Curs 8

Programare Paralela si Distribuita

OpenMP

- OpenMP
  - http://www.openmp.org

  OpenMP 5.0
  - https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-111802-web.pdf

- Tutoriale:
  - TutorialOpenMP.pdf pentru OpenMP 3.0 in

  “Class Materials” - Cursuri

Motivatie

- managementul explicit al threadurilor este de multe ori dificil si poate fi automatizat
  - Consideram exemplul: ‘adunare vectori’
    - Din codul secvential putem deduce ce trebuie executat in parallel-ciclul for
    - conversia spre codul paralel se poate face mecanic
      - trebuie doar sa specificam ca respectivul ciclu se poate face in parallel
      - lasam compilatorul sa realizeze transformarea
    - OpenMP executa aceste automatizari!!!

Ce este OpenMP?

- Acronimul - OpenMP ?
  - Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia.
- OpenMP => Application Program Interface (API) pentru ***multi-threaded, shared memory parallelism.***
    - Componente:
      - Compiler Directives,
      - Runtime Library Routines,
      - Environment Variables
- OpenMP
  - directive-based method to invoke parallel computations on share-memory multiprocessors

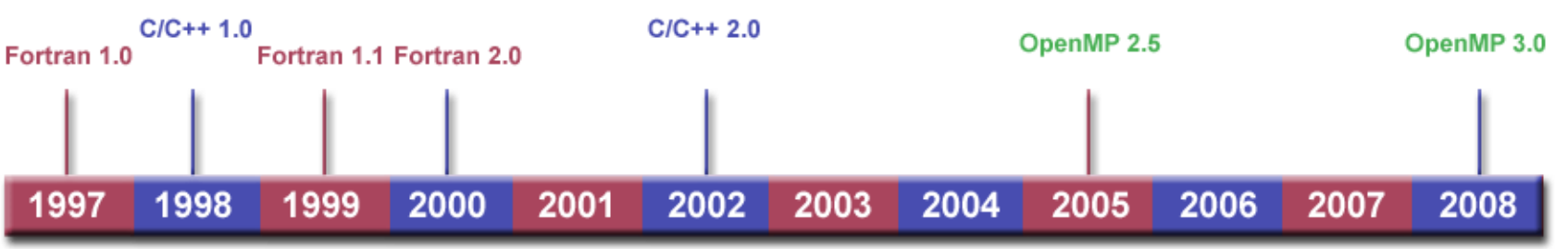
History of OpenMP

- The OpenMP standard specification started in the spring of 1997, taking over where ANSI X3H5 had left off, as newer shared memory machine architectures started to become prevalent.
- Led by the **OpenMP Architecture Review Board** (ARB). Original ARB members included: *(Disclaimer: all partner names derived from the OpenMP web site)*
  - Compaq / Digital
  - Hewlett-Packard Company
  - Intel Corporation
  - International Business Machines (IBM)
  - Kuck & Associates, Inc. (KAI)
  - Silicon Graphics, Inc.
  - Sun Microsystems, Inc.
  - U.S. Department of Energy ASCI program

Other Contributors

- Endorsing application developers
  - ADINA R&D, Inc.
  - ANSYS, Inc.
  - Dash Associates
  - Fluent, Inc.
  - ILOG CPLEX Division
  - Livermore Software Technology Corporation (LSTC)
  - MECALOG SARL
  - Oxford Molecular Group PLC
  - The Numerical Algorithms Group Ltd.(NAG)
- Endorsing software vendors
  - Absoft Corporation
  - Edinburgh Portable Compilers
  - GENIAS Software GmBH
  - Myrias Computer Technologies, Inc.
  - The Portland Group, Inc. (PGI)

OpenMP Release History



Scop: OpenMP

- Standardizare
  - furnizeaza un standard pentru arhitecturi/platforme de tip “shared memory”
- Productivitate
  - stabileste un set limitat de directive simple
  - se poate ajunge la un nivel semnificativ de paralelizare doar folosind 3 or 4 directives.
- Simplu de folosit
  - paralelizare incrementala
  - permite -> coarse-grain and fine-grain parallelism
- Portabilitate
  - Suporta Fortran (77, 90, and 95), C, si C++
- Public forum pentru API si participare

OpenMP?

