ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΜΗΧΑΝΙΚΗ ΜΑΘΗΣΗ

Παράλληλοι υπολογισμοί σε αρχιτεκτονικές κοινής μνήμης

Ακαδημαϊκό έτος 2019-20

Εισαγωγή

- Παράλληλοι υπολογισμοί και επίδοση
- Σχεδιασμός παράλληλων προγραμμάτων
- Παράλληλος προγραμματισμός σε κοινή μνήμη με OpenMP

Παράλληλοι υπολογισμοί και επίδοση

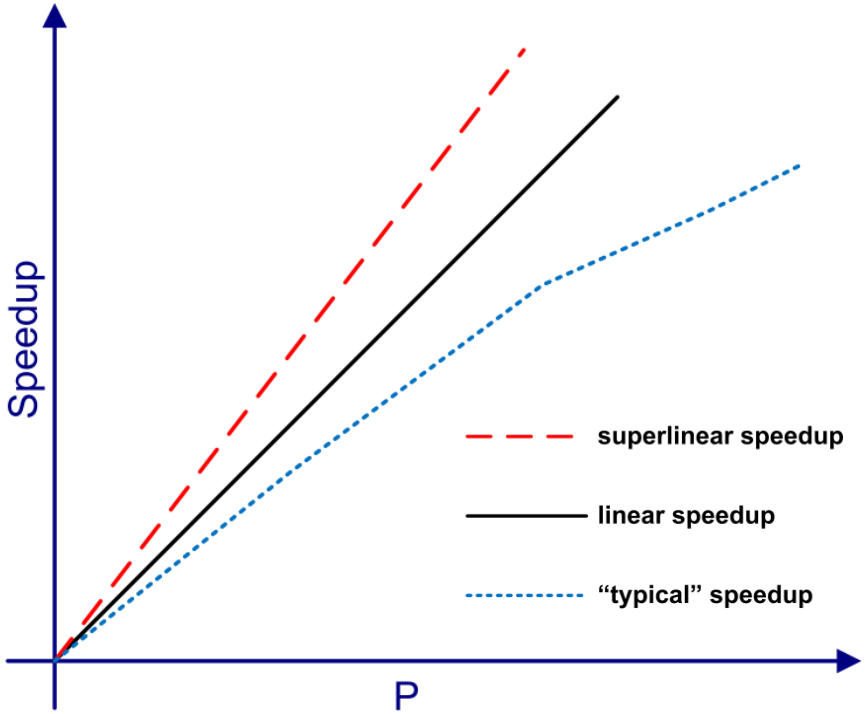
Επιτάχυνση (Speedup)

- Εκτέλεση σειριακού προγράμματος σε έναν πυρήνα
  - Χρόνος εκτέλεσης σειριακού προγράμματος: $\mathbf{T_s}$
- Εκτέλεση παράλληλης έκδοσης του προγράμματος σε p πυρήνες
  - Χρόνος εκτέλεσης παράλληλου προγράμματος: $\mathbf{T_p}$
- **Επιτάχυνση** (speedup) $\mathbf{S = T_s / T_p}$
  - Δείχνει πόσες φορές πιο γρήγορο είναι το παράλληλο πρόγραμμα από το σειριακό

- Ποιο είναι το μέγιστο speedup που μπορώ να πάρω σε p πυρήνες;

Επιτάχυνση (Speedup)

- **Επιτάχυνση** (speedup) **S** = $\mathbf{T_s}$ / $\mathbf{T_p}$
  - Δείχνει πόσες φορές πιο γρήγορο είναι το παράλληλο πρόγραμμα από το σειριακό
- Σε **p** επεξεργαστές, *ιδανικά* **S** = **p**
  - Γραμμικό (linear) speedup
- Σε **p** επεξεργαστές, *τυπικά* **S** ≤ **p**
  - Αν S > p → superlinear speedup
  - Αν προκύψει, πρέπει να το ερμηνεύσουμε προσεκτικά



Αποδοτικότητα (Efficiency)

- Επιτάχυνση (speedup) **S** = $\mathbf{T_s}$ / $\mathbf{T_p}$
  - Δείχνει πόσες φορές πιο γρήγορο είναι το παράλληλο πρόγραμμα από το σειριακό

- **Αποδοτικότητα** (efficiency) **E** = **S** / **p**
  - Δείχνει πόσο επιτυχημένη είναι η παραλληλοποίηση
  - Δείχνει τι ποσοστό του χρόνου κάθε πυρήνας κάνει χρήσιμη δουλειά

- Τυπικά: **E** ≤ **1**

Νόμος του Amdahl

- Επιτάχυνση **S** = $\mathbf{T_s}$ / $\mathbf{T_p}$ **S** -> *τυπικά* **S ≤ p**
- Αποδοτικότητα **E** = **S** / **p** -> *τυπικά* **E ≤ 1**

- Υπάρχει «πάντα» ένα μέρος του προγράμματος που δεν παραλληλοποιείται

Νόμος του Amdahl

- Σειριακό πρόγραμμα

```
data = read_from_file()
result = compute(data)
write_to_file(result)
```

- Παράλληλο πρόγραμμα

```
data = read_from_file()
result = PARALLEL_compute(data)
write_to_file(result)
```

- 1 πυρήνας

- 2 πυρήνες

- 4 πυρήνες

Νόμος του Amdahl

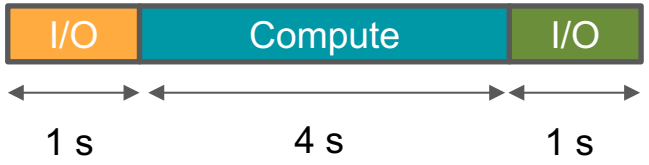
- Χρόνος καλύτερου σειριακού προγράμματος: $\mathbf{T_s}$
- **f** το κλάσμα του χρόνου ενός σειριακού προγράμματος που δεν παραλληλοποιείται

- **Νόμος Amdahl**

$$T_p = fT_s + \frac{(1-f)T_s}{p}$$

$$S = \frac{T_s}{T_p} = \frac{1}{f + \frac{(1-f)}{p}}$$

Νόμος του Amdahl



- Χρόνος σειριακού προγράμματος
  $\mathbf{T_s}$ = 1 + 4 + 1 = 6 s

- Ποσοστό που δεν παραλληλοποιείται:
  f = (1 + 1) / 6 = 1/3

- 2 πυρήνες
  $\mathbf{T_p}$ = **f** * $\mathbf{T_s}$ + ((1-**f**) * $\mathbf{T_s}$)/p = 4 s
  S = $\mathbf{T_s}$ / $\mathbf{T_p}$ =1.5, **E** = **S** / **p** = 0.75

- 4 πυρήνες
  $\mathbf{T_p}$ = 3 s
  S = 2, **E** = 0.5

Νόμος του Amdahl

- Συνέπειες του νόμου του Amdahl:
  - Παραλληλοποιούμε (και γενικά βελτιστοποιούμε) τμήματα του κώδικα που καταλαμβάνουν **το μεγαλύτερο ποσοστό** του χρόνου εκτέλεσης
  - Αναζητούμε παραλληλία **παντού**! (π.χ. 1% σειριακός κώδικας → μέγιστο speedup 100!)

Γιατί δεν κλιμακώνει το πρόγραμμά μου;

1. Δεν έχει παραλληλοποιηθεί το κατάλληλο τμήμα κώδικα
   - Υπάρχει μεγάλο σειριακό μέρος
2. Κόστος συγχρονισμού / επικοινωνίας
   - Μπορεί να οδηγήσει ακόμα και σε αύξηση του χρόνου εκτέλεσης
3. Συμφόρηση στο διάδρομο μνήμης (για αρχιτεκτονικές κοινής μνήμης)
4. Ανισοκατανομή φορτίου (load imbalance)
5. Κόστος από την παραλληλοποίηση (επιπλέον εργασία στον παράλληλο αλγόριθμο)
   - Επιπλέον λειτουργίες κατανομής εργασίας (στατικά ή δυναμικά)
   - Διαχείριση οντοτήτων εκτέλεσης (νήματα, διεργασίες, κλπ)
   - Επιλογή

Ο νόμος του Amdahl βοηθά στην παραπάνω ανάλυση!

Σχεδιασμός παράλληλων προγραμμάτων

Σχεδιασμός παράλληλων προγραμμάτων

- **Στόχος:** Η μετατροπή ενός σειριακού αλγορίθμου σε παράλληλο
  1. Κατανομή υπολογισμών σε υπολογιστικές εργασίες (tasks)
  2. Ορισμός ορθής σειράς εκτέλεσης (χρονοδρομολόγηση)
  3. Διαμοιρασμός των δεδομένων - Οργάνωση πρόσβασης στα δεδομένα (συγχρονισμός / επικοινωνία)
  4. Ανάθεση εργασιών (απεικόνιση) σε οντότητες εκτέλεσης (processes, threads)
- Δεν υπάρχει αυστηρά ορισμένη μεθοδολογία για το σχεδιασμό και την υλοποίηση παράλληλων προγραμμάτων
- Ο σχεδιασμός επηρεάζεται από την αρχιτεκτονική και το προγραμματιστικό μοντέλο

Στάδιο 1: Κατανομή υπολογισμών

- **Στόχος:** κατανομή σε υπολογιστικές εργασίες (tasks)

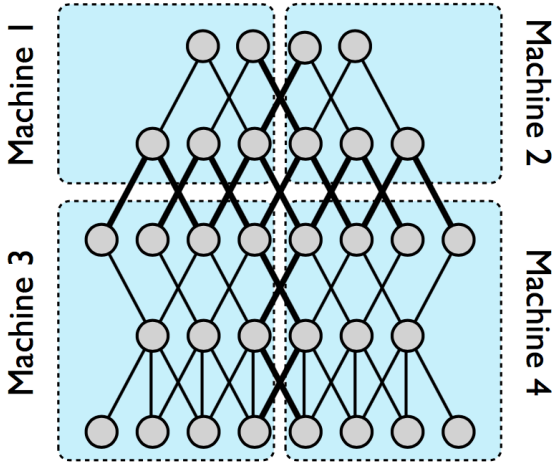
- 3 βασικές προσεγγίσεις:
  - **Data-centric:** Μοιράζονται τα δεδομένα σε μονάδες επεξεργασίας και στη συνέχεια οι υπολογισμοί
  - **Task-centric:** Μοιράζονται οι υπολογισμοί σε μονάδες επεξεργασίες και στη συνέχεια τα δεδομένα
  - **Function-centric:** Μοιράζονται διαφορετικές λειτουργίες/στάδια σε διαφορετικές μονάδες επεξεργασίας
- Σε κάθε περίπτωση προκύπτουν υπολογιστικές εργασίες (tasks)
- Επιλογή στρατηγικής ανάλογα με τον αλγόριθμο!

