OpenMP- model de programare paralelă a sistemelor cu memorie partajată

OpenMP este cel mai răspândit model pentru programarea paralelă a maşinilor cu memorie partajată, oferind paralelism de tip fork-join explicit în care un thread master creează și coordonează multiple threaduri, model parametrizabil prin intermediul unor construcții de tip directive și clauzele asociate acestora , respectiv utilizând variabile de mediu și funcții de bibliotecă pentru crearea de taskuri paralele/concurente cu execuție controlată prin construcțiile de sincronizare.

3.1. Obiective

- Studiul modelului de programare paralelă OpenMP
- Implementarea paralelă a unor algoritmi în modelul cu memorie partajată

3.2. Concepte

Modelul OpenMP a fost lansat în 1997 (gestionat de OpenMP ARBwww.openmp.org) și a evoluat continuu ajungând în prezent la versiunea 5.0.(2018) , fiind utilizat pentru dezvoltarea aplicațiilor paralele pe platforme diverse de la sisteme desktop până la supercomputere. În cadrul acestui model, threaduri cooperante rezolvă sarcini (taskuri) prin execuție simultană pe procesoare sau core-uri multiple.
Modelul de programare OpenMP se bazeaza pe:

- *memorie partajata și paralelism bazat pe threaduri* : un proces cu memorie partajată poate consta în fire de execuție multiple. OpenMP se bazează pe existența firelor multiple în paradigma programării cu partajarea memoriei.
- *paralelism explicit*: OpenMP este un model de programre explicit (nu automat), care oferă programatorului deplinul control asupra paralelizării.
- *modelul fork-join* : OpenMP uzează de un model al execuției paralele alcătuit din ramificații și joncțiuni vezi (vezi Figura 3.1.)

Directivele compilator sunt utilizate ca extensii pentru limbajele secvențiale C++/Fortran oferind construcții pentru creare de taskuri, partajare sarcini , sincronizarea threadurilor .

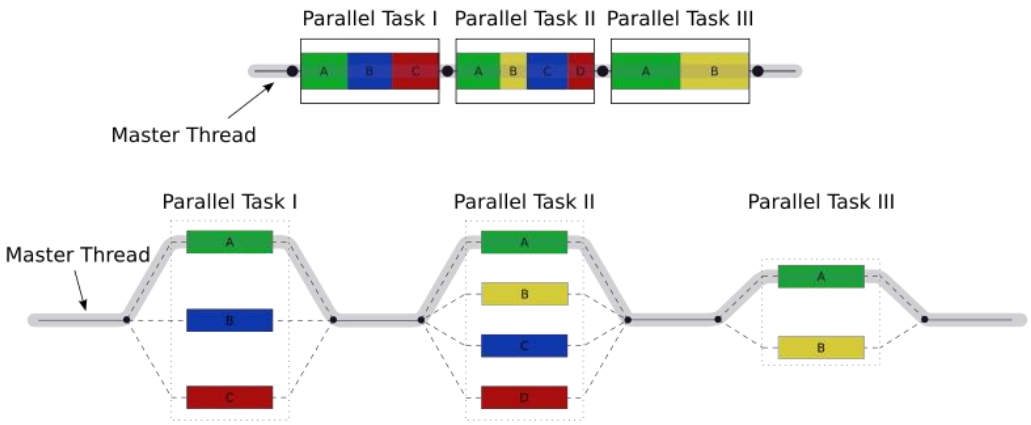

Figura 3.1. Modelul fork-join OpenMP [1]

Setul de directive OpenMP permite specificarea acţiunilor compilator necesare execuţiei codului program în paralel fără însă a verifica *dependenţele, conflictele, blocajele* posibile, programatorul fiind astfel responsabil în totalitate de proiectarea corectă a aplicaţiei. Modelul nu este aplicabil sistemelor cu memorie distribuită, nu este implementat în mod identic de către toți producătorii (de ex. Intel/Microsoft) și nu este garantat că ar asigura cea mai eficientă utilizare a memoriei partajate deoarece nu există pentru moment constructori de localizare a datelor , dar este de așteptat a deveni în viitor un standard ANSI.

3.2.1. Modelul de programare și modelul de memorie

Standardul OpenMP oferă o interfaţă (API) de *programare independentă de platformă* (există implementări Linux/Windows), integrând o serie de construcţii pentru managementul proceselor, distribuţia taskurilor, concurenţă și sincronizare a threadurilor şi gestionarea datelor. Modelul de programare oferă deasemenea construcţii pentru controlul partajării și consistenţei memoriei utilizate de aplicație și suport pentru paralelism imbricat.

Regiunea paralelă este construcţia ce defineşte o secţiune de program sub forma unui *bloc structurat* (marcat prin intermediul acoladelor*), executat în paralel de procesoarele sistemului.* Programul îşi începe execuţia cu un singur thread, iar în momentul întâlnirii primei regiuni paralele, thread-ul master, având id-ul 0 , creează o echipă de thread-uri (după modelul fork / join prezentat în figura 3.1.) şi se include pe sine în echipă, threaduri ce vor executa regiunea paralelă prin planificarea (alocarea) iteraţiilor ciclurilor paralele acestor threaduri. Această operație de tip fork presupune că la finele regiunii paralele există o barieră *implicită* pentru sincronizare și doar threadul master continuă execuția ulterior acesteia. *Numărul de threaduri* poate fi specificat în directivă, stabilit folosind variabile de mediu sau în mod dinamic la runtime prin funcţii OpenMP. Dacă un thread modifică un obiect partajat, acest fapt va afecta nu doar propriul context de execuţie ci şi pe cele ale celorlalte threaduri ale programului care au acces și pot utiliza respectivul obiect. Asignarea specifică de taskuri diferitelor threaduri este deasemenea posibilă prin intermediul unor directive specifice ce vor fi prezentate în continuare.

Modelul de memorie distinge între *memorie partajată și memorie privată*, toate threadurile având acces la memoria partajată (modelul implicit), iar pentru evitarea blocajelor sau a conflictelor sunt prevăzute mecanisme de sincronizare. Complementar memoriei partajate, un thread poate utiliza variabile private în memoria privată proprie, variabile ce nu pot fi accesate de alte threaduri.

Extensiile de limbaj oferite de standardul OpenMP sunt structurate în următoarele categorii : control paralel, partajare sarcini, mediu de date, sincronizare funcţii runtime și variabile de mediu (vezi Figura 3.2.) și vor fi abordate în continuare.

