PARASOL - Simulator de sisteme paralele şi distribuite

Valentin Cristea *Gavril Godza*
valentin@cs.pub.ro *gavrilg@cs.pub.ro*

La ora actuală sunt disponibile foarte multe instrumente software pentru modelarea şi simularea sistemelor, indiferent de tipul lor. Există simulatoare de uz general (de exemplu GPSS) care permit "abordarea" oricărui tip de sistem. Însă în momentul în care atenţia se focalizează pe o anume clasă de sisteme, un instrument prea general poate îngreuna procesul de simulare. De aceea se recomandă folosirea simulatoarelor specializate, oriunde se poate acest lucru. PARASOL este un astfel de "ajutor", conceput special pentru simularea sistemelor de calcul (reţele de calculatoare, calculatoare paralele sau distribuite).

Ce este PARASOL?

PARASOL este un instrument software care permite modelarea şi simularea sistemelor moderne de calcul. El a fost proiectat la Universitatea Carleton din Ottawa, Canada şi este disponibil prin Internet, sub licenţă GNU-OSF. Se pune problema dacă un astfel de sistem specializat este util. Răspunsul este imediat, dacă ţinem cont de faptul că studiul modelului unui sistem de calcul este mult mai convenabil (şi mai ieftin:-) decât studiul sistemului fizic. PARASOL este un simulator de granularitate medie - mică, permiţând astfel analiza fină a sistemului, proiectat special pentru sisteme de calcul distribuite şi / sau paralele. Poate fi privit atât ca orientat pe *procese*, cât şi ca orientat pe *obiecte* (însă nu în sensul OOP). El permite ca utilizatorul să controleze descrierea şi comportamentul entităţilor hardware şi software implicate, ceea ce îl face util atât pentru evaluarea performanţelor, cât şi pentru testarea algoritmilor paraleli sau distribuiţi. De fapt, PARASOL este o bibliotecă de funcţii C, care pot fi utilizate în programe C sau C++, pe diferite platforme UNIX.

Ce poate PARASOL?

PARASOL pune la dispoziţia utilizatorului posibilitatea de a descrie o reţea formată din *noduri* interconectate, pe care rulează (în paralel!) *task*-urile specificate de utilizator. Fiind totuşi un simulator, el este "dotat" cu generatoare de numere aleatoare, mecanisme de colectare a statisticilor, rutine de gestionare a evenimentelor, etc. În afară de aceste facilităţi care se regăsesc în orice simulator de uz general, există un număr de facilităţi specifice. Astfel, utilizatorul poate să-şi definească uşor partea hardware precum şi topologia reţelei, având apoi la dispoziţie o serie de apeluri sistem pentru gestiunea task-urilor, pentru comunicaţia între aceste task-uri, pentru sincronizarea lor, etc. În fapt PARASOL rulează ca un singur proces UNIX, emulându-se caracteristicile unui sistem de operare distribuit. Principalele entităţi PARASOL (hardware şi software) sunt:

- noduri de calcul, care pot conţine unul sau mai multe procesoare
- conexiuni de tip magistrală între mai multe noduri (bus)
- conexiuni de tip legătură unidirecţională între două noduri (link)
- task-uri
- porturi de comunicaţie
- evenimente şi mesaje
- mecanisme de sincronizare (variabile spin-lock şi semafoare)
- statistici

Respectând filozofia UNIX, toate aceste entităţi sunt identificate prin numere întregi, care se alocă în momentul creării lor şi sunt folosite apoi pentru a referi aceste obiecte.

O vedere simplificată asupra unui sistem simulat cu PARASOL pune în evidență existența unui *driver* (supervizor) care răspunde de derularea experimentului în conformitate cu dorințele utilizatorului, a unui *mediu de execuție* care constă dintr-un set de componente hardware modelate de utilizator prin constructori PARASOL şi a unui set de task-uri utilizator.

Cum se foloseşte PARASOL?

Cel care doreşte să folosească PARASOL nu trebuie să plece de la zero. El trebuie să fie familiarizat cu modul de operare pe un sistem UNIX şi, evident, să ştie să programeze în C sau C++. Dacă aceste condiții sunt îndeplinite se poate trece la etapa următoare, realizarea modelului. Pentru aceasta, se pleacă de la sistemul fizic avut în vedere şi se vizează obținerea unei reprezentări care să fie cât mai uşor "tradusă" în PARASOL. E bine ca această reprezentare să fie "salvată" pe un suport "extern" de memorare (de exemplu pe o foaie de hârtie:-), unde se pot face anumite validări primare. Această fază este cea mai lipsită de reguli de abordare, astfel încât ingeniozitatea programatorului este solicitată din plin. Pentru a ilustra cum poate decurge această fază, vom considera un exemplu concret, şi anume dirijarea (rutarea) mesajelor în cazul unui calculator paralel.

