat respectivament. L'ordre en que guarda les tasques és de més cost a menys cost perqué la primera tasca ha d'anar assignada en el primer processador amb menys cost.

La implementació de l'algorisme del mapping de balanceig de CPU és mostra a la figura 5.5.

La implementació de l'algorisme es molt semblant al de l'apartat anterior. Inicialment també comproba els possibles errors que hi puguin haver abans de fer el mapping. Segonament agafa la instància de la classe *TaskCostTask* del primer node de la llista de tasques per obtenir la tasca amb el cost de CPU més alt i dins el bucle agafa la primera instància de classe *ProcessorCostTask* de la llista que és el que té el processador amb el cost assignat més baix. A continuació si no existeix anteriorment un mapping fet de la tasca actual llavors assigna en el processador la nova tasca. Per fer-ho comprova si existeix una assignació feta en aquest processador. En cas afirmatiu comprova que la tasca no estigui anteriorment mapejada llavors assigna la nova tasca al processador i el seu cost associat sinó no fa res i deixa com estava el mapping anterior. Altrament en cas negatiu afegeix el processador en el mapping, busca la classe que representa l'assignació d'aquest processador i assigna la tasca i el seu cost. Després dins del bucle elimina el primer element de les llistes ordenades de tasques i processadors, llavors insereix el pro-

```
public void CalculateMap()
{ if (CheckErrors())          return ;
  CreateSortedLists();
  Nodo nodecosttask=TaskList.firstnodo;
  while(nodecosttask!=null)
  {
    TaskCostTask ctask=(TaskCostTask)nodecosttask.getNodo();
    Task task = ctask.Task;
    Nodo nodeproc=ProcList.firstnodo;
    TaskCostProcessor proccost=(TaskCostProcessor)nodeproc.getNodo();
    Assig assig=Map.BuscaAssig(proccost.Proc.get_Id());
    if (assig!=null)
    {
      if (!Map.isTaskAssigned(task.getId()))
      {
        assig.addAssign(task.getId());
        assig.add_cost(task.get_cost());
      }
    }
    else
    {
      if (!Map.isTaskAssigned(task.getId()))
      {
        Map.addNewProcessor(proccost.Proc.get_Id());
        assig = Map.BuscaAssig(proccost.Proc.get_Id());
        assig.addAssign(task.getId());
        assig.add_cost(task.get_cost());
      }
    }
    TaskList.removeFirst();
    ProcList.removeFirst();
    InsertProcessorList(proccost.Cost+(task.get_cost()/
    proccost.Proc.get_CPU_Period()),proccost.Proc);
    nodecosttask=nodecosttask.next;
  }
 }
```

Figura 5.5: Implementació de l'algorisme de balanceig de CPU.

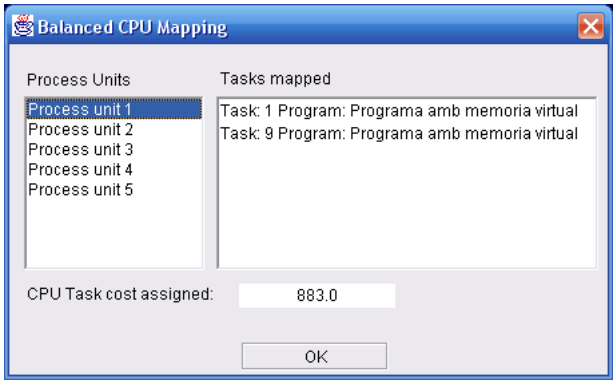

Figura 5.6: Disseny de l'interficie de l'algorisme de balanceig de CPU.

cessador que s'ha eliminat un altre cop en la llista en la posició que li pertoca segons el cost de CPU assignat dividit pel factor de període. Finalment busca el següent node de la llista de tasques per la proxima iteració. Quant hagi assignat totes les tasques, l'algorisme para i retorna el mapping calculat.

