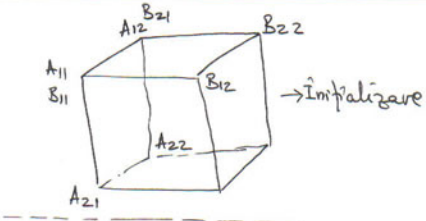
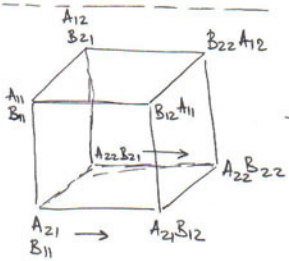


→ Inițializare



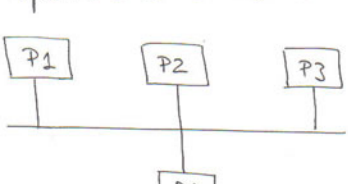
→ A în dreapta, B în jos; fiecare înmulțește ce a primit

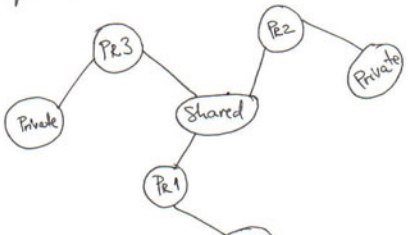
inițializare minimală

SPM.18

Standardul MPI

Open MPI se adresează sist. multiprocesor.





Fiecare fir de execuție beneficiază de un set de variabile proprii. Ptr. comunicație ∃ memoria partajată.

Oricum am face implementarea, MPI-s vor avea o zonă privată de memorie și vor partaja o zonă comună, atât la progr. multitasking cât și la progr. pe proces.

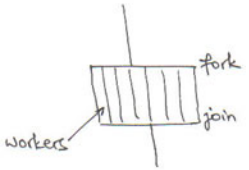
Stdrdul. MPI-e un stdrd. deschis → www.openmp.org

Standardul își propune:

→ standardizarea directivelor de calcul || (C/C++ & Fortran) ⇒ nu se lucrează cu reg. critice, bariere, etc → toate instrucțiunile sunt aduse în aceste directive;

→ mai ∃ și librării accesibile care permit apelarea unor funcții, ptr. a rafina programarea // ⇒ semafoare, bariere, etc.;

Stilul de tipul Fork-Join. Directiva Fork împarte (sparge) procesul principal în mai multe fire paralele. Când s-au sincronizat, acestea se unesc din nou (Join).



MP inițial e master, urm. ulterioare sunt workers. Toți workers formează o echipă (team). Masterul este singurul care deține ID-urile de proces ale tuturor workers ⇒ master este sg. care poate face handle.

Ex.: • directivele încep cu #pragma omp. Urmează clauzele.

```
#pragma omp parallel for default (none) private (i,j,sum)
                         shared (m,n,a,b,c)
for (i=0; i<m; i++)        //înmulțire mtx. cu vector
  { sum=0;
    for (j=0; j<n; j++)
        sum = sum + b[i][j] * c[j];
    a[i]= sum;
  }
```

directiva parallel ⇒ specifică cum va fi executată secțiunea; ~~f~~

for ⇒ specif. că sect. se va exe. // ptr. toate procesele

default (none) ⇒ face referire la cum sunt considerate variabilele proc. maintea acestei secțiuni

nu se folosesc (aici se ignoră tot ce a fost declarat anterior în secțiunea de calcul //)

shared ⇒ setul de var. din mem. comună (var. comune)

private ⇒ dtlc. separate ale fiecărui fir de exe. (copii distincte ptr. fiec. fir)

Acest stilul permite ca un progr. scris în C sub Linux compatibil Open MP să fie portat pe o platf. C sub Windows compatibl. Open MP.

Aceste directive ne scutesc de testarea situațiilor de deadlock etc. (elimină analiza de pb apărute la programarea // ptr. că librăriile ~~s~~ sunt perfect sigure).

∃ directive legate de granularitate → câte µPs sunt, câte taskuri → procese
pb. rezolvate automat.

La compilare, for-ul inițial e desfăcut, în cazul în care avem 2 µPs, în 2 for-uri separate:

for (i=0; i<[m/2]; i++)
for (i=[m/2]+1; i<m; i++)

⇒ for a fost distribuit la compilare

[parallel arată că instr. urm. va fi exe. în ||]

Granularitatea poate fi specif. de programator, dar în cazul în care nu e specif. compilatorul împarte automat totul la câte fire de execuție avem.

! La exam: sub. legat de ↑ e făcut de Radu Pietraru ⇒ nu va da teorie!!! ⇒ tb. citite tutorialele (unul bun e cel de la SUN).
(ce a predat azi)

Ex.: #pragma omp parallel if (condiție) ...