- OpenMP API are specificatii pentru C/C++ si Fortran.
- OpenMP nu modifica codul secvential initial
  - Fortran -> comments
  - C/C++ -> pragmas
- OpenMP website: http://www.openmp.org
  - diverse tutoriale
    - slide-urile folosesc imagini din acestea

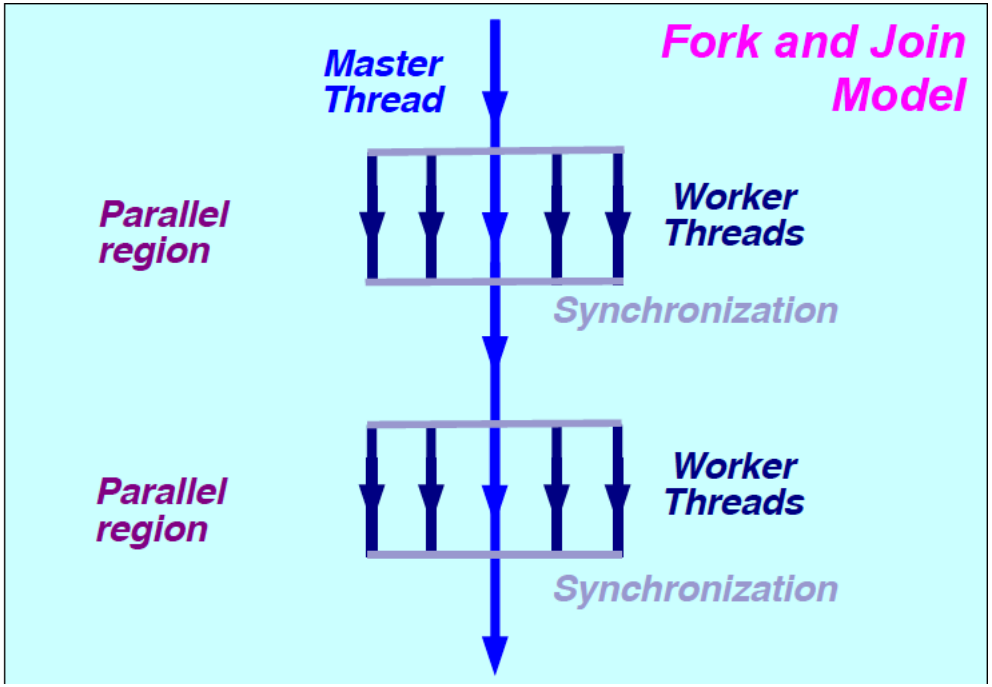
Compilare si executie

- gcc 4.2 and above suporta OpenMP 3.0
  - **gcc –fopenmp a.c**

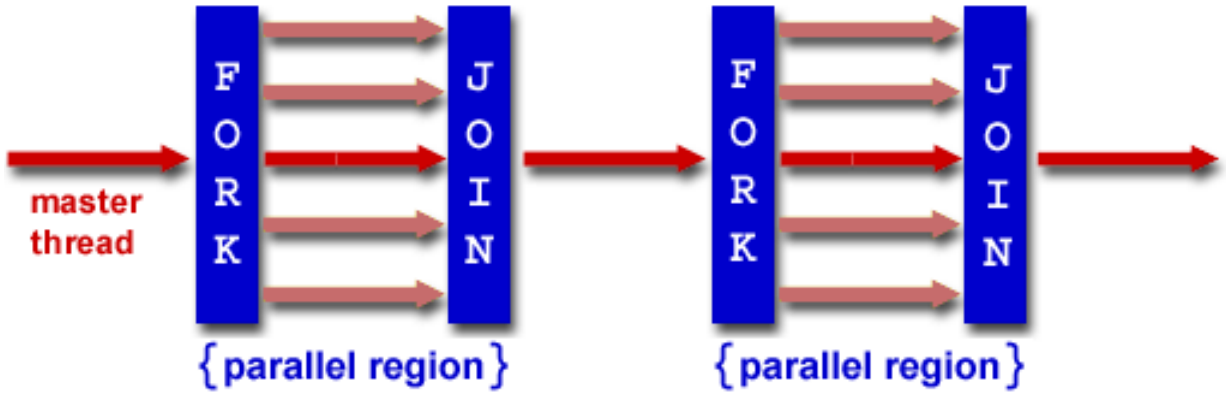
- executie – ca si la programele secventiale
  - ‘a.out’

- Compiler documentation
  - IBM: www-4.ibm.com/software/ad/fortran
  - Cray: http://docs.cray.com/ (Cray Fortran Reference Manual)
  - Intel: www.intel.com/software/products/compilers/
  - PGI: www.pgroup.com
  - PathScale: www.pathscale.com
  - GNU: gnu.org