Στάδιο 1: Κατανομή υπολογισμών

- Παράδειγμα: Παραλληλοποίηση της παρασκευής Ν ταψιών πίτσας
  - «Πρόγραμμα»: Εντολές (συνταγή) + Δεδομένα (υλικά)
  - Μονάδες επεξεργασίας: Μάγειρες
- **task-centric** προσέγγιση: μοιράζω τη δουλειά
  - 1 task = παρασκευή 1 ταψιού πίτσας
- **data-centric** προσέγγιση: μοιράζω τα υλικά
  - η κατανομή της δουλειάς προκύπτει σαν φυσικό επακόλουθο του διαμοιρασμού των δεδομένων
  - μοιράζω τα υλικά σε τμήματα (π.χ. σε Κ μέρη, ή ανά μάγειρα, ή ανά πίτσα)
- **function-centric** προσέγγιση: διαχωρίζω τις διαφορετικές δουλειές
  - παρασκευή ζύμης, παρασκευή σάλτσας, προετοιμασία λοιπών υλικών, άπλωμα ζύμης, τοποθέτηση σάλτσας, ...

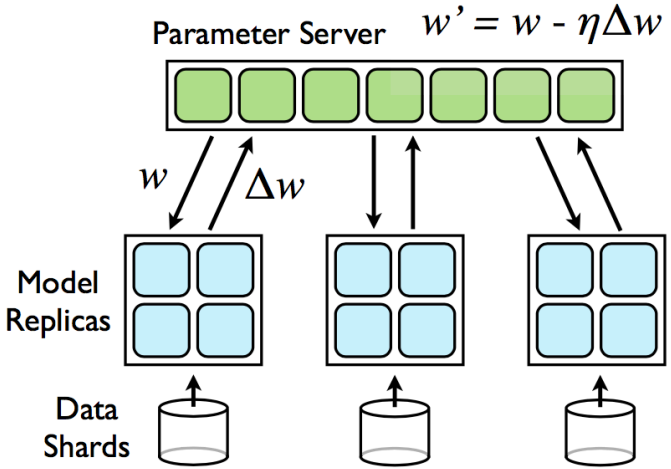
Παράδειγμα: παραλληλισμός στην εκπαίδευση νευρωνικών δικτύων

- **Task-centric**: παραλληλισμός σε επίπεδο μοντέλου
  - Κάθε μονάδα επεξεργασίας αναλαμβάνει να εκπαιδεύσει ένα υποσύνολο των παραμέτρων του μοντέλου με όλα τα δεδομένα εισόδου



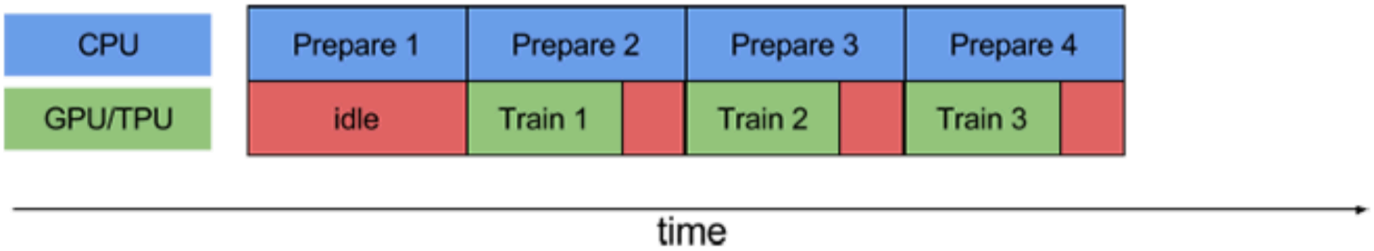
Παράδειγμα: παραλληλισμός στην εκπαίδευση νευρωνικών δικτύων

- **Data-centric**: παραλληλισμός σε επίπεδο δεδομένων εισόδου
  - Κάθε μονάδα επεξεργασίας διατηρεί ένα αντίγραφο του μοντέλου και εκπαιδεύει τις παραμέτρους με ένα υποσύνολο των δεδομένων εισόδου



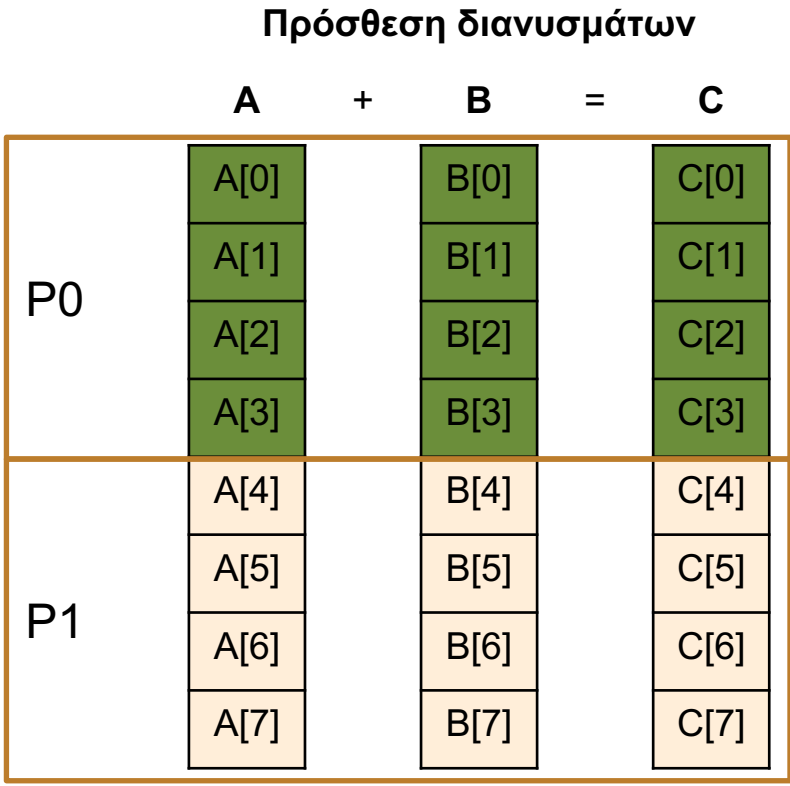
Παράδειγμα: παραλληλισμός στην εκπαίδευση νευρωνικών δικτύων

- **Function-centric**: παραλληλισμός σε επίπεδο δεδομένων λειτουργιών
  - Μία μονάδα επεξεργασίας αναλαμβάνει την προ-επεξεργασία (pre-processing) των δεδομένων και μια άλλη μονάδα επεξεργασίας αναλαμβάνει την εκπαίδευση του μοντέλου με τα δεδομένα που προκύπτουν από την προ-επεξεργασία



Rule of thumb

- Όταν το πρόβλημα είναι “embarrassingly parallel” (π.χ. συμπίεση Ν αρχείων, επίλυση Ν ανεξάρτητων συστημάτων) η data centric προσέγγιση είναι προφανής.
  - Οδηγεί σε data parallel υλοποιήσεις
- Αλγόριθμοι σε κανονικές δομές δεδομένων (π.χ. regular computational grids, αλγεβρικοί πίνακες) συχνά ευνοούν τη **data-centric** κατανομή:
  - Πολλαπλασιασμός πινάκων (γενικά βασικές αλγεβρικές ρουτίνες)
  - Επίλυση γραμμικών συστημάτων



Στάδιο 2: Ορισμός ορθής σειράς εκτέλεσης

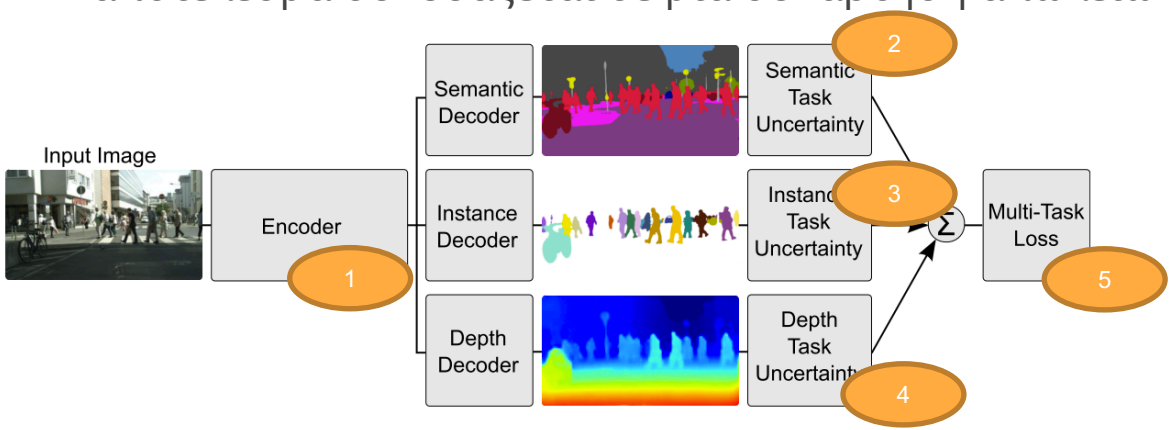
- Οι εργασίες που ορίστηκαν στο προηγούμενο στάδιο πρέπει να μπουν στη σωστή σειρά ώστε να εξασφαλίζεται η ίδια σημασιολογία με το σειριακό πρόγραμμα
- Το στάδιο αυτό συχνά αναφέρεται και ως **χρονοδρομολόγηση** = ανάθεση εργασιών σε χρονικές στιγμές
- Απαιτείται **εντοπισμός των εξαρτήσεων** ανάμεσα στις εργασίες και κατάστρωση του γράφου των εξαρτήσεων (task dependence graph)

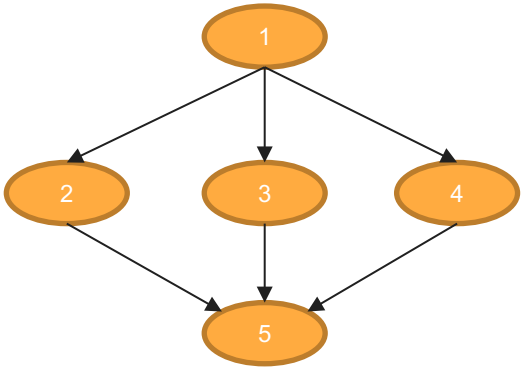
Εξαρτήσεις

- Εξαρτήσεις δεδομένων υπάρχουν ήδη από το σειριακό πρόγραμμα
- Εξάρτηση υπάρχει όταν 2 εντολές αναφέρονται στα ίδια δεδομένα (θέση μνήμης)
- 4 είδη εξαρτήσεων
  - Read-After-Write (RAW) ή true dependence
  - Write-After-Read (WAR) ή anti dependence
  - Write-After-Write (WAW) ή output dependence
  - Read-After-Read (not really a dependence)
- Η παράλληλη εκτέλεση πρέπει να σεβαστεί τις εξαρτήσεις του προβλήματος
- Η κατανομή των tasks δημιουργεί κατά κανόνα εξαρτήσεις ανάμεσα στα tasks
  - Παράδειγμα: το task1 πρέπει να διαβάσει δεδομένα που παράγει το task2
- Διατήρηση των εξαρτήσεων: σειριοποίηση μεταξύ tasks

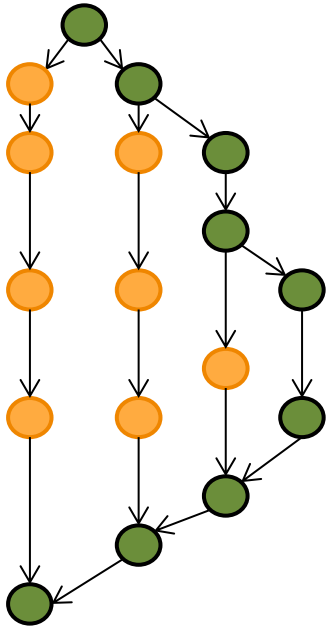
Γράφος εξαρτήσεων: παράδειγμα στην εκπαίδευση νευρωνικών δικτύων

- **Γράφος εξαρτήσεων:** multi-task learning – για την ίδια εικόνα, υπολογίζεται η αβεβαιότητα για διαφορετικές εργασίες κατηγοριοποίησης (classification) και το αποτέλεσμα συνδυάζεται σε μία συνάρτηση απωλειών





Γράφοι εξαρτήσεων εργασιών: ιδιότητες



- **T_1:** Συνολική εργασία **(work)**, ο χρόνος που απαιτείται για την εκτέλεση σε 1 επεξεργαστή
- **T_p:** Χρόνος εκτέλεσης σε ***p*** επεξεργαστές
- Κρίσιμο μονοπάτι (critical path): Το μέγιστο μονοπάτι ανάμεσα στην πηγή και τον προορισμό του γράφου
- **T_∞ :** Χρόνος εκτέλεσης σε **∞** επεξεργαστές **(span)** και χρόνος εκτέλεσης του κρίσιμου μονοπατιού
- Ισχύει:
  - **$T_p \geq T_1 / p$**
  - **$T_p \geq T_\infty$**
  - **Μέγιστο speedup T_1 / T_∞**

Στάδιο 3: Οργάνωση πρόσβασης στα δεδομένα

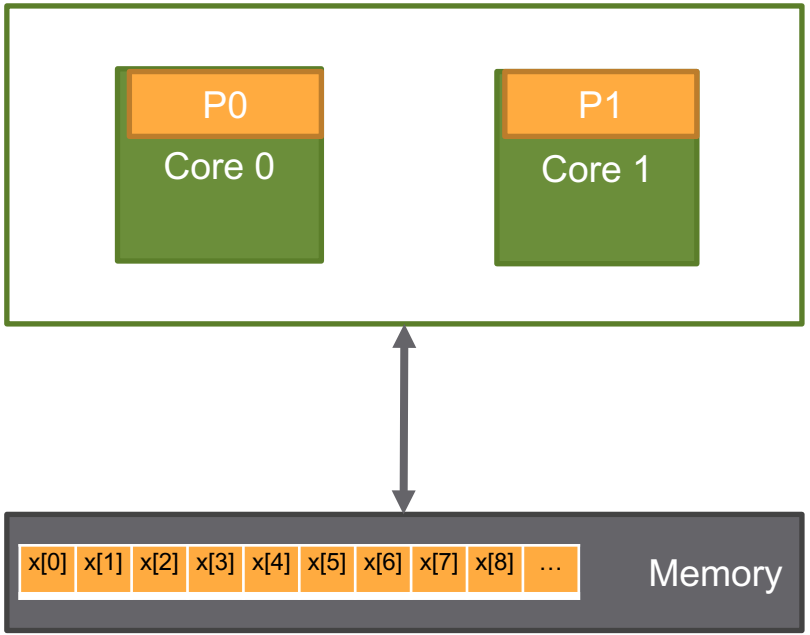
- Τα δεδομένα του προβλήματος μπορούν να είναι:
  - **Μοιραζόμενα** (shared data):
    - Μπορεί να υποστηριχθεί μόνο σε «κοινή μνήμη»
    - Αποτελεί ένα πολύ βολικό τρόπο «κατανομής»
    - Χρειάζεται ιδιαίτερη προσοχή για τον εντοπισμό race conditions
  - **Κατανεμημένα** (distributed data):
    - Τα δεδομένα κατανέμονται ανάμεσα στα tasks
    - Αποτελεί τη βασική προσέγγιση στα προγραμματιστικά μοντέλα ανταλλαγής μηνυμάτων
    - Επιβάλλει επιπλέον προγραμματιστικό κόστος
  - **Αντιγραμμένα** (replicated data):
    - Για αντιγραφή μικρών read-only δομών δεδομένων
    - Για αντιγραφή υπολογισμών

Κοινή μνήμη
και μοιραζόμενα δεδομένα

- Παράδειγμα μοιραζόμενων δεδομένων:

```
for (i = 0 ; i < 100 ; i++)
        x[i] = f(x[i])
```

- Εκτελείται από δύο νήματα/διεργασίες (processes)
- Κάθε μία εκτελείται σε διαφορετικό πυρήνα
- Μοιράζονται τον πίνακα x – βρίσκεται στην κοινή κύρια μνήμη
- Κάθε διεργασία θα εκτελέσει κάποια βήματα του for loop – κάθε βήμα του for loop είναι μια εργασία

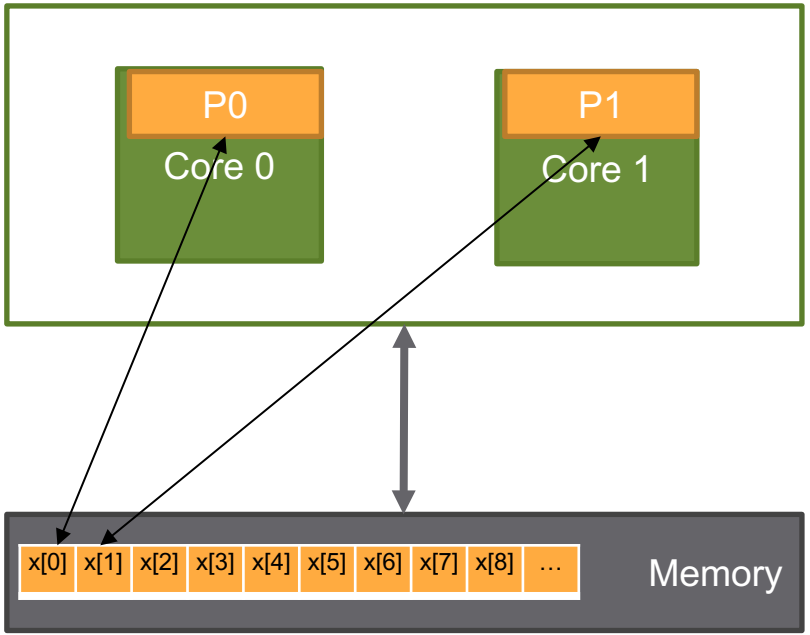


Κοινή μνήμη και μοιραζόμενα δεδομένα

- Παράδειγμα μοιραζόμενων δεδομένων:

```
for (i = 0 ; i < 100 ; i++)
    x[i] = f(x[i])
```

P0 (Core 0)	P1 (Core 1)
x[0] = f(x[0])	x[1] = f(x[1]

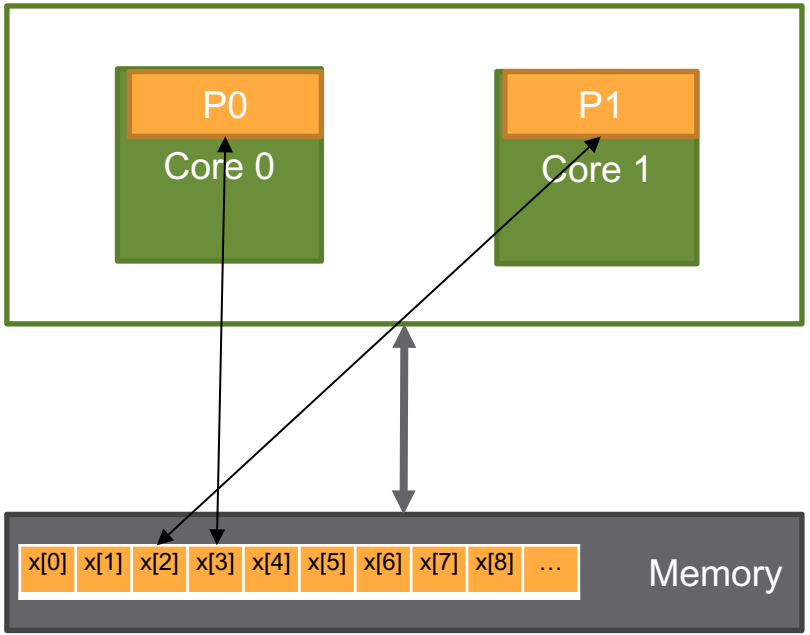


Κοινή μνήμη και μοιραζόμενα δεδομένα

- Παράδειγμα μοιραζόμενων δεδομένων:

```
for (i = 0 ; i < 100 ; i++)
    x[i] = f(x[i])
```

P0 (Core 0)	P1 (Core 1)
x[0] = f(x[0])	x[1] = f(x[1]
x[3] = f(x[3])	x[2] = f(x[3]

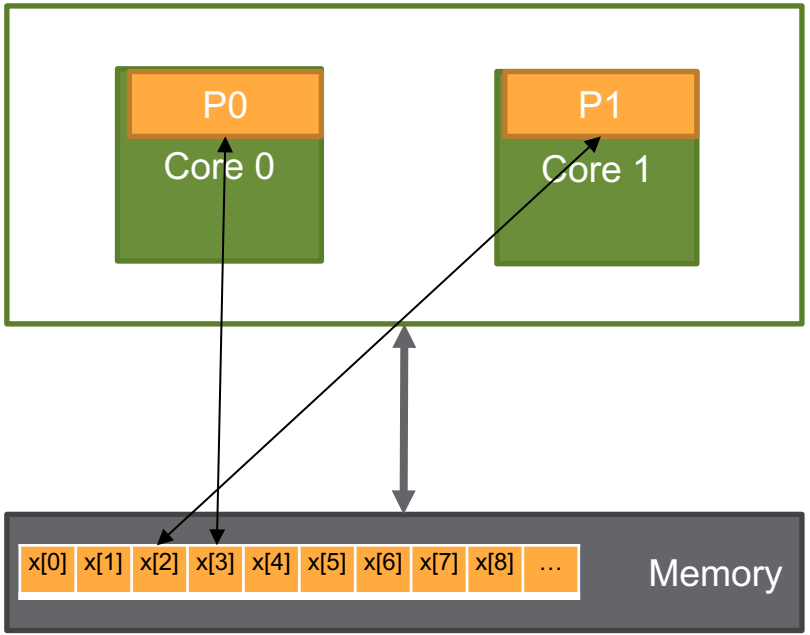


Κοινή μνήμη και μοιραζόμενα δεδομένα

- Παράδειγμα μοιραζόμενων δεδομένων:

```
for (i = 0 ; i < 100 ; i++)
    x[i] = f(x[i])
```

P0 (Core 0)	P1 (Core 1)
x[0] = f(x[0])	x[1] = f(x[1]
x[3] = f(x[3])	x[2] = f(x[3]
...	....



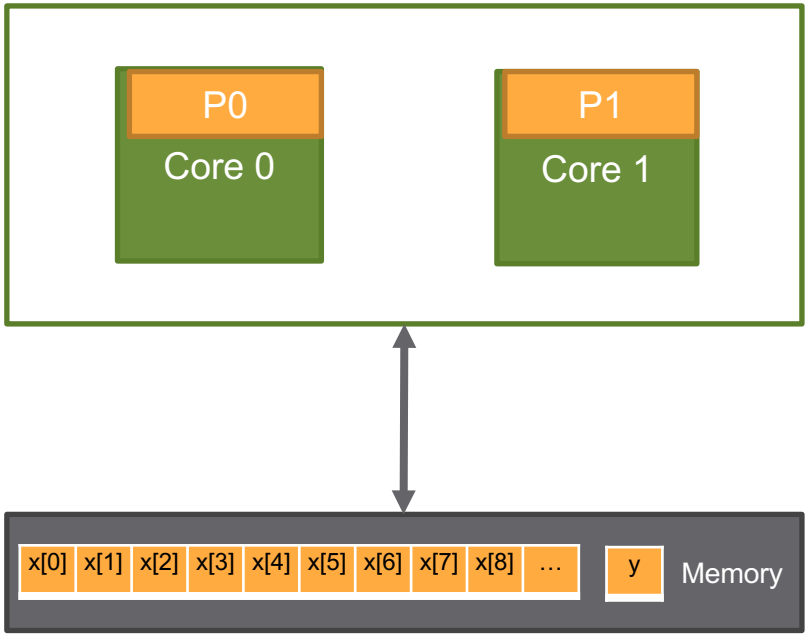
- Τα δεδομένα είναι μοιραζόμενα, αλλά κάθε διεργασία διαβάζει και γράφει από/σε διαφορετική θέση μνήμης!

Κοινή μνήμη και μοιραζόμενα δεδομένα

- Παράδειγμα μοιραζόμενων δεδομένων **με race condition**:

```
y = 0
for (i = 0 ; i < 100 ; i++)
        y = y + f(x[i])
```

- Εκτελείται από δύο νήματα/διεργασίες (processes)
- Κάθε μία εκτελείται σε διαφορετικό πυρήνα
- Μοιράζονται τον πίνακα x – βρίσκεται στην κοινή κύρια μνήμη
- **Μοιράζονται τη μεταβλητή y!**
- Κάθε διεργασία θα εκτελέσει κάποια βήματα του for loop – κάθε βήμα του for-loop είναι μια εργασία

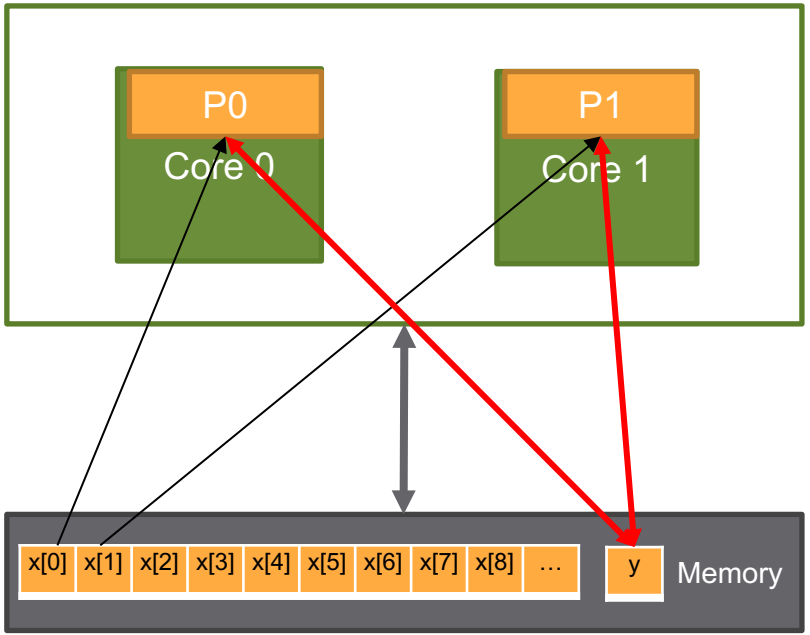


Κοινή μνήμη και μοιραζόμενα δεδομένα

- Παράδειγμα μοιραζόμενων δεδομένων **με race condition**:

```
y = 0
for (i = 0 ; i < 100 ; i++)
        y = y + f(x[i])
```

P0 (Core 0)	P1 (Core 1)
y = y + f(x[0])	y = y + f(x[1])

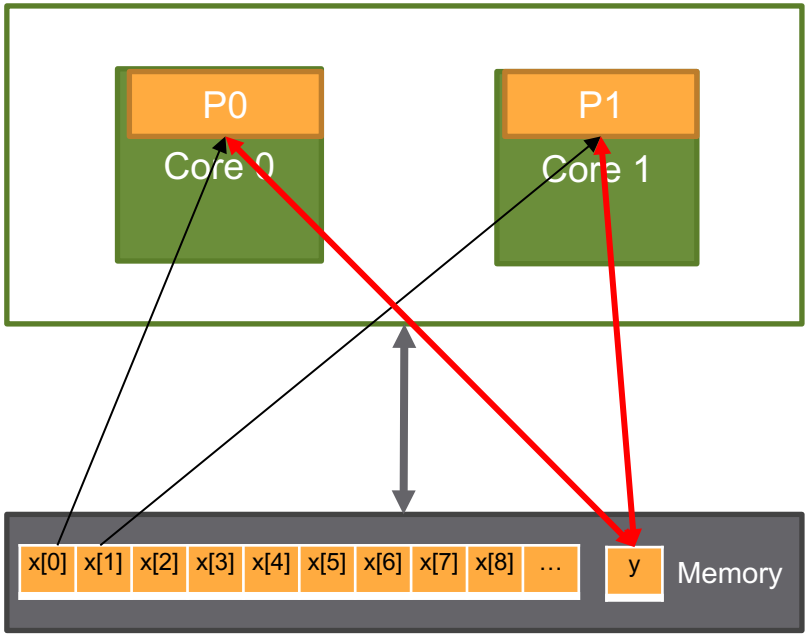


Κοινή μνήμη και μοιραζόμενα δεδομένα

- Παράδειγμα μοιραζόμενων δεδομένων **με race condition**:

```
y = 0
for (i = 0 ; i < 100 ; i++)
        y = y + f(x[i])
```

P0 (Core 0)	P1 (Core 1)
y = y + f(x[0])	y = y + f(x[1])



- **Πρόβλημα!**
  - Ποια τιμή για τη μεταβλητή y διαβάζει η P0;
  - Ποια τιμή για τη μεταβλητή y διαβάζει η P1;

Στάδιο 3: Οργάνωση πρόσβασης στα δεδομένα

- Η κατανομή των υπολογισμών και των δεδομένων καθώς και ο χαρακτηρισμός τους δημιουργεί ανάγκες για επικοινωνία και συγχρονισμό
- Συγχρονισμός:
  - Απαιτείται είτε λόγω του γράφου εξαρτήσεων (σειριοποίηση) είτε λόγω καταστάσεων συναγωνισμού (race conditions)
  - Σειριοποίηση:
    - Barriers, condition variables, semaphores
  - Εντοπισμός καταστάσεων συναγωνισμού και εισαγωγή κατάλληλου σχήματος ελέγχου ταυτόχρονης πρόσβασης (concurrency control)
  - Αμοιβαίος αποκλεισμός:
    - Critical section, Locks, readers-writers

Στάδιο 4: Ανάθεση εργασιών σε οντότητες εκτέλεσης

- Το στάδιο αυτό αναλαμβάνει να αναθέσει εργασίες (tasks) σε οντότητες εκτέλεσης (process, tasks, κλπ)
- Ο τρόπος με τον οποίο ανατίθενται τα tasks σε οντότητες εκτέλεσης μπορεί να επηρεάσει σημαντικά την εκτέλεση:
  - Παραλληλισμός και ισοκατανομή φορτίου
  - Τοπικότητα δεδομένων
  - Κόστος συγχρονισμού και επικοινωνίας
  - Κόστος διαχείρισης

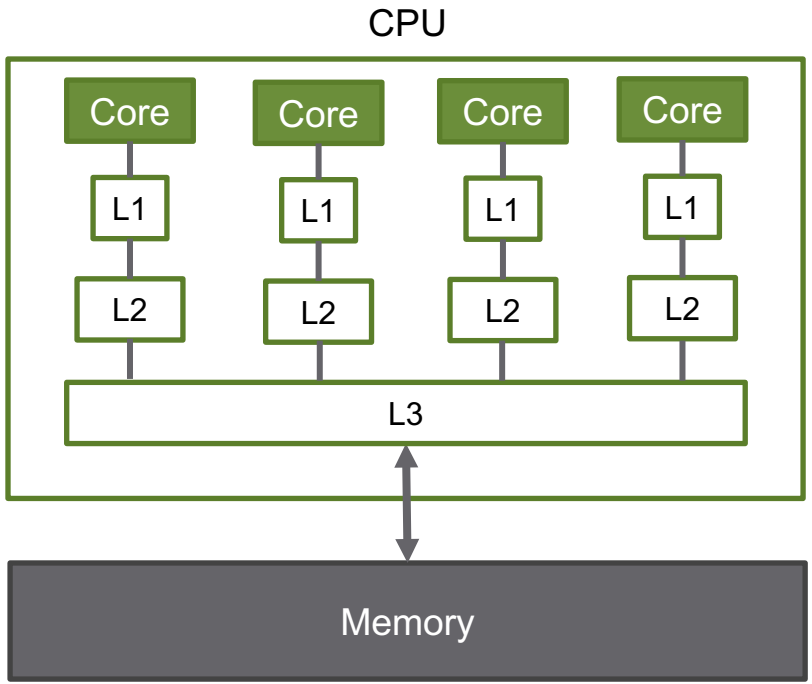
Παράλληλος προγραμματισμός σε κοινή μνήμη με OpenMP

«Μιλώντας» παράλληλα

- *Πρώτο βήμα*: σχεδιασμός παράλληλων προγραμμάτων = σκέψη
- *Δεύτερο βήμα*: υλοποίηση παράλληλων προγραμμάτων = ομιλία
  - Όπως και στις φυσικές γλώσσες, η σκέψη και η ομιλία είναι δύο στενά συνδεδεμένες λειτουργίες
- Τι προγραμματιστικές δομές χρειάζομαι για να μιλήσω παράλληλα;
- Πώς δημιουργώ και τερματίζω εργασίες (tasks) και οντότητες εκτέλεσης (processes, threads);
- Τι προγραμματιστικές δομές υπάρχουν για το χαρακτηρισμό των δεδομένων, τον έλεγχο πρόσβασης, το συγχρονισμό και την επικοινωνία ανάμεσα στις εργασίες/οντότητες εκτέλεσης;
- Πώς θα πετύχω παράλληλη επίδοση;

Σύγχρονες CPUs

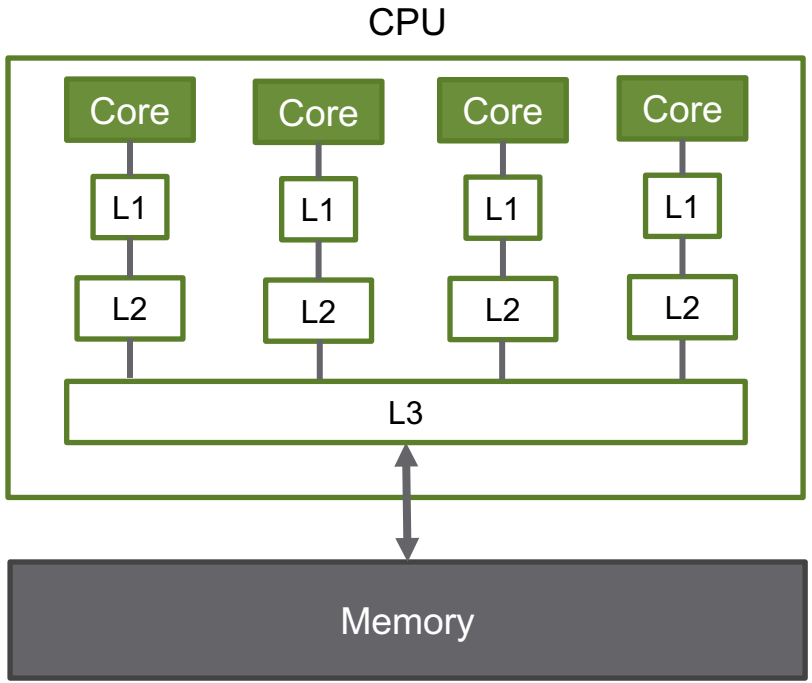
- Πολλοί παράλληλοι πυρήνες
- Βαθιές ιεραρχίες κρυφής μνήμης



Παράλληλος προγραμματισμός σε σύγχρονες CPU

- Κοινή μνήμη – πολλοί πυρήνες
- Όλοι οι πυρήνες έχουν πρόσβαση στα *ίδια δεδομένα*
- Σε κάθε πυρήνα, ανατίθεται ένα *νήμα εκτέλεσης* ενός προγράμματος που χρησιμοποιεί αυτά τα δεδομένα

- Οι κρυφές μνήμες (μοιραζόμενες ή μη) λειτουργούν βοηθητικά για επιτάχυνση της πρόσβασης στα δεδομένα
- Δε διαχειριζόμαστε προγραμματιστικά τα δεδομένα στις κρυφές μνήμες



Παράλληλος προγραμματισμός σε σύγχρονες CPU

- Προγραμματιστικά μοντέλα κοινού χώρου διευθύνσεων
  - Υποστηρίζουν κοινά δεδομένα ανάμεσα στα νήματα
  - Επιταχύνουν τον προγραμματισμό
  - Μπορούν να οδηγήσουν σε δύσκολα ανιχνεύσιμα race conditions

- Σε αρχιτεκτονικές κοινής μνήμης, ο παράλληλος προγραμματισμός είναι εύκολος
  - Αξιοποιούμε προγραμματιστικά μοντέλα κοινού χώρου διευθύνσεων
  - Ο προγραμματιστής σημειώνει παράλληλες εργασίες και ορίζει τη χρήση των μεταβλητών
  - Ο μεταγλωττιστής και το σύστημα χρόνου εκτέλεσης αναλαμβάνουν την εκτέλεση

OpenMP: προγραμματισμός σε κοινή μνήμη

- Πρότυπο για προγραμματισμό σε μοιραζόμενη μνήμη
- Ορίζει συγκεκριμένη διεπαφή (API) και όχι υλοποίηση
  - οδηγίες σε μεταγλωττιστή (compiler directives)
  - βιβλιοθήκη χρόνου εκτέλεσης (run-time library)
  - μεταβλητές συστήματος (environment variables)
- Ο παραλληλισμός δηλώνεται ρητά (explicitly) από τον προγραμματιστή
- Γλώσσες: C/C++, Fortran
- MP = MultiProcessor
- Τα προγράμματα του OpenMP:
  - Μπορούν να μεταφραστούν από μεταγλωττιστή που δεν το υποστηρίζει.
  - Μπορούν να εκτελεστούν σειριακά.
- Εφαρμόζεται κυρίως σε εφαρμογές με μεγάλους πίνακες

OpenMP: Βασικές έννοιες

- Παράλληλη Περιοχή (Parallel Region)
  - Κώδικας που εκτελείται από πολλαπλά νήματα
- Κατανομή Εργασίας (Work Sharing)
  - Η διαδικασία κατά την οποία κατανέμεται η εργασία στα νήματα μια παράλληλης περιοχής
- Οδηγία Μεταγλωττιστή (Compiler Directive)
  - Η διεπαφή για την χρήση του OpenMP σε προγράμματα.
  - Για την C:

    ```
    #pragma omp <directive> <clauses>
    ```
- Construct:

  ```
  #pragma omp ...
  <C statement>
  ```

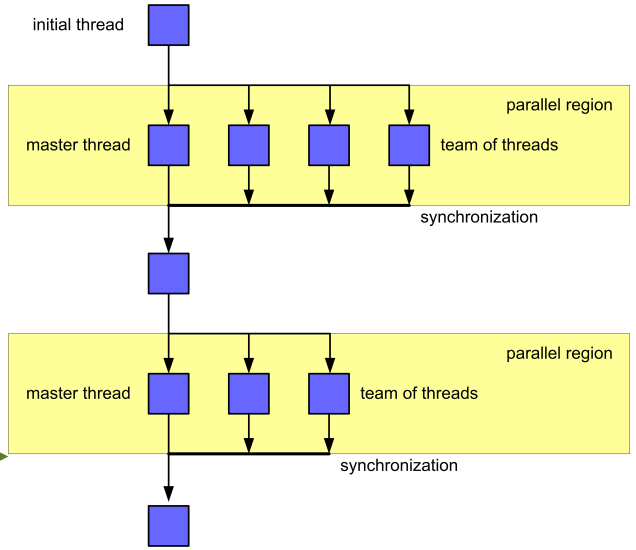
OpenMP: μοντέλο εκτέλεσης

- Η εκτέλεση ξεκινά από 1 **initial thread**
- Όταν το initial thread συναντήσει μία *παράλληλη περιοχή*
  - Δημιουργείται μία ομάδα νημάτων (**team of threads**) που περιλαμβάνει το initial thread (τώρα λέγεται **master thread**) και 0 ή περισσότερα άλλα threads
  - Κάθε νήμα αναλαμβάνει την εκτέλεση του μπλοκ εντολών που περιλαμβάνει η παράλληλη περιοχή (υπάρχουν κατάλληλες οδηγίες που διαφοροποιούν την εκτέλεση των νημάτων, βλ. συνέχεια)
- Στο τέλος της παράλληλης περιοχής τα νήματα συγχρονίζονται
- Οι παράλληλες περιοχές μπορεί να είναι φωλιασμένες (nested)
  - Αν η υλοποίηση το υποστηρίζει η φωλιασμένη ομάδα νημάτων μπορεί να περιέχει περισσότερα του ενός thread

OpenMP: Μοντέλο εκτέλεσης

```
#include <omp.h>

    main(){
  …
  #pragma omp parallel num_threads(4)
  {
    …
  }
  …
  omp_set_num_threads(3);
  #pragma omp parallel
  {
    …
  }
  …
}
```



OpenMP: Προγραμματιστικές δομές

- Το OpenMP μού επιτρέπει να φτιάξω νήματα εκτέλεσης που θα εκτελεστούν παράλληλα σε διαφορετικούς πυρήνες
- Για να προγραμματίσω παράλληλα, χρειάζομαι τρόπους να εκφράσω την κατανομή εργασίας
  - Παράλληλες εργασίες που θα ανατεθούν αυτόματα στα νήματα εκτέλεσης
  - Προγραμματιστικές δομές
- Το OpenMP προσφέρει δύο βασικές προγραμματιστικές δομές για διευκόλυνση του παράλληλου προγραμματισμού:
  - parallel for
  - fork/join

parallel for

- Οι δομές for είναι πολύ συνηθισμένες στον προγραμματισμό
- Η παραλληλοποίηση των for-loops αποτελεί σημαντικότατη προσέγγιση στο σχεδιασμό και την υλοποίηση ενός παράλληλου προγράμματος
- Είναι ευθύνη του προγραμματιστή να αποφασίσει αν ένα loop είναι παράλληλο!

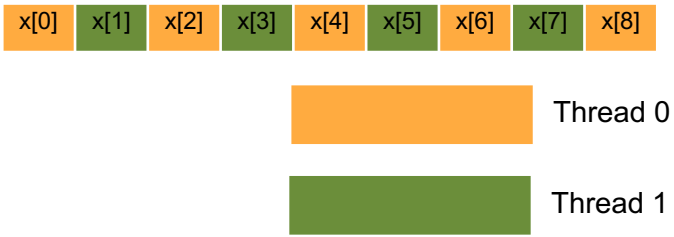
parallel for

- **Παράδειγμα 1:** Υπολογισμοί σε διάνυσμα x με n στοιχεία

```
for (i = 0 ; i < n ; i++)
        x[i] = sqrt(x[i])
```

- Το loop είναι παράλληλο – μπορώ να διαμοιράσω τον υπολογισμό των στοιχείων σε νήματα

```
Parallel for (i = 0 ; i < n ; i++)
        x[i] = sqrt(x[i])
```



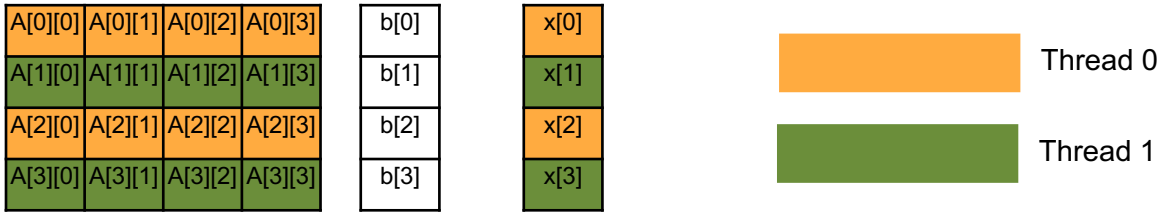
parallel for

- **Παράδειγμα 2:** Πολλαπλασιασμός πίνακα A (n x n) με διάνυσμα b (n x 1)

```
for (i = 0 ; i < n ; i++)
        for (j = 0 ; j < n ; j++)
                x[i] = A[i][j] * b[j]
```

- Το εξωτερικό loop είναι παράλληλο – μπορώ να διαμοιράσω τον υπολογισμό κάθε στοιχείου του x σε νήματα

```
Parallel for (i = 0 ; i < n ; i++)
        for (j = 0 ; j < n ; j++)
                x[i] = x[i] + A[i][j] * b[j]
```



parallel for

- Οι δομές for είναι πολύ συνηθισμένες στον προγραμματισμό
- Η παραλληλοποίηση των for-loops αποτελεί σημαντικότατη προσέγγιση στο σχεδιασμό και την υλοποίηση ενός παράλληλου προγράμματος
- Είναι ευθύνη του προγραμματιστή να αποφασίσει αν ένα loop είναι παράλληλο!

fork/join

- Αφορά εφαρμογές με δυναμική ανάγκη για δημιουργία / τερματισμό tasks
- Τα tasks δημιουργούνται (fork) και τερματίζονται (join) δυναμικά

fork/join

- **Παράδειγμα 1:** Υπολογισμοί σε διάνυσμα x με n στοιχεία

```
for (i = 0 ; i < n ; i++)
        x[i] = sqrt(x[i])
```

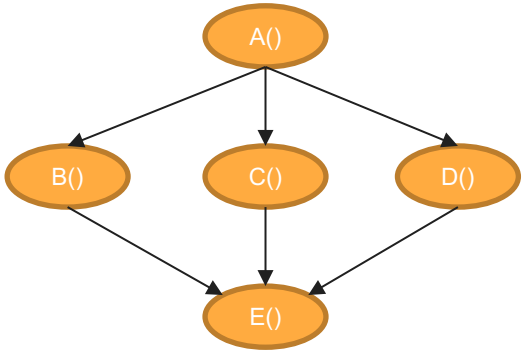
- Μπορώ να δημιουργήσω μία νέα εργασία για κάθε στοιχείο του διανύσματος

```
for (i = 0 ; i < n ; i++)
        fork(f(x[i]))
join()
```

- Παραλληλοποίηση ακριβώς όπως στο parallel for

fork/join

- **Παράδειγμα 2:** Παράλληλος υπολογισμός του ακόλουθου γράφου εργασιών
  - Οι συναρτήσεις B, C, D μπορούν να εκτελεστούν παράλληλα
    - Μετά την A
  - Η συνάρτηση E μπορεί να εκτελεστεί μόνο μετά την εκτέλεση των B, C, D




```
A();


fork B();
fork C();
fork D();
join

E();
```

- Αν υπάρχουν περισσότερα από 3 νήματα, οι συναρτήσεις B, C, D θα εκτελεστούν παράλληλα

OpenMP: components

Directives

- Parallel regions
- Work sharing
- Synchronization
- Tasks
- Data-sharing attributes
  - private
  - firstprivate
  - lastprivate
  - shared
  - reduction

Environment variables

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism

Runtime environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Timers
- API for locking

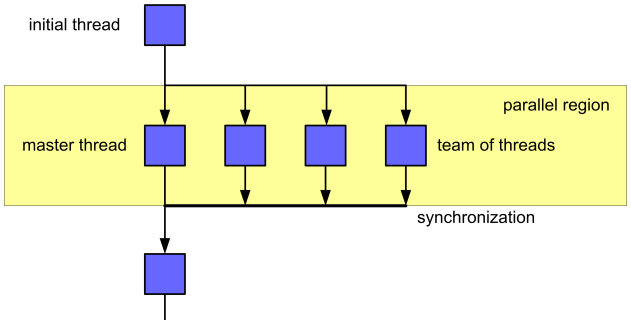
OpenMP: Παράλληλες περιοχές

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
  structured-block
```

- where clause is one of the following:
  - if (scalar-expression)
  - num_threads (integer-expression)
  - default (shared | none)
  - private (list)
  - firstprivate (list)
  - shared (list)
  - copyin (list)
  - reduction (operator: list)
- Ο αριθμός των νημάτων καθορίζεται:
  - Από το num_threads clause
  - Με τη χρήση της omp set_num_threads()
  - Με τη μεταβλητή περιβάλλοντος OMP_NUM_THREADS (Χρόνος Εκτέλεσης)
- Υπονοείται *barrier* στο τέλος της περιοχής
- *Το barrier υπονοεί flush*

OpenMP: Barrier

- Barrier: μέθοδος συγχρονισμού όλων των νημάτων εκτέλεσης
- Όταν στον κώδικα υπάρχει ένα barrier, ένα νήμα εκτέλεσης που φτάνει εκεί, θα μπλοκάρει και θα περιμένει ωσότου όλα τα νήματα φτάσουν εκεί στην εκτέλεσή τους
- Επιβάλει *καθολικό συγχρονισμό*
- Στο τέλος κάθε παράλληλης περιοχής, υπάρχει barrier



OpenMP: Flush

- Το barrier στο OpenMP υπονοεί *flush*
- Στο τέλος μιας παράλληλης περιοχής, τα νήματα πρέπει να έχουν καθολική εικόνα της κοινής μνήμης
  - Αν ένα νήμα έχει πραγματοποιήσει αλλαγές σε μοιραζόμενα δεδομένα, τα άλλα νήματα πρέπει να ενημερωθούν για τις αλλαγές
    - Ένα νήμα μπορεί να έχει πραγματοποιήσει αλλαγές σε δεδομένα στην κρυφή του μνήμη – οι αλλαγές πρέπει να μεταφερθούν στην κύρια μνήμη για να γίνουν ορατές στα άλλα νήματα
  - Η ενημέρωση γίνεται με τη λειτουργία flush στη μνήμη

OpenMP: Κατανομή εργασίας

- Directives
  - `#pragma omp for`
  - `#pragma omp single`
  - `#pragma omp task`

- Τα directives για την κατανομή εργασίας, περιέχονται σε μία παράλληλη περιοχή.
- Δεν δημιουργούνται νέα νήματα
- Δεν υπονοείται barrier στην είσοδο

OpenMP: #pragma omp for

```
#pragma omp for [schedule(…)] [nowait]
  for-loop
```

- Κατανέμει επαναλήψεις εντολής ***for*** σε ομάδα νημάτων
- Εντολή for σε κανονική μορφή (canonical form)
- schedule: καθορίζει τρόπο κατανομής επαναλήψεων
  - static[,chunk]: round-robin στατική κατανομή
  - dynamic[,chunk]: δυναμική κατανομή σε ανενεργά νήματα
  - guided[,chunk]: δυναμική κατανομή με εκθετική μείωση
  - runtime: κατανομή καθορίζεται σε χρόνο εκτέλεσης
- nowait: αποτρέπει συγχρονισμό (barrier) κατά την έξοδο

OpenMP: #pragma omp for

- Παράδειγμα

```
for(i=1;i<n;i++)
    b[i]=(a[i]+a[i-1])/2.0;

#pragma omp parallel
  {
    #pragma omp for
      for(i=1;i<n;i++)
        b[i]=(a[i]+a[i-1])/2.0;
  }
```

OpenMP: #pragma omp single

```
#pragma omp single [nowait]
  structured-block
```

- Ορίζει τμήμα κώδικα που εκτελείται από μόνο ένα νήμα της ομάδας
- nowait: αποτρέπει συγχρονισμό κατά την έξοδο

OpenMP: #pragma omp single

- Παράδειγμα

```
#pragma omp parallel
{
  #pragma omp single
    printf("Beginning work1.\n");
  work1();
  #pragma omp single
    printf("Finished work1.\n");
  #pragma omp single nowait
    printf("Finished work1, beginning work2.\n");
  work2();
}
```

OpenMP: Συντομεύσεις

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

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

```
#pragma omp parallel
#pragma omp sections
```

```
#pragma omp parallel sections
```

OpenMP: Συγχρονισμός

- `#pragma omp barrier:`
  Συγχρονισμός νημάτων
- `#pragma omp master:`
  Κώδικας που εκτελείται μόνο από το κύριο νήμα
- `#pragma omp critical:`
  Κώδικας που δεν εκτελείται παράλληλα
- `#pragma omp atomic:`
  Ατομική λειτουργία σε θέση μνήμης (++,–,+=,...)
- `#pragma omp flush:`
  Επιβολή συνεπούς εικόνας των μοιραζόμενων αντικειμένων
- `#pragma omp ordered:`
  Επιβολή σειριακής εκτέλεσης structured block

OpenMP: #pragma omp barrier / master

```
#pragma omp parallel
{
   #pragma omp barrier
   #pragma omp master
      gettimeofday(start,(struct timezone*)NULL);
   work();
   #pragma omp barrier
   #pragma omp master
   {
      gettimeofday(finish,(struct timezone*)NULL);
      print_stats(start,finish);
   }
}
```

OpenMP: #pragma omp atomic

```
#pragma omp atomic new-line
  expression-stmt
```

```
#pragma omp parallel for shared(x, y, index, n)
{
   for(i=0;i<n;i++)
      #pragma omp atomic
         x[index[i]] += work1(i);
      y[i]+=work2(i);
}
```

OpenMP: #pragma omp ordered

#pragma omp ordered *new-line*

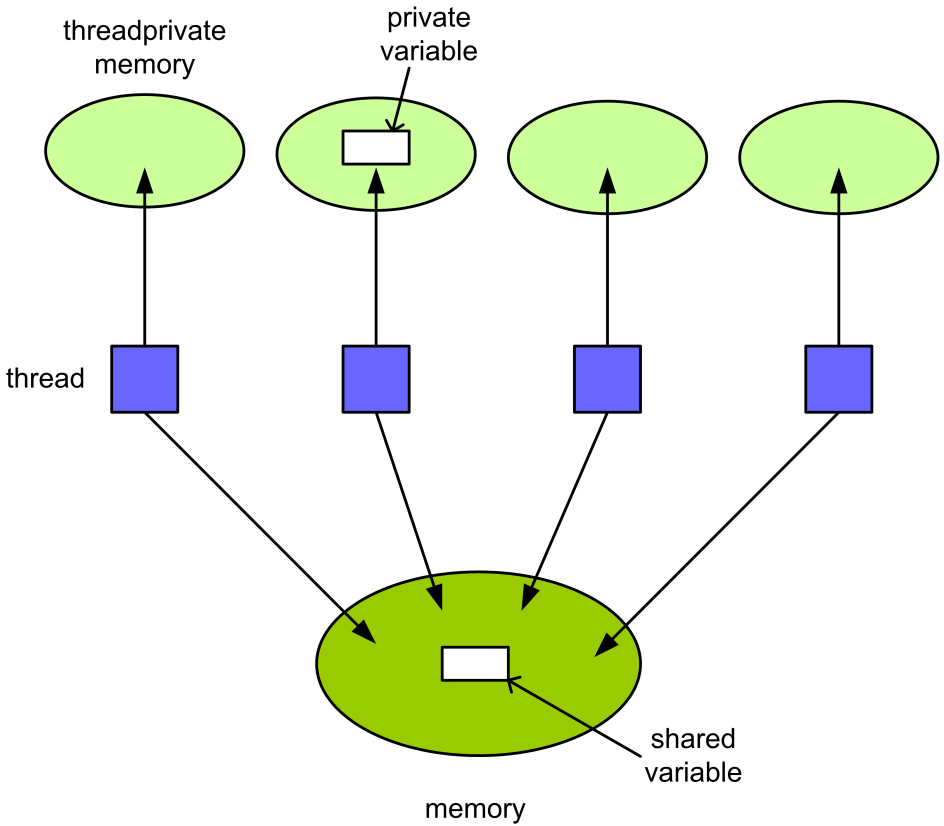
structured-block

```
#pragma omp parallel
{
   #pragma omp for ordered
      for(i=0;i<N;i++){
         a[i]=compute(i);
         #pragma omp ordered
         printf("a[%d]=%d\n",i,a[i]);
      }
}
```

OpenMP: Μοντέλο δεδομένων

- Σε μία παράλληλη περιοχή, υπάρχουν δύο ήδη μεταβλητών, shared και private
- Οι αλλαγές στα αντικείμενα που βρίσκονται στην κοινή μνήμη (shared) δεν γίνονται απαραίτητα αντιληπτές στο σύνολο των νημάτων
- Κάθε νήμα έχει μία τοπική εικόνα των δεδομένων
- Η λειτουργία flush επιβάλει συνέπεια ανάμεσα στις τοπικές εικόνες και στην κεντρική μνήμη

OpenMP: Μοντέλο δεδομένων



OpenMP: Περιβάλλον δεδομένων

- private (*variable-list*)
  - Ανάθεση νέου αντικειμένου για κάθε νήμα
  - Το πρότυπο αντικείμενο έχει απροσδιόριστη τιμή κατά την είσοδο και έξοδο στο construct, και δεν πρέπει να τροποποιείται
- firstprivate (*variable-list*)
  - Σαν private, κάθε νέο αντικείμενο **αρχικοποιείται** (εισέρχεται στην παράλληλη περιοχή) με την τιμή του προτύπου ακριβώς πριν την έναρξη της παράλληλης περιοχής
- lastprivate (*variable-list*)
  - Σαν private, το πρότυπο αντικείμενο εξέρχεται από την παράλληλη περιοχή με την τιμή που κατέχει το thread που εκτέλεσε την τελευταία επανάληψη (σε parallel loop) ή το τελευταίο section (σε parallel sections)

OpenMP: Περιβάλλον δεδομένων

- shared (*variable list*)
  - Μοιραζόμενη μεταβλητή για όλα τα νήματα της ομάδας
- reduction (*op: variable-list*)
  - Αναφέρεται σε εντολές τις μορφής x = x op expr, όπου op ένας από τους *, -, &, ^, |, &&, ||
  - Κάθε μεταβλητή το πολύ σε μια reduction clause
  - Για κάθε μεταβλητή δημιουργείται αντίστοιχη τοπική μεταβλητή σε κάθε νήμα και αρχικοποιείται ανάλογα με τελεστή op
- default (shared | none)
  - shared: ισοδύναμο με τον ορισμό κάθε μεταβλητής που δεν υπάρχει σε κανέναν περιβάλλον (shared, private, reduction, κλπ) , σαν shared
  - none: αν μία μεταβλητή δεν έχει ενταχθεί σε κάποιο περιβάλλον τότε ο compiler «χτυπάει» λάθος

OpenMP: reduction

```
#pragma omp parallel for reduction(+:sum)
   for(i=1;i<n;i++)
      sum = sum + a(i);
```

OpenMP: Βιβλιοθήκη χρόνου εκτέλεσης

- Περιβάλλον εκτέλεσης
  - omp_set_num_threads
  - omp_get_thread_num
  - omp_set_dynamic
- Συγχρονισμός με κλειδώματα
  - omp_init_lock
  - omp_set_lock / omp_test_lock
  - omp_unset_lock
  - nested
- Χρονομέτρηση
  - omp_get_wtime
  - omp_get_wtick

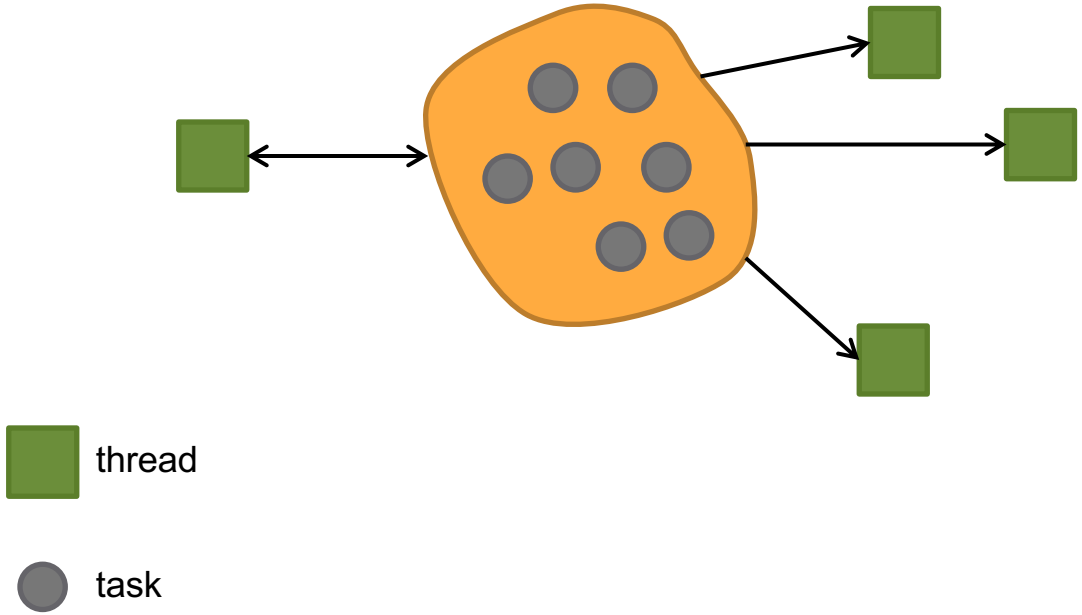
OpenMP: Μεταβλητές περιβάλλοντος

- Δρομολόγηση
  - export OMP_SCHEDULE=“static”
  - export OMP_SCHEDULE=“static,100”
  - setenv OMP_SCHEDULE “dynamic,20”
  - setenv OMP_SCHEDULE “guided,50”
- Δυναμική πολυνηματική εκτέλεση
  - export OMP_DYNAMIC=TRUE
  - setenv OMP_DYNAMIC FALSE
- Πλήθος νημάτων
  - export OMP_NUM_THREADS=2

OpenMP: tasks

- Η παραλληλοποίηση με χρήση tasks ξεκίνησε να υποστηρίζεται από το OpenMP στο τελευταίο πρότυπο (OpenMP 3.0) – May 2008
- Παρέχει τη δυνατότητα παραλληλοποίησης για εφαρμογές που παράγουν δουλειά δυναμικά
- Παρέχει ένα ευέλικτο μοντέλο για μη κανονικό (irregular) παραλληλισμό
- Ευκαιρίες για παραλληλισμό σε:
  - While loops
  - Recursive structures

OpenMP: tasks



OpenMP: #pragma omp task

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

όπου clause:

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

- Το thread που συναντά ένα `#pragma omp task` directive δημιουργεί ένα task με τον κώδικα που περιέχει το `structured-block` και το βάζει σε ένα task pool
- Το thread μπορεί να εκτελέσει ή όχι ένα task που συναντά

OpenMP: #pragma omp task

```
void process_list_items(node * head)
{
   #pragma omp parallel
   {
      #pragma omp single
      {
         node *p = head;
         while (p) {
            #pragma omp task
               process(p);
            p = p->next;
         }
      }
   }
}
```

OpenMP: Συγχρονισμός tasks

```
#pragma omp taskwait
```

το τρέχον task σταματά την εκτέλεσή του μέχρι όλα τα tasks που έχουν δημιουργηθεί μέχρι στιγμής από το τρέχον (παιδιά) να ολοκληρώσουν την εκτέλεσή τους

Ισχύει μόνο για τα άμεσα παιδιά (π.χ. όχι για τα εγγόνια)

(**Σημείωση:** βλέπε taskgroups στο OpenMP 4.0 για την αναμονή και άλλων απογόνων)

OpenMP: Μοντέλο εκτέλεσης tasks

- Προσοχή στις έννοιες **δημιουργία** / **εκτέλεση** task!
- Κάθε **task** μπορεί να εκτελεστεί από ένα από τα **threads** της ομάδας που το δημιούργησε
- Κάθε thread της ομάδας δημιουργεί ένα αρχικό (implicit) task
- Άρα κάθε λειτουργία σχετική με tasks έχει νόημα μόνο σε παράλληλες περιοχές
- Όταν ξεκινήσει η εκτέλεση ενός task by default είναι προσδεμένο (tied) με ένα thread
    - Αυτό μπορεί να αλλάξει (βλ. untied)
- Ένα task αναστέλλει τη λειτουργία του όταν υποχρεωθεί να **εκτελέσει** ένα άλλο task (βλ. If (0)) ή αν συναντήσει ένα taskwait

OpenMP: Μοντέλο εκτέλεσης tasks

- Προσοχή στις έννοιες **δημιουργία** / **εκτέλεση** task!
- Κάθε **task** μπορεί να εκτελεστεί από ένα από τα **threads** της ομάδας που το δημιούργησε
- Κάθε thread της ομάδας δημιουργεί ένα αρχικό (implicit) task
- Άρα κάθε λειτουργία σχετική με tasks έχει νόημα μόνο σε παράλληλες περιοχές
- Όταν ξεκινήσει η εκτέλεση ενός task by default είναι προσδεμένο (tied) με ένα thread
  - Αυτό μπορεί να αλλάξει (βλ. untied)
- Ένα task αναστέλλει τη λειτουργία του όταν υποχρεωθεί να **εκτελέσει** ένα άλλο task (βλ. If (0)) ή αν συναντήσει ένα taskwait