ΥΟ7 – Παράλληλα Συστήματα
2011-12

2/4/2012
Προγραμματισμός συστημάτων κοινόχρηστης μνήμης (II)
OpenMP

Β. Δημακόπουλος

Shared address space / shared variables

- Τι χρειάζεται κανείς για να προγραμματίσει σε αυτό το μοντέλο:
  - Οντότητες εκτέλεσης (νήματα, διεργασίες)
    - Δημιουργία, διαχείριση
    - ***Όχι άμεση έννοια στο OpenMP***
  - Κοινές μεταβλητές μεταξύ των οντοτήτων
    - Ορισμός μεταβλητών (τι είναι κοινό και πως ορίζεται)
    - Τις διαβάζουν και τις τροποποιούν όλες οι διεργασίες
    - ***Οι καθολικές μεταβλητές (και όχι μόνο) στο OpenMP***
  - Αμοιβαίος αποκλεισμός
    - Π.χ. κλειδαριές
    - ***Κλειδαριές και στο OpenMP***
  - Συγχρονισμός
    - Π.χ. κλήσεις φραγής (barrier calls)
    - ***Κλήσεις φραγής, άμεσες και έμμεσες***

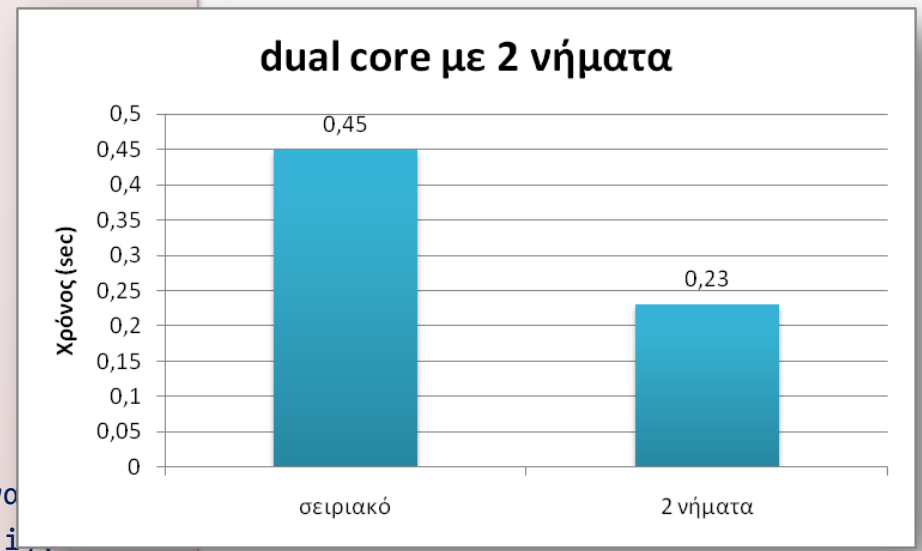
Γιατί OpenMP;

- Απλό!
- «Αυξητικό» (δηλαδή απλά προσθέτεις λίγο-λίγο παραλληλισμό στο υπάρχον σειριακό πρόγραμμα)
  - Όχι πάντα τόσο απλό, βέβαια!
- Υποστηρίζεται σχεδόν παντού πλέον
- Τρέχουσα έκδοση: 3.1

Με νήματα, βελτιστοποιημένο

```c
#include <pthread.h>
#define NPROCS 2          /* dual core */
#define N 10000000        /* Για ακρίβεια (ίδια με σειριακό) */
#define WORK N/NPROCS
double pi = 0.0, W = 1.0/N;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *thrfunc(void *iter) {
  int    i, me = (int) iter;
  double mysum = 0.0;
  for (i = me*WORK; i < (me+1)*WORK; i++)
    mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
  pthread_mutex_lock(&lock);
  pi += mysum;
  pthread_mutex_unlock(&lock);
}

main() {
  int i;
  pthread_t tids[NPROCS];

  for (i = 0; i < NPROCS; i++)  /* νήματα = # επεξεργο
    pthread_create(&tids[i], NULL, thrfunc, (void *) i);
  for (i = 0; i < NPROCS; i++)
    pthread_join(tids[i], NULL);
  printf("pi = %.10lf\n", pi);
}
```

```c
#define N 1000000
double pi = 0.0, W = 1.0/N;

main() {
  int i;
  for (i = 0; i < N; i++)
    pi += 4*W/(1+(i+0.5)*(i+0.5)*W*W);
  printf("pi = %.10lf\n", pi);
}
```



Με OpenMP

```
#define N 1000000
double pi = 0.0, W = 1.0/N;

main() {
  int i;
  for (i = 0; i < N; i++)
    pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
  printf("pi = %.10lf\n", pi);
}
```

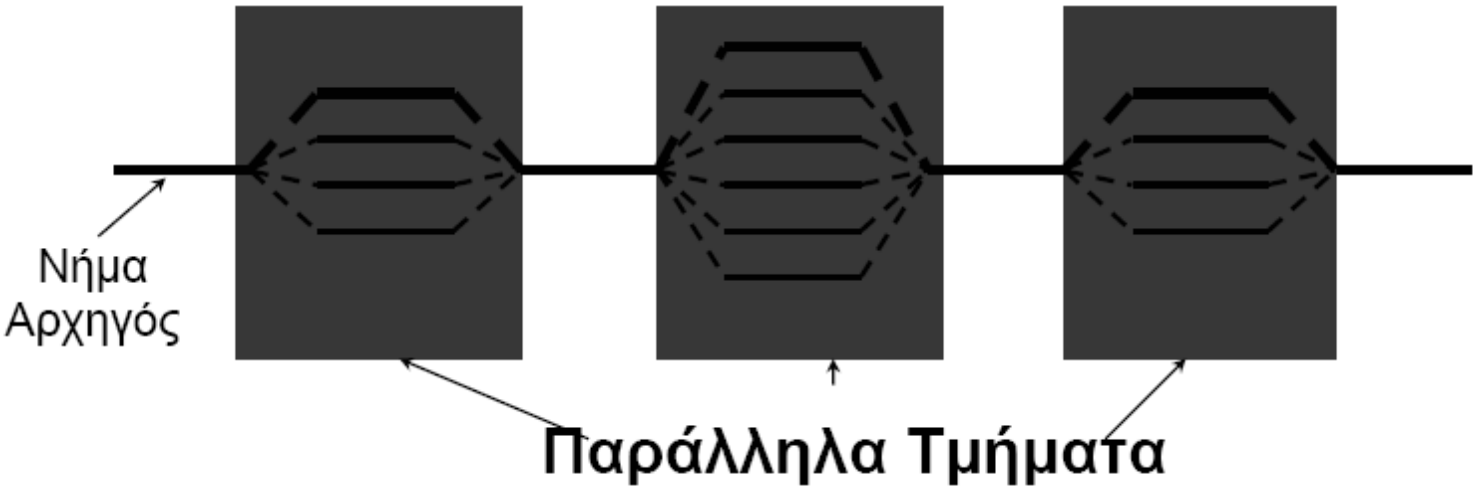
```
#include <omp.h>
#define N 1000000
double pi = 0.0, W = 1.0/N;

main () {
  int i;
  #pragma omp parallel for reduction(+:pi)
    for (i=0; i < N; i++)
      pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
  printf("pi = %.10lf\n", pi);
}
```