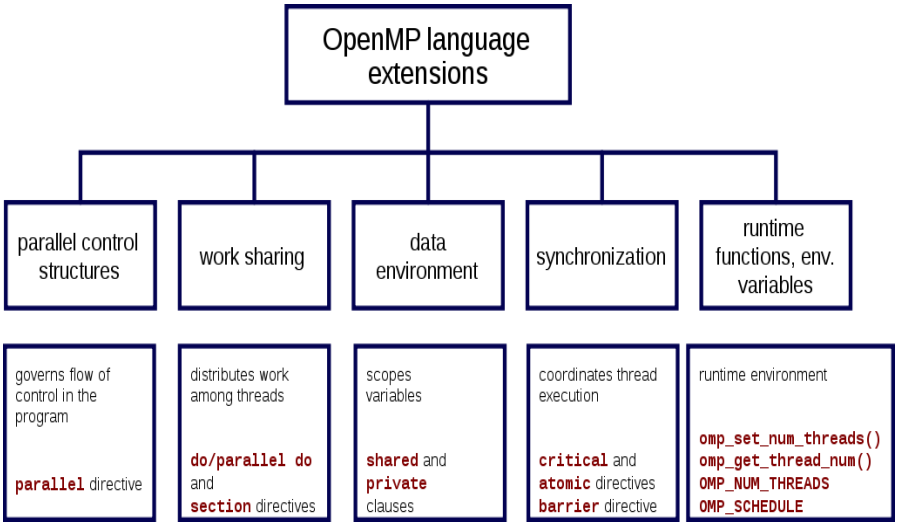

Figura 3.2. Tipuri de construcţii OpenMP [1]

Directive. O directivă este o linie specială de cod sursă având semantică utilă doar anumitor compilatoare ce integrează suport pentru expandarea și execuția sa. Directivele sunt case - sensitive, iar ordinea execuţiei clauzelor directivei este nesemnificativă.

Orice directivă este de forma: **#pragma omp.** Sintaxa directivelor este:

#pragma omp *[clauză[[,]clauză]. . .]linie nouă*

Clauze. Specificarea de informaţii adiţionale în diverse directive, inclusiv directiva de creare a unei regiuni paralele este realizată prin *clauze* ce pot fi specificate sub forma unei liste. Lista clauzelor permite specificarea paralelizării condiţionale, a numărului de threaduri de execuţie şi a modului de gestionare a datelor, cele mai importante categorii de clauze fiind:

- Specificarea datelor : acestea pot fi date partajate şi private. În regiunile paralele, variabilele pot fi *partajate sau private.* Toate thread-urile văd aceeaşi copie a variabilelor partajate, pot citi sau scrie variabilele partajate, însă fiecare thread are propria copie a variabilelor private, invizibile pentru celelalte thread-uri, astfel o variabilă privată poate fi citită sau scrisă numai de thread-ul căruia îi aparţine.
- Specificarea ciclurilor paralele. Ciclurile sunt principala sursă de paralelism, dacă iteraţiile unui ciclu sunt *independente* (pot fi executate în orice ordine), atunci se pot partaja iteraţiile între thread-uri diferite.
- Specificare unor operații pe date ,de exemplu reducţii. Reducţia este operaţia prin care se generează o singură valoare din operaţii asociative. Permiţând unui singur thread la un moment dat să actualizeze o variabilă partajată. Aceasta presupune eliminarea totală a paralelismului, dar este posibil ca fiecare thread să acumuleze în propria copie privată, iar apoi aceste copii sunt reduse pentru a furniza un rezultat final.

Compilare condiţională. Directivele respectă convențiile din standardele C/C++ pentru directivele de compilare. Sensibilă la upper/lower case. Pentru o directivă, numai un nume de directivă poate fi specificat. Fiecare directivă se aplică la cel mult o declarație următoare, care poate fi un bloc structurat. Liniile-directivă lungi pot fi continuate pe linii următoare prin combinarea caracterelor newline (linie-nouă) cu un caracter "\" la finalul liniei-directivă.

Domeniile de aplicabilitate a directivelor sunt :

- *domeniu static (lexical):* Codul inclus textual între începutul si finalul unui bloc structurat care urmează directiva. Domeniul static al unei directive nu acoperă rutine multiple sau fisiere de cod.
- *directive orfane:* Despre o directivă OpenMP care apare independent de o altă directivă care o include se spune că este o directivă orfană. Ea există în afara domeniului static (lexical) al altei directive. Acoperă rutine si posibile fisiere de cod.
- *domeniu dinamic:* Domeniul dinamic al unei directive include atât domeniul ei static (lexical) cât si domeniile orfanelor ei

Variabile de mediu. Variabilele de mediu permit controlul execuţiei paralele. Cele mai uzuale sunt:

OMP_NUM_THREADS -setează numărul de threaduri posibil a fi utilizate în timpul execuţiei programului.

OMP_SCHEDULE - setează tipul planificării run-time a execuției threadurilor (cu alocare de un anumit tip a taskurilor a threadurilor)

OMP_DYNAMIC - activează-dezactivează ajustarea dinamică a numărului de threaduri care pot fi utilizate pentru executarea unor zone diferite în paralel

OMP_NESTED - activează-dezactivează paralelismul imbricat (utilizarea imbricată a directivelor de tip parallel for)

Variabile partajate şi private. În interiorul unei regiuni paralele variabilele pot fi ***partajate*** - *toate thread-urile văd aceeaşi copie (modelul implicit),* sau ***private*** - *fiecare thread are propria copie asupra căreia operează* și al cărei conținut poat deveni accesibil ulterior celorlalte threaduri.
Clauzele SHARED, PRIVATE, DEFAULT pot fi utilizate pentru specificarea tipului variabilelor, conform sintaxei: *shared(list), private(list), default(shared|none).*

Câteva observaţii se impun referitor la declararea tipurilor variabilelor:

- majoritatea variabilelor sunt partajate (este modelul implicit)
- indicii ciclurilor sunt privaţi
- variabilele temporare ale ciclurilor sunt private

3.2.2. Regiuni paralele

Codul sursă din interiorul unei regiuni paralele este executat de toate thread-urile. Sintaxa directivei de creare a unei regiuni paralele este:

#pragma omp parallel *[clauză[[,]clauză]. . .]linie nouă*
bloc_ structurat

iar clauza poate lua una din valorile

if(*expresie _scalară***)**
private(listă _var**)**
firstprivate(listă _var**)**
default(shared | none)
shared(listă _var**)**
copyin(listă _var**)**
reduction(*operato***r:** listă _var**)**
num_threads(*expresie int.* **)**

Numărul de threaduri dintr-o regiune paralel este determinat de factorii următori, în ordinea prezentată: (1)se utilizează functia de bibliotecă omp_set_num_threads(), (2)se setează variabila de mediu OMP_NUM_THREADS, (3)implementarea default.