Descrierea problemei

Presupunem că studiem un calculator paralel dotat cu *N* procesoare identice, fiecare având propria sa memorie de lucru. Aceste procesoare conlucrează pentru rezolvarea unei probleme, prin execuția unui algoritm paralel. Evident, în timpul prelucrării un procesor poate avea nevoie de anumite date deținute de alt procesor, modul în care le poate obține fiind schimbul de mesaje. Acest schimb de mesaje induce un *overhead*, un timp suplimentar în timpul global de rezolvare a algoritmului. Prin urmare ar fi foarte bine să cunoaştem influența diferiților parametri de configurare (număr de procesoare, topologie, algoritm, etc.) asupra acestui timp suplimentar, cu scopul reducerii lui. Pentru aceasta evident nu vom testa sistemul fizic, ci vom folosi un model PARASOL.

Topologiile folosite pentru realizarea calculatoarelor paralele sunt deosebit de diverse, mergând de la structuri simple de tip plasă sau inel până la structuri mai complicate, de tip tor sau hipercub. Topologia aleasă în acest exemplu este de tip plasă (*mesh*) având N=9 noduri identice, dispuse într-o matrice de dimensiune 3x3 noduri. Această topologie se poate foarte uşor transforma în tor (se leagă nodurile extreme de pe fiecare linie şi coloană) sau chiar în hipercub. Modul ales de etichetare al nodurilor este prezentat în Figura 1. Etichetarea lor nu s-a făcut la întâmplare, ci în concordanță cu algoritmul de rutare pe care îl testăm. În cazul de față am ales un algoritm *wormhole* ("galerie de vierme" în engleză). Esența acestui algoritm este următoarea: un mesaj este trimis fragmentat în secvențe numite *flit*-uri, primul flit având misiunea de a găsi un drum de la sursă către destinație. Pe măsură ce acest prim flit avansează, el este urmat de celelalte flit-uri, fenomenul apărut fiind asemănător cu deplasarea unui vierme. Tot în conformitate cu algoritmul ales se face configurarea rețelei de comunicație. Pentru aceasta folosim numai *link*-uri, etichetate astfel încât orice legătură de la un nod cu eticheta mai mică spre unul cu eticheta mai mare să fie *pară*, iar orice legătură de la un nod cu eticheta mai mare spre unul cu eticheta mai mică să fie *impară*. În acest fel se pun în evidență două subrețele, care vor fi utilizate în cadrul algoritmului pentru a evita deadlock-ul (blocarea sistemului prin aşteptare reciprocă).

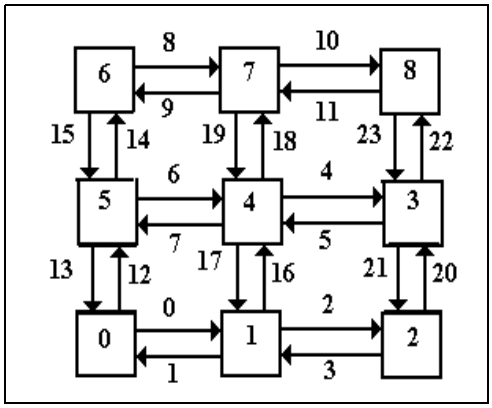

Figura 1. Topologia folosită

Fiecare nod va fi înzestrat cu un singur procesor, caracterizat de o anumită viteză de lucru, aceeaşi pentru toate procesoarele. Pe acest procesor se vor rula toate task-urile, atât cele specifice, locale nodului, cât şi cele destinate vehiculării mesajelor în rețea. Aceasta nu e unica alegere, putându-se opta pentru folosirea a două procesoare într-un nod (un procesor specializat pentru execuția task-urilor locale şi unul de intrare-ieşire pentru dirijarea mesajelor) sau chiar prin folosirea a cinci procesoare într-un nod (un procesor specializat pentru execuția task-urilor locale şi patru procesoare de intrare-ieşire, câte unul pentru fiecare legătură a nodului).

Pe fiecare nod (procesor în cazul acesta) se vor crea două task-uri: un *task_server*, responsabil cu traficul de mesaje şi un *task_local*, care va simula calcule ce necesită şi informații din alte noduri. Acesta va emite mesaje spre alte noduri din rețea prin intermediul server-ului şi va primi de la server mesajele adresate nodului curent. Comunicația între aceste procese se va face *local*. Task-ul local va genera mesaje la anumite intervale de timp, obținute folosind distribuțiile puse la dispoziție de PARASOL. Recepția mesajelor se va face cu aşteptarea unui interval de timp (*timeout*). Aceasta presupune că dacă un mesaj nu a sosit, procesul respectiv este capabil să facă alte calcule, pentru o mai bună utilizare a timpului procesor. Procesul server va trebui să distingă mesajele adresate nodului local de cele pentru rețea. Un mesaj local va fi trimis task-ului local de pe acel nod, în timp ce unul destinat altui nod va fi trimis mai departe, pe o legătură aleasă pe baza algoritmului de dirijare.

