Procesare multicore
Eficienta paralelizarii in sistemele MultiProcessor SoC (MPSoC)

RAPAN Adrian
Master of Information Engineering and Computing Systems
"Polytechnics" University of Bucharest
No. 1-3 Iuliu Maniu, 061071, Bucharest, Romania,
adrian_rapan2002@yahoo.com

Cuvinte cheie: MPSoC, paralelizare, thread POSIX, afinitate CPU

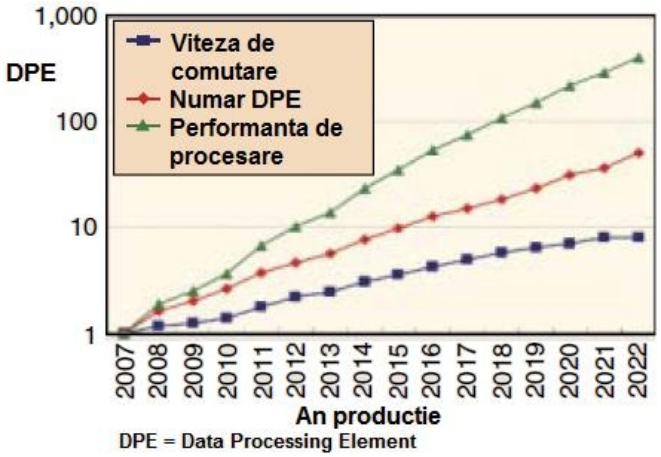
Abstract. Cresterea exploziva a continutului media (video, imagini) se poate observa atat in dispozitivele folosite in mod uzual cat si pe Internet. Astfel noi provocari au aparut in ceea ce priveste compresia, analiza si sinteza continului media in timp real. Datoria resurselor computationale necesare foarte mari sistemele MPSoC s-au dovedit a fi o solutie pentru imbunatatirea performantelor computationale ale sistemelor dedicate la un consum redus de energie. Pe langa aceste performante hardware este necesara si o strategie de programare adecvata in vederea obtinerii maximului de eficienta prin alocarea optima de procese resurselor existente. Prezenta unui numar mare de nuclee de procesare intr-un singur chip aduce de la sine nevoia de aplicatii cu un grad inalt de paralelism. Utilizand algoritmi potriviti de alocare a proceselor catre nucleele de procesare poate duce la o imbunatatirea performantei si in acelasi timp ca un consum redus de energie. Am utilizat libraria de threaduri POSIX pentru parelelizarea de operatii a 2 matrice de mari dimensiuni,valorile fiind incarcate din nivelul de gri al pixelilor unor imagini satelitare de rezolutie mare. S-a realizat o comparatie intre timpul secvential de rulare, timpul de prelucrare paralela cu alocarea dinamica a nucleelor de procesare, prelucrarea paralela cu alocarea nucleelor de procesare lasata la latitudinea mecanismului de guvernare a proceselor implementat in sistemul de operare Linux (Kubuntu). Procesor folosit:Intel Core2Duo 2 GHz

INTRODUCERE

Procesarea de semnal este aplicatia primara pentru tehnologia VLSI (Very Large Scale Integration) si SoC (System on Chip), si astfel nu trebuie sa fie o surpriza efortul imens ce se depune in dezvoltarea unor noi si mai performante arhitecturi. Nevoia de programabilitate si de performanta in timp real la o putere scazuta au dus la o constanta imbunatatire a acestor arhitecturi. Desi procesoarele multicore au aparut recent pe piata desktop - urilor si a serverelor, acestea au o istorie bogata in parte de calcul integrat (embedded computing) datorita cerintelor stricte a sistemelor. MPSoC - urile (Multiprocessor SoC) s-au dezvoltat ca raspuns la nevoia de procesare de semnal mai rapida si a aplicatiilor multimedia din ce in ce mai complexe si robuste. Alegerea unui algoritm de procesare trebuie facuta asa in fel incat sa se preteze corect pe platforma dorita cunoscand in profunzime atat avantajele cat si limitarile pentru a ajunge la cele mai bune rezultate.