Există o serie de funcţii complementare cu ajutorul cărora poate fi controlată execuţia în cadrul regiunilor paralele prin stabilirea/determinarea numărului de threaduri active la un moment dat, astfel:

int omp_get_num_threads(void) - determină numărul de threaduri, funcţia returnează 1 dacă este apelată în exteriorul unei regiuni paralele

int omp_get_thread_num(void) - determină numărul threadului curent ,ia valori între 0 şi OMP_GET_NUM_THREADS() – 1

int omp_set_num_threads(num_threads) - setarea numărului de threaduri utilizabile în regiunile paralele următoare

int omp_get_max_threads(void) - funcţie ce returnează limita superioară a numărului de threaduri ce poate fi creat de directiva PARALLEL

int omp_get_num_procs(void) - funcţie ce returnează numărul maxim de procesoare ce pot fi utilizate pentru execuţia programului

Fire dinamice. Implicit, un program cu regiuni paralele multiple utilizează acelasi număr de fire pentru a executa fiecare dintre regiuni. Această comportare poate fi modificată pentru a permite la momentul executiei modificarea dinamică a firelor create pentru o anumită sectiune paralelă. Cele două metode disponibile pentru a permite fire dinamice sunt: fie utilizarea functiei de bibliotecă omp_set_dynamic(), respectiv setarea variabilei de mediu OMP_DYNAMIC.

Iniţializarea variabilelor private. Atunci când o variabilă este declarată privată, threadul primește adresa unde va stoca valorile variabilei pe durata regiunii paralele. La incheierea regiunii paralele, memoria este dealocată și aceste variabile nu mai există. Dacă se dorește reținerea valorii acestor variabile anterior/ulterior regiunii paralele se vor folosi clauzele FIRSTPRIVATE și

LASTPRIVATE. Variabilele private sunt neiniţializate la începutul regiunii paralele. Dacă se doreşte iniţializarea lor, se poate folosi clauza FIRSTPRIVATE: firstprivate(*list*).

Clauza de reducţie. O reducţie produce o singură valoare din operaţii asociative precum: adunarea, înmulţirea, max, min, ŞI (logic), SAU (logic). Operaţia este echivalentă următorului set de etape: fiecare thread reduce într-o copie privată, iar apoi toate aceste copii se reduc pentru a furniza rezultatul final, clauza utilizată este REDUCTION. Este permisă utilizarea şirurilor ca variabile de reducţie.
Sintaxa clauzei este*: reduction(op:list)*

Clauza IF. Directiva de regiune paralelă poate fi condiţională, specificare utilă situaţiilor în care nu există suficientă sarcină de procesare pentru a utiliza efectiv paralelismul.
Sintaxa este: *if (scalar expression)*

3.2.3. Directive de partajare a lucrului

Directivele de partajare a lucrului (FOR, SECTIONS, SINGLE) distribuie execuţia între membrii echipei de threaduri. Aceste directive sunt plasate în *interiorul unei regiuni paralele* şi indică modul în care este divizat lucrul. Directivele nu lansează noi threaduri, iar la începutul unei asemenea construcţii nu se consideră în mod implicit plasarea unei bariere, secvenţa de construcţii de partajare a lucrului şi directivele de tip barieră trebuie să fie similare pentru grupul de threaduri.

Directiva FOR. Deoarece ciclurile sunt cea mai comună sursă de paralelism în majoritatea programelor, ele permit *divizarea iteraţiilor ciclului între thread-uri.* Ciclurile identifică o construcţie de partajare a sarcinilor iterativă ce permite specificarea execuţiei în paralel a iteraţiilor asociate ciclului şi distribuite threadurilor deja existente**.**

Sintaxa este:
#pragma omp for *[clauză [,] clauză]. . .] linie _nouă*
for-loop

iar clauzele pot fi:
private(listă _var**)**
firstprivate(listă _var**)**
lastprivate(listă _var**)**
reduction(*operato***r:** listă _var**)**
ordered
schedule(*tip***[,** *chunk_size***])**
nowait

Semnificaţia clauzelor în contextul acestei directive este următoarea: *firstprivate* specifică copii locale ale variabilei private threadului, iar *lastprivate* permite specificarea actualizării variabilei la ultima iteraţie a ciclului for. Directiva plasează restricţii asupra structurii buclei for corespondente, astfel ,aceasta trebuie să respecte o formă canonică: *for (var = a; var op_logic b; incr-exp)*

unde op_logic poate fi: <, <=, >, >=
şi incr-expr este var = var +/- incr sau echivalenţi semantici precum var++, ce nu poate fi modificată în corpul ciclului. Forma canonică permite ca numărul de iteraţii să fie determinat la intrarea ciclului.

Deoarece construcţia este larg utilizată, există o formă scurtă care combină regiunea paralelă cu directivele Do/For astfel rezultă construcţia combinată:

#pragma omp parallel for ***[clauze]***
for loop

Fără clauze adiţionale, directiva Do/For va *partiţiona iteraţiile în mod echilibrat* între threaduri, însă implementarea este dependentă de sistem. Un test ce permite verificarea dacă un ciclu este paralel este următorul: se verifică dacă ciclul oferă aceleaşi rezultate dacă este rulat în ordine inversă (atunci este mai mult ca sigur paralel, salturile în afara ciclului nefiind permise).
Variabila de index a ciclului paralel este PRIVATĂ prin definiţie, iar directiva PARALLEL DO/FOR acceptă toate clauzele directivei PARALLEL.

Clauza SCHEDULE oferă o varietate de opţiuni pentru specificarea *divizării iteraţiilor ciclurilor* care vor fi executate de fiecare thread. Sintaxa acesteia este**:**

schedule (*tip[, size]***)**

unde *tip* poate fi: STATIC, DYNAMIC, GUIDED sau RUNTIME, iar *chunksize* este o expresie întreagă pozitivă, ce reflectă modul în care a fost divizat spaţiul de iteraţii.