5.4.2.1 Disseny de l'interficie de l'algorisme de balanceig de CPU

La interficie de l'algorisme de balanceig de CPU es molt similar a la de l'algorisme de balanceig de memòria. Al començament també demana seleccionar el programa per realitzar el mapping individual. Després de seleccionar-lo apareix el mapping final amb un formulari i a la part de sota indica el cost de CPU assignat en cada processador.

En la figura 5.6 es mostra el disseny de l'interficie de l'algorisme de balanceig de CPU.

5.5 Experimentació

Aquesta secció es composa de dos apartats que realitzen l'experimentació dels dos algorismes de mapping implementats en el simulador ESPPADA.

En l'apartat 5.5.1 es comenta detalladament les proves realitzades amb l'algorisme de mapping de balanceig de memòria i es compara els resultats de la simulació amb la planificació Round Robin.

En l'apartat 5.5.2 s'explica les proves realitzades amb l'algorisme de balanceig de CPU i també es compara els resultats de la simulació amb la planificació Round Robin.

5.5.1 Experimentació amb l'algorisme de balanceig de memòria

L'algorisme de balanceig de memòria és més efectiu quan tenim varis processadors amb memòries principals de diferent tamany, moltes tasques a assignar i una sobrecàrrega de faltes de pàgina bastant alta. La funció principal de l'algorisme de mapping és evitar el màxim possible que s'alliberi memòria principal utilitzant l'algorisme de reemplaçament de pàgines, ja que a conseqüencia es produeixen més faltes de pàgina en les tasques

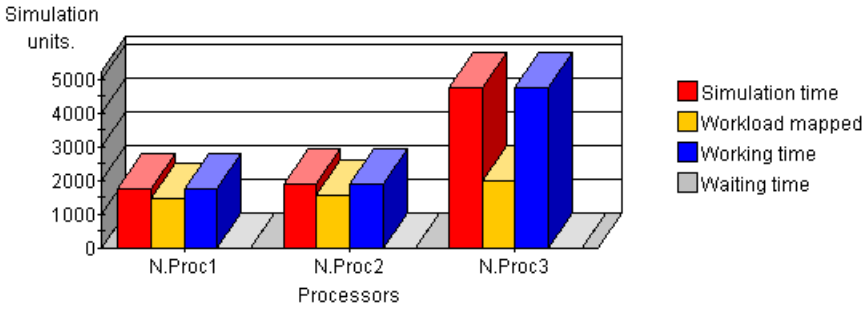

Figura 5.7: Resultats gràfics de la simulació amb el mapping de balanceig de memòria.

perquè aquestes necessiten la memòria que s'ha alliberat i pertant deteriora el rendiment del programa paral·lel degut a la sobrecàrrega total de les faltes de pàgines.

Si tenim varis processadors amb la mateixa memòria i poques tasques els resultats no seran tant diferents comparat amb una planificació distribuïda com la Round Robin perquè aquestes poden anar assignades a qualsevol processador sense tenir en compte la memòria virtual que necessiten perquè els processadors tenen la mateixa memòria. Es notarà més la diferència quan s'assignin moltes tasques ja que la memòria lliure dels processadors varia segons les tasques assignades i com que l'algorisme de mapping té en compte a cada moment quin és el maxim de memòria disponible a tots els processadors sempre buscarà el processador més adeqüat per cada tasca. Com a resultat final de fer el mapping tindrem els processadors amb la memòria disponible bastant igualada i per tant el paral·lelisme es notarà més i pujarà el *speed up*.

L'arquitectura utilitzada per les dues simulacions de diferent mapping conté tres processadors amb 256, 128 i 384 unitats de memòria respectivament, tots tres processadors amb una sobrecàrrega de faltes de pàgina de 30 cicles i sense tasca local i l'algorisme d'apropiació de CPU és el Round Robin amb un quantum de 200. Les altres parts de l'arquitectura no es comenten perquè no son utilitzades per la simulació.

Sols s'ha fet servir un programa paral·lel per mesurar l'efectivitat de l'algorisme, aquest és el mateix que l'utilitzat a l'Experimentació amb tasques locals, i es mostra la seva definició amb XML en la figura 4.7.