OpenMP

- Μοντέλο προγραμματισμού για παράλληλα συστήματα
- Οδηγίες προς τον μεταγλωττιστή ώστε να παράγει κώδικα με νήματα
- Όλες οι μεγάλες εταιρίες το υποστηρίζουν ή/και αποφασίζουν για την εξέλιξή του
  - Intel, SUN, IBM, HP, SGI, …
  - Microsoft (Visual Studio 2005)
  - GNU GCC
- Ερευνητικοί compilers
  - Omni (Ιαπωνία), NANOS/Mercurium (Ισπανία), OpenUH (ΗΠΑ)
  - OMPi (Ελλάδα - UoI) 

Προγραμματιστικό Μοντέλο

- Παραλληλισμός τύπου Fork-Join:
  - Το νήμα αρχηγός δημιουργεί ομάδα νημάτων σύμφωνα με τις ανάγκες
  - Ο παραλληλισμός προστίθεται βαθμιαία
    - το ακολουθιακό πρόγραμμα εξελίσσεται σε παράλληλο πρόγραμμα



Σύνταξη Οδηγιών

- Οι περισσότερες «εντολές» OpenMP είναι directives (οδηγίες) προς τον compiler ή pragmas.
- Για την C και C++, τα pragmas έχουν τη μορφή:
  - #pragma omp construct [clause [clause]…]
- Για τη Fortran, τα directives έχουν μία από τις ακόλουθες μορφές:
  - C$OMP construct [clause [clause]…]
  - !$OMP construct [clause [clause]…]
  - *$OMP construct [clause [clause]…]
- Αφού οι «εντολές» είναι directives και pragmas
  - ένα πρόγραμμα OpenMP μπορεί να μεταγλωττιστεί από compilers που δεν υποστηρίζουν OpenMP
  - οι τελευταίοι απλά αγνοούν τα directives / pragmas

Παράλληλα Τμήματα

- Νήματα δημιουργούνται στο OpenMP (στη C/C++) με το pragma “omp parallel”
- Για παράδειγμα, για να δημιουργηθεί ένα παράλληλο τμήμα με 4 νήματα:

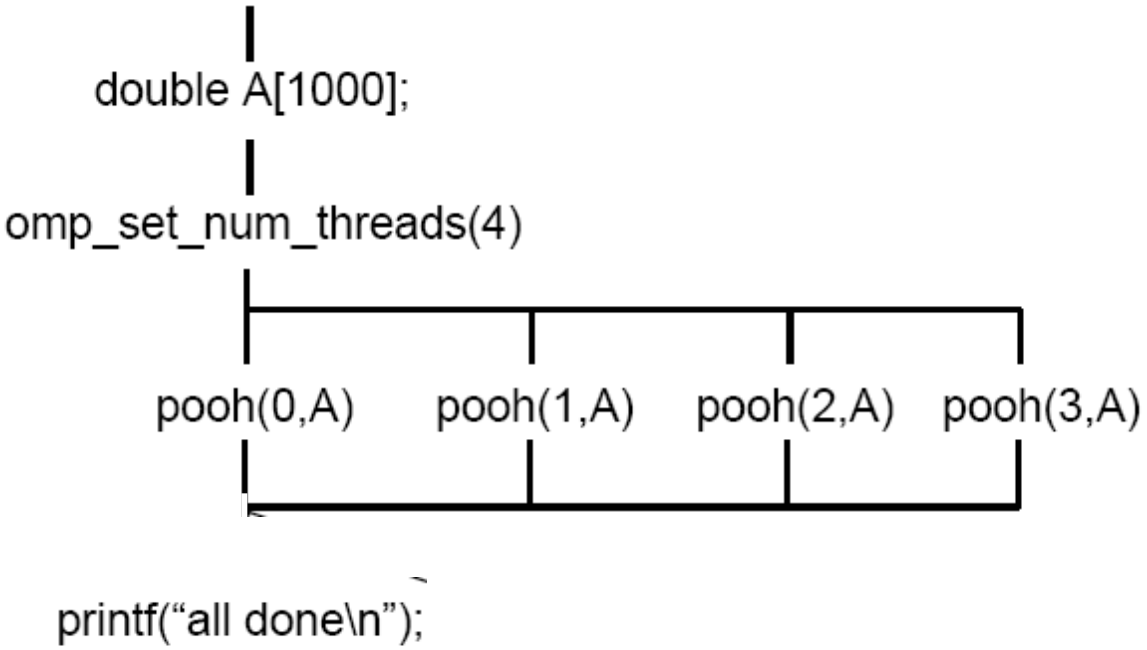
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_thread_num();
   pooh(ID,A);
}
```

- Κάθε νήμα εκτελεί για λογαριασμό του τον κώδικα μέσα στο δομημένο block του παράλληλου τμήματος
- Κάθε νήμα καλεί την pooh(ID) για ID = 0 έως 3

Παράλληλα Τμήματα

- Κάθε νήμα εκτελεί για λογαριασμό του τον ίδιο κώδικα
- Ένα μοναδικό αντίγραφο του A μοιράζεται μεταξύ των νημάτων
- Η εκτέλεση προχωρά μόνο όταν έχουν τελειώσει όλα τα νήματα (barrier)

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_thread_num();
        pooh(ID,A);
}
printf("all done\n");
```



Οδηγίες Διαμοίρασης Έργου (workshare directives)

- Ακολουθιακός κώδικας

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

- Παραλληλοποιημένος με OpenMP:

```
#pragma omp parallel
{
  int id, i, Nthrds, istart, iend;
  id = omp_get_thread_num();
  Nthrds = omp_get_num_threads();

  istart = id * N / Nthrds;    /* Οι επαναλήψεις που μου αντιστοιχούν */
  iend = (id+1) * N / Nthrds;
  for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
}
```

- Αντί αυτού, το OpenMP έχει κάτι πιο εύκολο:

```
#pragma omp parallel
  #pragma omp for schedule(static)
    for(i=0;i<N;i++) { a[i] = a[i] + b[i]; }
```

Οδηγίες Διαμοίρασης Έργου: for

- Το omp “for” κατανέμει τις επαναλήψεις ενός loop μεταξύ των νημάτων μιας ομάδας

```
#pragma omp parallel
{
   ...
  #pragma omp for
    for (I=0;I<N;I++) {
     NEAT_STUFF(I);
    }
   ...
}
```