Cerinţe pentru planificare statică

- dacă *size* nu este specificat, spaţiul de iteraţie este divizat în părţi egale, şi o parte este asignată fiecărui thread (planificare pe block)
- dacă *size* este specificat, spaţiul de iteraţie este divizat în părţi, fiecare parte de un număr egal cu *chunksize* iteraţii, iar părţile sunt asignate în mod ciclic fiecărui thread (planificare tip block-cyclic)

Cerinţe pentru planificare dinamică

- planificarea dinamică divide spaţiul de iteraţie în părţi de dimensiunea *size* şi le asignează thread-urilor după regula primul-venit-primul-servit, astfel dacă un thread termină o parte, îi este asignată următoarea parte din listă.
- dacă *size* nu este specificat, se consideră având valoarea 1.

Cerinţe pentru planificare ghidată (GUIDED)

- este similară celei dinamice, dar secvenţele planificate sunt mari la început şi apoi se micşorează exponenţial.
- dimensiunea următoarei părţi reprezintă (generic) numărul de iteraţii rămase, divizat la numărul thread-urilor.
- câmpul *size* specifică dimensiunea minimă a unei părţi, dacă *size* nu este specificat, se consideră ca fiind 1.

Planificarea *RUNTIME* permite alegerea planificării în momentul rulării, când este determinată de valoarea variabilei de mediu OMP_SCHEDULE.

Observaţii utile în alegerea unei anumite planificări sunt următoarele:

- se utilizează planificarea STATIC pentru cicluri *echilibrate* ca încărcare, utilă pentru cicluri cu încărcare uşoară, dar care poate induce partajare falsă.
- este potrivită planificarea DYNAMIC dacă iteraţiile au *încărcare variabilă* la scară largă, dar distruge localitatea datelor, iar planificarea GUIDED, mai puţin costisitoare decât DYNAMIC oferă posibilitatea unei planificări dirijate.
- în mod uzual se utilizează modul RUNTIME pentru experimentări diverse.

Standardul OpenMP propune şi soluţii de paralelism dinamic implicit**,** astfel este posibil să lăsăm sistemul să decidă câte thread-uri execută fiecare regiune paralelă, pentru ca acesta să realizeze *optimizările* necesare alocării resurselor. Numărul thread-urilor va fi egal sau mai mic decât cel setat de utilizator şi va rămâne fix pe durata fiecărei regiuni paralele.

Paralelismul dinamic se poate seta cu rutina OMP_SET_DYNAMIC, sau cu variabila de mediu OMP_DYNAMIC. Valoarea implicită este dependentă de implementare. Dacă codul depinde de utilizarea unui anumit număr de thread-uri, atunci este necesară dezactivarea paralelismului dinamic.

Alături de funcţiile ce permit identificarea threadurilor create şi a numărului lor sunt utile funcţiile ce permit *alterarea dinamică* a numărului de threaduri *int omp_get_dynamic()*, respectiv activarea paralelismului imbricat *int omp_get_nested().*

Directiva SECTIONS. Directiva permite partiţionarea spaţiului iteraţiilor între threaduri, în acest mod blocuri separate, independente de cod pot fi executate în paralel (ex. diferite subrutine independente). Prin intermediul acestei dircetive, OpenMP oferă asignarea noniterativă a taskurilor paralele, iar codul sursă este cel ce determină cantitatea de paralelism pusă la dispoziţie. Aceste directive sunt relativ rar utilizate, cu excepţia implementării paralelismului imbricat.

```
#pragma omp sections [clauze]
{
        [ #pragma omp section ]
                bloc structurat
        [ #pragma omp section
                bloc_ structurat
        . . . ]}
```

Directiva *Sections* acceptă clauzele PRIVATE, FIRSTPRIVATE şi LASTPRIVATE, REDUCTION, NOWAIT fiecare secţiune trebuie însă să conţină un bloc structurat. La sfârşitul directivei o barieră implicită realizează sincronizarea ,în absenţa unei clauze NOWAIT. Forma scurtă a acestei directive este:

```
#pragma omp parallel sections [clauze]
        {
        . . . }
```

3.2.4.Directive de sincronizare şi consistenţă a memoriei

Directiva SINGLE. Este necesar ca pentru anumite situaţii o anumită parte de procesare din regiunea paralelă să fie executată de *un singur thread*, fiind utilă pentru calcul de date globale, respectiv operaţii de intrare-ieşire. Primul thread care ajunge la directiva SINGLE va executa blocul, iar celelalte thread-uri vor aştepta până când blocul a fost executat. Directiva acceptă clauzele directivelor PRIVATE şi FIRSTPRIVATE, în mod implicit la sfârşitul directivei se găseşte o barieră. Această directivă este utilă pentru calcule de date globale sau operaţii I-O.

Sintaxa:

```
#pragma omp single [clauze]
bloc structurat
```

iar clauza este una din următoarele:

private(*lista_var*)
firstprivate(*lista_var*)
copyprivate(*lista_var*)
nowait

Directiva MASTER. Directiva Master este o specializare a directivei SINGLE, ea indică faptul că un bloc de cod trebuie să fie executat numai de thread-ul master (thread 0), iar celelalte thread-uri evită acest bloc şi îşi continuă execuţia. Sintaxa:

```
#pragma omp master
bloc structurat
```

Directiva ORDERED. Directiva permite specificarea codului unui ciclu ce trebuie executat *în ordinea în care ar fi fost executat secvenţial*. Deoarece directiva referă o execuţie de tip *in-order* a unui

ciclu FOR, aceasta poate apărea numai în interiorul unei directive ParallelFor care are clauza ORDERED specificată. Directiva introduce un punct de serializare în program, astfel un singur thread poate accesa secvenţa de cod astfel definită, doar atunci când threadurile anterioare au ieşit din ciclu. Sintaxa:

#pragma omp ordered
bloc structurat

Directiva BARRIER. Implementarea unei bariere presupune că nici un thread nu poate trece de barieră până când aceasta nu a fost atinsă şi de către celelalte threaduri. Există o barieră implicită la sfârşitul directivelor DO/FOR, SECTIONS şi SINGLE, astfel fie toate thread-urile, fie nici unul trebuie să atingă bariera, în caz contrar apare o blocare (blocare circulară - DEADLOCK). Sintaxa:

#pragma omp barrier

Clauza NOWAIT. Poate fi folosită pentru a elimina barierele implicite de la sfârşitul directivelor DO/FOR, SECTIONS şi SINGLE (există anumite situaţii în care barierele sunt costisitoare). Directiva indică faptul că threadurile pot trece la instrucţiunea următoare fără a aştepta ca celelalte threaduri să termine execuţia ciclului for. Sintaxa:

#pragma omp for nowait
for loop

Observaţie. Este foarte uşor a se omite utilizarea unei bariere necesare, fiind posibil astfel să se genereze un comportament nedeterminist. Prin folosirea directivei NOWAIT şi explicitarea tuturor barierelor se poate utiliza un stil de programare care să evite un astfel de comportament al aplicației.