OpenMP Model



OpenMP execution model



- OpenMP foloseste modelul ‘fork-join’ de executie paralela
  - Toate programele OpenMP incep cu un singur thread - master thread.
  - master thread se executa secvential pana cand intalneste o regiune paralela - parallel region, cand creeaza un set de threaduri team of parallel threads (FORK).
  - Atunci cand regiunea paralela se termina threadurile se sincronizeaza si se ‘termina’ (JOIN).
    - implementarile pot folosi un thread pool care gestioneaza setul de threaduri care se folosesc in regiunea paralela – este posibil sa mentina aceste threaduri

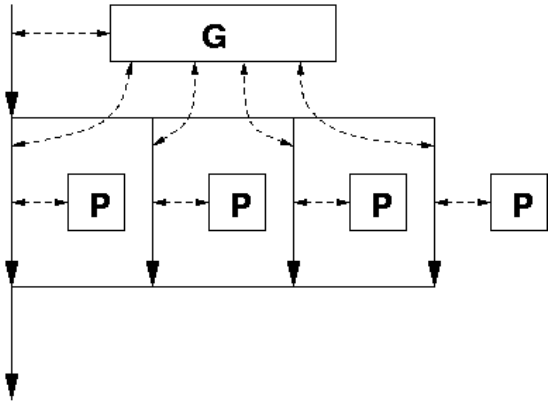
OpenMP- structura generala cod

```
#include <omp.h>
main () {
  int var1, var2, var3;
  Serial code
  . . .
 /* Beginning of parallel section. Fork a team of threads. Specify variable scoping*/
  #pragma omp parallel private(var1, var2) shared(var3)
 {
    /* Parallel section executed by all threads */

      ...

   /* All threads join master thread and disband*/
   }
   Resume serial code
   . . .
}
```

Data model



P = private data space
G = global data space

Variabile: private + shared

• global data space – accesibile de catre toate threadurile (shared variables).

• private space – accesibile doar la nivel de thread (private variables)

•exista variatiuni care depend de modul de initializare

Memory Model

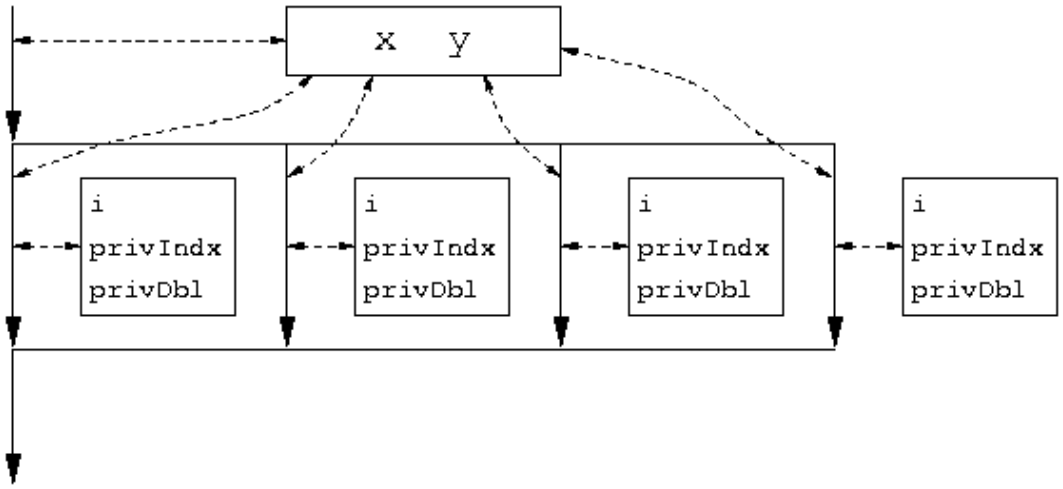
- OpenMP furnizeaza
  - "relaxed-consistency" +
  - "temporary" view of thread memory
- adica thread-urile pot face "cache" pe date si nu se mentine o consistenta exacta cu memoria globala tot timpul.