Sistemele integrate cu un singur procesor nu mai pot satisface cerintele in continua crestere a aplicatiilor software, insa nanotehnologia a facut posibila integrarea de mai multe procesoare intr-un singul chip pentru a indeplini cerintele software la o eficienta energetica foarte mare. Un MPSoC, spre deosebire de traditionalul SoC, este bine structurat, astfel incat probleme de genul consumului de energie sunt adresate separat de catre unitatile de procesare DS (Distributed Systems). In ciuda potentialului extraordinar, un mare impediment in dezvoltarea lor raman toate uneltele de programare si dezvoltare de o complexitate foarte mare.

Progrese exista in crearea de arhitecturi configurabile dar si in metodologia de dezvoltare, insa inginerii sunt inca nevoiti sa aloge manual sarcini specifice unei aplicatii pe diferite procesoare, si in acelasi timp ca particularizeze traficul pe un anumit procesor astfel incat cerintele de performanta sa fie atinse.



[FIG 1] Previziuni asupra performantelor MPSoC

PUNEREA PROBLEMEI. PROVOCARI

Cresterea exploziva a continutului media (video,imagini) se poate observa atat in dispozitivele folosite in mod uzual cat si pe Internet. Astfel noi provocari au aparut in ceea ce priveste compresia, analiza si sinteza continului media in timp real. Datoria resurselor computationale necesare foarte mari s-au dezvoltat o serie de algoritmi de procesare specifici calcului paralel.

Astazi industria semiconductoarelor a trecut de la cresterea frecventelor ceasurilor interne la marirea numarului de nuclee de procesare. Procesoare dual-core si quad-core sunt acum des intalnite. Spre exemplu procesorul Intel Core i7 are 4 nuclee si ofera un varf de putere computationala de 140 GFLOPS (operatii in giga virgula mobila si precizie singulara) si o banda in afara chipului a memoriei de 32 GB/s. Aceasta evolutie extraordinara a hardwarului pune o mare presiune pe dezvoltatorii de aplicatii, care acum sunt nevoiti sa expuna catre chip un paralelism suficient de performant. La extrema, exista numeroase sisteme precum platformele programabile dezvoltate de Nvidia, Amd sau Intel, care prezinta mai multe de 10 nuclee de procesare.

METODA

In vederea dezvoltarii de aplicatii pe asemenea sisteme multiprocesor trebuie considerate 3 aspecte: gradul de paralelizare, resursele hardware, contrangeri in timp real. Pentru ca un program sa ruleze in paralel trebuie sa fie partitionat intr-o serie de fire de executie care comunica intre ele si depind unul de altul. Aceasta impartire implica pe de o parte partitionare dar si planificare a proceselor. Planificarea este procedeul de distributie pe fiecare procesor a unui fir de executie si poate fi : statica sau dinamica. In cazul planificarii statice firele de executie si ordinea de executie sunt cunoscute inaintea executiei propriuzise. Algoritmii de planificare statica necesita un volum de comunicare mai mic intre procese si sunt pretabile pentru aplicatii in care comunicarea interproces implica costuri ridicate. In cazul planificarii dinamice firele de executie cat si alocarea lor este realizata in timpul rularii. Aceasta tehnica permite o incarcare egala a procesoarelor si ofera flexibilitate in utilizarea unui numar variabil de procesoare. Incovenientele existente pot fi reprezentate de structura programului ce devine foarte dificil de inteles, volumul de comunicare,sincronizare si competitie intre procese este unul imens, detectia atingerii maximului de incarcare computationala pe un procesor, analiza performantei in timpul rularii devenind cateodata inposibila datorita alocarii dinamice a firelor de executie.