- Εξ’ ορισμού υπονοείται barrier στο τέλος του omp for
- Για να αφαιρεθεί το barrier χρησιμοποιούμε την φράση “nowait”

Οδηγίες Διαμοίρασης Έργου: sections

- Η δομή διαμοίρασης έργου sections αναθέτει ένα διαφορετικό δομημένο block σε κάθε νήμα

```
#pragma omp parallel
{
   ...
   #pragma omp sections
   {
           #pragma omp section
                     x_calculation();
           #pragma omp section
                     y_calculation();
           #pragma omp section
                     z_calculation();
   }
   ...
}
```

- Εξ' ορισμού υπονοείται barrier στο τέλος του omp sections
- Για να αφαιρεθεί το barrier χρησιμοποιούμε την φράση "nowait"

Οδηγίες «Διαμοίρασης» Έργου: single

- Η οδηγία διαμοίρασης έργου single αναθέτει τον κώδικα που ακολουθεί σε ένα και μοναδικό νήμα

```
#pragma omp parallel
{
   ...
   #pragma omp single
   {
           calc();
   }
   ...
}
```

- Οποιοδήποτε από τα νήμα συναντήσει το single, μπορεί να το εκτελέσει – ενώ τα υπόλοιπα όχι.
- Εξ' ορισμού υπονοείται barrier στο τέλος του omp single
- Για να αφαιρεθεί το barrier χρησιμοποιούμε τη φράση "nowait"

Συνδυασμοί Εντολών

- Για διευκόλυνση, όλα σε ένα pragma
- Συνδυασμός της Εντολής parallel με δομές διαμοίρασης έργου

```
#pragma omp parallel for
   for (I=0;I<N;I++){
     NEAT_STUFF(I);
   }
```

- Υπάρχει και “parallel sections”
- Δεν υπάρχει “parallel single” (δεν έχει και νόημα)

Υπολογισμός του π (μέχρι τώρα)

```
#define N 1000000
double pi = 0.0, W = 1.0/N;

main() {
  int i;
  for (i = 0; i < N; i++)
    pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
  printf("pi = %.10lf\n", pi);
}
```

```
#define N 1000000
double pi = 0.0, W = 1.0/N;

main () {
  #pragma omp parallel
  {
    int i, mysum = 0.0;
    #pragma omp for
      for (i=0; i < N; i++)
        mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    #pragma omp critical
      pi += mysum;
  }
  printf("pi = %.10lf\n", pi);
}
```

Περιβάλλον δεδομένων

- Προγραμματιστικό μοντέλο κοινής μνήμης:
  - Οι περισσότερες μεταβλητές είναι εξ ορισμού κοινές
- Οι global μεταβλητές είναι κοινές μεταξύ των νημάτων
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- Όμως δεν είναι τα πάντα κοινά...
  - Οι stack variables σε υπο-προγράμματα που καλούνται από παράλληλα τμήματα είναι ιδιωτικές
  - Οι automatic variables μέσα σε ένα δομημένο block εντολών είναι ιδιωτικές.

Αλλαγή Χαρακτηριστικών Αποθήκευσης

- Ο προγραμματιστής μπορεί να αλλάξει τα χαρακτηριστικά αποθήκευσης των μεταβλητών χρησιμοποιώντας μία από τις ακόλουθες φράσεις
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - THREADPRIVATE
- Η τιμή μιας ιδιωτικής μεταβλητής εντός ενός παράλληλου loop μπορεί να «μεταδοθεί» σαν καθολική τιμή εκτός του loop με τη:
  - LASTPRIVATE
- Η default συμπεριφορά μπορεί να μεταβληθεί με τη:
  - DEFAULT (PRIVATE | SHARED | NONE)
- Όλες οι εντολές δεδομένων εφαρμόζονται σε παράλληλα τμήματα και δομές διαμοίρασης έργου εκτός της “shared” η οποία εφαρμόζεται μόνο σε παράλληλα τμήματα.
- Όλες οι παραπάνω εντολές έχουν ισχύ στο lexical extent της εντολής OpenMP (δηλαδή ανάμεσα από τα άγκιστρα που καθορίζουν το block).

Περιβάλλον Δεδομένων

- Παράδειγμα με χρήση των PRIVATE και FIRSTPRIVATE

```
int A, B, C;
A = B = C = 1;
#pragma omp parallel shared(A) private(B) firstprivate(C)
{
   #pragma omp single
     A++;
   B++; C++;
   printf("%d, %d, %d", A,B,C);
}
printf("%d, %d, %d", A,B,C);
```

- Τι θα τυπωθεί μέσα στο παράλληλο τμήμα
  - 2 (ή 1), <τυχαία τιμή>, 2
- Τι θα τυπωθεί μετά το παράλληλο τμήμα (OpenMP > 2.5)?
  - 2, 1, 1

Παράδειγμα

Είναι σωστός ο υπολογισμός του π;

```
#define N 1000000
double pi = 0.0, W = 1.0/N;

main () {
  int i, mysum = 0.0;

  #pragma omp parallel
  {
    #pragma omp for
      for (i=0; i < N; i++)
        mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    #pragma omp critical
      pi += mysum;
  }
  printf("pi = %.10lf\n", pi);
}
```

Σωστές επιλογές:

```
main () {
  int i, mysum = 0.0;

  #pragma omp parallel private(i)\
       firstprivate(mysum)
  {
```

```
main () {
  #pragma omp parallel shared(W,pi)
  {
    int i, mysum = 0.0;
```

Εντολή threadprivate

- Κάνει global δεδομένα ιδιωτικά σε κάθε νήμα
  - Fortran: COMMON blocks
  - C: File scope και static variables
- Διαφορετική συμπεριφορά από το PRIVATE
  - Με το PRIVATE οι global μεταβλητές αποκρύπτονται.
  - Το THREADPRIVATE διατηρεί το global scope σε κάθε νήμα
- Οι μεταβλητές threadprivate μπορούν να αρχικοποιηθούν χρησιμοποιώντας εντολές COPYIN ή DATA

Φράση reduction

- Επηρεάζει στην ουσία τον τρόπο «διαμοίρασης» των μεταβλητών:
  - reduction (op : list)
- Οι μεταβλητές στο “list” πρέπει να είναι shared στο παράλληλο τμήμα που βρισκόμαστε.
- Εντός μια δομής parallel ή διαμοίρασης εργασίας:
  - Δημιουργείται τοπικό αντίγραφο κάθε μεταβλητής της λίστας και αρχικοποιείται (ανάλογα με την πράξη “op” π.χ. 0 για “+”)
  - Τα τοπικά αντίγραφα συνδυάζονται ώστε να εκφυλιστούν σε ένα μοναδικό global αντίγραφο στο τέλος της δομής