Directiva CRITICAL. O secţiune critică este un bloc de cod care poate fi executat de un singur thread la un moment dat şi este folosită pentru a proteja variabilele partajate. Alături de funcţiile de bibliotecă specifice managementului lacătelor, directiva CRITICAL permite denumirea secţiunilor critice, astfel dacă un thread este într-o secţiune critică cu un nume dat, atunci nici un alt thread nu poate pătrunde într-o secţiune critică cu acelaşi nume. Dacă regiunea critică nu este numită secţiunea va primi un nume implicit, numele regiunilor critice este global. Regiunile critice reprezintă zone de serializare în cod ce trebuie reduse pentru creşterea performanţei programului. Sintaxa**:**

#pragma omp critical *[***(** *nume* **)***]*
bloc structurat

Directiva ATOMIC. Directiva este folosită pentru a proteja o singură actualizare a unei variabile partajate, aplicată unui singur bloc. Utilizarea acesteia poate fi mai eficientă decât folosirea directivei CRITICAL, de ex. dacă diferite elemente ale unui şir pot necesita protejare distinctă.

#pragma omp atomic
bloc structurat

Este de preferat utilizarea directivei ATOMIC, deoarece permite cea mai bună optimizare, dacă nu este posibil, se poate folosi directiva CRITICAL, având grijă ca numele date să fie pe cât posibil diferite.

Directiva FLUSH. Oferă un mecanism ce permite implementarea consistenţei memoriei între threaduri diferite, astfel în acest mod întregul set de threaduri posedă o viziune consistentă a anumitor obiecte din memorie necesare aplicației. Dacă obiectele ce necesită sincronizarea sunt desemnate ca variabile, acestea pot fi specificate în lista opţională. Directiva fără o listă de variabile va sincroniza toate obiectele.

#pragma omp flush *[***(***list***)***]*

unde *list* specifică o listă de variabile care trebuiesc golite. Dacă nu este specificată nici o listă, atunci toate variabilele partajate vor fi golite.

O directivă FLUSH este utilizată de o directivă BARRIER la intrarea şi ieşirea dintr-o secţiune CRITICAL sau ORDERED, PARALLEL, PARALLEL FOR şi la sfârşitul directivelor PARALLEL, DO/FOR, SECTIONS şi SINGLE (mai puţin atunci când este prezentă clauza NOWAIT). Directiva nu este implicită pentru intrarea clauzelor: FOR, MASTER, SECTIONS, SINGLE.

3.2.5. Date în aplicații OpenMP

Performanţa programelor este în mod direct influenţată de manipularea datelor de către threaduri. Anumite directive acceptă clauze ce permit utilizatorului controlul atributelor de partajare a variabilelor pentru o regiune, aplicate doar pentru extensia lexicală specifică directivei. Lista clauzelor valide pentru o anumită directivă sunt specifice directivei. Există un set de directive ce permit controlul mediului de date pe durata execuţiei regiunii paralele, cele mai importante fiind THREADPRIVATE ce defineşte domeniul de adresare ca fiind fişier, spaţiu de nume respectiv static şi un set de clauze ce permit controlul atributelor de partajare a variabilelor. Dacă o anumită variabilă este vizibilă la apariţia unei construcţii de partajare a lucrului şi nu este specificată în lista THREADPRIVATE sau într-o clauză de atribute, atunci variabila este partajată.

Directiva THREADPRIVATE. Pentru anumite aplicaţii poate fi convenabil ca fiecare thread să posede *propria copie* a variabilelor cu scop global (variabile file-scope şi namespace-scope în C/C++). În afara regiunilor paralele şi în directivele MASTER, accesele la aceste variabile, referă copia thread-ului master.

Există situaţii în care este necesar ca anumite variabile să fie accesibile prin menţinerea lor persistentă între regiuni paralele, *fără a fi copiate în spaţiul de date al threadului master*. Această clasă de variabile persistente este realizată folosind directiva THREADPRIVATE, ce implică faptul că toate variabilele listei sunt locale fiecărui thread şi sunt iniţializate înainte de a fi accesate într-o regiune paralelă, ele fiind persistente. Utilizarea directivei presupune o serie de restricţii între care, directiva trebuie să aibă scop la nivel de fişier sau de spaţiu de nume, după toate declaraţiile de variabile din *list* şi înainte de orice referinţe la variabilele din *list*.

Sintaxa este:**#pragma omp threadprivate (***list***)**

Clauza PRIVATE. Clauza declară variabilele din listă private pentru fiecare thread, alocând în acest sens un nou obiect persistent ale cărui caracteristici sunt determinate de tipul variabilei. Sintaxa: **private(list)**

Clauza FIRSTPRIVATE. Clauza oferă un superset aferent funcţionalităţii oferite de clauza PRIVATE. Această directivă este utilizată atunci când este necesară valoarea unei variabile private la intrarea într-un ciclu Sintaxa: **firstprivate(list)**

Clauza LASTPRIVATE. Această directivă este utilizată atunci când este necesară valoarea unei variabile private la ieşirea dintr-un ciclu. Sintaxa:**lastprivate(list)**

Clauza SHARED. Clauza defineşte variabile ce apar în listă ca fiind partajate de grupul de threaduri ce pot accesa simultan aceeaşi zonă de stocare a acestora. Sintaxa: **shared(list)**

Clauza REDUCTION. O reducţie produce o singură valoare din operaţii asociative precum: adunarea, înmulţirea, max, min, ŞI (logic), SAU (logic). Operaţia este echivalentă următorului set de etape: fiecare thread reduce într-o copie privată, iar apoi toate aceste copii se reduc pentru a furniza rezultatul final. Sintaxa **reduction(*op*:*list*)**