Există o singură funcție server şi o singură funcție locală, care vor deveni task-uri pe fiecare nod din rețea, în felul acesta făcându-se o importantă economie de cod şi crescând claritatea programului.

Reguli de implementare

După ce s-a ales sistemul ce trebuie modelat, urmează scrierea propriu-zisă a programului, care trebuie să respecte unele reguli. Ca orice program C, şi cel ce foloseşte PARASOL trebuie să posede o funcție *main()*. Aceasta poate fi scrisă fie de utilizator, fie poate fi furnizată automat de sistem. Evident, a doua variantă este mai convenabilă şi implică o responsabilitate mai mică din partea celui ce scrie programul. Dacă este furnizată automat, funcția main() va cere introducerea de la tastatură a unor valori ce caracterizează experimentul, şi anume valoarea pe baza căreia se inițializează generatorul de numere aleatoare şi durata simulării, în unități specifice. Timpul gestionat de PARASOL diferă de cel real, al maşinii UNIX, el scurgându-se în trepte, odată cu apariția diferitelor evenimente.

Utilizatorul care doreşte să controleze mai adânc proprietățile sistemului modelat va trebui să furnizeze funcția main(), specifică dorințelor sale. Descrierea modelului se va face în ambele cazuri în cadrul funcției *ps_genesis()*, funcție apelată de către main(). Această funcție este de fapt un task, strămoşul tuturor task-urilor ce vor fi create. Ca o observație, toate funcțiile PARASOL au numele precedat de "**ps_**", aceasta fiind o bună măsură de prevenire a unor conflicte de nume între funcții sistem şi funcții utilizator, care cel mai adesea sunt dificil de identificat.

Funcția ps_genesis() va trebui să *construiască* suportul hardware al modelului (noduri, bus-uri, link-uri) şi apoi să *pornească* simularea prin lansarea în execuție a anumitor task-uri, pe diferite noduri. Nodurile

care se vor crea prin intermediul acestei funcţii vor fi numeroate de la 1, deoarece nodul 0 (nodul PARASOL) este unul special, rezervat, pe el rulând chiar funcţia ps_genesis(). Pe acest nod special mai rulează în mod obligatoriu încă un task, numit **reaper** (vă mai amintiţi de *Jack the Reaper*?) care asistă procesele ce se termină prin "sinucidere" (şi o face cu un scop cât se poate de nobil: eliberarea memoriei ocupată de acestea). Acest nod poate prelua spre execuţie şi alte task-uri. Task-urile utilizator care vor fi lansate în execuţie sunt funcţii C obişnuite, care pot apela funcţiile bibliotecii PARASOL. O caracteristică a acestor task-uri este aceea că ele nu au argumente şi nu întorc nici un rezultat. Dacă se impune pasarea unor valori task-ului, acest lucru se va face prin trimitere de mesaje sau prin variabile globale. *Comunicarea* între task-uri se face la nivel de porturi, un *port* fiind un fel de "cutie poştală" unde se pun mesajele destinate unui task. Pentru aceasta, în afară de portul standard asociat fiecărui nod, se pot crea dinamic noi porturi.

Tot în funcţia ps_genesis() este bine să se precizeze *statisticile* dorite. Utilizatorul poate alege între cele oferite implicit de PARASOL sau poate să definească statistici proprii. În cazul celor din urmă, se vor specifica explicit (în celelalte task-uri) momentele la care se înregistrează informaţiile necesare statisticii. De asemenea, utilizatorul va avea responsabilitatea asigurării "avansului" timpului, prin apeluri specifice. Planificatorul încorporat va face gestiunea tuturor momentelor de timp la care se produc evenimentele, prin gestiunea unor tabele interne, asigurând scurgerea cronologică a timpului.

La terminarea duratei stabilite pentru simulare toate statisiticile înregistrate vor fi afişate pe ecran, înaintea terminării programului propriu-zis.

Pentru a se compila sursele C ce compun modelul PARASOL se poate folosi orice compilator de C. Lucrul este mult uşurat dacă se folosesc fişiere Makefile (ca cel din Figura 2) în care se descrie modul de compilare (de fapt important este în ce director se găseşte biblioteca PARASOL folosită).

```
# Make file for Parasol Version 2.0
# Parasol system location (may be any directory)
PHOME=..
SRC=  rutare
CFLAGS= -O -L$(PHOME) -I$(PHOME)/include -lparasol -lm
CC= gcc
all: $(SRC)
$(SRC): $(PHOME)/libparasol.a
        $(CC)  $(@).c -o $(@) $(CFLAGS)
clean:
        rm -f *.o $(SRC) core
```

Figura 2. Fişierul Makefile

Cum se face configur2area hardware?