PRELUCRARILE IN PARALEL

Maparea unui set de algoritmi pe o platforma multicore necesita untilizarea unui model de programare paralela care sa controleze comunicarea, sincronizarea, concurenta a tuturor componentelor din cadrul aplicatiei. Modelele de progrmare paralela s-au dezvoltat alaturi de noile arhitecturi multicore. Ele sunt de regula oferite drept o extensie a limbajelor de programare actuale, precum si prin interfete de programare (API) alaturi de C/C++, decat separat ca un now limbaj de programare paralela. Modele precum MPI(Message Passing Interface) sau UPC (Unified Parallel C) sunt utile pentru sisteme distribuite de scala mare spatiu de adresare separat, ce nu se gasesc in hardware-ul utilizat zi de zi. In sistemele de mici dimensiuni unde toate nucleele au acces la un singur spatiu de adrese, programatorii dezvolta aplicatii utilizand suportul thread-urilor, pentru ca un thread poate accesa datele altui thread. Modelul de programare paralela cu threaduri POSIX ofera mecanisme sofisticate de sincronizare si blocare, fiind dintre toate cel mai putin restrictiv si mai ales expresiv. Totodata utilizarea corecta a acestor

metode sincronizare si blocare constituie insa o mare provocare pana si pentru cei mai experimentati programatori.

Threadurile (fire de execuţie) reprezintă o modalitate software de îmbunătăţire a performantelor de calcul prin reducerea costului de comutare a proceselor. Un thread este un proces mai uşor, cu o stare redusă. Reducerea stării se obţine prin gruparea unui număr de threaduri corelate intre ele pentru a partaja diferite resurse de calcul, ca de exemplu, memoria şi fişierele. În sistemele bazate pe threaduri, threadul devine cea mai mică entitate de planificare, iar procesul serveşte ca un mediu de execuţie a threadurilor. În astfel de sisteme, un proces cu un singur thread este identic cu un proces clasic. Fiecare thread reprezintă un flux separat de execuţie şi este caracterizat prin propria sa stivă şi stare hardware(registre, flaguri). De vreme ce toate celelalte resurse, cu excepţia procesorului, sunt gestionate de către procesul care le înglobează, comutarea între threadurile care aparţin aceluiaşi proces, care implică doar salvarea, respectiv restaurarea,stării hardware şi a stivei, este rapidă şi eficientă. Totuşi comutarea între threadurile care aparţin unor procese diferite implică tot costul de comutare a proceselor. Threadurile, după implementare, se împart în kernelspace şi userspace.

Când threadurile sunt implementate în kernel, schimbarea contextului se face de către kernel fără ca aplicaţia să aibă cunoştinţă de aceasta şi de accea aceste implementări sunt preemptive. În schimb, când threadurile sunt implementate în userspace atunci schimbarea contextului se face de către librăria threadului, la nivelul aplicaţiei, şi în cazul acesta threadurile pot fi preemptive sau nonpreemptive, sau se pot alege în funcţie de implementare diferite versiuni. De asemenea e posibilă combinaţia kernel space şi userspace în momentul în care se lansează mai multe threaduri la nivel userspace în interiorul unui thread kernelspace. Dezavantajul unui thread la nivel user: nu poate beneficia de multiprocesoare. Insa putem restrictiona excutarea threadului pe un anume procesor, prin exlcuderea celorlalte.

Threadurile sunt un mecanism eficient de exploatare a concurenţei programelor. Un program poate fi împărţit în mai multe threaduri, fiecare cu o execuţie mai mult sau mai puţin independentă. Threadurile comunică între ele prin accesul la spaţiul de adresă a memoriei procesului, pe care îl partajează. Threadul poate fi privit ca o funcţie specială şi astfel toate caracteristicile funcţiilor din C sunt regăsite şi la threaduri. În sistemele cu memorie partajată (multiprocesoare) execuţia fiecărui thread de către un procesor separat (dacă sunt implementate la nivel de kernel space) asigură accerarea paralelă a programelor.