=>

- Atunci cand este critic ca toate threadurile sa vada identic o variabila partajata este responsabilitatea programatorului sa asigure aceasta prin comanda FLUSH ...

```
int y[MAX];
#pragma omp parallel for private( privIndx, privDbl )
  for ( i = 0; i < arraySize; i++ ) {
     for ( privIndx = 0; privIndx < 16; privIndx++ ) {
                    privDbl = ( (double) privIndx ) / 16;
                    y[i] = sin( exp( cos( - exp( sin(x[i]) ) ) ) ) + cos( privDbl );
     }
  }
```

Parallel for loop index is Private by default.



execution context for "arrayUpdate_II"

OpenMP Programming Model

- Directivele OpenMP in C and C++ se bazeaza pe directivele de compilare `#pragma`
- Format:

```
#pragma omp directive [clause list]
```

- programele OpenMP se executa secvential pana la intalnirea unei regiuni paralele (`parallel` directive)

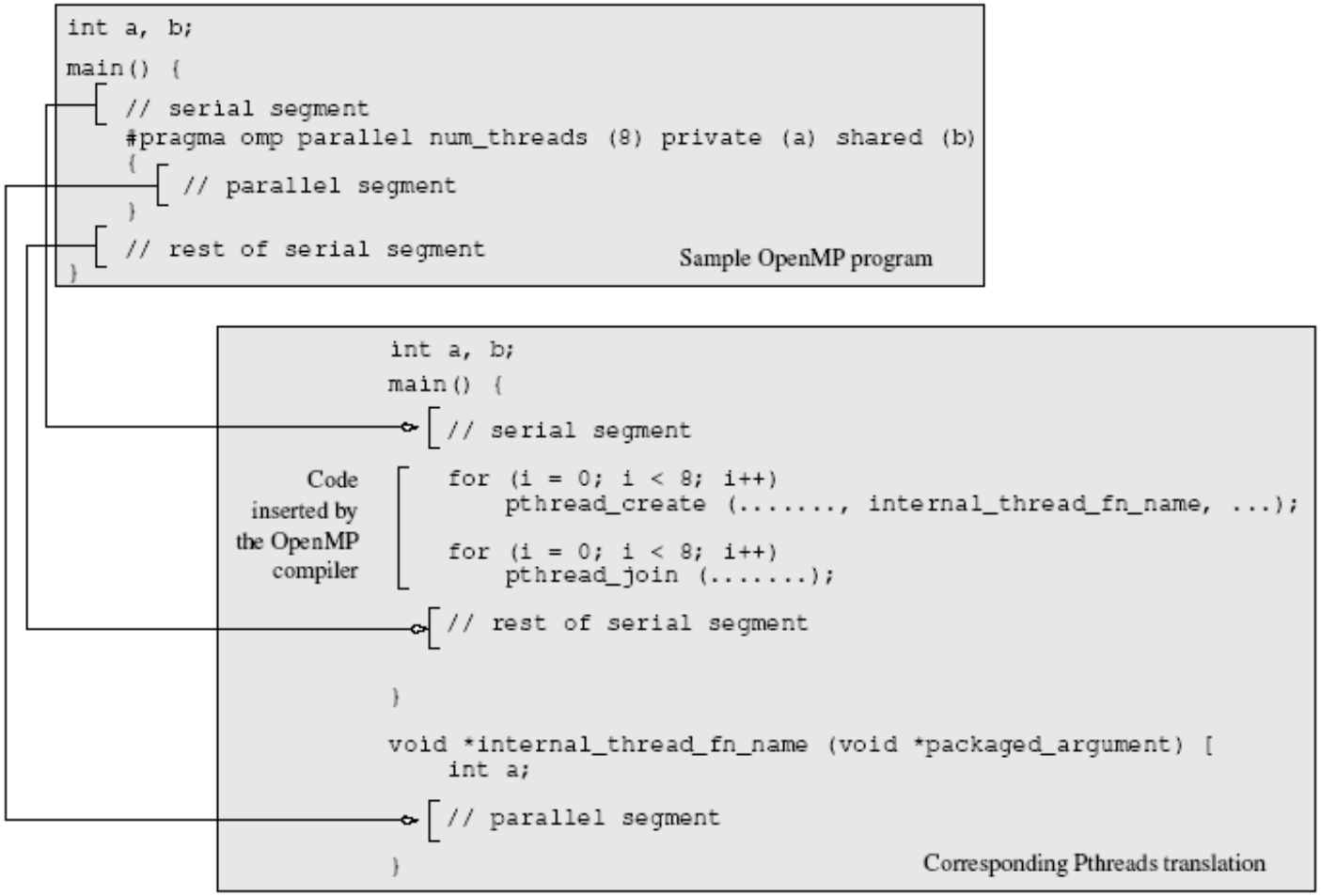
```
#pragma omp parallel [clause list]
/* structured block */
```