Utilizatorul poate crea entităţi hardware folosind funcţiile *ps_build_xxx()*, unde xxx se înlocuieşte cu numele fiecărei componente şi primeşte argumente specifice. Aceste componente au o capacitate finită de prelucrare, aşa că se vor forma nişte cozi de aşteptare pentru serviciile oferite. Să vedem care sunt caracteristicile acestor entităţi, exemplificând cu modelul deja prezentat.

Gestiunea nodurilor

Nodul este entitatea hardware de bază deoarece pe el rezidă task-urile care se execută şi el se conectează la alte noduri prin tipurile de conexiuni puse la dispoziţie de PARASOL. Prin urmare el va fi prima entitate creată. Funcţia care construieşte un nod este **ps_build_node()**, care primeşte o serie de argumente pentru caracterizarea nodului:

```
SYSCALL ps_build_node (char *name, int ncpu, double speed, double quantum, int discipline, int stat_flag);
```

Parametrul name este numele nodului, un şir de caractere, ataşat din motive de claritate (statisticile asociate sunt mai uşor de interpretat, depanarea este mai uşoară). Numărul de procesoare din nod este dat

de argumentul *ncpu* şi trebuie să fie cel puţin 1. Nodul special PARASOL simulează totdeauna un singur procesor. Viteza procesoarelor poate fi controlată prin valoarea argumentului *speed*. Coada de aşteptare asociată unui nod va conține *task*-uri în starea "gata de execuție". Politica ce guvernează coada poate fi FCFS (First-Come-First-Served în care primul sosit este primul servit), cu prioritate fără prelevare forțată (HOL în care apariția unui task cu prioritate mai mare decât a celui ce se execută nu îl va face pe acesta să piardă procesorul) şi cu prelevare forțată (PR în care un task cu prioritate mai mare decât a celui ce se execută va câştiga procesorul). Argumentul *discipline* specifică politica folosită, în timp ce argumentul *quantum* precizează dacă se foloseşte sau nu planificarea round-robin. În fine, ultimul argument, *stat_flag*, are valoarea TRUE dacă se doreşte activarea statisticii de utilizare a procesoarelor. Funcția întoarce fie un identificator (număr) intreg pozitiv pe baza căruia va fi referit nodul în continuare, fie valoarea specială SYSERR în cazul în care nu s-a reuşit construirea nodului, din cauza parametrilor incorecți. Fiecare nod are o singură coadă de aşteptare, însă utilizatorul poate alege procesorul pe care un anume task să ruleze.

Nodurile care formează sistemul propus se generează cu secvența de cod din Figura 3 (ele se vor numi Nod_0, Nod_1, ... ,Nod_8 şi vor fi păstrate intr- un vector):

```
#include <parasol.h>
#define SPEED    1.0              /* viteza CPU */
#define QUOTA    0.5              /* cuanta de timp Round Robin */
#define NNODE    9                /* numărul de noduri */
. . . . .
for(i=0;i<NNODE;i++)
     {     sprintf(buf,"Nod_%d",i);
           node[i]=ps_build_node(buf, 1, SPEED, QUOTA, FCFS, TRUE);
     }
```

Figura 3. Generarea reţelei de noduri

Gestiunea legăturilor

Aşa cum am mai amintit există două tipuri de entități care permit conectarea nodurilor: *link*-uri (legături unidirecționale) şi *bus*-uri (magistrale). Cozile asociate acestor entități conțin *mesaje* şi nu task-uri. Politica acestor cozi este FCFS, însă se poate folosi şi o politică aleatoare, de exemplu pentru simularea tehnicii CSMA/CD (Carrier Sense Multiple Access with Collision Detection) de alocare a magistralei.

Crearea acestor entități se face cu funcțiile **ps_build_bus()**, respectiv **ps_build_link()**.

```
SYSCALL ps_build_bus(char *name, int ncount, int node_array[], int packet_size,
double trans_rate, int discipline, int stat_flag);
```

O magistrală va avea un nume asociat (şirul de caractere *name*) şi va conecta un număr de *ncount* noduri care sunt descrise în vectorul *node_array*. Mesajele sunt fragmentate automat în bucăți de dimensiunea *packet_size* şi sunt trimise cu o rată de transfer de *trans_rate* caractere / secundă. Fragmentele aceluiaşi mesaj sunt trimise pe magistrală continuu, iar ultimul fragment este completat la nevoie până la lungimea stabilită. Parametrul *discipline* specifică modul de gestiune al cozii de mesaje iar *stat_flag* dacă se doreşte colectarea de statistici despre bus. Funcția returnează identificatorul magistralei, care va fi folosit apoi pentru trimiterea mesajelor. Dacă valorile argumentelor sunt invalide funcția va returna SYSERR.

```
SYSCALL ps_build_link(char *name, int source, int destination, int packet_size,
double trans_rate, int stat_flag,)
```