Chiar şi în sistemele uniprocesor, împărţirea unui program în threaduri poate aduce îmbunătăţiri prin execuţia concurentă a mai multor sarcini de calcul (de exemplu, în cazul interfeţelor grafice). În sistemul de operare Unix tradiţional nu sunt definite threaduri, fiecare proces conţine un singur thread, iar alocarea timpului de execuţie al procesorului se face între procesele active, după diferiţi algoritmi de planificare. În sistemele de operare derivate din Unix, dezvoltate pentru multiprocesoare şi multicalculatoare, sau implementat diferite facilităţi pentru controlul şi partajarea resurselor multiple (procesoare, memorie), între threaduri. Cea mai utilizată modalitate de implementare a threadurilor în sistemele derivate din UNIX sau pe sistemele multiprocesoare este aceea definită în standardul POSIX (IEEE POSIX Standard 1003.4a), care asigură portabilitatea între platformele hardware diferite. Threadul POSIX este denumit pthread. O aplicaţie cu threaduri POSIX este un program C care utilizează diferite funcţii din biblioteca POSIX pentru crearea, definirea atributelor, controlul stărilor (terminare, detaşare) threadurilor. Aceste funcţii sunt definite întro bibliotecă (libpthread.a), care trebuie adaugată (la linkare) programului apelant (prin opţiunea de compilare –lpthread). Lucrările descrise în continuare se pot executa în orice sistem de operare care are prevăzută extensia POSIX pentru threaduri.

Acest articol pune accentul pe programarea utilizand threadurile POSIX utilizand sistemul de operare Linux. Numita simplu, CPU affinity reprezinta tendinta unui proces de a rula pe un CPU definit atat timp cat este posibil. Kernelul din Linux de la versiunea 2.6 are un planificator de procese ce poate forta aceasta referinta catre un anumit procesor, sau un anumit set de procesoare, utilizand un anume set de functii . Toate procesele au asociata o structura de date importanta pentru toate setarile ce se pot realiza. Una dintre aceste setari este masca *cpu_allowed* ce consta intr-o serie de biti pentru fiecare procesor logic din sistem.

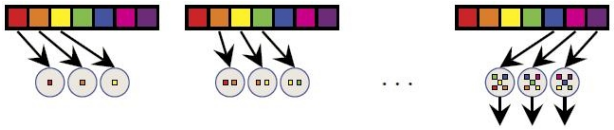
•pthread_attr_set_affinity_np(&attr,sizeof(cpu_set_t),&mask)) (seteaza afinitatea in atributul unui thread)

•sched_get_affinity() (returneaza afinitatea curenta)

• CPU_ZERO initializeaza toti bitii din masca la zero (CPU_ZERO_S(sizeof (cpu_set_t), &mask);)

• CPU_SET seteaza doar bitul corespunzator unui cpu in masca (CPU_SET_S(CPU_number, sizeof

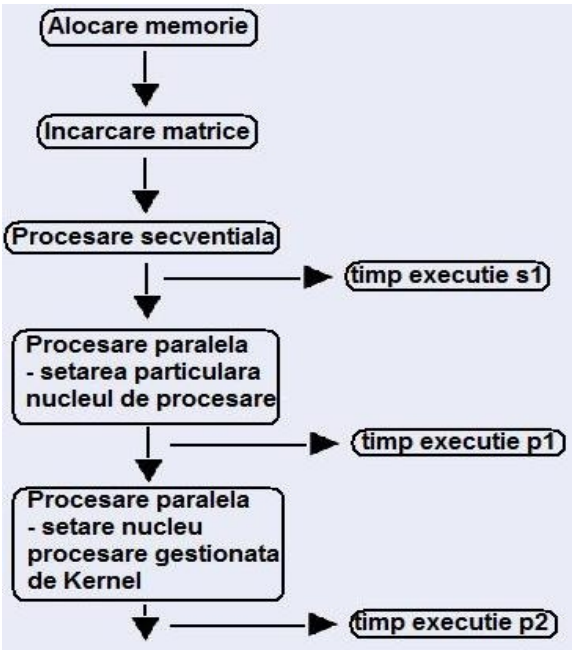
(cpu_set_t), &mask);)