Clauza COPYIN. Această clauză permite ca valorile datelor private ale thread-ului master să fie copiate la toate celelalte thread-uri la începutul regiunii paralele. Sintaxa: **copyin(***list***)**

Clauza COPYPRIVATE. Clauza oferă un mecanism de utilizare a unei variabile private pentru a transmite în mod broadcast o valoare de la un thread către celelalte threaduri din grup. Clauza poate apărea doar corelat cu directiva SINGLE. Efectul clauzei asupra listei de variabile apare după execuţia blocului structurat asociat construcţiei SINGLE dar înainte ca oricare din threadurile din grup să părăsească bariera. Astfel, fiecare variabilă din listă devine definită (asemeni asignării) cu valoarea aferentă din threadul ce a executat blocul structurat. *Clauza COPYPRIVATE* trimite valoarea unei variabile private tuturor thread-urilor, la sfârşitul unei directive **SINGLE,** fiind astfel cea mai folositoare pentru citirea în variabile private. Sintaxa: **#pragma omp single copyprivate(***list***)**

3.2.6. Legarea şi imbricarea directivelor

Directivele PARALLEL permit crearea concurentă de threaduri, iar directivele FOR şi SECTIONS permit distribuirea taskurilor threadurilor create. În cazul în care nu există directiva PARALLEL, secvenţa de cod aferent directivelor SECTIONS şi FOR se va executa serial.

Pentru a evita ambiguitatea referitoare la directiva de regiune paralelă la care se face referire, este nevoie de un set de reguli pentru legarea directivelor, cele mai importante sunt:

- directivele DO/FOR, SECTIONS, SINGLE, MASTER şi BARRIER se leagă de cea mai apropiată directivă PARALLEL.
- directiva ORDERED de leagă de cea mai apropiată directivă DO.

Paralelism imbricat (NESTED). Standardul oferă construcţii pentru specificarea paralelismului imbricat, ce poate fi activat cu variabila de mediu OMP_NESTED sau cu rutina OMP_SET_NESTED.

Imbricarea directivelor OpenMP respectă următoarele reguli:

- dacă este întâlnită o directivă PARALLEL în interiorul unei alte directive PARALLEL va fi creată o nouă echipă de thread-uri, însă noua echipă va conţine numai un thread, până la activarea paralelismului imbricat.
- directivele FOR, SECTIONS şi SINGLE legate la aceeaşi directivă PARALLEL nu pot fi imbricate reciproc
- directivele CRITICAL cu acelaşi nume nu pot fi imbricate reciproc.

Directive orfane. Directivele sunt active în scopul *dinamic* (de execuţie) a regiunii paralele, nu doar în scopul *lexical (sintactic)* , proprietate deosebit de utilă deoarece permite un stil modular de programare, însă care poate fi foarte confuz, dacă arborele de apeluri este complicat.

Din aceste considerente există un set de reguli suplimentare referitor la domeniul de aplicabilitate numit *scop de date* (data scope attributes), astfel atunci când se apelează o subrutină din interiorul unei regiuni paralele:

- variabilele globale şi blocurile COMMON sunt partajate, mai puţin în cazul în care sunt declarate cu THREADPRIVATE
- variabilele locale statice în C/C++ sunt partajate
- variabilele din lista de argumente moştenesc atributele cu scop de dată din rutina apelantă.
- toate celelalte variabile locale sunt private.

3.2.7.Funcţii de bibliotecă pentru sincronizare

Deoarece sunt situaţii în care este necesară mai multă flexibilitate decât cea oferită de directivele CRITICAL şi ATOMIC, pot fi utilizate lacătele.

Un lacăt (lock) este o variabilă specială care poate fi setată de un thread, astfel nici un alt thread nu mai poate seta lacătul, până în momentul în care acesta este resetat. Setarea unui lacăt poate fi blocantă sau non-blocantă, un lacăt trebuie iniţializat înainte de a fi utilizat, şi poate fi distrus când nu mai este necesar. Este posibilă utilizarea de *lacăte imbricate* ce pot fi utilizate multiplu de către acelaşi

thread. Aceste funcţii permit aceluiaşi thread să seteze un lacăt de mai multe ori, înainte de a-l reseta de acelaşi număr de ori, rutinele corespunzătoare vor citi sau actualiza valoarea cea mai recentă a variabilei, nefiind astfel necesară utilizarea explicită a directivei flush pentru a asigura consistenţa variabilei lacăt între diferite threaduri.

```
void omp_init_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_init_nest_lock (omp_nest_lock_t *lock);
void omp_destroy_nest_lock (omp_nest_lock_t *lock);
void omp_set_nest_lock (omp_nest_lock_t *lock);
void omp_unset_nest_lock (omp_nest_lock_t *lock);
void omp_test_nest_lock (omp_nest_lock_t *lock);
```

Pentru *analiza de performanță* și instrumentarea codului aplicației cu scopul de a identifica timpul necesar procesării este utilă funcția *omp_get_wtime()* ce returnează valoarea timpului raportat la ceasul sistem.

Deasemenea este util Analizorul de concurență pentru Visual Studio care poate fi descărcat ca și componentă distinctă de la adresa https://docs.microsoft.com/en-us/visualstudio/profiling/concurrency-visualizer?view=vs-2019 și care va fi instalat pentru integrare în Visual Studio- IDE, cu scopul de a vizualiza variația gradului de paralelism pentru aplicație și consumul de memorie la execuția codului.

3.3. Exemple

3.3.1.Implementarea unei aplicații pentru *calculul paralel a numărului Pi* prin integrare numerică folosind directive OpenMP (sursa [1])

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#define omp_get_num_threads() 1
#endif

#define NUM_THREADS 4
static long num_steps = 100000000;
double step;

int main() {
  double x;
  double pi;
  int thread_count;
  int i;

  //change in x (i.e. width of rectangle)
  step = 1.0/(double)num_steps;

  #ifdef _OPENMP
  omp_set_num_threads(NUM_THREADS);
  #endif
  #pragma omp parallel
  { int thread_id;
    double sum; //create a local sum to eliminate false sharing
    int t_count; //local copy of thread count
    double x;
    int i;
    thread_id = omp_get_thread_num();
    t_count = omp_get_num_threads();
```