Acest apel construieşte o legătură **unidirecțională** cu numele *name*, de la nodul *source* spre nodul *destination*. Este deci foarte importantă ordinea specificării nodurilor, primul va fi nodul sursă, iar al doilea nodul destinație. Ceilalți parametri ai link-ului sunt similiari celor de la bus. O legătură bidirecțională se va crea prin două legături unidirecționale, în care cele două noduri conectate sunt pe rând sursă şi destinație, aşa cum se observă şi în fragmentul de cod din Figura 4:

```
link[0]=ps_build_link("Leg_0",node[0],node[1], 100, 200000, TRUE);
link[1]=ps_build_link("Leg_1",node[1],node[0], 100, 200000, TRUE);
```

Figura 4. Construirea unei legături bidirecţionale

Cum se face configurarea software?

Având asigurată *infrastructura hardware* a modelului se poate trece la "instalarea" software-ului. Cea mai importantă entitate software este task-ul PARASOL, prin care se pot specifica diferitele acțiuni ce trebuie executate.

Gestiunea task-urilor

În sistemul PARASOL un task este caracterizat de *stiva* proprie, de *cod* propriu (furnizat sub forma unei funcții C) şi ocupă o intrare în tabela proceselor. Aici sunt memorate toate informațiile necesare sistemului pentru a planifica pentru execuție un task: starea task-ului, nodul şi procesorul pe care rulase anterior, contextul task-ului, prioritatea lui etc. Stările task-urilor PARASOL se pot împărți în două categorii: stări care implică executarea task-ului şi stări care nu implică acest lucru. Un task este trecut într-o stare care nu implică execuția sa la crearea şi la distrugerea lui, dacă e gata de execuție dar îi lipseşte procesorul, dacă aşteaptă sosirea unui mesaj sau dacă el cere explicit acest lucru (auto-adormire). În afară de starea HOT, în care un task se rulează efectiv pe un procesor, celelalte stări implicând execuția sunt specifice punctului de vedere PARASOL.

Task-urile nu sunt specifice modelării entităților hardware, însă un utilizator mai "ghiduş" poate să le folosească pentru a înlocui variante de noduri, magistrale şi legături. Ele există pentru a modela orice altceva. Crearea unui task PARASOL este o operație ce presupune execuția mai multor paşi. În primul rând trebuie să existe o funcție C care să "candideze" la postul de task, funcție scrisă de utilizator şi care nu primeşte argumente şi nu întoarce nici un rezultat. Apoi funcția trebuie "avansată" la gradul de task prin apelul ps_create() şi, în fine activată prin apelul ps_resume(). Un task poate fi distrus prin apelul ps_kill() sau suspendat prin apelul ps_suspend(). Controlul timpului se face folosind ps_compute(), ps_hold(), ps_sync() şi ps_sleep(), care permit unui task să "consume" un anumit număr de tick-uri în diferite moduri. Astfel, durata specificată poate să fie mai mare sau mai mică (mică însemnând reducerea granularității!), poate să fie întreruptibilă sau neîntreruptibiă, poate duce sau nu la pierderea resurselor.

```
SYSCALL ps_create(char *name; int node; int cpu; void (*class_code)(void); int priority);
```

La crearea unui task se vor specifica numele task-ului (*name*), nodul (*node*) şi procesorul (*cpu*) pe care acesta va rula. Dacă nodul pe care va rula trebuie specificat obligatoriu, alegerea procesorului poate fi lăsată la latitudinea sistemului, folosind constanta ANY_HOST. Urmează numele funcției C care conține codul task-ului şi se mai precizează prioritatea acestuia (*priority*). Stiva unui astfel de task va avea dimensiunea implicită. Dacă cineva doreşte o stivă de altă dimensiune va folosi apelul ps_create2(), care are un parametru suplimentar, dimensiunea stivei. Alegerea apelului potrivit se realizează de obicei prin încercări (sistemul poate avea comportări bizare, deoarece PARASOL nu detectează depăşirea de stivă). Dacă valorile parametrilor sunt valide, task-ul e creat şi suspendat; în caz contrar funcția semnalează eroare. Crearea şi lansarea proceselor server pe nodurile deja existente se face conform codului din Figura 5. Aşa cum se vede din figură, aceste task-uri vor purta numele *Server_0*, ..., *Server_8*. Pentru a permite o mai mare flexibilitate a programului, în exemplul ales se folosesc doi vectori de lungimi egale, unul de funcții iar celălalt cu identificatorii task-urilor.

```
for(i=0;i<NNODE;i++)
{sprintf(buf,"Server_%d",i);
 ps_resume(t[i]=ps_create(buf,node[i],ANY_HOST, (*server[i]),0));
}
```