[FIG 2] Exemplu paralelism

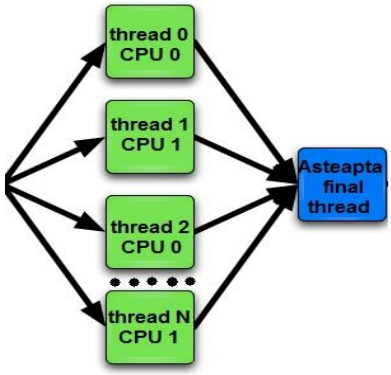
SIMULARE

In vederea simularii am utilizat libraria de threaduri POSIX pentru paralelizarea de operatii pe 2 matrice de mari dimensiuni,valorile fiind incarcate din nivelul de gri al pixelilor unor imagini satelitare de rezolutie mare. Programul a fost dezvoltat in limbajul C si s-a realizat o comparatie intre timpul secvential de rulare, timpul de prelucrare paralela cu alocarea dinamica a nucleelor de procesare, prelucrarea paralela cu alocarea nucleelor de procesare lasata la latitudinea mecanismului de guvernare a proceselor implementat in sistemul de operare Linux (Kubuntu).



[FIG 3] Schema bloc a programului

Procesorul folosit in simulari au fost Intel Core2Duo 2 GHz iar rularea in paralel se executa astfel : numarul de threaduri ce se doresc a fi lansate este variabil, fiecare thread gestionand inmultirea unui procent de linii proportional cu numarul de threaduri. Rulare in paralel cu selectarea nucleului de procesarea lanseaza cate un thread pe fiecare procesor setandu-se afinitatea threadului catre un anumit procesor prin intermediul mastii ce desemneaza CPU – urile active.. Rulare in paralel cu algoritmul de distribuire pe procesoare implementat de kernelul din Linux lanseaza toate threadurile lansand alocarea nucleului de procesare la latitudinea guvernatorului de procese ce monitorizeaza activitatea procesorului.



[FIG 4] Paralelizarea operatiilor

REFERINTE

[1] NITA Iulian, LAZARESCU Vasile, CONSTANTINESCU Rodica, *Dynamic intelligent task mapping for heterogeneous MPSoC applications,* Conference Journal : Electronics, Computers and Artificial Intelligence 3-5 June, 2009, Piteşti, ROMÂNIA

[2] MICHELA BECCHI, PATRICK CROWLEY, *Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures*, Conference Journal of Instruction-Level Parallelism 10 (2008) 1-26, Submitted 10/06; published 6/08, 2008.

[3] GEOFFREY BLAKE, RONALD G. DRESLINSKI, TREVOR MUDGE, *A Survey of Multicore Processors*, IEEE Signal Processing Magazine (Volume 26 Number 6 November 2009 : Multicore Platforms in the signal processing world, Part 1)

[4] WAYNE WOLF, *Multiprocessor System-On-Chip Technology*, IEEE Signal Processing Magazine (Volume 26 Number 6 November 2009 : Multicore Platforms in the signal processing world, Part 1)

[5] DENNIS LIN, XIAOHUANG HUANG, QUANG NGUYEN, JOSHUA BLACKBURN, CHRISTOPHER RODRIGUES, THOMAS HUANG, MINH N. DO, SANJAY J. PATEL, WEN-MEI W. HWU, The parallelization of video processing, IEEE Signal Processing Magazine (Volume 26 Number 6 November 2009 : Multicore Platforms in the signal processing world, Part 1)