- Threadul care intalneste/executa directive paralela devine master pentru grupul de threaduri care o executa si are ID 0

Clauze

- [clause list] specifica
  - *conditional parallelization*
  - *number of threads*
  - *data handling*
- **Conditional Parallelization:**
  - clause `if (scalar expression)`
- **Degree of Concurrency:**
  - clause `num_threads(integer expression)`
- **Data Handling:**
  - clause `private (variable list`
  - clause `firstprivate (variable list)`
    - private si initializate si valoare corespunzatoare variabilei (cf nume) inainte de regiunea paralela
  - clause `shared (variable list)`

Exemplu de translatare a unui cod OpenMP.



OpenMP Code Structure – C/C++

```
#include <omp.h>
main () {
int var1, var2, var3;
Serial code
 .
 .
 .
Beginning of parallel section. Fork a team of threads. Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
     {
     Parallel section executed by all threads
      .
      .
      .
      All threads join master thread and disband
      }
Resume serial code
 .
 .
 .
}
```

PARALLEL Region – Number of Threads

- Numarul de threaduri care se folosesc intr-o regiune paralela depinde de urmatorii factori in urmatoarea ordine de precedenta :
  1. **IF** clause
  2. **NUM_THREADS** clause
  3. apel **omp_set_num_threads()** library function

     se apeleaza in sectiune de cod seriala inainte de o regiune parallel (void omp_set_num_threads(int num_threads))
  4. setare **OMP_NUM_THREADS** environment variable
  5. implementare implicita (eventual) – nr de core-uri.
- Threads sunt numerotate de la 0 (master thread) to N-1

OpenMP runtime environment

- omp_get_num_threads
- omp_get_thread_num
- omp_in_parallel
- Routines related to locks
- ……

OMP_GET_NUM_THREADS() and OMP_GET_THREAD_NUM()

- OMP_GET_NUM_THREADS()
  - Returneaza numarul de thread-uri care se executa intr-o regiune paralela curenta

C/C++	#include <omp.h> int omp_get_num_threads(void)

- OMP_GET_THREAD_NUM()
  - Returneaza numarul threadului care apeleaza functia si care actioneaza intr-o regiune paralela
  - returneza o valoare intre OMP_GET_NUM_THREADS-1.
  - master thread -> thread 0.

OMP_GET_MAX_THREADS()

- Returneaza valoare maxima care poate fi returnata de un apel al functiei OMP_GET_NUM_THREADS
- in general reflecta numarul de threaduri setat prin variabila de mediu OMP_NUM_THREADS environment variable sau de functia OMP_SET_NUM_THREADS()
- Poate fi apelata atat din regiune paralala cat si din sectiune secventiala

OpenMP environment variables

- OMP_NUM_THREADS
- OMP_SCHEDULE

OMP_GET_THREAD_LIMIT() and OMP_GET_NUM_PROCS()

- OMP_GET_THREAD_LIMIT()
  - New with OpenMP 3.0
  - Returns the maximum number of OpenMP threads available to a program
- OMP_GET_NUM_PROCS()
  - Returns the number of processors that are available to the program

Exemplificare

```
int a=1, b=2, c=3, d;
#pragma omp parallel if (is_parallel== 1) num_threads(8)
  \
  private (a) shared (b) firstprivate(c) lastprivate(d)

{
 d = omp_get_thread_num();
  /* structured block */
}
```

d=?

PRIVATE and SHARED Clauses

- PRIVATE Clause
  - un nou obiect in fiecare thread
  - toate referintele la obiectul original sunt redirectionate
  - trebuie sa se asume ca nu sunt initializate
- SHARED Clause
  - o variabila *shared* exista doar intr-o locatie de memorie
  - este responsabilitatea programatorului sa asigure accesul correct (eventual via CRITICAL sections daca este necesar)

FIRSTPRIVATE / LASTPRIVATE Clauses

- FIRSTPRIVATE Clause
  - Combina PRIVATE clause cu automatic initialization

- LASTPRIVATE Clause
  - Combina PRIVATE clause cu o copierea valorii din threadul care face ultima iteratie sau sectiune in variabila initiala din codul secvential anterior

PRIVATE Variables

```
main()
{
    int A = 10;
    int B, C;
    int n = 20;
    #pragma omp parallel
    {
        #pragma omp for private(i) firstprivate(A) lastprivate(B)
        for (int i=0; i<n; i++)
        {
            ....
            B = A + i;   /* A undefined unless declared firstprivate */
            ....
        }
        C = B;     /* B undefined unless declared lastprivate */
    }  /* end of parallel region */
}
```

Restrictii pentru PARALLEL Region

- parallel region bloc – trebuie sa fie un bloc structurat
  - nu se poate intra sau iesi fortat (not branch into or out of a parallel region) – no goto
- doar o clauza IF
- doar o clauza NUM_THREADS

Exemplu - PARALLEL Region

```
#include <omp.h>
main () {
int nthreads, tid;
/* Fork a team of threads with each thread having a private tid variable */
omp_set_num_threads(4);
#pragma omp parallel private(tid)
    {
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
       nthreads = omp_get_num_threads();
       printf("Number of threads = %d\n", nthreads);
       }
    } /* All threads join master thread and terminate */
}
```

Reduction Clause in OpenMP

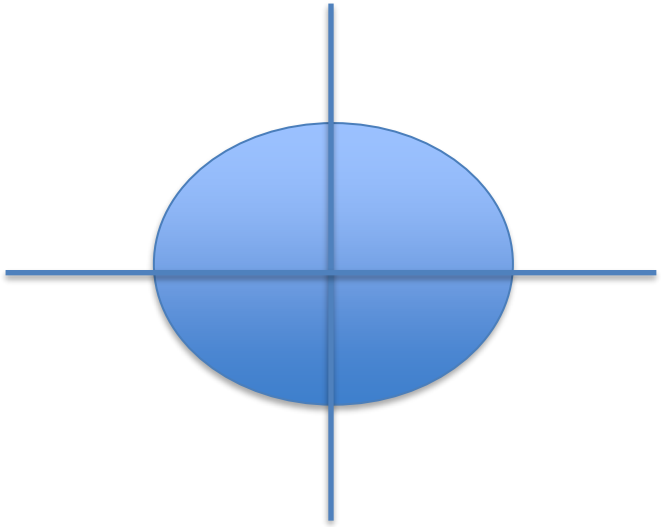
- `reduction` clause specifica cum se pot combina valorile locale fiecarui thread (ale unor variabile private) intr-o singura valoare in threadul master atunci cand regiunea paralela se termina

`reduction (operator: variable list)`

- `operator: +, *, -, &, |, ^, &&,` and `||`
- OpenMP generareaza cod astfel incat sa nu apara *race condition*

```
sum = 0.0;
#pragma parallel default(none) shared (n, x) private (I) reduction(+ : sum)
{
for(I=i_start; I<i_final; I++) sum = sum + x(I);
}
```

Approximare Pi – secvential



```
double PI(long npoints) {
double sum = 0;
int num_threads = omp_get_num_threads();
sum = 0;
    double rand_no_x, rand_no_y;
    unsigned long seed = time(NULL);
    srand(seed);
    for (long i = 0; i < npoints; i++) {
        rand_no_y = (double)(rand()) / (double)(RAND_MAX);
        rand_no_x = (double)(rand()) / (double)(RAND_MAX);
        if (((rand_no_x ) * (rand_no_x ) + (rand_no_y ) * (rand_no_y
        )) < 1) sum++;
     }
}
return 4*sum / npoints;
}
```

OpenMP Programming: Example

```
/* ****************************************************
An OpenMP version of a threaded program to compute PI.
**************************************************** */
double PI(long npoints, int no_threads) {
double sum = 0;
#pragma omp parallel shared (npoints) reduction(+: sum) num_threads(no_threads)
{
     int num_threads = omp_get_num_threads();
     long sample_points_per_thread = npoints / num_threads;
     sum = 0;
     double rand_no_x, rand_no_y;
std::time_t nt = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
thread_local unsigned long seed = time(&nt);
srand(seed);
     for (long i = 0; i < sample_points_per_thread; i++) {
          rand_no_y = (double)(rand()) / (double)(RAND_MAX);
          rand_no_x = (double)(rand()) / (double)(RAND_MAX);
          if (((rand_no_x ) * (rand_no_x ) + (rand_no_y ) * (rand_no_y )) < 1) sum++;
      }
}
return 4*sum / npoints;
}
```

Exemplu REDUCTION Clause

```
#include <omp.h>

main () {
int i, n, chunk;
float a[100], b[100], result;
/* Some initializations */
n = 100;
chunk = 10;
result = 0.0;
for (i=0; i < n; i++) {
  a[i] = i * 1.0;
  b[i] = i * 2.0;
}
#pragma omp parallel for default(shared) private(i) \
   schedule(static,chunk)  reduction(+:result)
   for (i=0; i < n; i++)
     result = result + (a[i] * b[i]);
// final regiune paralela
printf("Final result= %f\n",result);
}
```