El mapping calculat per l'algorisme de balanceig de memòria és el següent: En el processador 1 les tasques 7,4,8 que és el que té 256 unitats de memòria; En el processador 2 les tasques 1,3,5 i té 128 unitats de memòria i en el processador 3 les tasques 6,9 i 2 amb 384 unitats de memòria.

Com es pot observar les tasques del processador 3 són les que necessiten més memòria, les del processador 1 que són les que estan en la meitat de la memòria necessaria i la del processador 2 que són les que en necessiten menys.

Eb la figura 5.7 es mostra els resultats gràfics de la simulació amb la planificació de balanceig de memòria.

Els resultats gràfics de la figura 5.7 ens indiquen que el primer processador en acabar és el número 1 amb 1763 cicles, el segon el número 2 amb 1900 cicles i el tercer el número 3

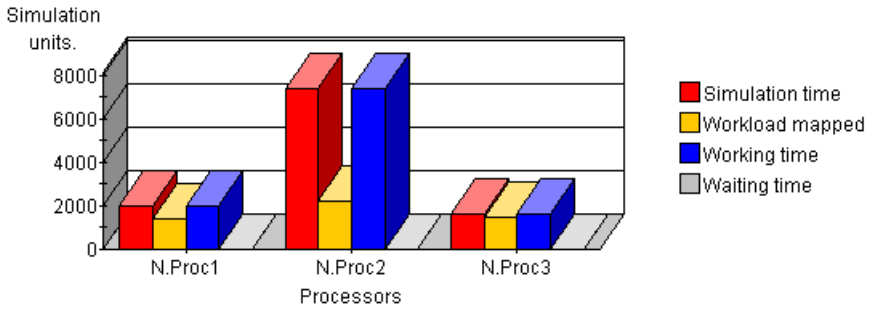

Figura 5.8: Resultats gràfics de la simulació amb el maping Round Robin.

amb 4746 cicles, una gran diferencia respecte els dos anteriors. Això es degut a que aquest processador conté la tasca 2 que produeix moltes faltes de pàgina perquè té el pagefault interval baix a 5 cicles i pertant augmenta molt el temps total de sobrecarrega de les faltes de pàgines. En els dos primers processadors el *Worlkload Mapped* es bastant semblant al *Simulation Time*, això vol dir que tot el temps de CPU es destina practicament a les primitives d'execució de les tasques. Al tercer processador hi ha molta diferencia de temps degut al temps de sobrecàrrega de faltes de pàgines.

El *speed up* no és gaire alt, 1,07 degut en general al temps de les faltes de pàgina sobretot del processador número 3 on el seu temps és del 57,5% respecte el temps total del processador 3. El processador 1 i 2 el temps de paginació es del 15,3% i 17,3%, valors relativament més baixos comparats amb el processador 3.

En la figura 5.8 es mostren els resultats gràfics de la simulació del mateix programa paral·lel i arquitectura però amb el maping Round Robin. El maping resultant és el següent: les tasques 7, 1 i 9 al processador 1; les tasques 4, 3 i 2 al processador 2 i les tasques 6, 5 i 8 al processador 3.

En aquest cas el primer processador en acabar és el número 3 amb 1653 cicles, després finalitza immediatament el número 1 amb 1956 cicles i finalment el número 2 que finalitza al cap de bastant temps, 7380 cicles a causa de la tasca 2 que produeix moltes faltes de pàgines.

El *speed up* en aquest cas és de 0,68 un 35,7% menys que en la primera simulació. El temps de paginació en el primer processador és del 27,6%, en el segon processador el 69,9% i el tercer processador el 12,7%. Són valors més alts que els de la simulació amb balanced memory mapping excepte el processador 3 que el mapping Round Robin ha assignat per cuasolitat tasques amb pocs requeriments de memòria virtual.

Si comparem els temps de finalització dels processadors de les dues simulacions amb diferents mappings podem observar que el processador 1 a la primera simulació triga un 9,8% menys respecte el mateix processador però de la segona simulació. El processador 2 triga un 74,2% menys i el processador 3 un 65,1% més. Els resultats que valen són els finals i si comparem els dos temps de simulació final podem veure que la primera simulació triga 35,69% menys que la segona.

Per tant podem concloure que l'algorisme de balanceig de memòria millora el rendiment

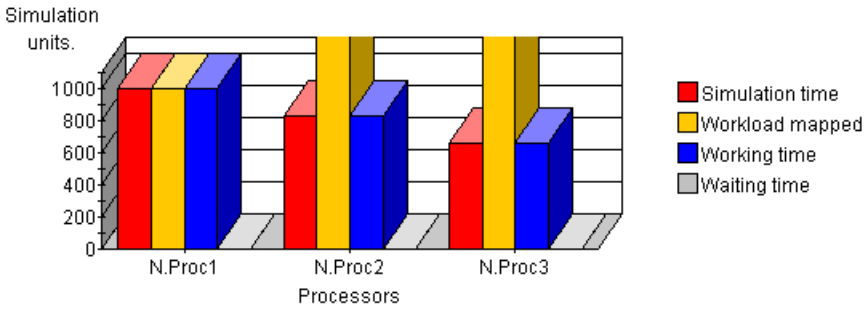

Figura 5.9: Resultats gràfics de la simulació amb el mapping de balanceig de CPU.

general dels programes paral·lels amb requeriments de memòria i si la memòria dels processadors està bastant limitada encara és més efectiu.

5.5.2 Experimentació amb l'algorisme de balanceig de CPU

L'algorisme de balanceig de CPU té en compte el cost de cada tasca i intenta igualar la suma dels costs de les tasques assignades a cada processador. Com més igualats estiguin aquests costs més rapid serà el rendiment de la màquina paral·lela perquè tots els processadors acabar en un temps bastant similar ja que estarà repartit entre tots els processadors. L'algorisme és més efectiu quan els processadors són de diferentes velocitats perquè té en compte la velocitat de compút de cada processador i si aquest és més ràpid li assignarà més tasques en el processador.

Per realitzar aquesta experimentació s'ha utilitzat el mateix programa paral·lel que l'experimentació de balanceig de memòria. El que canvia és l'arquitectura que ara la memòria no està activa i a més existeix 3 processadors amb diferentes velocitats, el primer amb el paràmetre **cpu_period** igual a 1, el segon amb aquest paràmetre igual a 2 i el tercer igual a 3. Això vol dir que el segon i tercer processador van el doble i el triple de ràpids que el primer processador. La resta de paràmetres de l'arquitectura són iguals a l'anterior experimentació.

Després s'ha aplicat el maping de balanceig de CPU i el resultat és el següent: El processador 1 té assignada la tasca 2, el processador 2 té assignades les tasques 4, 6 i 3 i el processador 3 les tasques 7, 1, 5, 9 i 8.

En la figura 5.9 es mostra els resultats gràfics de la simulació amb el maping de balanceig de CPU.

Com es pot observar el primer processador en acabar es el número 3 amb 664 cicles, el segon el processador 2 amb 825,75 cicles i el tercer el processador 1 amb 1000 cicles. En el primer processador el *Workload Mapped* és igual al *Simulation Time* perquè el periode de CPU és igual a 1. En canvi als altres processadors el *Workload Mapped* és més alt que el *Simulation Time* perquè el processador duplica o triplica la velocitat d'execució d'aquesta carga de treball i per tant el temps de simulació és mes baix.