SIMULARE

TS : timp serial ; TP : timp paralel ; 0/1 : metoda

Dim. Matr. / Nr. Thread	TS (s) (serial)	TP0 (s) / Castig	TP1 (s) / Castig
300 / 30	0.313621	0.217343	0.244995
		1.442977	1.280112
300 / 100	0.315248	0.262647	0.299183
		1.200273	1.053696
300 / 300	0.339238	0.357866	0.594193
		0.947947	0.570922
1000 / 100	20.343330	9.479294	9.545338
		2.146081	2.131232
1000 / 250	17.635264	9.615566	9.484441
		1.834033	1.859389
3000 / 300	507,12	321.561792	323,99
		1.577059	1.565247
3000 / 350	502,64	308,56	310.486816
		1.629001	1.618881

[TABEL 1] Date simulare Seriala / Paralela

Se realizeaza detectia numarului de procesoare din sistem si se executa inmultirea a 2 matrice de dimensiuni mari selectabile. Executia are 3 etapa : etapa secventiala, etapa paralela cu selectia dinamica a procesorului pe care se lanseaza un thread, etapa paralela cand threadurile sunt lansate iar guvernarea lor este lasata la indemana kernelului.

Rezultatele sunt exportante in secunde pentru o comparatie clara.

```
Introduceti dim matricei: 5000
Alocare de memorie ....
GENERARE DONE ....
Calcul secvential in derulare ....
Calcul secvential : DONE ....
INFO: Sistemul are 2 procesoare !

Vor fi lansate 640 threaduri
Calcul paralel 1 in derulare ...
S-au lansat toate threadurile ...
S-au asteptat toate threadurile. CP1 : DONE
Verificare terminata
Calcul paralel 2 in derulare ... S-au asteptat toate threadurile. CP2 : DONE

Timp serial vs. Timp paralel cu alocare de CPU
Timp 1 =3.875816 s, timp 2 =2.216318 s
Speedup =1.748763 cu 640 threaduri

Timp serial vs. Timp paralel cu CPU alocat de kernel
Timp 1 =3.875816 s, timp 2 =2.885338 s
Speedup =1.343280 cu 640 threaduri
```