Υπολογισμός του π

```
#define N 1000000
double pi = 0.0, W = 1.0/N;

main() {
  int i;
  for (i = 0; i < N; i++)
    pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
  printf("pi = %.10lf\n", pi);
}
```

```
#define N 1000000
double pi = 0.0, W = 1.0/N;

main () {
  #pragma omp parallel
  {
    int i, mysum = 0.0;
    #pragma omp for
      for (i=0; i < N; i++)
        mysum += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    #pragma omp critical
     pi += mysum;

ntf("pi = %.10lf\n", pi);
```

```
#define N 1000000
double pi = 0.0, W = 1.0/N;

main () {
  int i;
  #pragma omp parallel for private(i) reduction(+:pi)
    for (i=0; i < N; i++)
      pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
  printf("pi = %.10lf\n", pi);
}
```

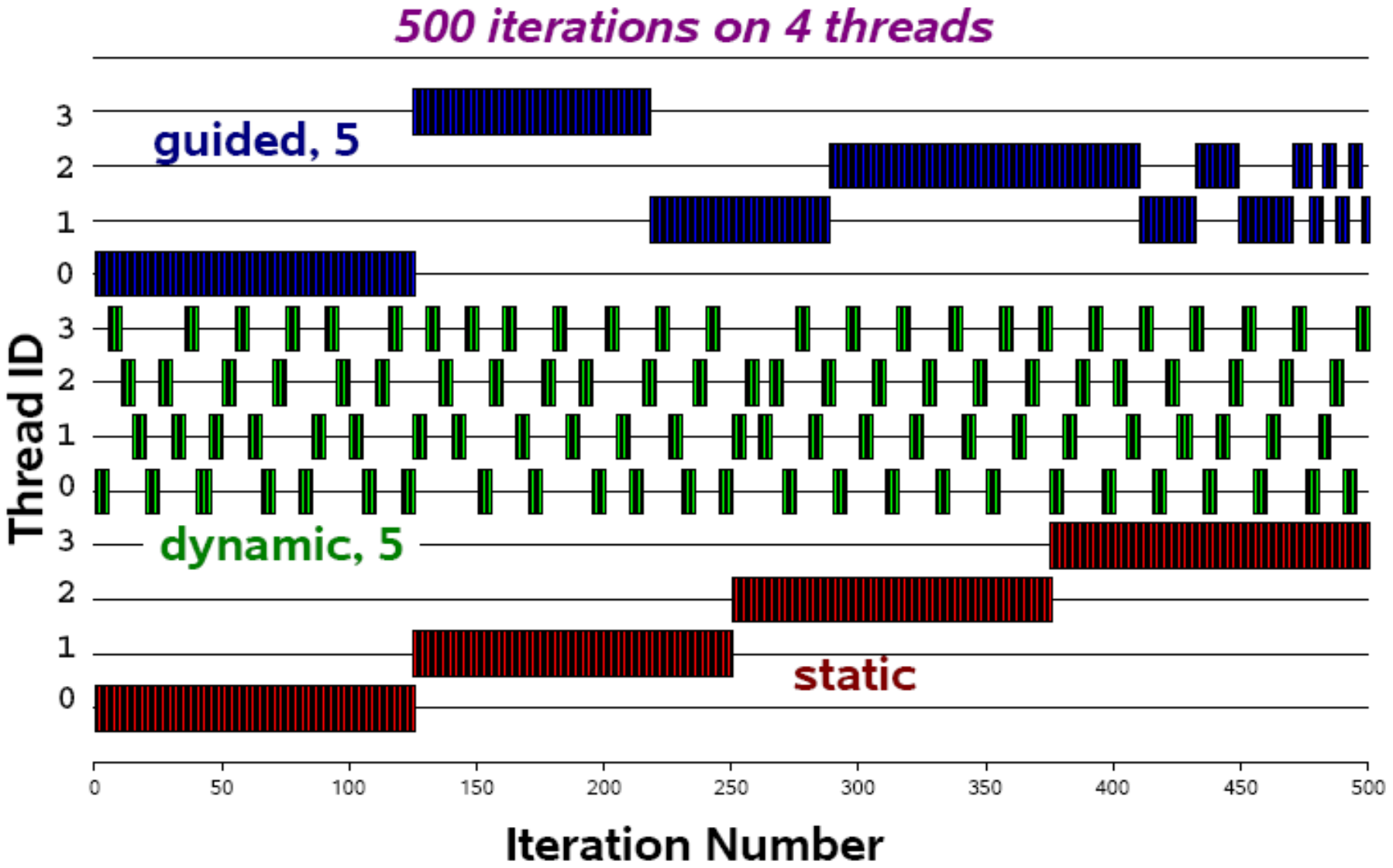
Εξισορρόπηση Φόρτου

- Αποτελεί βασικό παράγοντα απόδοσης
- Για συνηθισμένες λειτουργίες, π.χ. πρόσθεση διανυσμάτων, η εξισορρόπηση εργασίας δεν αποτελεί ζήτημα
- Για λιγότερο ομαλές λειτουργίες πρέπει να δοθεί ιδιαίτερη σημασία στην διαμοίραση της εργασίας μεταξύ των νημάτων
- Παράδειγμα μη ομαλών (irregular) λειτουργιών:
  - Πολλαπλασιασμός αραιών πινάκων
  - Παράλληλες αναζητήσεις σε μία διασυνδεδεμένη λίστα
- Για τέτοιες περιπτώσεις, η δομή διαμοίραση for χρησιμοποιείται με την οδηγία schedule που καθορίζει διάφορους αλγορίθμους δρομολόγησης των επαναλήψεων

Οδηγία schedule

- Χρήση
  - schedule (static | dynamic | guided [, chunk])
  - schedule (runtime)
- static [,chunk]
  - Διαμοιράζει τις επαναλήψεις, που έχουν χωριστεί σε τμήματα μεγέθους "chunk«, μεταξύ των νημάτων με κυκλικό τρόπο
  - Αν δεν ορίζεται το "chunk", αυτό ορίζεται κατά προσέγγιση ίσο με N/P και κάθε νήμα εκτελεί ένα τμήμα επαναλήψεων
- dynamic [,chunk]
  - Διαχωρίζει τις επαναλήψεις σε τμήματα μεγέθους "chunk“
  - Κάθε νήμα μόλις τελειώσει ένα τμήμα, παίρνει δυναμικά το επόμενο
- guided [,chunk]
  - Παρόμοια με dynamic, αλλά το μέγεθος του τμήματος μειώνεται εκθετικά
  - σε κάθε βήμα το μέγεθος του τμήματος είναι *ανάλογο* του (# unassigned iterations / # threads), αλλά όχι λιγότερο από chunk.
- runtime
  - Ο αλγόριθμος δρομολόγησης καθορίζεται κατά τον χρόνο εκτέλεσης ελέγχοντας τη μεταβλητή περιβάλλοντος OMP_SCHEDULE

Παράδειγμα


500 iterations on 4 threads
guided, 5
dynamic, 5
static
Thread ID
Iteration Number
0
50
100
150
200
250
300
350
400
450
500

Αμοιβαίος αποκλεισμός και συγχρονισμός

- Το OpenMP ορίζει τις ακόλουθες οδηγίες:
  - atomic
  - critical section
  - barrier
  - flush
  - ordered
  - master (στην ουσία δεν είναι εντολή συγχρονισμού και ΔΕΝ υπονοείται barrier στο τέλος του!)