Figura 5. Crearea şi lansarea proceselor server

Argumentul funcției ps_kill() care distruge un task este identificatorul acestui task. Un task nu poate distruge decât descendenții proprii. Distrugerea unui task are ca efect distrugerea tuturor descendenților, precum şi a tuturor entităților software deținute de acesta. Din această cauză "sinuciderea" este folosită pentru a asigura terminarea în bune condiții a experimentului de simulare, şi se face prin apelul **`ps_kill(ps_myself)`**. Argumentul folosit este unul dintre numeroasele macrouri puse la dispoziție

de PARASOL pentru identificarea mai uşoară a unor entităţi. Din motive de bună funcţionare, ps_kill() nu are efect asupra task-ului ps_genesis().

Alte funcţii de gestiune a task-urilor permit trecerea lor de pe nodul curent pe alt nod şi procesor (migrare) sau modificarea dinamică a priorităţii. De asemenea există funcţii şi macrouri care întorc informaţii despre configuraţia sistemului: legăturile de intrare şi ieşire ale unui task, bus-urile asociate task-ului, task-ul părinte şi task-urile copii, nodul asociat unui task, numele şi prioritatea unui task, starea lui, etc.

Comunicaţia în PARASOL

PARASOL foloseşte porturi pentru comunicaţia inter-task-uri. Entităţile numite porturi sunt receptorii de mesaje, ceea ce înseamnă că un mesaj nu este adresat unui task, ci unui port. Porturile au o capacitate teoretic nelimitată de stocare a mesajelor în cozi de aşteptare FCFS. Trimiterea de mesaje poate fi cu destinaţie unică, cu destinaţie multiplă (multicast) sau cu difuzare (broadcast), poate fi făcută local sau la "distanţă", pe link-uri sau bus-uri, fiind disponibile următoarele funcţii: ps_send(), ps_resend(), ps_bus_send(), ps_link_send(), ps_broadcast(), ps_localcast() şi ps_multicast(). Dacă un task deţine un port, prin ps_receive() poate recepţiona mesajele care sosesc acolo. Fiecare task deţine de la creare un *port standard*, care în majoritatea cazurilor este suficient pentru nevoile sale de comunicare. Însă există funcţii pentru crearea de porturi suplimentare, private sau partajate, şi de alocare a lor task-urilor. Porturile pot fi grupate pentru uşurarea gestiunii mesajelor sau pot fi transferate de la un task la altul, împreună cu toate mesajele ce există în coada asociată. Porturile private sunt dealocate automat la distrugerea task-ului, în timp ce cele partajate trebuie eliberate explicit. Nu se poate elibera portul standard al unui task.

Trimiterea unui mesaj se face la un port anume fie de pe acelaşi nod fie pe un bus sau link spre alt nod (în acest ultim caz trebuie să existe conexiune între noduri). Din cauza similitudinilor existente prezentăm doar comunicaţia pe link.

```
SYSCALL ps_link_send(int link, int port, int type,int size, char *text, int
ack_port);
```

Mesajul va avea tipul *type*, va fi trimis pe legătura *link* şi pe portul *port*. Mesajul poate să conţină în corpul său orice caractere, cunoscânu-se dimensiunea *size* a sa şi adresa de început *text*. El poate să conţină şi o "ştampilă de timp" (funcţia ps_resend()), însă obligatoriu conţine portul *ack_port* la care se aşteptă confirmarea mesajului. Dacă nu se doreşte confirmare, valoarea indicată pentru acest port va fi NULL_PORT. Dacă trimiterea eşuează, valoarea întoarsă de funcţie va fi SYSERR.

Dacă un mesaj trebuie să ajungă la mai mulţi destinatari, el poate fi distribuit tuturor task-urilor de pe nodul local, tuturor task-urilor din sistem, sau tuturor descendenţilor (direcţi sau indirecţi) unui task, recepţia făcându-se doar la portul standard. Este foarte important de remarcat că nici unul dintre apelurile de trimitere mesaje **nu blochează** task-ul care îl apelează.

Recepţia unui mesaj la portul *port* se face cu următorul apel:

```
SYSCALL ps_receive(int port, double time_out, int* type, double* ts, char *tp, int*
ack_port);
```

În urma lui, *type* va conţine tipul mesajului, *ts* ştampila de timp (momentul la care a fost emis), iar *tp* va conţine corpul mesajului. La recepţia unui mesaj se poate specifica modul de acţiune în caz că nu există mesaje disponibile. Astfel valoarea NEVER a argumentului *time_out* înseamnă aşteptarea un timp nedefinit a mesajului, valoarea IMMEDIATE presupune doar inspectarea cozii şi eventual culegerea mesajului, iar orice altă valoare reprezintă timpul de aşteptare.