[ANEXA 1 : CODUL SURSA C]

```
#define __USE_GNU
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/sysinfo.h>
#include <sys/mman.h>
#include <sched.h>
#include <ctype.h>
#include <string.h>

int dim;                /* dimensiunea matricelor               */
int nr_thread;          /* numar threaduri                      */
float *mata,*matb;      /* matricele a si b                     */
float *matc;            /* matricea pentru lucrul secvential    */
float *matc1;           /* matricea pentru lucrul paralel       */

pthread_t *vector;
cpu_set_t mask;
pthread_attr_t *attr;

void *thread_calcul(void *in) { /* calcul produs */
      int where=*(int *)in;
      int i,j,k,l;
      for(i=where;i<dim;i=i+nr_thread)
         for(j=0;j<dim;j++) {
                     matc1[i*dim+j]=0.0;
                     for(k=0;k<dim;k++)
                        //printf(" %d %d %d \n",i,j,k); fflush(stdout);
                        matc1[i*dim+j] += mata[i*dim+k]*matb[k*dim+j];

         }
      pthread_exit(NULL);
} // end thread_calcul

void speedup(struct timeval t1ser,struct timeval t2ser,struct timeval
t1par,struct timeval t2par,int P);

int main() {
  int       i,j,k;
  int       *index;
  float *temp;
  struct timeval t1s,t2s;               /* timpi pentru calcul secvential  */
  struct timeval t1p,t2p,t1p2, t2p2;    /* timpi pentru calcul paralel     */

  int       STACKSIZE;

  printf("Introduceti dim matricei: ");  fflush(stdout);
  scanf("%d",&dim); fflush(stdin);
  printf("Alocare de memorie .... "); fflush(stdout);
  /* alocare de memorie pentru matrici */
  mata      = (float *)calloc(dim*dim,sizeof(float));
  matb      = (float *)calloc(dim*dim,sizeof(float));
```

```
matc      = (float *)calloc(dim*dim,sizeof(float));
matc1     = (float *)calloc(dim*dim,sizeof(float));

/*generarea matricilor */
for(i=0;i<dim;i++)
  for(j=0;j<dim;j++) {
    mata[i*dim+j]= 1.0 * (float)(15.0*rand()/(3+1.0));
    matb[i*dim+j]= 1.0 * (float)(15.0*rand()/(4+1.0));
  }
printf("\nGENERARE DONE .... "); fflush(stdout);

/* Calcul secvential */
gettimeofday(&t1s,NULL);
for(i=0;i<dim;i++) {
    for(j=0;j<dim;j++) {
          matc[i*dim+j]=0.0;
          for(k=0;k<dim;k++)
                matc[i*dim+j]+=mata[i*dim+k]*matb[k*dim+j];
    }
}
gettimeofday(&t2s,NULL);
printf("\nCalcul secvential : DONE .... "); fflush(stdout);

/* Determina numarul de procesoare ale sistemului */
int NUM_PROCS = sysconf(_SC_NPROCESSORS_CONF);
printf("\nINFO: Sistemul are %i procesoare !\n", NUM_PROCS); fflush(stdout);

/*zona paralela*/

printf("Introduceti nr de threaduri = ");    fflush(stdout);
scanf("%d",&nr_thread);                  fflush(stdin);
printf("\nVor fi lansate %d threaduri \n ",nr_thread); fflush(stdout);

/* vectorul de id-uri */
vector = (pthread_t *)calloc(nr_thread,sizeof(pthread_t));
index  = (int *)calloc(nr_thread,sizeof(int));
attr       = (pthread_attr_t *)calloc(nr_thread,sizeof(pthread_attr_t));

int core_number = 0;
size_t stacksize;
                              /*    zona de calcul paralel  */
gettimeofday(&t1p,NULL);      /*    start calcul paralel    */
for(i=0;i<nr_thread;i++) {

  index[i] = i;
  /* CPU_ZERO initializeaza toti bitii din masca la zero */
  __CPU_ZERO_S(sizeof (cpu_set_t), &mask );

  /* CPU_SET seteaza doar bitul corespunzator unui cpu in masca */
  __CPU_SET_S( core_number, sizeof (cpu_set_t), &mask );

  if(!pthread_attr_init(&attr[i])) {
    /* seteaza afinitatea prin attr pentru un CPU anume */
    if ((pthread_attr_setaffinity_np(&attr[i],sizeof(cpu_set_t),&mask)) < 0) {
          printf("WARNING:CPU Affinity nu s-a putut seta ! Continuing...\n");
          fflush(stdout);
    }