Αμοιβαίος αποκλεισμός – critical, atomic

- Μόνο ένα νήμα μπορεί κάθε στιγμή να μπει στο critical section

```
#pragma omp parallel for private(b) shared(res)
    for (i =0; i<niters; i++) {
        b = doit(i);
        #pragma omp critical
        {
            consume(b, &res);
        }
    }
```

- Το atomic είναι ειδική περίπτωση του critical section που μπορεί να χρησιμοποιηθεί για ορισμένες απλές εντολές. Εφαρμόζεται μόνο για την τροποποίηση μίας θέσης μνήμης (την ανανέωση του x στο ακόλουθο παράδειγμα)

```
#pragma omp parallel private(b)
{
    b = doit(i);
    #pragma omp atomic
      x = x + b;
}
```

Συγχρονισμός – barriers

- Barrier: Κάθε νήμα περιμένει να φτάσουν όλα τα υπόλοιπα νήματα πριν προχωρήσει.

```
#pragma omp parallel shared (A, B, C) private(id)
{
  id=omp_get_thread_num();
  A[id] = big_calc1(id);
  #pragma omp barrier

  #pragma omp for
    for (i=0;i<N;i++){
      C[i]=big_calc3(I,A);
    }    /* υπονοούμενο barrier */

  #pragma omp for nowait
    for (i=0;i<N;i++){
      B[i]=big_calc2(C, i);
    }    /* χωρίς barrier */
  A[id] = big_calc3(id);
} /* υπονοούμενο barrier */
```

Συγχρονισμός – flush

- Η εντολή flush δηλώνει ένα σημείο στο οποίο το νήμα επιχειρεί να δημιουργήσει συνεπή εικόνα της μνήμης.
  - Όλες οι πράξεις μνήμης (αναγνώσεις και εγγραφές) που ορίζονται *πριν* από αυτό το σημείο πρέπει να ολοκληρωθούν.
  - Όλες οι πράξεις μνήμης (αναγνώσεις και εγγραφές) που ορίζονται *μετά* από αυτό το σημείο πρέπει να εκτελεστούν *μετά* το flush
  - Οι μεταβλητές σε registers ή write buffers πρέπει να εγγραφούν στη μνήμη
- Τα ορίσματα του flush ορίζουν ποιες μεταβλητές θα γίνουν flush. Αν δεν υπάρχουν ορίσματα όλες οι ορατές στο νήμα μεταβλητές γίνονται flush.
- Πρόκειται για memory fence που επιβάλλει συνέπεια μνήμης

Υπονοούμενος Συγχρονισμός

- Barriers υπονοούνται μετά τις ακόλουθες εντολές OpenMP:
  - end parallel
  - end do (except when nowait is used)
  - end sections (except when nowait is used)
  - end critical
  - end single (except when nowait is used)
- Flush υπονοείται μετά τις ακόλουθες εντολές OpenMP:
  - barrier
  - critical, end critical
  - end do
  - end parallel
  - end sections
  - end single
  - ordered, end ordered

Συναρτήσεις Βιβλιοθήκης

- Συναρτήσεις locks
  - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock()
- Συναρτήσεις περιβάλλοντος χρόνου εκτέλεσης:
  - Αλλαγή/Έλεγχος του αριθμού των νημάτων
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Ενεργοποίηση/απενεργοποίηση εμφωλευμένου παραλληλισμού και dynamic mode
    - omp_set_nested(), omp_set_dynamic(),
    - omp_get_nested(), omp_get_dynamic()
  - Έλεγχος εκτέλεσης σε παράλληλο τμήμα
    - omp_in_parallel()
  - Αριθμός επεξεργαστών στο σύστημα
    - omp_num_procs()

Συναρτήσεις βιβλιοθήκης

- Για να ελεγχθεί ο αριθμός των νημάτων που εκτελούν ένα πρόγραμμα αρχικά απενεργοποιείται το dynamic mode και κατόπιν καθορίζεται ο αριθμός των νημάτων.

```
#include <omp.h>
void main()
{
      omp_set_dynamic(0);
      omp_set_num_threads(4);
      #pragma omp parallel
      {
            int id=omp_get_thread_num();
            do_lots_of_stuff(id);
      }
}
```

Μεταβλητές Περιβάλλοντος

- Έλεγχος scheduling των loops εφόσον έχει καθοριστεί “omp for schedule(RUNTIME)”.
  - OMP_SCHEDULE “schedule[, chunk_size]”
- Καθορισμός του default αριθμού νημάτων.
  - OMP_NUM_THREADS int_literal
- Ενεργοποίηση/απενεργοποίηση dynamic mode
  - OMP_DYNAMIC TRUE || FALSE
- Ενεργοποίηση/απενεργοποίηση παράλληλης εκτέλεσης εμφωλευμένων παράλληλων τμημάτων
  - OMP_NESTED TRUE || FALSE

Χρήση

- Κώδικας OpenMP

```
#include <omp.h>
#include <stdio.h>
void main() {
   #pragma omp parallel
   {
     printf("Hello world from thread %d of %d\n",
              omp_get_thread_num(), omp_get_num_threads();
   }
}
```

- Παραγωγή εκτελέσιμου αρχείου:

```
$ ompicc –o hello hello.c
$ gcc –fopenmp –o hello hello.c
$ icc –openmp –o hello hello.c
$ suncc –xopenmp=parallel –o hello hello.c
```

Εκτέλεση :

```
$ export OMP_NUM_THREADS=4
$ ./hello
Hello world from thread 0 of 4
Hello world from thread 2 of 4
Hello world from thread 1 of 4
Hello world from thread 3 of 4
$ export OMP_NUM_THREADS=1
$ ./hello
Hello world from thread 0 of 1
```

Πίνακας επί πίνακα (4 CPUs)

```
for (i = 0; i < N; i++)
{
  for (j = 0; j < N; j++)
    for (k = sum = 0; k < N; k++)
      sum += A[i][k]*B[k][j];
  C[i][j] = sum;
}
```

Χρόνος: 130msec

```
#pragma omp parallel for
  for (i = 0; i < N; i++)
  {
    for (j = 0; j < N; j++)
      for (k = sum = 0; k < N; k++)
        sum += A[i][k]*B[k][j];
    C[i][j] = sum;
  }
```

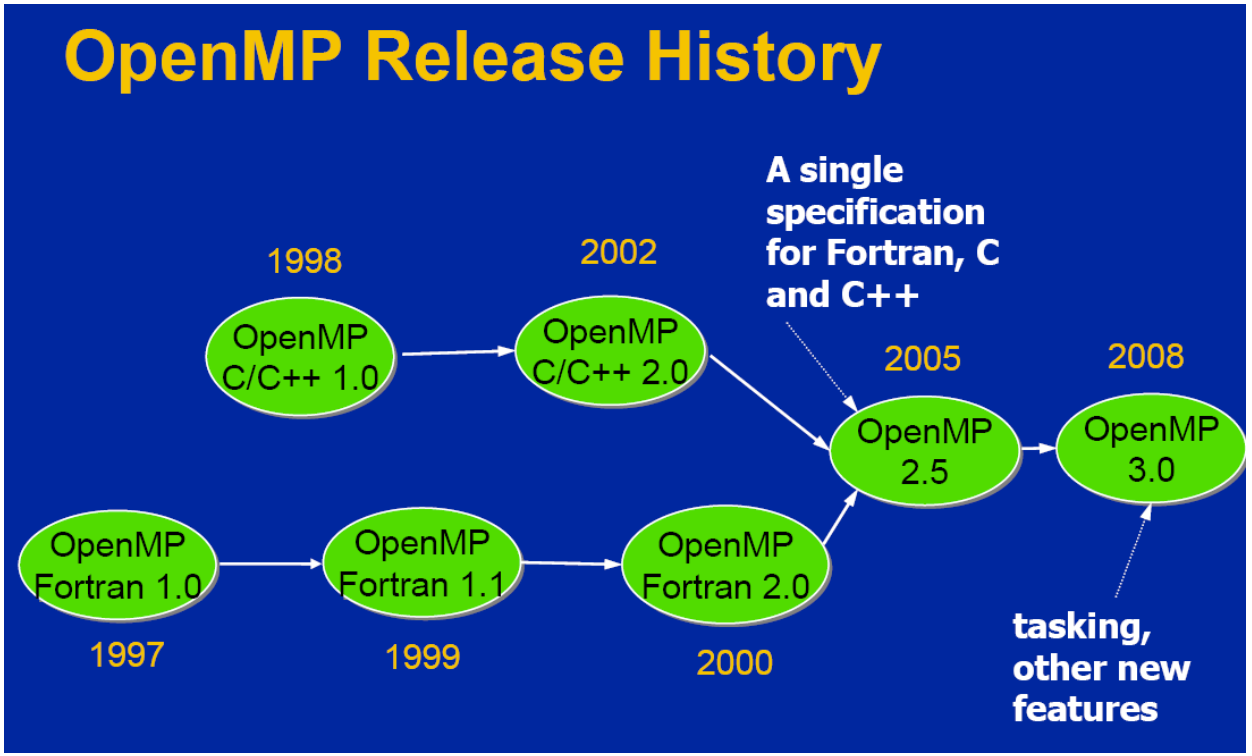
Χρόνος: 700msec

```
#pragma omp parallel for private(j,k,sum)
  for (i = 0; i < N; i++)
  {
    for (j = 0; j < N; j++)
      for (k = sum = 0; k < N; k++)
        sum += A[i][k]*B[k][j];
    C[i][j] = sum;
  }
```

Χρόνος: 40msec

OpenMP 3.0 & Tasks

- Στο OpenMP 3.0 ξεκαθαρίστηκαν κάποιες λεπτομέρειες, προστέθηκαν κάποιες νέες συναρτήσεις που μπορούν να κληθούν από τα προγράμματα (π.χ. εύρεση του nesting level).
- Το βασικότερο όμως ήταν η προσθήκη των *tasks*!



OpenMP Tasks

- Γενική ιδέα:
  - «ορισμός» μιας δουλείας που πρέπει να γίνει από κάποιο thread, κάποια στιγμή (και όχι οπωσδήποτε άμεσα από το νήμα που την όρισε)
- Βολεύει πολύ στην παραλληλοποίηση πολλών εφαρμογών
- Επίσης με αυτό τον τρόπο μπορεί σε αρκετές περιπτώσεις να αποφευχθεί η χρήση εμφωλευμένων παράλληλων περιοχών που δεν τις χειρίζονται καλά οι περισσότεροι compilers…

Απλή εφαρμογή: παραλληλοποίηση λειτουργιών σε συνδεδεμένη λίστα

```
        ........

while(my_pointer) {


        (void) do_independent_work (my_pointer);

  my_pointer = my_pointer->next ;
} // End of while loop

        ........
```

- Πώς παραλληλοποιείται στο OpenMP 2.5?
  - Πρώτα ένα πέρασμα για να βρεθεί το πλήθος των επαναλήψεων
  - Στην συνέχεια **#pragma omp parallel for**
  - Θα πρέπει να φυλάμε επίσης έναν πίνακα με όλους τους pointers

Με tasks

```
my_pointer = listhead;

#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer) {
          #pragma omp task firstprivate(my_pointer)
          {
              (void) do_independent_work (my_pointer);
          }
          my_pointer = my_pointer->next ;
        }
    } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

OpenMP Task is specified here (executed in parallel)

OpenMP tasks

- Ένα task έχει:
  - Κώδικα που πρέπει να εκτελεστεί:

```
#pragma omp task
{
  κώδικας
}
```

  - Μεταβλητές πάνω στις οποίες θα δουλέψει
    - Αφού θα εκτελεστεί πιθανώς αργότερα, θα πρέπει να «κουβαλάει» μαζί του και τις τιμές των μεταβλητών που υπήρχαν κατά τον ορισμό του
  - Ένα καθορισμένο νήμα
    - Που θα το εκτελέσει τον κώδικα
    - Όχι όμως πάντα προ-καθορισμένο.

OpenMP tasks

- Δύο φάσεις: (α) ορισμός/πακετάρισμα και (β) εκτέλεση
  - Το νήμα που συναντά το **#pragma omp task** πρέπει να δημιουργήσει μία δομή που περιέχει τον κώδικα αλλά και τα δεδομένα με τις τρέχουσες τιμές τους
  - Κάποια στιγμή, κάποιο νήμα θα πάρει το «πακέτο» και θα το εκτελέσει

(ο κόπος για το πακετάρισμα δεν χρειάζεται αν το νήμα που συναντά το **#pragma omp task** εκτελέσει απευθείας το task)

Γενική σύνταξη

task Construct

