Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχ. και Μηχανικών Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Παράλληλος προγραμματισμός:
Σχεδίαση και υλοποίηση παράλληλων προγραμμάτων

Συστήματα Παράλληλης Επεξεργασίας

9ο Εξάμηνο

National Technical University of Athens
CSLab

Σύνοψη παρουσίασης

- Παράλληλες υπολογιστικές πλατφόρμες
  - PRAM: Η ιδανική παράλληλη πλατφόρμα
  - Η ταξινόμηση του Flynn
  - Συστήματα κοινής μνήμης
  - Συστήματα κατανεμημένης μνήμης
- Ανάλυση παράλληλων προγραμμάτων
  - Μετρικές αξιολόγησης επίδοσης
  - Ο νόμος του Amdahl
  - Μοντελοποίηση παράλληλων προγραμμάτων
- Οργάνωση πρόσβασης στα δεδομένα
- Παράλληλα προγραμματιστικά μοντέλα
  - Κοινού χώρου διευθύνσεων
  - Ανταλλαγής μηνυμάτων

Σύνοψη παρουσίασης

- Σχεδίαση και υλοποίηση παράλληλων προγραμμάτων
  - Παραλληλισμός σε επίπεδο εργασίας (task parallelism)
  - Παραλληλοποίηση σε επίπεδο δεδομένων (data parallelism)
  - Παραλληλοποίηση σε επίπεδο βρόχου (loop parallelism)
  - Παραλληλοποίηση σε επίπεδο συνάρτησης (function parallelism)
- Γλώσσες και εργαλεία
  - POSIX threads, MPI, OpenMP, Cilk, Cuda, Γλώσσες PGAS
- Αλληλεπίδραση με το υλικό
  - Συστήματα κοινής μνήμης
  - Συστήματα κατανεμημένης μνήμης και υβριδικά

Παραλληλισμός σε επίπεδο εργασίας (task parallelism)

- Αποτελεί τη γενικότερη προσέγγιση για την υλοποίηση ενός παράλληλου προγράμματος
- Βασίζεται στην εις βάθος μελέτη του αλγορίθμου για την κατανόηση/ανακάλυψη όλου του εγγενούς παραλληλισμού
- Ακόμα και αν τελικά προτιμηθούν άλλες απλούστερες προσεγγίσεις (βλ. data parallelism, loop parallelism στη συνέχεια) τα πρώτα στάδια σχεδιασμού με τη λογική του task parallelism καλό είναι να ακολουθούνται για την κατανόηση του παραλληλισμού
- Υπάρχει πολύ καλή υποστήριξη για προγραμματιστικά μοντέλα κοινού χώρου διευθύνσεων
- Η υλοποίηση στη γενική περίπτωση δεν είναι τόσο προφανής σε προγραμματιστικά μοντέλα ανταλλαγής μηνυμάτων

Βήματα σχεδιασμού / υλοποίησης

- **Βήμα 1:** Κατανομή υπολογισμών
- **Βήμα 2:** Ορισμός ορθής σειράς εκτέλεσης
- **Βήμα 3:** Χαρακτηρισμός δεδομένων
- **Βήμα 4:** Ανάθεση εργασιών σε οντότητες εκτέλεσης

Βήμα 1: Κατανομή υπολογισμών

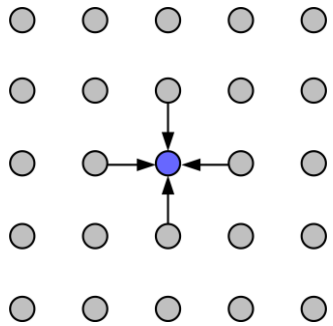
- **Στόχος:** Κατανομή σε υπολογιστικές εργασίες (tasks)
- **Παράδειγμα:** Πολλαπλασιασμός πίνακα με διάνυσμα ($y = A * x$, $y_i = \Sigma a_{ij} * x_j$)
  - *1 task = υπολογισμός της τιμής του* y_i *:*
  - 1 *task* = πολλαπλασιασμός, 1 *task* = πρόσθεση

Tip: Ξεκινάμε από το μικρότερο δυνατό granularity εργασίας (σαν να είχαμε PRAM!)