Work-sharing constructs

- #pragma omp for [clause …]
- #pragma omp section [clause …]
- #pragma omp single [clause …]

- distributie automata intre threaduri
- trebuie sa fie incluse in regiuni paralele
- nu este implicit bariera la inceput dar este o bariera implicita la iesire daca nu se specifica altfel prin clauza nowait
- The work is distributed over the threads

omp for directive

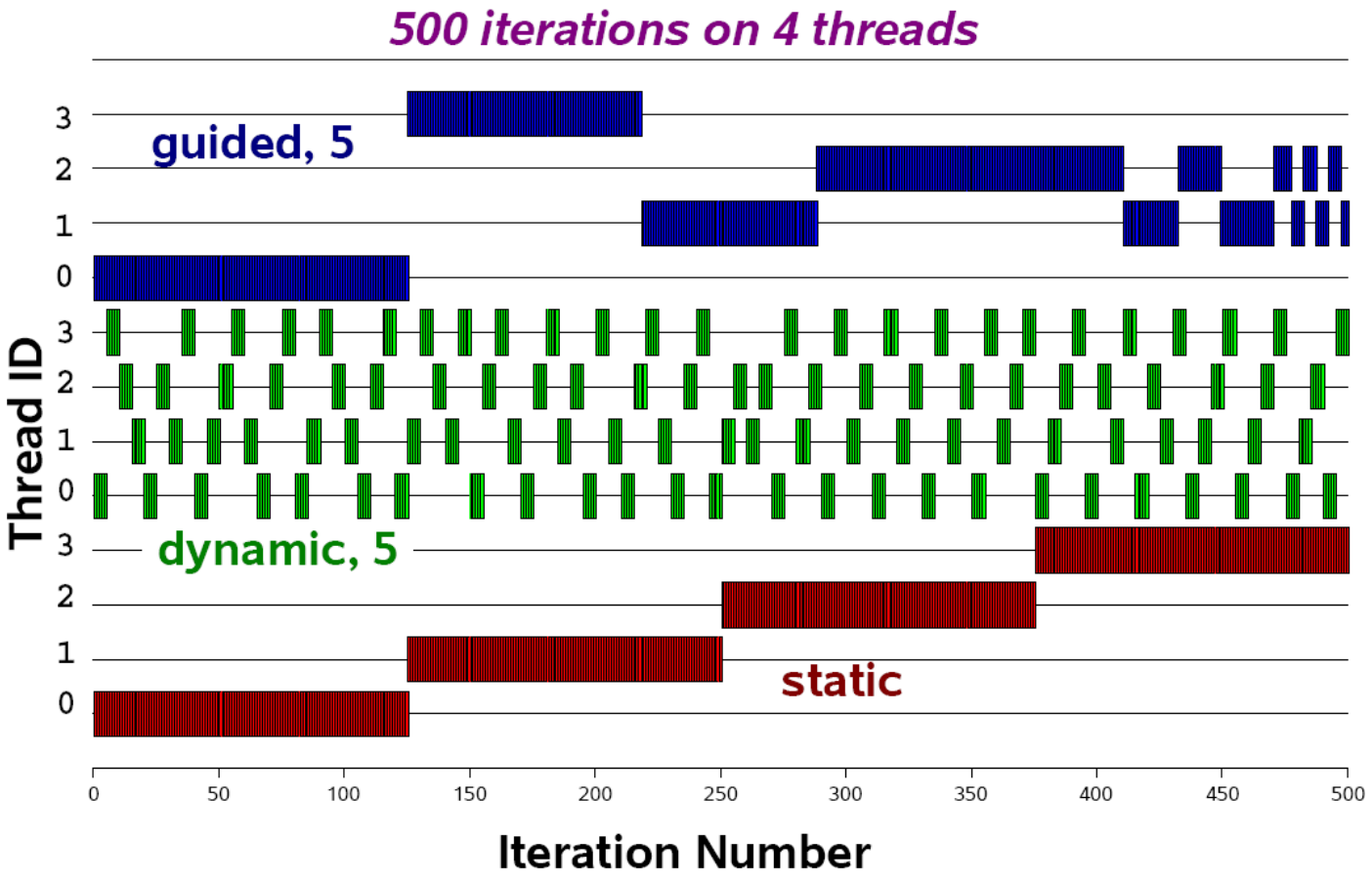
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)