```
#pragma omp task [clause[[,]clause] ...]
        structured-block
```

where *clause* can be one of:

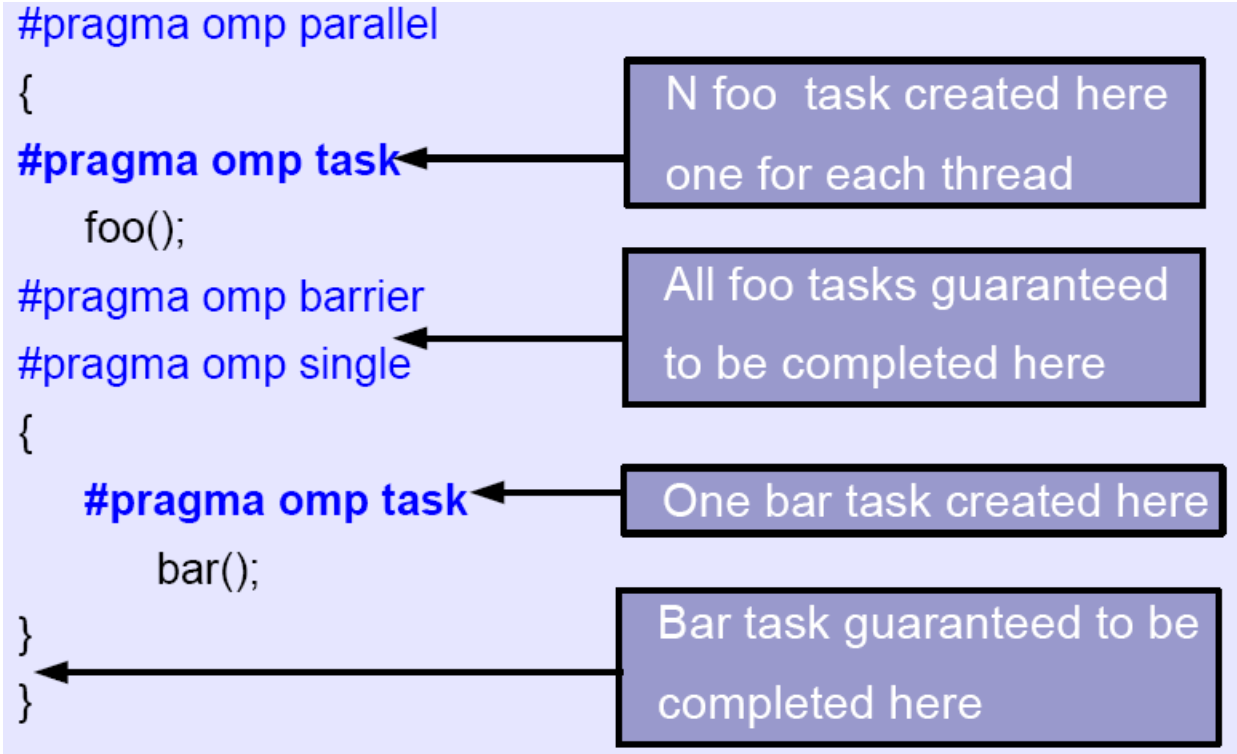
```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none  )
```

Φράσεις

- Φράση if(συνθήκη)
  - Αν η συνθήκη είναι false τότε το νήμα εκτελεί άμεσα το task
    - Π.χ. αν το κόστος της δημιουργίας ενός task είναι μεγάλο σε σχέση με τους υπολογισμούς που περιλαμβάνει
    - Ή π.χ. για να cache/memory affinity
- Φράση untied
  - Κανονικά, ένα task είναι «δεμένο» (“tied”) *με το νήμα που θα ξεκινήσει να το εκτελεί* – δηλαδή θα εκτελεστεί από το νήμα αυτό μέχρι τέλους
  - Σε ένα ελεύθερο task (“untied”), η εκτέλεσή του μπορεί να διακόπτεται και να συνεχίζει την εκτέλεσή του άλλο νήμα, κλπ.
- Λεπτομέρειες:
  - Το πρόγραμμα είναι όλο ένα task.
  - Όταν ένα νήμα συναντά μία παράλληλη περιοχή, φτιάχνει N tasks (implicit):
    - Είναι tied, ένα task σε κάθε ένα από τα N νήματα που θα δημιουργηθούν
  - Ένα task που εκτελείται μπορεί να δημιουργήσει άλλα tasks

Συγχρονισμός των tasks

- Στο τέλος ενός barrier, όλα τα tasks που έχει δημιουργήσει μία ομάδα νημάτων θα έχουν ολοκληρωθεί.
- Το task που συναντάει task barrier (**#pragma omp taskwait**) μπλοκάρει μέχρι να ολοκληρωθούν όλα τα tasks που δημιούργησε



Γενική σύνταξη – περιβάλλον δεδομένων

- Αν δεν υπάρχει το default(), τότε οι global μεταβλητές είναι κλασικά shared. Οι υπόλοιπες θεωρούνται shared μόνο αν είναι shared σε όλες τις περιβάλλουσες περιοχές του κώδικα μέχρι την πιο πρόσφατη παράλληλη περιοχή. Αλλιώς είναι *firstprivate*.

task Construct

```
#pragma omp task [clause[[,]clause] ...]
        structured-block
```

where *clause* can be one of:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none  )
```

fibonacci

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;

    x = fib(n-1);

    y = fib(n-2);

    return x+y;;
}
```

fibonacci

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
#pragma omp task
    x = fib(n-1);
#pragma omp task
    y = fib(n-2);
#pragma omp taskwait
    return x+y;;
}
```

guarantees results are ready

fibonacci

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
#pragma omp task
    x = fib(n-1);
#pragma omp task
    y = fib(n-2);
#pragma omp taskwait
    return x+y;;
}
```

Correct

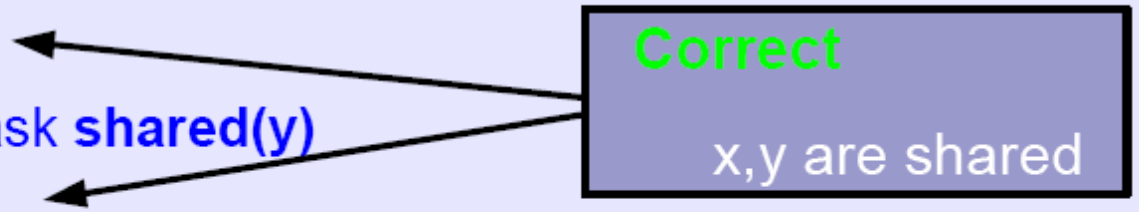
n is firstprivate

Wrong!

x,y are firstprivate

fibonacci

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
#pragma omp task shared(x)
    x = fib(n-1);
#pragma omp task shared(y)
    y = fib(n-2);
#pragma omp taskwait
    return x+y;;
}
```



Διάσχιση λίστας

```
List l;
Element e;
#pragma omp parallel
#pragma omp single
{
    for ( e = l->first; e ; e = e->next )
    #pragma omp task
        process(e);
}
```