En aquesta simulació el speed up es de 5,079 molt alt tenint en compte que el rendiment

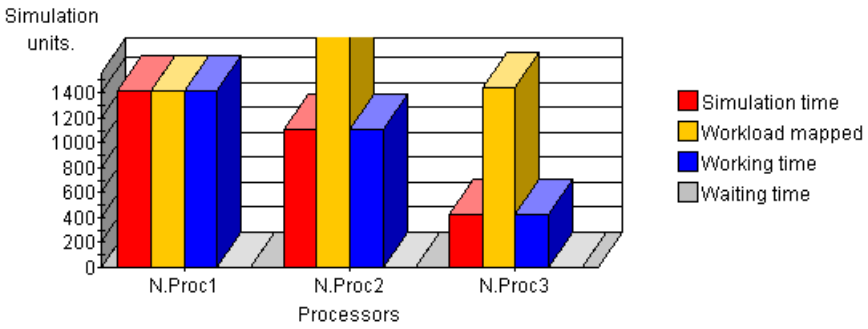

Figura 5.10: Resultats gràfics de la simulació amb el mapping Round Robin.

efectiu dels tres processadors equival a 6 processadors amb periode de cpu igual a 1. El temps final de simulació és el del processador més lent, el número 3 es a dir 1000 cicles.

Ara continuarem l'analisi dels resultats de l'experimentació de la simulació amb el mapping Round Robin per comparar-los i així demostrar que l'algorisme de mapping de balanceig de CPU és millor. En la figura 5.10 es mostra els resultats gràfics de la simulació amb el mapping Round Robin. El mapping resultant és el següent: les tasques 7, 1 i 9 en el processador 1, les tasques 4, 3 i 2 en el processador 2 i les tasques 6, 5 i 8 en el processador 3.

El primer processador en acabar és el número 3 amb bastanta diferencia respecte els altres. Els cicles de finalització d'aquest processador són de 427 unitats. El segon processador en acabar és el número 2 amb 1105 cicles i el tercer el processador 1 amb 1416 cicles. El speed up obtingut en aquest cas es de 3,586, un 29,3% menys respecte l'anterior. El temps final de simulació és de 1416 cicles, el 29,3% més que a l'anterior algorisme.

Pels resultats obtinguts l'algorisme de balanceig de CPU dona millors temps que el Round Robin perquè té en compte les caracteristiques dels processadors i a més intenta igualar el cost assignat a cada CPU per maximizar el paral·lelisme. L'únic inconvenient que hi pot haver és que no té en compte l'enviament i recepció de missatges i per tant podria decaure el rendiment del sistema si per cuasolitat les tasques que depenen del pas de missatges no estan gens sincronitzades amb altres tasques que també depenen del pas de missatges. Per tant l'algorisme de balanceig de CPU és efectiu quan hi han moltes primitives d'execució en les tasques i poques comunicacions.

6 Conclusions i treball futur

L'objectiu d'aquest treball ha estat estudiar el funcionament dels simuladors i en especial del simulador ESPPADA per tal d'integrar noves funcionalitats en el simulador i aconseguir millorar la simulació d'un o varis programes paral·lels en un sistema paral·lel distribuït.

Per a poder realitzar aquest estudi primer hem hagut d'estudiar el següents punts:

- Que és un simulador, quina és la seva utilitat i les diferentes tècniques per solventar el paradigma de la programació paral·lela.
- Les caràcteristiques de funcionament i disseny del simulador ESPPADA a nivell teòric, entendre quines són les entrades del simulador, les seves respostes de simulació i els seus resultats.
- El codi font de les parts del simulador implicades per dur a terme la implementació de la gestió de la memòria virtual, la multiprogramació a nivell de tasques locals i paral·leles i els mappings de balanceig de memòria i CPU.

Un cop entés el significat del concepte simulador i les caràcteristiques de l'ESPPADA s'ha proseguit amb l'elaboració de la primera part funcional al simulador, que és la integració de la memòria virtual:

- S'ha estudiat com funciona tot el subsistema de gestió de memòria virtual a trets generals.
- S'ha estudiat quin és el millor algorisme de memòria virtual adaptat a les implementacions ja realitzades del simulador.
- S'ha implementat l'algorisme de gestió de memòria virtual i totes les parts que intervenen en el seu funcionament, com el disseny de l'interfície.
- S'ha realitzat una experimentació per mostra els resultats obtinguts i demostrar que l'algorisme funciona correctament.