Rularea efectivă pe o anumită mașină e lăsată în seama compilatorului (nu se pune pb. de câte µPs. am) ⇒ dc. vreau exe. || mai mult de pr. nivel, nu mai pot vedea (verifica) ce face compilatorul

```
#pragma omp parallel if (cond) default (none) ~~shared(m,a,b,c,x~~
        shared(m,a,b,c,x,y,z) private(fi,scale)
{ #pragma omp for //restul e precizat înainte ↑
      { for (i=0;i<m;i++)   //acest for se distribuie
            z[i] = x[i] + y[i];
      }
  #pragma omp for nowait  //dc. vrem să nu se sincronizeze
                          //automat procesele
}
```

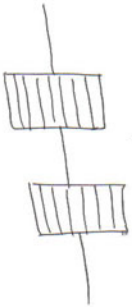
⇓

un thread executat nu mai tb. să le aștepte și pe celelalte

Obs.: când paralelizarea se face pt. un set f. mic de dte., e mai avantajos calculul ||.

var. private → au durata de viață limitată la durata firului de exe.
(∄ înainte și după execuția firului respectiv)

var. shared → sunt vizibile tuturor firelor de execuție, cu specificații specifice fiecărui fir; apoi au o stare nedefinită.



→ așa se comportă programul de mai sus; tot ce e în afara progr. e cu totul altceva.

Ex.: Dc. vrem să fol. o var. de dinaintea lui #progma... ∃ 2 opțiuni private:

A=10; // va fi vzt. în toate firele de execuție

#progma omp parallel for private(i) firstprivate(A) lastprivate(B)

↳ B va fi disponibil și după exe. threads

lastprivate → va fi dat de ultimul thread care se încheie

private, ptr. că se fac copii după val respectivă

Dc. în loc de private(i) am default atunci A=10 nu va fi modificat (când toate threads s-au încheiat, A=10 este văzută în continuare).

default(none) → A=10 nu este folosită, adică se fac copii după ea, și se folosesc acele copii.

O altă directivă este reduction(operator: list) ⇒ se face o colectare a mai multor valori din threaduri.

list → listă parametri

operator → crează o var. privată ptr. fiecare thread și fol. respectiva operație

Ex.: #progma omp parallel

- for // încearcă paralelizare de dte.
- sections // permite distribuția ⇒ paralelizare fețională (altă secv. de instr. în alt fir)
- single // secv. de instr. e exe. în master → nu crează alt fir

Deci în locul lui for se poate folosi oricare din celelalte două.

Forțarea granularității:

#pragma omp parallel schedule | static (chunk)
dynamic (chunk)
guided (chunk)

→ secvența de împărțire

ex.: chunk = 2:

P_1	P_2
0-1	2-3
4-5	6-7
⋮	⋮

dc 0-1 nu s-ar fi terminat ar fi primit tot P_2 pe 4-5

Compilatorul face împărțirea constantă a volumului de lucru ptr. fiecare thread. Dar dc. avem intrv. sup/inf. Δ unele threads muncesc, altele nu ⟹ ⟹ schedule se fol. ptr remediere.

Schedule → static: distribuția se face la compilare
→ dynamic: ——— la rulare → librăriile permit implem. unor controale referitoare la gestiunea proceselor
→ guided

chunk = setul de dte. / nr. de procesoare

La guided alocarea se face ca și la dynamic, dar P_1 și P_2 primesc câte o porție variabilă, iar aceasta porție e redusă exponențial.

Mecanisme de barieră:

```
for { for (i=0; i<N; i++)
        a[i]=b[i]+c[i];
    }
for { for (i=0; i<N; i++)
        d[i] = a[i] + b[i]
    }
```

⟹ apare dependență de dte.!
⟹ th. declarată o barieră care așteaptă încheierea (↑) proces diferit de master (sincronizare forțată a proceselor, până rămâne doar masterul)

Bariera: #pragma omp barrier

Dacă nu pun no wait toate procesele așteaptă → pun no wait și apoi o barieră forțată care nu permite trecerea procesului master până când toate celelalte nu sunt gata. În mod implicit fiecare for are deja o barieră deja setată.

Ex.: directive sections, fiec. proces calc. un termen al unei sume (paralel)

```
#pragma omp critical
    sum = sum + ...
/* în mom. acesta, în bucata asta de cod se efectuează un sg. task
   => nu e foarte eficient */
```

Și și directivă de operație atomică:

```
#pragma omp atomic // tot ce urmează nu poate fi întrerupt
#pragma omp ordered // op. din secțiunea respectivă sunt serializate
#pragma omp flush // asignră coerență unei var comune
                  // are referire la mem. cache
```

Variabile compilator → pot fi modificate

OMP_NUM_THREADS → câte fire de execuție pot fi rulate în // (implicit 2)

OMP_SCHEDULE → nr. iterații

OMP_DYNAMIC → implicit e true (alocarea se face dinamic)

OMP_NESTED → dc. se acceptă sau nu implicări de directive // (implicit e FALSE)

Pe lângă aceste directive, se mai poate lucra cu funcții incluse în bibliotecă.

Ex.: omplib.h (omp_lib.h)

omp_init_locks
- nu destroy - u
- - set - u

omp_uset_locks

Mai sunt disponibile și parametri ce pot fi modif. dinamic în timpul rulării: