

ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Κοινόχρηστος χώρος διευθύνσεων

Κεφάλαιο 4

Μέρος II: OpenMP



❖ Τι χρειάζεται κανείς για να προγραμματίσει σε αυτό το μοντέλο:

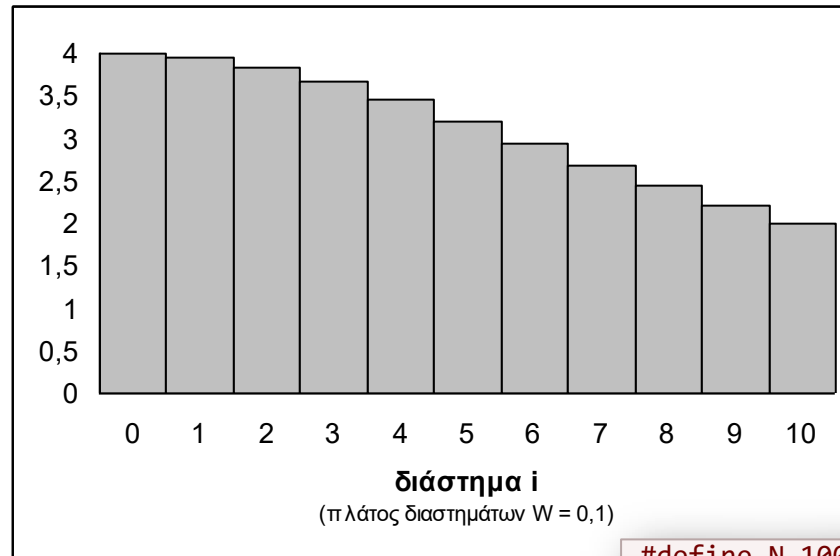
- Οντότητες εκτέλεσης (νήματα, διεργασίες)
 - ✧ Δημιουργία, διαχείριση
 - ✧ **Όχι άμεση έννοια στο OpenMP**
- Κοινές μεταβλητές μεταξύ των οντοτήτων
 - ✧ Ορισμός μεταβλητών (τι είναι κοινό και πως ορίζεται)
 - ✧ Τις διαβάζουν και τις τροποποιούν όλες οι διεργασίες
 - ✧ **Οι καθολικές μεταβλητές (και 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**

- ❖ Απλό!
- ❖ «Αυξητικό» (δηλαδή απλά προσθέτεις λίγο-λίγο παραλληλισμό στο υπάρχον σειριακό πρόγραμμα)
 - Όχι πάντα τόσο απλό, βέβαια!
- ❖ Υποστηρίζεται σχεδόν παντού πλέον, ιδιαίτερα δημοφιλές
- ❖ Πρόσφατα (v4.0, v4.5) η εμβέλειά του επεκτείνεται και στις επιπλέον *συσκευές / devices* ενός συστήματος (συνεπεξεργαστές, επιταχυντές, GPUs, κλπ)

Υπενθύμιση: υπολογισμός του $\pi = 3,14\dots$

❖ Αριθμητική ολοκλήρωση

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



$$\approx \sum_{i=0}^{N-1} \frac{4W}{1 + [(i + \frac{1}{2})W]^2}$$

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

int 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);
    return 0;
}
```

Με νήματα, βελτιστοποιημένο

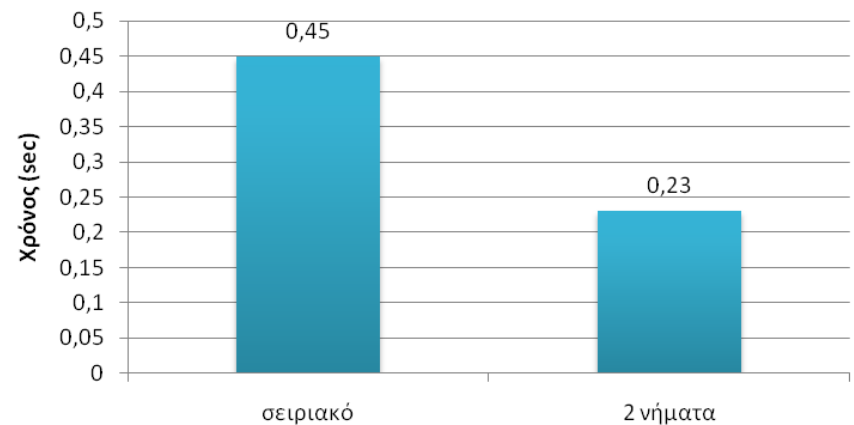
```
#define NPROCS 2          /* dual core */
#define N 10000000        /* Για ακρίβεια (ίδια με σειριακό) */
#define WORK N/NPROCS
double pi = 0.0, W = 1.0/N;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

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);
}

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

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

dual core με 2 νήματα



Με OpenMP

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

int 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);
    return 0;
}
```

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

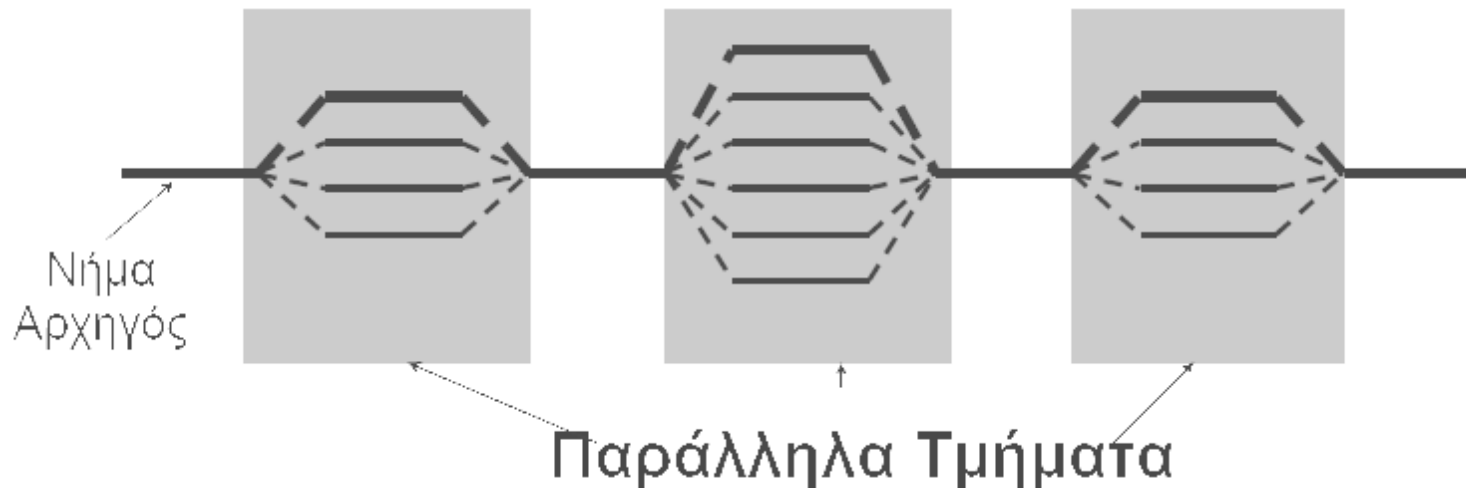
int main () {
    int i;
    #pragma omp parallel for reduction(+:sum)
    for (i=0; i < N; i++)
        pi += 4*W / (1 + (i+0.5)*(i+0.5)*W*W);
    printf("pi = %.10lf\n", pi);
    return 0;
}
```

- ❖ OpenMP: API για τη συγγραφή πολυνηματικών εφαρμογών
 - Σύνολο **οδηγιών** προς τον μεταγλωττιστή και **συναρτήσεων** βιβλιοθήκης, διαθέσιμο στον προγραμματιστή παράλληλων συστημάτων
 - Διευκολύνει τη συγγραφή πολυνηματικών προγραμμάτων σε Fortran, C and C++, **χωρίς να τροποποιεί** τη βασική γλώσσα
 - Πρότυπο που συγκεντρώνει την εμπειρία αρκετών χρόνων σε προγραμματισμό πολυεπεξεργαστικών συστημάτων
- ❖ Εδώ και > 15 χρόνια υποστήριξη από μεγάλες εταιρείες / οργανισμούς:
 - Intel, SUN/ORACLE, IBM, HP, SGI, ...
 - GNU GCC >= 4.2
- ❖ Επίσης, 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.
- Σε κάθε περιοχή του κώδικα που χρειάζεται:
 1. Δημιουργεί νήματα
 2. Συμμετέχει στους υπολογισμούς
 3. Περιμένει τον τερματισμό όλων των νημάτων της ομάδας
- Ο παραλληλισμός προστίθεται βαθμιαία
 - ❖ Το ακολουθιακό πρόγραμμα εξελίσσεται σε παράλληλο πρόγραμμα



❖ Σημαντικός στόχος του OpenMP αποτελεί η εύκολη παραλληλοποίηση βρόχων επανάληψης:

- Βρες τα πιο χρονοβόρα loops.
- Μοίρασε τις επαναλήψεις μεταξύ νημάτων.

Διαίρεση loop μεταξύ πολλαπλών νημάτων

```
int main() {  
    double Res[1000];  
  
    for (int i=0;i<1000;i++) {  
        do_huge_comp(Res[i]);  
    }  
    ...  
}
```

Ακολουθιακό πρόγραμμα

```
int main() {  
    double Res[1000];  
    #pragma omp parallel for  
    for (int i=0;i<1000;i++) {  
        do_huge_comp(Res[i]);  
    }  
    ...  
}
```

Παράλληλο πρόγραμμα

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

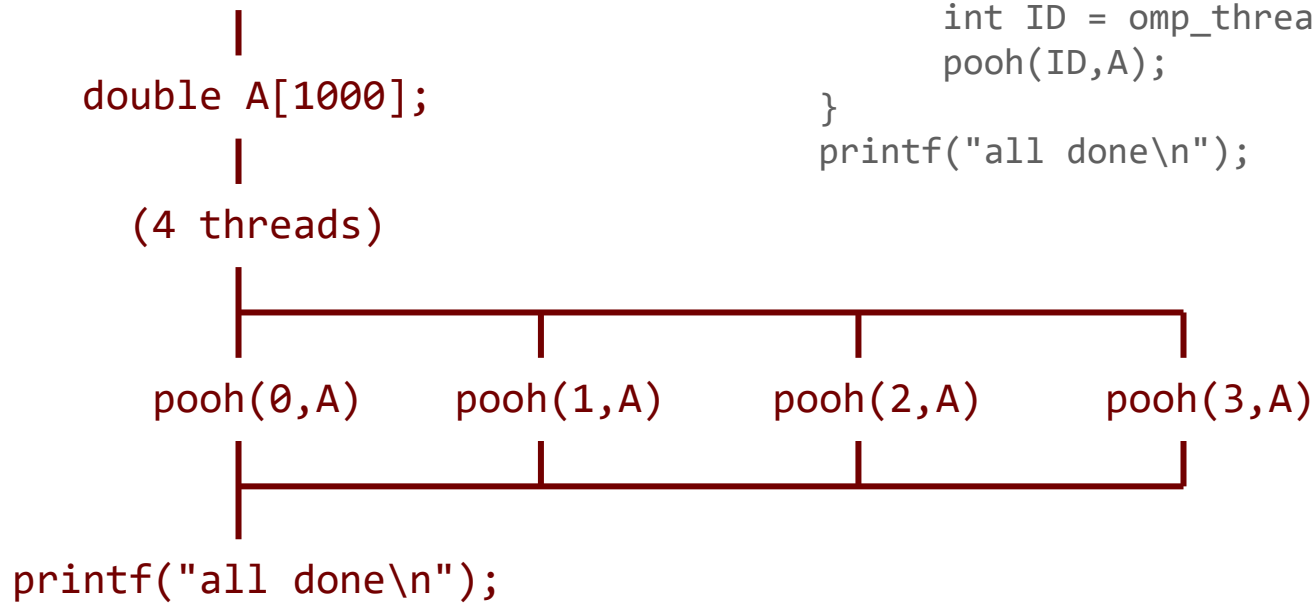
- ❖ Νήματα δημιουργούνται στο OpenMP (στη C/C++) με την οδηγία `omp parallel`.
- ❖ Για παράδειγμα, για να δημιουργηθεί ένα παράλληλο τμήμα με 4 νήματα:

```
double A[1000];  
#pragma omp parallel num_threads(4)  
{  
    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 (κοινόχρηστο) μεταξύ των νημάτων
- ❖ Η εκτέλεση συνεχίζεται μόνο όταν έχουν τελειώσει όλα τα νήματα (barrier)



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

❖ Dynamic mode (default):

- Ο αριθμός των νημάτων που χρησιμοποιούνται για την εκτέλεση παράλληλων τμημάτων μπορεί να διαφέρει μεταξύ διαφορετικών τμημάτων
- Ο ορισμός του αριθμού των νημάτων (`omp_set_num_threads()`) αφορά στον μέγιστο αριθμό νημάτων και ενδεχομένως η εκτέλεση να γίνει με λιγότερα νήματα

❖ Non-dynamic mode:

- Ο αριθμός των νημάτων είναι ακριβώς αυτός που καθορίζεται από τον προγραμματιστή (δεν επιτρέπεται στον μεταφραστή να «παίξει»)

❖ Το OpenMP υποστηρίζει εμφωλευμένα παράλληλα τμήματα, όμως...

- Ένας compiler ενδεχομένως να επιλέξει να εκτελέσει σειριακά όλα τα επίπεδα μετά το 1ο

Οδηγίες Διαμοίρασης Έργου (workshare directives)

- ❖ Ακολουθιακός κώδικας

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

- ❖ Παραλληλοποιημένος με OpenMP, τμηματική δρομολόγηση:

```
#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. Υποχρεωτικά ακολουθεί βρόχος for.

```
#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** στην οδηγία **omp for**.

Οδηγίες Διαμοίρασης Έργου: sections

- ❖ Η δομή διαμοίρασης έργου `omp sections` αναθέτει ένα διαφορετικό δομημένο block σε κάθε νήμα της ομάδας

```
#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`

- ❖ Η οδηγία διαμοίρασης έργου `omp single` αναθέτει τον κώδικα που ακολουθεί σε ένα και μοναδικό νήμα της ομάδας

```
#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`”

- ❖ Για διευκόλυνση, χρήση ενός pragma αντί δύο.
- ❖ Συνδυάζεται η οδηγία omp parallel με δομές διαμοίρασης έργου omp for ή omp sections.
- ❖ Παράδειγμα:

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

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

Υπολογισμός του π (μέχρι τώρα)

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

int 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);
    return 0;
}
```

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

int 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);
    return 0;
}
```

- ❖ Οι global μεταβλητές είναι κοινόχρηστες μεταξύ των νημάτων
- ❖ Όμως δεν είναι τα πάντα κοινά...
 - Οι μεταβλητές στη στοίβα του κάθε νήματος είναι ιδιωτικές
 - Οι μεταβλητές που ορίζονται μέσα σε ένα δομημένο block εντολών είναι ιδιωτικές.
- ❖ Το OpenMP επιτρέπει την αλλαγή των *χαρακτηριστικών κοινοχρησίας (sharing attributes)* των μεταβλητών

- ❖ Τα χαρακτηριστικά κοινοχρησίας των μεταβλητών καθορίζονται είτε *έμμεσα* (implicitly) είτε *άμεσα* (explicitly)
- ❖ Έμμεσα καθορίζονται ανάλογα με το πώς ορίζονται / χρησιμοποιούνται.
- ❖ Για παράδειγμα, στις παράλληλες περιοχές:
 - όποια μεταβλητή είναι ορισμένη **πριν** από μία παράλληλη περιοχή και χρησιμοποιείται μέσα σε αυτήν, θεωρείται αυτόματα ως κοινόχρηστη μεταξύ των νημάτων της ομάδας, **ακόμα και αν δεν είναι global**.
- ❖ Ο προγραμματιστής μπορεί, όμως, να παρέμβει και να καθορίσει άμεσα την κοινοχρησία των μεταβλητών με ειδικές *φράσεις (clauses)*.

Φράσεις καθορισμού χαρακτηριστικών κοινοχρησίας (I)

- ❖ Ο προγραμματιστής μπορεί να καθορίσει ρητά τα χαρακτηριστικά αποθήκευσης των μεταβλητών χρησιμοποιώντας μία από τις ακόλουθες φράσεις

`shared(<μεταβλητές>)`

`private(<μεταβλητές>)`

`firstprivate(<μεταβλητές>)`

- ❖ Η τιμή μιας ιδιωτικής μεταβλητής εντός μίας περιοχής διαμοίρασης έργου μπορεί να «μεταδοθεί» εκτός της περιοχής με την:

`lastprivate(<μεταβλητές>)`

- ❖ 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).

shared(x,y)

- Οι μεταβλητές x, y θα είναι κοινόχρηστες στην παράλληλη περιοχή που ακολουθεί.

private(x,y)

- Οι μεταβλητές x, y θα είναι ιδιωτικές για κάθε νήμα στην περιοχή που ακολουθεί.

firstprivate(x,y)

- Οι μεταβλητές x, y θα είναι ιδιωτικές για κάθε νήμα στην περιοχή που ακολουθεί, αλλά θα αρχικοποιηθούν με την τιμή που έχει η αντίστοιχη αρχική (original) μεταβλητή.

lastprivate(x,y)

- Οι μεταβλητές x, y θα είναι ιδιωτικές για κάθε νήμα στην περιοχή που ακολουθεί, η οποία είναι υποχρεωτικά *omp for* ή *omp sections*. Στο τέλος της περιοχής, η αντίστοιχη αρχική (original) μεταβλητή θα τροποποιηθεί από το νήμα που θα εκτελέσει την τελευταία επανάληψη του loop (*omp for*) ή το τελευταίο section (*omp sections*).

threadprivate(x,y)

- Δεν είναι φράση, αλλά αυτόνομη οδηγία που τοποθετείται κοντά στο σημείο που δηλώνονται οι μεταβλητές.
- Οι μεταβλητές x, y θα είναι ιδιωτικές σε κάθε νήμα που θα δημιουργηθεί (ορίζεται μόνο μία φορά, στο σημείο που γίνεται η δήλωση των global μεταβλητών x και y).

Παράδειγμα φράσεων 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
 - Το κάθε νήμα θα τύπωνε: **2, <τυχαία τιμή>, 2**
- ❖ Τι θα τυπωθεί μετά το παράλληλο τμήμα?
 - **2, 1, 1**

Αν είχαμε

```
#pragma omp single nowait  
τι θα τυπωνόταν;
```


- ❖ Κανονικά, όλες οι μεταβλητές πρέπει να προσδιοριστούν μέσα από κάποια φράση (*shared*, *private*, *firstprivate* κλπ) προκειμένου να καταλάβει ο μεταφραστής πώς να τις χειριστεί
- ❖ Το OpenMP επιτρέπει να ΜΗΝ προσδιοριστούν (έμμεσος καθορισμός χαρακτηριστικών κοινοχρησίας).
 - Όλες αυτές οι μεταβλητές θεωρούνται *shared*
- ❖ Μπορεί να αλλάξει ο έμμεσος καθορισμός με τη φράση *default()*.
 - *default(shared)*: όσες δεν ορίζονται ρητά, θεωρούνται *shared*.
 - *default(private)*: όσες δεν ορίζονται ρητά, θεωρούνται *private*.
 - *default(none)*: όσες δεν ορίζονται ρητά, προκαλούν συντακτικό λάθος κατά τη μετάφραση – αναγκάζει τον προγραμματιστή να ορίσει ρητά τα πάντα.

Παράδειγμα

Είναι σωστός ο υπολογισμός του π ;

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

int 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);
    return 0;
}
```

Σωστές επιλογές:

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

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

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

- ❖ Μετατρέπει τα κοινόχρηστα global δεδομένα σε ιδιωτικά για κάθε νήμα
 - 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 τη φράση copyin().

- ❖ 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 ιδιωτικό αντίγραφο κάθε μεταβλητής της λίστας και αρχικοποιείται (ανάλογα με την πράξη “op” π.χ. 0 για “+”)
 - Τα ιδιωτικά αντίγραφα συνδυάζονται και δίνουν την τελική τιμή στην αντίστοιχη (κοινόχρηστη) αρχική μεταβλητή στο τέλος της δομής.

Υπολογισμός του π

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

int 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);
    return 0;
}
```

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

int 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);
    return 0;
}
```

Υπολογισμός του π

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

int 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);
    return 0;
```

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

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

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

int 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);
    return 0;
```

Υπολογισμός του π

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

int 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);
    return 0;
}
```

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

int 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);
    return 0;
}
```

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

int 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);
    return 0;
}
```

- ❖ Αποτελεί βασικό παράγοντα απόδοσης
- ❖ Για συνηθισμένες λειτουργίες, π.χ. πρόσθεση διανυσμάτων, η εξισορρόπηση εργασίας δεν αποτελείται 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

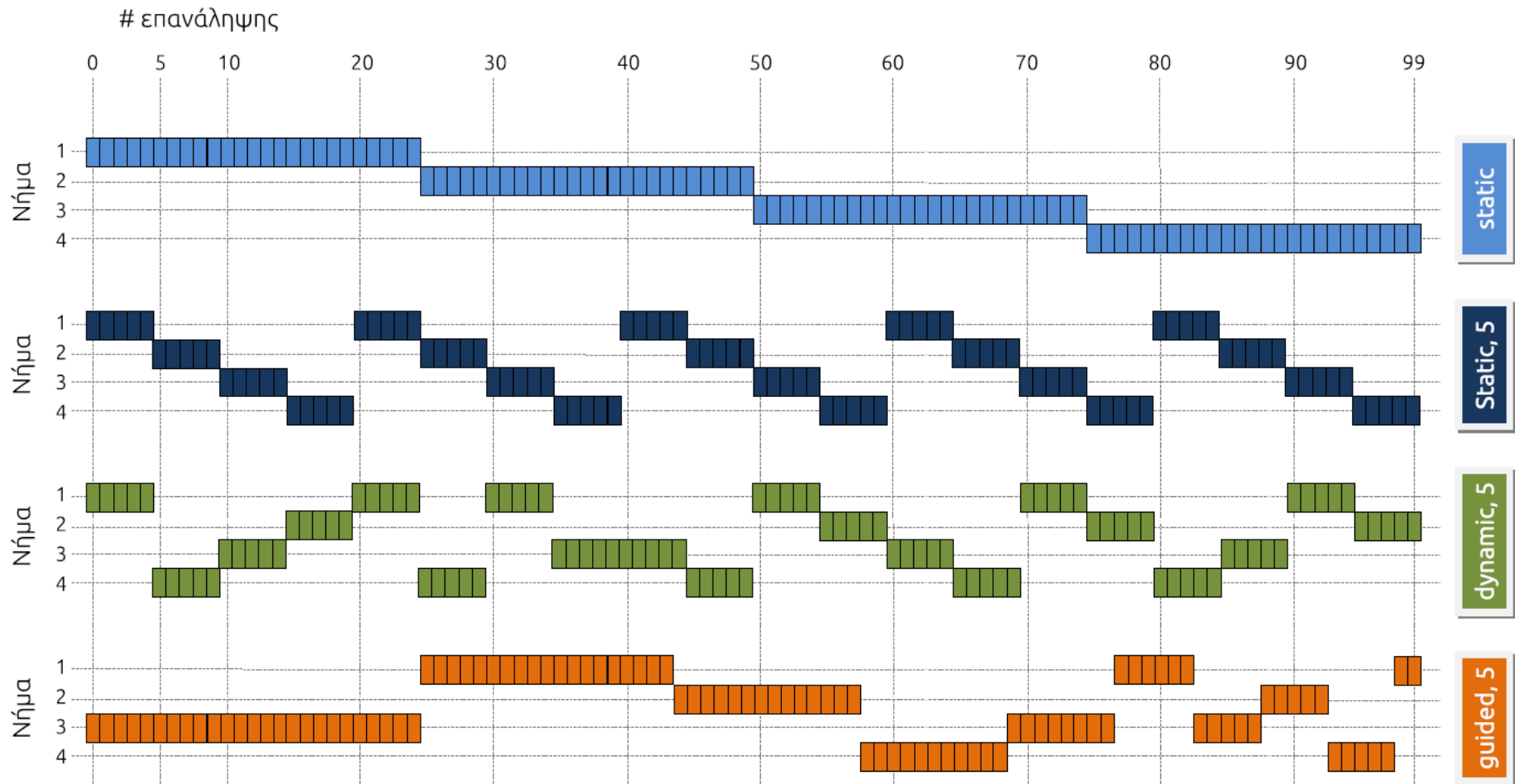
❖ `guided [,chunk]`

- Παρόμοια με `dynamic`, αλλά το μέγεθος του τμήματος μειώνεται εκθετικά
- σε κάθε βήμα το μέγεθος του τμήματος είναι *ανάλογο* του $(\# \text{ unassigned iterations} / \# \text{ threads})$, αλλά όχι λιγότερο από `chunk`.

❖ `runtime`

- Ο αλγόριθμος δρομολόγησης καθορίζεται κατά τον χρόνο εκτέλεσης ελέγχοντας τη μεταβλητή περιβάλλοντος `OMP_SCHEDULE`

Παράδειγμα



❖ Το OpenMP ορίζει τις ακόλουθες οδηγίες:

➤ `atomic`

- ✧ Για ατομική (αδιαίρετη) πράξη σε μία μεταβλητή (π.χ. αύξηση)

➤ `critical`

- ✧ Για ορισμό κρίσιμων περιοχών

➤ `barrier`

- ✧ Για συγχρονισμό νημάτων

➤ `flush`

- ✧ Για συνέπεια μνήμης (το μοντέλο του OpenMP υποθέτει εξαιρετικά χαλαρή συνέπεια)

➤ `master`

- ✧ Για εκτέλεση κώδικα από τον αρχηγό μίας ομάδας νημάτων (σαν το `single`, μόνο που πάντα το εκτελεί το νήμα 0 και **δεν** υπονοεί `barrier` στο τέλος)

- ❖ 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` είναι τα ονόματα των μεταβλητών που θα πρέπει να γίνουν `flush`. Αν δεν υπάρχουν ορίσματα όλες οι ορατές στο νήμα μεταβλητές γίνονται `flush`.
- ❖ Πρόκειται για `memory fence` που επιβάλλει συνέπεια μνήμης

❖ **Barrier υπονοείται** αμέσως μετά το τέλος των παρακάτω οδηγιών:

- `parallel`
- `for` (εκτός αν είχε δοθεί η φράση `nowait`)
- `sections` (εκτός αν είχε δοθεί η φράση `nowait`)
- `single` (εκτός αν είχε δοθεί η φράση `nowait`)
- `critical`

❖ **Flush υπονοείται** αμέσως μετά το τέλος των παρακάτω οδηγιών:

- `parallel`
- `for`
- `sections`
- `single`
- `critical`
- `barrier`
- `ordered`

❖ 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()`

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

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

❖ Κώδικας OpenMP

```
#include <omp.h>
#include <stdio.h>
int main() {
    #pragma omp parallel
    {
        printf("Hello world from thread %d of %d\n",
               omp_get_thread_num(), omp_get_num_threads());
    }
    return 0;
}
```

❖ Παραγωγή εκτελέσιμου αρχείου για OMPI, GNU GCC 4.2, Intel Compiler)

```
$ ompicc -o hello hello.c
$ gcc -fopenmp -o hello hello.c
$ icc -openmp -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
```

Παράδειγμα 1 (τι θα τυπώσει;)

```
#include <stdio.h>
#include <omp.h>

int main()
{
    int x = 2;
    #pragma omp parallel num_threads(2) shared(x)
    {
        if (omp_get_thread_num() == 0) {
            x = 5;
        } else {
            printf("1: Thread# %d: x = %d\n", omp_get_thread_num(), x );
        }

        #pragma omp barrier

        if (omp_get_thread_num() == 0) {
            printf("2: Thread# %d: x = %d\n", omp_get_thread_num(), x );
        } else {
            printf("3: Thread# %d: x = %d\n", omp_get_thread_num(), x );
        }
    }
    return 0;
}
```

Παράδειγμα 2

```
#include <omp.h>

int main()
{
    omp_set_dynamic(1);
    #pragma omp parallel num_threads(10)
    {
        /* do work here - at most 10 threads */
    }
    return 0;
}
```



Παράδειγμα 3

```
void work(int i, int j) {}

void good_nesting(int n)
{
    int i, j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
        for (i=0; i<n; i++) {
            #pragma omp parallel shared(i, n)
            {
                #pragma omp for
                for (j=0; j < n; j++)
                    work(i, j);
            }
        }
    }
}
```

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;
}
```

ΧΡΟΝΟΣ: 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;
}
```

ΧΡΟΝΟΣ: 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;
}
```

ΧΡΟΝΟΣ: 40msec

(λάθος στην κοινοχρησία μεταβλητών)

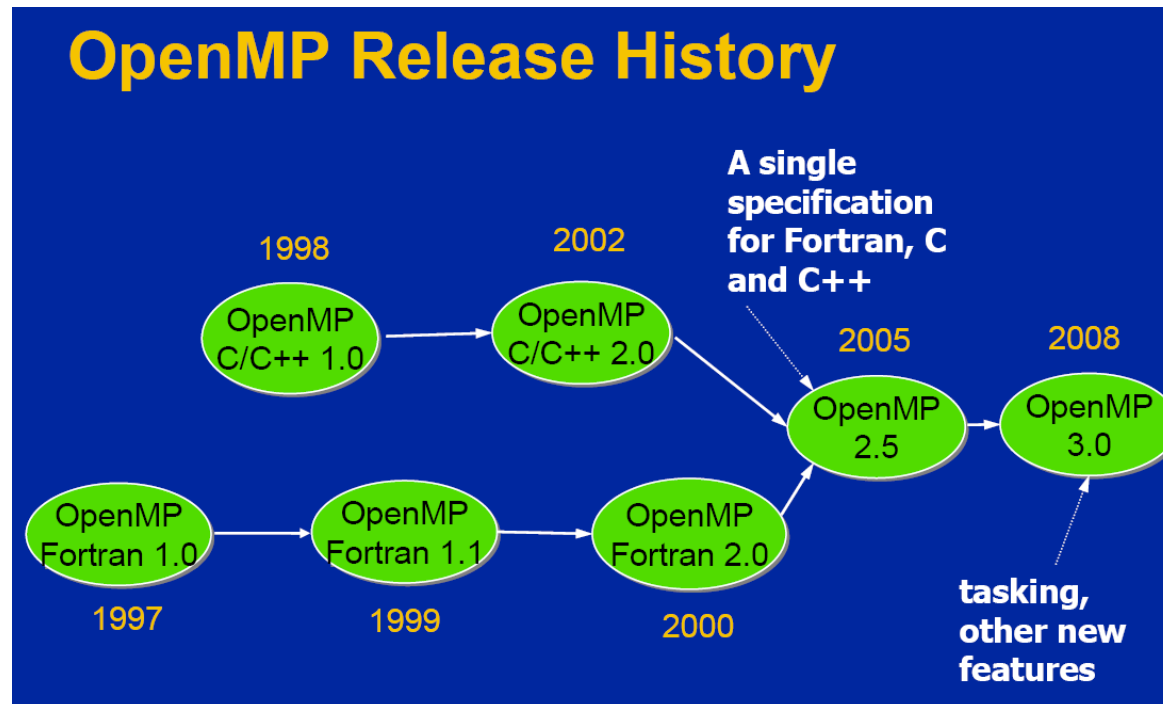
OpenMP tasks



MYE023

OpenMP 3.0

- ❖ Στο OpenMP 3.0 ξεκαθάρισαν 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**.



❖ 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 αρκετών 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...



```
.....  
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

```
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)**



❖ 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.

❖ 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)

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 )
```

❖ 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`

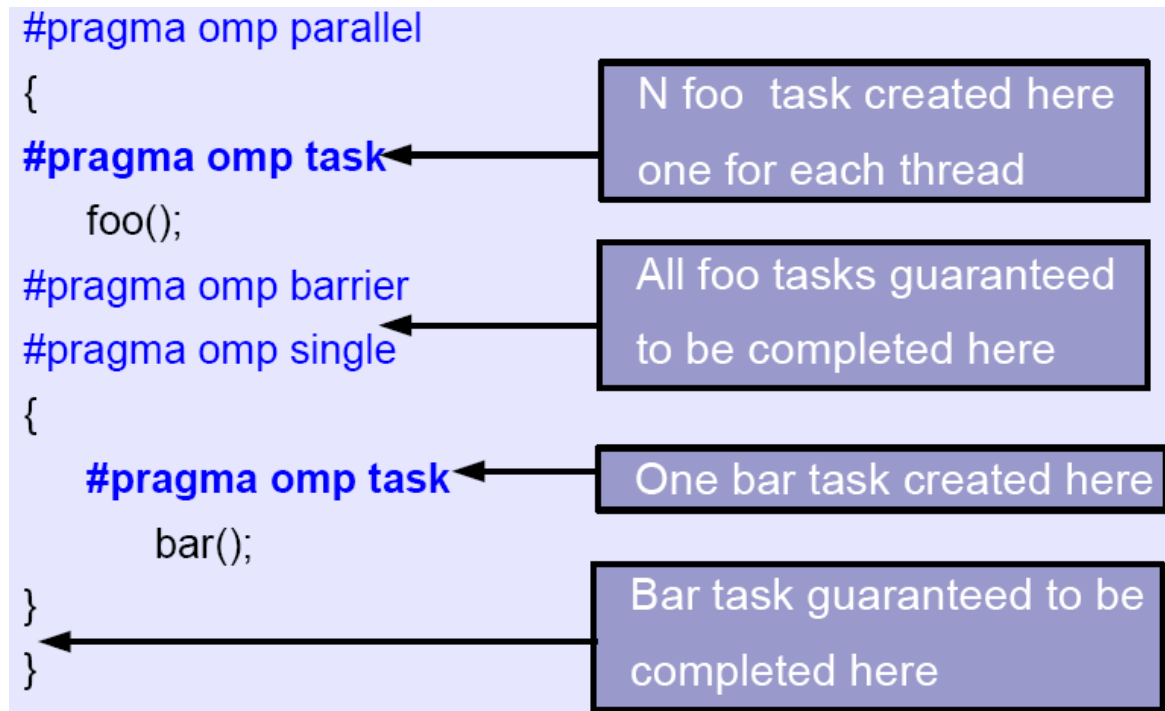
- Κανονικά, ένα `task` είναι «δεμένο» (“`tied`”) με το νήμα που θα ξεκινήσει να το εκτελεί – δηλαδή θα εκτελεστεί από το νήμα αυτό μέχρι τέλους
- Σε ένα ελεύθερο `task` (“`untied`”), η εκτέλεσή του μπορεί να διακόπτεται και να συνεχίζει την εκτέλεσή του άλλο νήμα, κλπ.

❖ 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



- ❖ 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 )
```



```
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;;  
}
```

Correct

x,y are shared

```
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



```
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);  
}
```

Right!

e is firstprivate



Μερικά ακόμα για το OpenMP



❖ GCC

- Υποστήριξη tasks από την έκδοση 4.3.x (όμως άσχημη υλοποίηση)
- Καλή υλοποίηση από έκδοση 4.4.x

❖ ICC

- Intel compiler & tools
- Εξαιρετικά εργαλεία, ταχύτατα σειριακά προγράμματα για x86
- Έχει εξαγοράσει την Cilk Arts

❖ SUNCC

- SUN C compiler (πλέον ORACLE)
- Γενικά καλές και σταθερές επιδόσεις, όχι όμως ο καλύτερες

❖ OMPI

- Ο «δικός» μας, μέσα από πτυχιακές, μεταπτυχιακές και διδακτορικές εργασίες
- Ανοιχτού κώδικα, πολύ καλές επιδόσεις, επεκτάσεις κ.α.
- <http://paragroup.cse.uoi.gr/>

❖ Άλλοι ερευνητικοί, ελεύθερου κώδικα:

- OpenUH
- Mercurium (Nanaos)

Εκδόσεις του OpenMP