```
    if (thread_id == 0) {
      thread_count = t_count;}
    //calculate the summation of F(x)
    // (i.e. sum of rectangles)
    //in the approximation of pi

    for (i=thread_id, sum = 0.0; i < num_steps; i = i+ t_count) {
      //calculate height
      x = (i+0.5)*step;
      sum = sum + 4/(1.0+x*x); //sum F(x)}
    sum = sum * step;
    #pragma omp atomic
      pi = pi + sum; //ensures calculation is atomic }
  printf("pi = %f", pi);}
```

3.3.2.Implementarea *produsului dintre o matrice şi un vector* folosind construcţiile de partajare a sarcinilor OpenMP și serializarea acceselor la variabila totală partajată ce cumulează rezultatele. (sursa [1])

```
#include "omp.h"
#define SIZE 10
main ()
{float A[SIZE][SIZE], b[SIZE], c[SIZE], total;
int i, j, tid;
total = 0.0;
for (i=0; i < SIZE; i++)
  {
  for (j=0; j < SIZE; j++)
    A[i][j] = (j+1) * 1.0;
    b[i] = 1.0 * (i+1);
    c[i] = 0.0; }
printf("\nStarting values of matrix A and vector b:\n");
for (i=0; i < SIZE; i++)
  {printf("  A[%d]= ",i);
  for (j=0; j < SIZE; j++)
    printf("%.1f ",A[i][j]);
    printf("  b[%d]= %.1f\n",i,b[i]); }
    printf("\nResults by thread/row:\n");

/* Create a team of threads and scope variables */
#pragma omp parallel shared(A,b,c,total) private(tid,i)
  {tid = omp_get_thread_num();

/* Loop work-sharing construct - distribute rows of matrix */
#pragma omp for private(j)
  for (i=0; i < SIZE; i++)
    {for (j=0; j < SIZE; j++)
      c[i] += (A[i][j] * b[i]);

    /* Update and display of running total must be serialized */
    #pragma omp critical
      {total = total + c[i];
      printf("  thread %d did row %d\t c[%d]=%.2f\t",tid,i,i,c[i]);
      printf("Running total= %.2f\n",total);}}
/* end of parallel i loop */}
/* end of parallel construct */