Després de realitzar aquest mòdul en el simulador s'ha proseguit amb l'extensió del simulador cap a un entorn multiprogramat a nivell de tasques locals i tasques paral·leles:

- Per millorar la simulació s'ha estudiat el codi font del simulador la capacitat de poder simular més d'un programa paral·lel i obtenir així els resultats de varis programes paral·lels mapejats a les diferentes CPU's de l'entorn paral·lel.

- S'ha implementat la multiprogramació a nivell de tasques locals.
- S'ha implementat la multiprogramació a nivell de tasques paral·leles.
- S'ha realitzat una experimentació per veure els resultats de la simulació de tasques locals, demostrar el seu correcte funcionament i veure com afecta el rendiment de la simulació d'un programa paral·lel a causa d'aquestes tasques.
- S'ha realitzat un altra experimentació per veure els resultats de la simulació de varis programes paral·lels carregats al simulador, demostrar el seu correcte funcionament i veure el resultats de la simulació de varis programes paral·lels.

Després de realitzar aquest dues noves implementacions s'ha continuat el treball amb la implementació de dos algorismes de mapping per millorar el rendiment dels programes paral·lels.

- S'ha decidit quins algorismes de mapping podrien influir directament amb el rendiment dels programes d'aquest simulador.
- S'ha estudiat quin era el millor algorisme per implementar el mapping de balanceig de memòria.
- També s'ha estudiat quin era el millor algorisme per implementar el mapping de balanceig de CPU.
- S'ha realitzat les dues implementacions dels algorismes adaptant-los a les característiques del simulador ESPPADA.
- S'ha realitzat l'experimentació de l'algorisme de balanceig de memòria comparant els resultats obtinguts amb l'algorisme de mapping Round Robin.
- S'ha realitzat l'experimentació de l'algorisme de balanceig de CPU comparant els resultats obtinguts amb l'algorisme de mapping Round Robin.

Una vegada acabat el treball i fet les diferentes experimentacions de cada implementació, podem concloure que:

- En el capítol primer sobre el simulador ESPPADA, podem concloure que aquest és un simulador que ens ensenya detalladament com funciona una maquina paral·lela i tot el que comporta com l'arquitectura, el programa, els algorismes de mapping, la xarxa d'interconnexió, la unitat de control, el graf TTIG, etc. També cal destacar que és un simulador asíncron ja que els processadors poden estar en diferents temps mentre es simula i això comporta certs avantatges, com realitzar la simulació a gran velocitat però també existeixien certs desavantages a nivell d'implementació com per exemple certes técniques de planificació no es poden implementar com el *coscheduling* ja que requereix les CPU's sincronitzades.

- En el capítol sobre la integració de la memòria virtual podem concloure gràcies a l'experimentació realitzada que s'ha de tenir en compte en la simulació la gestio de la memòria virtual perquè es una sobrecarrega important que afecta al rendiment dels programes paral·lels, sobretot aquells que tenen grans requeriments de memòria.

- En el capítol sobre l'extensió cap un entorn multiprogramat a nivell de tasques locals podem concloure gràcies a l'experimentació realitzada que la simulació de les tasques locals d'usuaris degrada el rendiment de les tasques paral·leles i que s'han de tenir en compte ja que existeixen en qualsevol entorn paral·lel com per exemple les tasques d'un sistema operatiu com el gestor de finestres, la cònsola, etc.

- En el mateix capítol sobre l'extensió cap un entorn multiprogramat però a nivell de tasques paral·leles podem concloure gràcies a l'experimentació que la simulació de varis programes paral·lels és important sobretot si volem avaluar el comportament de la gestió de la memòria virtual sotmesa a grans carregues de treball de varis programes paral·lels.