Διάσχιση λίστας

```
List l;
Element e;
#pragma omp parallel
#pragma omp single
{
    for ( e = l->first; e ; e = e->next )
    #pragma omp task
        process(e);
}
```

Wrong!

e is shared here

Διάσχιση λίστας

```
List l;
Element e;
#pragma omp parallel
#pragma omp single
{
    for ( e = l->first; e ; e = e->next )
    #pragma omp task firstprivate(e)
        process(e);
}
```

Γενικότερα προβλήματα στον προγραμματισμό συστημάτων κοινόχρηστης μνήμης

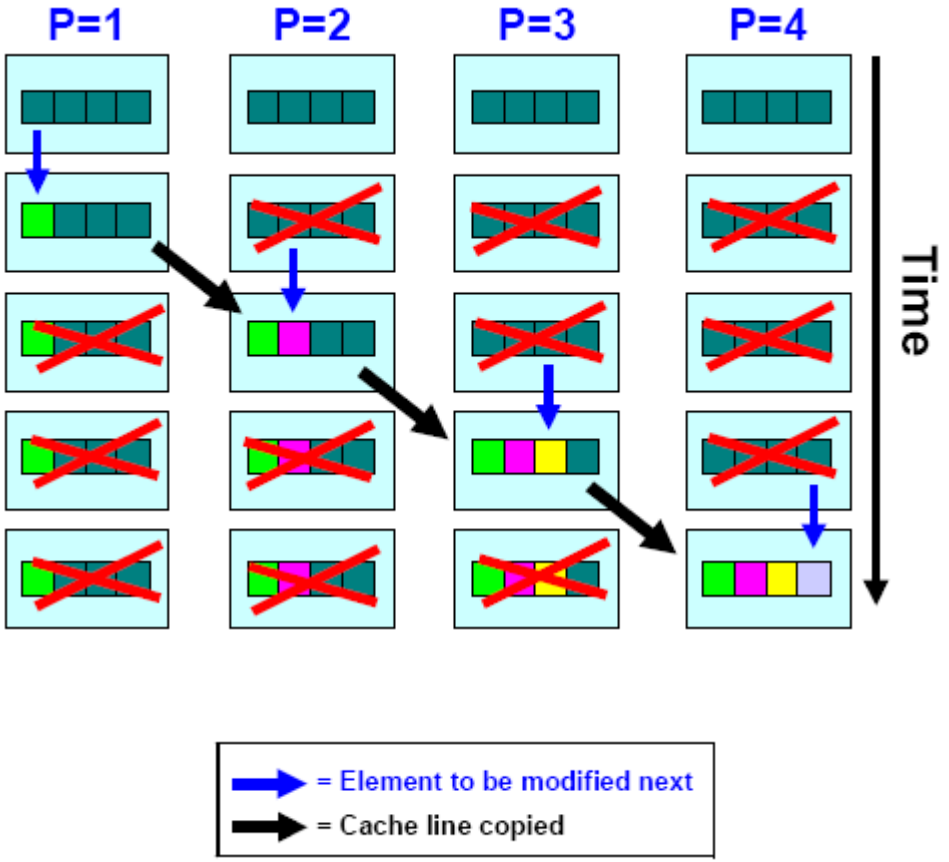
False sharing

- Οι caches δεν αποθηκεύουν 1 λέξη αλλά ένα ολόκληρο μπλοκ μνήμης σε κάθε γραμμή τους. Ας υποθέσουμε για παράδειγμα ότι έχουμε το παρακάτω πρόγραμμα και ότι κάθε γραμμή της cache χωράει 4 συνεχόμενες λέξεις.
- Έχουμε 4 νήματα (εκτελούνται σε 4 επεξεργαστές) και το νήμα i (i = 0, 1, 2, 3) εκτελεί:

```
1. sum = a[0]+a[1]+a[2]+a[3];    /* Όλος ο a[] στην cache */

2. a[i] = a[i] + i;
```

- Μετά την πρώτη εντολή, στις cache των 4 επεξεργαστών υπάρχει ολόκληρος ο πίνακας a[].

False sharing II



False sharing III

- Παρότι τα νήματα αλλάζουν διαφορετικά στοιχεία, υπάρχει πρόβλημα μιας και οι caches θεωρούν ότι άλλαξε η γραμμή που φυλάνε και εκκινούν τους μηχανισμούς συνέπειας!

- **ΔΙΔΑΓΜΑ**: ***υπάρχει κίνδυνος καθυστερήσεων ακόμα και όταν τα νήματα δεν τροποποιούν κάτι κοινό***

- **Λύση;**
  - Τα δεδομένα να έχουν μέγεθος όσο και η γραμμή της cache και
  - Να είναι στοιχισμένα σε θέσεις μνήμης που είναι ακέραια πολλαπλάσια του μεγέθους της γραμμής.

NUMA & συνέπειες

- Όλοι οι επεξεργαστές έχουν caches. Τι γίνεται αν κάποια στιγμή ο επεξεργαστής φέρει ένα δεδομένο από έναν άλλο επεξεργαστή;
  - Το δεδομένο θα πάει στην cache
  - Το πρόβλημα της συνέπειας πώς λύνεται σε αυτή την περίπτωση (αφού δεν υπάρχει δίαυλος να «παρακολουθείται»);

- Δύο λύσεις:
  - ΑΠΑΓΟΡΕΥΕΤΑΙ τα απομακρυσμένα δεδομένα να μπαίνουν στην cache (π.χ. Cray T3D) ή
  - ΕΠΙΤΡΕΠΕΤΑΙ, αλλά επιπλέον χρησιμοποιούνται νέα πρωτόκολλα συνοχής, βασισμένα σε καταλόγους (**ccNUMA** – cache coherent NUMA)
    - Directory-based cache coherence

Τοποθέτηση δεδομένων / νημάτων

- Το false sharing έχει ακόμα μεγαλύτερες επιπτώσεις
- Βασικό μέλημα το πού θα τοποθετηθούν τα νήματα και πού τα δεδομένα
  - Αν τα δεδομένα που χρησιμοποιεί πιο συχνά ένα νήμα πάνε σε άλλον επεξεργαστή από ότι το νήμα, τότε θα έχουμε καθυστερήσεις (NUMA factor)
  - Συνήθης τακτική στα λειτουργικά συστήματα:
    - **First touch** (όποιο νήμα προσπελάσει πρώτο κάποιο δεδομένο, τότε «τραβάει» το δεδομένο αυτό στην μνήμη του επεξεργαστή που το εκτελεί. Το δεδομένο δεν μετακινείται από εκεί και πέρα).
  - **ΔΙΔΑΓΜΑ???**
    - Θέλει προσοχή στις *αρχικοποιήσεις*.
    - Δεν μπορεί τις αρχικοποιήσεις να τις κάνει μόνο ένα νήμα!!