- Schedule clause (decide how the iterations are executed in parallel):

  schedule (static | dynamic | guided [, chunk])



section directive

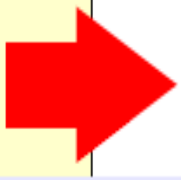
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```

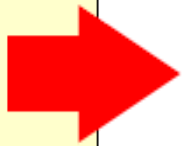
```
#pragma omp parallel
#pragma omp for
   for (...)
```



```
#pragma omp parallel for
for (....)
```

Single PARALLEL loop

```
#pragma omp parallel
#pragma omp sections
{ ...}
```



```
#pragma omp parallel sections
{ ... }
```

Single PARALLEL sections

Synchronization: barrier

```
For(=0; I<N; I++)
    a[I] = b[I] + c[I];

For(I=0; I<N; I++)
    d[I] = a[I] + b[I]
```

Both loops are in parallel region
With no synchronization in between.
What is the problem?

Fix:

```
For(I=0; I<N; I++)
    a[I] = b[I] + c[I];

#pragma omp barrier

For(I=0; I<N; I++)
    d[I] = a[I] + b[I]
```

Critical session

int sum=0

```
For(I=0; I<N; I++) {
   ……
   sum += A[I];
   ……
}
```

Cannot be parallelized if sum is shared.

Fix:

```
For(I=0; I<N; I++) {
   ……
   #pragma omp critical
  {
     sum += A[I];
   }
   ……
}
```

Sequential Matrix Multiply

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0;
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

OpenMP Matrix Multiply

```
#pragma omp parallel for private(j, k)
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0;
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[I][k] * b[k][j];
```

OpenMP Matrix Multiply - **collapse**

```
#pragma omp parallel for private(j, k) collapse(2)
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0;
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

se paralelizeaza ambele instructiuni for

FLUSH Directive

- Identifica un punct de sincronizare la care implementarea trebuie sa furnizeze o vedere consistenta a memoriei
- Variabilele care sunt vizibile threadurilor (globale) trebuie sa fie actualizare (written back to memory) daca sunt valori cashuite in threaduri
- Se instruieste compilatorul ca la acel punct o variabila trebuie sa fie “written to/read from the memory system” – nu mai poate fi tinuta intr-un registru CPU local
  - este posibil ca variabilele sa fie tinute intr-un registru atunci cand se executa un ciclu (loop) pentru a se eficientiza codul

FLUSH Directive (2)

C/C++: #pragma omp flush *(list)* newline

- The optional list contains a list of named variables that will be flushed in order to avoid flushing all variables. For pointers in the list, the pointer itself is flushed, not the object to which it points.
- Implementations must ensure any prior modifications to thread-visible variables are visible to all threads after this point; i.e., compilers must restore values from registers to memory, hardware might need to flush write buffers, etc.
- The FLUSH directive is implied for the directives shown in the table below. The directive is not implied if a NOWAIT clause is present.

FLUSH Directive (3)

- The FLUSH directive is implied for the directives shown in the table below.
  - The directive is not implied if a NOWAIT clause if present.

Fortran	C/C++
BARRIER END PARALLEL CRITICAL and END CRITICAL END DO END SECTIONS END SINGLE ORDERED and END ORDERED	`barrier` `parallel` – upon entry and exit `critical` – upon entry and exit `ordered` – upon entry and exit `for` – upon exit `sections` – upon exit `single` – upon exit

Synchronization Constructs in OpenMP

- OpenMP provides a variety of synchronization constructs:

```
#pragma omp barrier
#pragma omp single [clause list]
   structured block
#pragma omp master
   structured block
#pragma omp critical [(name)]
   structured block
#pragma omp ordered
   structured block
```

work construct - task

- foarte eficient
- se bazeaza pe o abordare de tip ‘thread-pool’
- avantaj fata de *section* – un task se poate tine in ‘asteptare’ (nefinalizat) pana cand alte taskuri se executa => implementare divide&impera

OpenMP Library Functions

```
/* controlling and monitoring thread creation */
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
/* mutual exclusion */
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

- all lock routines also have a nested lock counterpart for recursive mutexes.

rezumat

- #pragma omp parallel
  - regiune paralela
  - clauze:
- #pragma omp for
  - clauze
  - reduce
- #pragma omp sections
  - clauze
- #pragma omp barrier
- #pragma omp critical