    /* seteaza stiva threadului sa fie mai mica decat cea alocata de kernel */
```

```
    STACKSIZE = 65536*16; // 1MB Stiva
    pthread_attr_setstacksize(&attr[i],STACKSIZE);

    /* creaza si lanseaza thread */
    pthread_create(&vector[i],&attr[i],thread_calcul,(void *)&index[i]);

    if ( __CPU_ISSET_S (core_number, sizeof (cpu_set_t), &mask) != 0 ) {
          //printf("Pe CPU %d s-a pornit un thread ! \n",core_number);
          fflush(stdout);
    }

  } else {
    printf("Nu s-a putut crea threadul \n"); fflush(stdout);
  }

  core_number += 1;

  if (core_number == NUM_PROCS) core_number = 0;
}
printf("S-au lansat toate threadurile ... \n");    fflush(stdout);

for(i=0;i<nr_thread;i++) pthread_join(vector[i],NULL);
printf("S-au asteptat toate threadurile. CP1 : DONE\n"); fflush(stdout);
gettimeofday(&t2p,NULL); /* final calcul paralel*/


/* verificare rezultat */
for(i=0;i<dim;i++)
    for(j=0;j<dim;j++)
          if((matc[i*dim+j]-matc1[i*dim+j])>1.0e-5) {
                printf("eroare la elem %f!=%f\n", matc[i*dim+j],
                matc1[i*dim+j]);
                fflush(stdout);
          }
printf("Verificare terminata\n"); fflush(stdout);
speedup(t1s,t2s,t1p,t2p,nr_thread);

/* rulare in paralel lasand kernel - ul sa aloce rularea pe procesoare*/

/* vectorul de id-uri */
free(vector); free(index); free(attr);
vector = (pthread_t *)calloc(nr_thread,sizeof(pthread_t));
index  = (int *)calloc(nr_thread,sizeof(int));
attr       = (pthread_attr_t *)calloc(nr_thread,sizeof(pthread_attr_t));

gettimeofday(&t1p2,NULL);  /* start calcul paralel */
for(i=0;i<nr_thread;i++) {
    index[i] = i;

    pthread_attr_init(&attr[i]);
    STACKSIZE = 65536*16; // 1MB
    pthread_attr_setstacksize(&attr[i],STACKSIZE);

    pthread_create(&vector[i],&attr[i],thread_calcul,(void *)&index[i]);
}
for(i=0;i<nr_thread;i++) pthread_join(vector[i],NULL);
gettimeofday(&t2p2,NULL); /* final calcul paralel*/
printf("S-au asteptat toate threadurile. CP2 : DONE\n"); fflush(stdout);
```

```
  /* verificare rezultat */
  for(i=0;i<dim;i++)
      for(j=0;j<dim;j++)
            if((matc[i*dim+j]-matc1[i*dim+j])>1.0e-5){
                  printf("eroare la elem %f!=%f\n", matc[i*dim+j],
                  matc1[i*dim+j]);
                  fflush(stdout);
            }

  /* rezultate simulare */

  printf("\nTimp serial vs. Timp paralel cu alocare de CPU \n");
      fflush(stdout);
  speedup(t1s,t2s,t1p,t2p,nr_thread);
  printf("\nTimp serial vs. Timp paralel cu CPU alocat de
      kernel\n");fflush(stdout);
  speedup(t1s,t2s,t1p2,t2p2,nr_thread);

  /* eliberare memorie*/
  free(mata); free(matb); free(matc); free(matc1);

} // end main

void speedup(struct timeval t1ser,struct timeval t2ser,struct timeval
t1par,struct timeval t2par,int P) {
      long zecimal1,intreg1;
      long zecimal2,intreg2;
      float operand1,operand2;
      float speedup;
      if(t1ser.tv_usec>t2ser.tv_usec){
            zecimal1=1000000+t2ser.tv_usec-t1ser.tv_usec;
            intreg1=t2ser.tv_sec-t1ser.tv_sec-1;
      } else {
            zecimal1=t2ser.tv_usec-t1ser.tv_usec;
            intreg1=t2ser.tv_sec-t1ser.tv_sec;
      }
      if(t1par.tv_usec>t2par.tv_usec) {
            zecimal2=1000000+t2par.tv_usec-t1par.tv_usec;
            intreg2=t2par.tv_sec-t1par.tv_sec-1;
      } else {
            zecimal2=t2par.tv_usec-t1par.tv_usec;
            intreg2=t2par.tv_sec-t1par.tv_sec;
      }
      operand1=1000000*intreg1+zecimal1;
      operand2=1000000*intreg2+zecimal2;
      speedup=operand1/operand2;
      printf("Timp 1 =%f s, timp 2 =%f s\n",operand1/1000000.0,
      operand2/1000000.0);fflush(stdout);
      printf("Speedup =%f cu %d threaduri\n",speedup,P);fflush(stdout);
}
```