Şi în acest caz sunt disponibile funcţii pentru aflarea unor informaţii de configurare: care este task-ul ce deţine un anume port, care sunt porturile deţinute de un anume task, etc.

Sincronizarea task-urilor

Pe lângă sincronizarea asigurată de schimbul de mesaje, există două mecanisme de sincronizare, care folosesc variabile "zăvor" (spin-lock) sau semafoare. Funcțiile disponibile sunt cele uzuale, cu precizarea că aceste obiecte nu se alocă explicit, ele fiind pur şi simplu folosite. Prin urmare este responsabilitatea programatorului să folosească aceste entități asignându-le identificatori într-o plajă continuă, începând de la zero.

Culegerea de statistici

Statisiticile reprezintă finalitatea experimentului PARASOL, ele sintetizând modul în care a decurs acesta. Există statisitici predefinite, care indică utilizarea resurselor hardware şi care pot fi folosite sau nu. Utilizatorul are posibilitatea să-şi definească propriile sale statisitici, având de ales între două tipuri. Primul se numeşte "VARIABLE" deoarece ele urmăresc o anume variabilă pentru a-i calcula valoarea medie (de exemplu lungimea medie a unei cozi de aşteptare). Al doilea tip este "SAMPLE", deoarece lucrează cu eşantioane (se face o sumă a valorilor, marcând-se şi numărul de eşantioane). La deschiderea unei statistici folosind **`ps_open_stat()`** se specifică numele şi tipul statisticii, obținându-se un identificator. Culegerea datelor se face cu **`ps_record_stat()`**, indiferent de tipul statisticii. În timpul simulării utilizatorul poate să afle valoarea medie asociată unei statistici la momentul respectiv, să determine afişarea statisticii sau resetarea acesteia. La sfârşitul simulării statisticile sunt afişate pe ecran ca în Figura 6.

```
Simulation statistics for time = 11.
 Name                                Type    Mean  Obs(#|interv)
Nod_0 (cpu 0) Utilization             VAR    0.981727      11
Nod_1 (cpu 0) Utilization             VAR    0.999864      11
Nod_2 (cpu 0) Utilization             VAR    0.981818      11
Leg_12 Utilization                    VAR    0.000727273   11
Leg_13 Utilization                    VAR    0.00118182    11
Leg_14 Utilization                    VAR    0.000409091   11
Statistica timp                       SAMPLE    0.35       20
```

Figura 6. Exemplu de afişare a statisticilor

Numere aleatoare

Fiind vorba de un simulator, el trebuie să fie dotat cu posibilitatea controlării distribuției evenimentelor care apar în sistem. PARASOL răspunde acestei cerințe cu un set de macrouri. Distribuțiile disponibile sunt cea uniformă, exponențială şi Erlang. Numerele generate pe baza distribuție uniforme pot fi reale între 0 şi 1 sau între o valoarea minimă şi una maximă, respectiv întregi între 0 şi n-1. Pentru ca experimentul să poată fi repetat pentru diferite secvențe de numere aleatoare, valoarea de start (*seed*) va fi introdusă de utilizator înaintea începerii experimentului propriu-zis. Acest procedeu permite însă şi reproducerea unor experimente.