- En el capítol sobre algorismes de mapping adaptats a l'entorn multiprogramat podem concloure que el rendiment dels programes paral·lels augmenta considerablement gràcies a l'utilització màxima dels recursos limitats del sistema per part d'aquests dos algorismes. L'algorisme de balanceig de memòria té en compte la memòria principal de cada processador i els requeriments de memòria dels programes. L'algorisme de balanceig de CPU té en compte la velocitat de cada processador i els requeriments de cómput de cada tasca.

Tretes les conclusions es poden plantejar diverses línies d'investigació com a treball futur:

- Implementar nous algorismes de mapping que s'adaptin a les característiques de l'entorn i al comportament del programa paral·lel, com per exemple mappings que tenen en compte les comunicacions entre tasques.

- Un altra técnica important a desenvolupar és la planificació de tasques anomenada *coscheduling* que té en compte quines tasques s'han d'apropiar de la CPU per tal de sincronitzar-les en les comunicacions.

- Realitzar un treball sobre la simulació d'aplicacions en temps reals com per exemple aplicacions de video o de só tenint en compte i canviant si fa falta les diferents parts de la simulació que ofereix l'eina ESPPADA com els algorismes de mapping, els paràmetres dels programes paral·lels i l' arquitectura del sistema.

- Degut a que l'estructura de la màquina de simulació es semblant a la que posseixen els simuladors paral·lels per events discrets, pot ser interessant portar a cap les modificacions necessaries per adaptar-lo a la simulació paral·lela. D'aquesta manera seria possible realitzar d'una manera més eficient simulacions amb una carga de processament elevada i que d'un altra manera necessitaria un temps de còmput elevat.

Bibliografia

[1] Fernando Guirado Fernández. "Desarrollo de un Entorno de Simulación de Programas Paralelos sobre Arquitecturas Distribuidas." Tesis Doctorat. Barcelona. Septembre 2001

[2] Damià Castellà Martínez. "Subsistema de memòria virtual del Nucli de Linux." Projecte Fi Carrera d'Enginyeria Técnica en Informatica de Sistemes a l'Universitat de Lleida. Juny 2003

[3] Francesc Giné de Sola "Arquitectures no dedicades" Tesis Doctorat.

[4] R. Bagrodia, E. Deelman, S. Docy, T.Phan. "Perfromance Prediction of Large Parallel Applications Using Parallel Simulations. Proceedings ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming". Atlanta, May 1999.

[5] M. Dikaiakos, A. Rogers, K. Steiglitz. "FAST: A Functional Algorithm Simulation Testbed. International Workshop on Modelling, Analysis and Simulation of Computer and Telecomunication System". Mascots '94. 1994

[6] Yu-Kwong Kwok, Ahmad. "Dynamic Critical-Path Scheduling: An effective Tecnique for allocating Tasks Graphs to Multiprocessors." IEEE Transactions on Parallel and Distributed Systems. Vol. 8, n. 2, pp 119-129 Feb. 1997

[7] King-Jang Hwang. Yuan-Chien Chow, Frank. D. Anger and Chung-Yee Lee. "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times." SIAM J. Computing Vol. 18, n. 2 pp 244-257. Apr. 1989

[8] M.A. Senar Estrategias de Asignacion de Programas en Computadores Paralelos. Tesis Doctoral.Universidad Autònoma de Barcelona. 1996

[9] C. Roig, A. Ripoll, M. A. Senar, F. Guirado F. and E. Luque. "Modelling Message-Passing Programs for Static Mapping" Euromicro Workshop on Parallel and Distributed Processing, Jan 2000. Proceedings 8th . pp 229-236

[10] C. Roig, A. Ripoll, M.A. Senar, F. Guirado and E. Luque. "Exploting Knowledge of Temporal Behaviour in Parallel Programs for Improving Distributed Mapping." 6th Internacional Euro-Par conference. Lecture Notes in Computer Science, vol:1900, pp: 262-271. 2000

[11] "Extensible Markup Language" http://www.w3.org/XML/

[12] Remo Lucio Suppi Boldrito. "Modelado y Simulación de Sistemas Paral·lelos". Tesis Doctoral. Universitat Autònoma de Barcelona. 1996