printf("\nMatrix-vector total - sum of all c[] = %.2f\n\n",total);}
```

3.3.3.Se vor testa exemplele ce ilustrează modul de utilizare a API-ului OpenMP ,exemple pe care le puteți descărca de la adresa *https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/03_OpenMP*

3.4. Întrebări teoretice

3.4.1. Ce este o barieră implicită și cum este ea implementată în OpenMP?

3.4.2. În modelul de date OpenMP, ce reprezintă FIRSTPRIVATE și LASTPRIVATE? Imaginați un exemplu simplu de utilizare.

3.4.2. Identificați construcțiile din modelul de programare OpenMP prin intermediul cărora se poate seta și verifica gradul de paralelism .

3.4.2. Explicați diferența în semnificația și utilizarea clauzelor SINGLE și MASTER.

3.4.3. Cum poate verificat și activat paralelismul dinamic al threadrulor, dar suportul pentru execuția unor clauze Parallel For imbricate?

3.5. Probleme propuse

3.5.1.Implementaţi un algoritm de înmulţire a două matrici pătratice folosind planificarea statică a threadurilor. Modificaţi soluţia propusă pentru înmulţirea matricilor astfel încât folosind variabila de mediu OMP_NUM_THREADS să controlaţi numărul de threaduri, analizând totodată performanţele soluţiei la modificarea acestuia. Consideraţi următoarele cazuri de analiză: paralelizarea buclei exterioare, paralelizarea ambelor bucle, paralelizarea tuturor celor trei bucle.

3.5.2.Descrieti în pseudocod o formulare paralelă a înmulțirii matrice/vector pentru care matricea este partiționată 1D pe blocuri de-a lungul coloanelor , iar vectorul este egal partiționat proceselor, pentru un sistem cu memorie partajată. Determinați timpul de execuție pentru cele două alternative de partiționare pe blocuri 1D pe linie, respectiv pe coloane.Propuneți o soluție de implementare bazată pe OpenMP, identificând cele mai potrivite funcții de bibliotecă pentru implementarea soluției.

3.5.3.Implementați o coadă multiacces pentru un set de threaduri ce inserează, respectiv extrag conținut, folosind construcții de sincronizare de tip mutex.Cuantificați timpul necesar pentru 1000 de operații de inserare și 1000 operații de extragere generate de 64 threaduri ce produc date , respectiv 64 threaduri ce consumă date din coadă.

3.5.4.Ilustrați utilizarea în manieră recursivă a lacătelor folosind un algoritm de căutare de tip arbore binar. Programul va primi ca intrare o listă mare de numere ce va fi divizată multiplelor threaduri. Fiecare thread va încerca inserarea propriilor elemente în arbore folosind un singur lacăt asociat arborelui. Demonstrați că un singur lacăt devine o gâtuire și pentru un număr moderat de threaduri.

3.6. Miniproiect

3.6.1. Să se implementeze și evalueze versiuni paralele pentru algoritmii de sortare *quick/merge/ radix* folosind modelul de programare paralelă bazat pe partajarea memoriei și biblioteca OpenMP v3.0 (se vor propune și se vor implementa algoritmi astfel încât să beneficieze de abstracțiunea task introdusă în OpenMP v3) .Pentru fiecare algoritm este necesară descrierea soluției în pseudocod a algoritmului secvențial și paralel, oferind detalii referitoare la complexitate și specificând funcțiile de bibliotecă OpenMP utilizate în implementare. Să se evalueze performanța algoritmilor pentru diferite dimensiuni ale problemei și număr de nuclee (core), calculând accelerația și eficiența obținute.Se vor genera mai multe execuții menținând unul din cei doi parametrii variabili și anume timp , respectiv dimensiunea problemei la aceeași valoare și modificându-l pe celălalt. Evaluarea se va realiza prin reprezentare grafică , importantă fiind alegerea sa corectă pentru a permite o bună interpretare a rezultatelor.

3.7. Referințe bibliografice

1. Site oficial - API și exemple :http://openmp.org/wp/2009/04/download-book-examples-and-discuss/
2. Tutorial OpenMP : https://computing.llnl.gov/tutorials/openMP/
3. Microsoft OpenMP https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=vs-2019
4. Configurari Visual Studio pentru suport OpenMP: http://msdn.microsoft.com/en-us/library/fw509c3b.aspx
5. Chandra R., Parallel programming in OpenMP https://apps2.mdp.ac.id/perpustakaan/ebook/Karya%20Umum/Parallel_Programming_in_OpenMP.pdf

OpenMP Reference Sheet for C/C++

Constructs

<parallelize a for loop by breaking apart iterations into chunks>
#pragma omp parallel for [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr), ordered, schedule(type[,chunkSize])]
<A,B,C such that total iterations known at start of loop>
for(A=C;A<B;A++) {
    <your code here>

    <force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>
    #pragma omp ordered {
        <your code here>
    }
}

<parallelized sections of code with each section operating in one thread>
#pragma omp parallel sections [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr)] {
    #pragma omp section {
        <your code here>
    }
    #pragma omp section {
        <your code here>
    }
    ....
}

<grand parallelization region with optional work-sharing constructs defining more specific splitting of work and variables amongst threads. You may use work-sharing constructs without a grand parallelization region, but it will have no effect (sometimes useful if you are making OpenMP'able functions but want to leave the creation of threads to the user of those functions)>
#pragma omp parallel [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(private|shared|none), reduction(op:vars), copyin(vars), if(expr)] {
    <the work-sharing constructs below can appear in any order, are optional, and can be used multiple times. Note that no new threads will be created by the constructs. They reuse the ones created by the above parallel construct.>

    <your code here (will be executed by all threads)>

    <parallelize a for loop by breaking apart iterations into chunks>
    #pragma omp for [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), ordered, schedule(type[,chunkSize]), nowait]
    <A,B,C such that total iterations known at start of loop>
    for(A=C;A<B;A++) {
        <your code here>

        <force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>
        #pragma omp ordered {
            <your code here>
        }
    }

    <parallelized sections of code with each section operating in one thread>
    #pragma omp sections [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), nowait] {
        #pragma omp section {
            <your code here>
        }
        #pragma omp section {
            <your code here>
        }
        ....
    }

    <only one thread will execute the following. NOT always by the master thread>
    #pragma omp single {
        <your code here (only executed once)>
    }
}

Directives

shared(vars) *<share the same variables between all the threads>*
private(vars) *<each thread gets a private copy of variables. Note that other than the master thread, which uses the original, these variables are not initialized to anything.>*
firstprivate(vars) *<like private, but the variables do get copies of their master thread values>*
lastprivate(vars) *<copy back the last iteration (in a for loop) or the last section (in a sections) variables to the master thread copy (so it will persist even after the parallelization ends)>*
default(private|shared|none) *<set the default behavior of variables in the parallelization construct. shared is the default setting, so only the private and none setting have effects. none forces the user to specify the behavior of variables. Note that even with shared, the iterator variable in for loops still is private by necessity >*
reduction(op:vars) *<vars are treated as private and the specified operation(op, which can be +,*,-,&,|,&,&&,||) is performed using the private copies in each thread. The master thread copy (which will persist) is updated with the final value.>*

copyin(vars) *<used to perform the copying of threadprivate vars to the other threads. Similar to firstprivate for private vars.>*
if(expr) *<parallelization will only occur if expr evaluates to true.>*
schedule(type [,chunkSize]) *<thread scheduling model>*

type	*chunkSize*
static	*number of iterations per thread pre-assigned at beginning of loop (typical default is number of processors)*
dynamic	*number of iterations to allocate to a thread when available (typical default is 1)*
guided	*highly dependent on specific implementation of OpenMP*

nowait *<remove the implicit barrier which forces all threads to finish before continuation in the construct>*

Synchronization/Locking Constructs *<May be used almost anywhere, but will only have effects within parallelization constructs.>*

<only the master thread will execute the following. Sometimes useful for special handling of variables which will persist after the parallelization.>
#pragma omp master {
<your code here (only executed once and by the master thread).
}

<mutex lock the region. name allows the creation of unique mutex locks.>
#pragma omp critical [(name)] {
<your code here (only one thread allowed in at a time)>
}

<force all threads to complete their operations before continuing>
#pragma omp barrier

<like critical, but only works for simple operations and structures contained in one line of code>
#pragma omp atomic
<simple code operation, ex. a += 3; Typical supported operations are ++,--,+,,-,/,&,^,<<,>>,| on primitive data types>*

<force a register flush of the variables so all threads see the same memory>
#pragma omp flush[(vars)]

<applies the private clause to the vars of any future parallelize constructs encountered (a convenience routine)>
#pragma omp threadprivate(vars)

Function Based Locking *< nest versions allow recursive locking>*

void **omp_init_[nest_]lock**(omp_lock_t*) *<make a generic mutex lock>*
void **omp_destroy_[nest_]lock**(omp_lock_t*) *<destroy a generic mutex lock>*
void **omp_set_[nest_]lock**(omp_lock_t*) *<block until mutex lock obtained>*
void **omp_unset_[nest_]lock**(omp_lock_t*) *<unlock the mutex lock>*
int **omp_test_[nest_]lock**(omp_lock_t*) *<is lock currently locked by somebody>*

Settings and Control

int **omp_get_num_threads**() *<returns the number of threads used for the parallel region in which the function was called>*
int **omp_get_thread_num**() *<get the unique thread number used to handle this iteration/section of a parallel construct. You may break up algorithms into parts based on this number.>*
int **omp_in_parallel**() *<are you in a parallel construct>*
int **omp_get_max_threads**() *<get number of threads OpenMP can make>*
int **omp_get_num_procs**() *<get number of processors on this system>*
int **omp_get_dynamic**() *<is dynamic scheduling allowed>*
int **omp_get_nested**() *<is nested parallelism allowed>*
double **omp_get_wtime**() *<returns time (in seconds) of the system clock>*
double **omp_get_wtick**() *<number of seconds between ticks on the system clock>*
void **omp_set_num_threads**(int) *<set number of threads OpenMP can make>*
void **omp_set_dynamic**(int) *<allow dynamic scheduling (note this does not make dynamic scheduling the default)>*
void **omp_set_nested**(int) *<allow nested parallelism; Parallel constructs within other parallel constructs can make new threads (note this tends to be unimplemented in many OpenMP implementations)>*

<env vars- implementation dependent, but here are some common ones>
OMP_NUM_THREADS "number" *<maximum number of threads to use>*
OMP_SCHEDULE "type,chunkSize" *<default #pragma omp schedule settings>*

Legend

vars is a comma separated list of variables
[optional parameters and directives]
<descriptions, comments, suggestions>
.... above directive can be used multiple times
For mistakes, suggestions, and comments please email e_berta@plutospin.com