Codul aplicației PARASOL

În Figura 7 se prezintă codul obținut. Pentru claritate, o serie de aspecte legate de algoritm au fost prezentate sub formă de pseudo-cod.

```
#include <parasol.h>

#define NFLIT   16              /*Numar flit-uri*/
#define PK_SZ   100             /*Dimensiune pachet*/
#define LN_RT   200000          /*Rata de transmisie pe o legatura*/
#define SPEED   1.0             /*Viteza CPU*/
#define QUOTA   0.5             /*Cuanta Round Robin */
#define NNODE   9                    /*Numar noduri*/
#define NLINK   24              /*Numar legaturi in mesh*/
#define TMOUT   1

/* diferite variabile globale*/
 void server1(), local1(), (*local[NNODE])(), (*server[NNODE])();
 char buf[PK_SZ];
 int node[NNODE], t[NNODE], lt[NNODE], p[NNODE], lp[NNODE], link[NLINK];

/* task-ul ps_genesis() */
void ps_genesis(void)
{ for(i=0;i<NNODE;i++)
   { sprintf(buf,"Nod_%d",i);
     node[i] = ps_build_node(buf, 1, SPEED, QUOTA, FCFS, TRUE);
   }
 link[ 0] = ps_build_link("Leg_00", node[0], node[1], 100, 200000, TRUE);
 link[ 1] = ps_build_link("Leg_01", node[1], node[0], 100, 200000, TRUE);
 link[ 2] = ps_build_link("Leg_02", node[1], node[2], 100, 200000, TRUE);
 link[ 3] = ps_build_link("Leg_03", node[2], node[1], 100, 200000, TRUE);
/* si asa mai departe pana se construiesc toate cele 24 link-uri */

 for(i=0;i<NNODE;i++) server[i]=server1;
 for(i=0;i<NNODE;i++) local[i]=local1;
 for(i=0;i<NNODE;i++)
    {   sprintf(buf,"Server_%d",i);
        ps_resume(t[i]=ps_create(buf, node[i], ANY_HOST, (*server[i]), 0));
        p[i] = ps_std_port(t[i]);
    }
 for(i=0;i<NNODE;i++)
    {   sprintf(buf,"Local_%d",i);
        ps_resume(lt[i]=ps_create(buf, node[i], ANY_HOST, (*local[i]), 0));
        lp[i] = ps_std_port(lt[i]);
    }
 ps_send_links(ps_myself, &lcount, link_array);
 ps_receive_links(ps_myself, &lcount, link_array);
 ps_sleep(5);    ps_suspend(ps_myself);
}

/* task-ul local din fiecare nod*/
void local1()
{int lo, i, l, dest, di, type, ack_port, link_o[[4];
 double ts; char *tp, char mes[NFLIT*PK_SZ];
  ps_send_links(ps_myself, &lo, link_o);
  while(TRUE)
  { i=ps_choice(1+ps_myself<<1);
    pregateste_mesaj();
    alege_destinatie();
    if(ps_link_send(dest, di, 1, strlen(mes), mes, NULL_PORT)==SYSERR)
         printf("\nDetected error: invalid send\n");
    ps_receive(ps_my_std_port, TMOUT, &type, &ts, &tp, &ack_port);
    ps_compute(ps_exponential(ps_myself));
  }
}

/* task-ul server de comunicatie din fiecare nod */
void server1()
{int lo, li, link_o[4], link_i[4], i, l, d, di, ack_port, type;
 double ts; char *tp, mes[NFLIT*PK_SZ];
 ps_send_links(ps_myself, &lo, link_o);
 ps_receive_links(ps_myself, &li, link_i);
 while(TRUE)
 {ps_receive(ps_my_std_port, TMOUT, &type, &ts, &tp, &ack_port);
  if (d==ps_my_node)  ps_send(ps_my_std_port, type, tp, NULL_PORT);
```

```
  else   {selecteaza_legatura();
          if(ps_link_send(dest, di, 1, strlen(mes), mes, NULL_PORT)==SYSERR)
            printf("\nDetected error: invalid send\n");
          }
  ps_compute(ps_exponential(ps_myself));
  }
}
```

Figura 7. Codul aplicaţiei

Concluzii şi tendințe

Orice simulare poate ridica anumite probleme şi poate fi dificil de depanat. Modul în care a fost realizat PARASOL permite depistarea funcțiilor care nu reuşesc să-şi facă "datoria". De asemenea, simulatorul încorporat este de un real folos în punerea la punct a unei aplicații. Lansarea lui se face specificând în linia de comandă opțiunea "-t" sau "-s", iar dacă nu face față se pot folosi depanatoarele standard de pe platformele UNIX (dbx, gdb, etc.).

Orice simulare implică un consum relativ mare de timp procesor pe maşina gazdă. Din această cauză şi aplicațiile PARASOL trebuie optimizate. De exemplu nu se vor colecta statistici fără discernământ, deoarece această acțiune implică unele calcule în virgulă mobilă, deci consumatoare de timp. Studiul programelor PARASOL a arătat că marea majoritate a timpului se consumă cu comutarea de contexte şi operații de planificare (e mai bine ca un task să fie "suspendat" şi nu "adormit"; e mai bine să se folosească cuante nu prea mici la planificarea Round-Robin, etc.). Sunt necesare deci şi câteva rulări de probă, pentru a vedea unde se poate interveni.

Direcțiile de dezvoltare ale acestui produs vizează realizarea variantei VPARASOL (Visual PARASOL) sub mediul grafic XWindow, respectiv DC/P (Distributed Computing PARASOL) axat pe algoritmi distribuiți, însă acestea nu sunt singulare, existând multe alte propuneri de extindere a sa. Acestea însă nu sunt prea agreate de autorii săi, deoarece pot duce la aparția multor dialecte. Deoarece sistemul PARASOL încurajează scrierea modulară de cod şi reutilizarea acestuia, se aşteaptă apariția unor biblioteci în afara nucleului, care să conțină entități deja testate, pentru a permite dezvoltarea rapidă de modele complexe, de nivel înalt.

Autorii au testat acest produs sub sistemele de operare DEC-Ultrix, Sun-OS, respectiv Linux, folosind limbajul C. Instalarea lui a fost deosebit de facilă. Arhiva de la care s-a plecat are circa 240 kB (în format gzip) şi conține pe lângă bibliotecile PARASOL, o serie de programe demonstrative, precum şi documentația ce însoțeşte produsul.