Παράδειγμα: Εξίσωση θερμότητας

- Αλγόριθμος του Jacobi (2-διάστατο πλέγμα Χ * Υ)

```
A[step+1][i][j] = 1/5 (A[step][i][j] + A[step][i-1][j] +
                       A[step][i+1][j] + A[step][i][j-1] +
                       A[step][i][j+1])
```



- **1 task =** υπολογισμός θερμότητας για κάθε σημείο του χωρίου
- **1 task =** υπολογισμός θερμότητας για κάθε χρονικό βήμα
- **1 task =** υπολογισμός θερμότητας για μία γραμμή / στήλη / block του δισδιάστατου πλέγματος για όλες τις χρονικές στιγμές
- **1 task =** υπολογισμός θερμότητας για μία γραμμή / στήλη / block του δισδιάστατου πλέγματος για μία χρονική στιγμή

Βήμα 2: Ορισμός ορθής σειράς εκτέλεσης

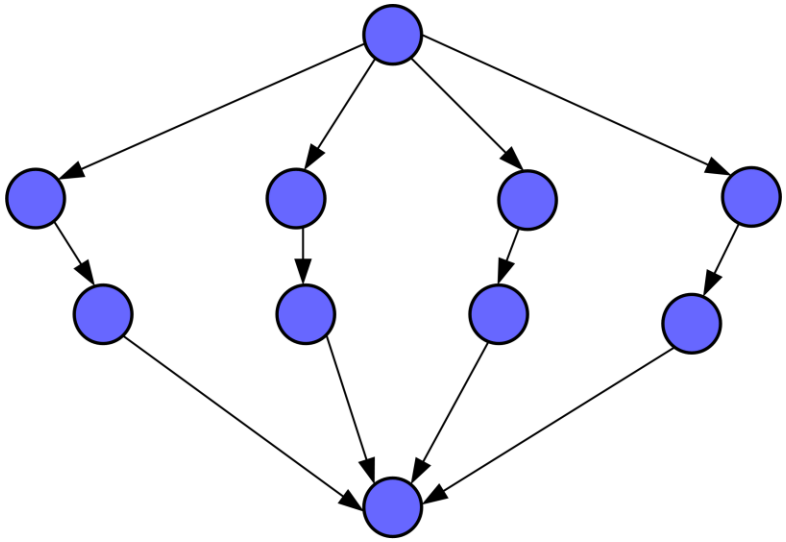
- Οι εργασίες που ορίστηκαν στο προηγούμενο στάδιο πρέπει να μπουν στη σωστή σειρά ώστε να εξασφαλίζεται η ίδια σημασιολογία με το σειριακό πρόγραμμα
- Το στάδιο αυτό συχνά αναφέρεται και ως χρονοδρομολόγηση = ανάθεση εργασιών σε χρονικές στιγμές
- Απαιτείται εντοπισμός των **εξαρτήσεων** ανάμεσα στις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

Εξαρτήσεις

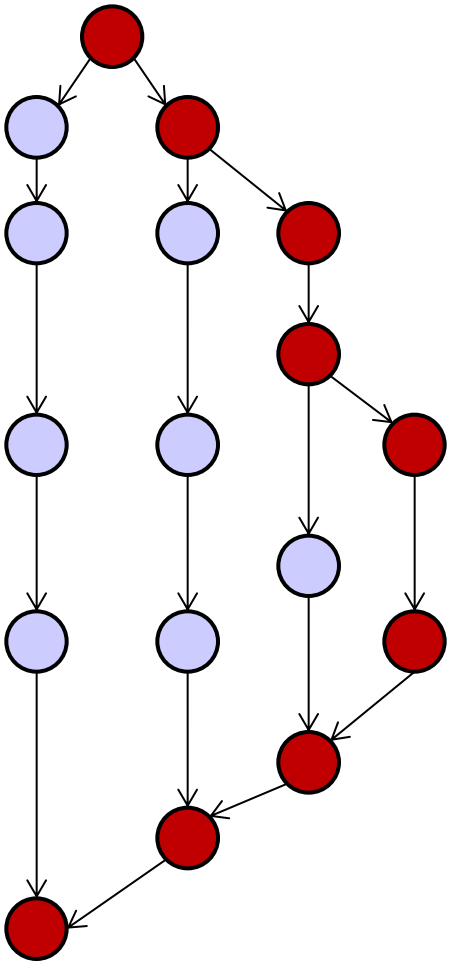
- Εξαρτήσεις δεδομένων υπάρχουν ήδη από το σειριακό πρόγραμμα
- Εξάρτηση υπάρχει όταν 2 εντολές αναφέρονται στα ίδια δεδομένα (θέση μνήμης)
- 4 είδη εξαρτήσεων
  - **Read-After-Write (RAW) ή true dependence**
  - Write-After-Read (WAR) ή anti dependence
  - Write-After-Write (WAW) ή output dependence
  - 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 (π.χ. το task1 πρέπει να διαβάσει δεδομένα που παράγει το task2)
- Διατήρηση των εξαρτήσεων: **σειριοποίηση** μεταξύ tasks

Γράφος εξαρτήσεων (task dependence graph)

- Ή απλά **task graph**
- Κορυφές: tasks
  - **Label κορυφής**: κόστος υπολογισμού του task
- Ακμές: εξαρτήσεις ανάμεσα στα tasks
  - **Βάρος ακμής**: όγκος δεδομένων που πρέπει να μεταφερθούν (στην περίπτωση της επικοινωνίας)



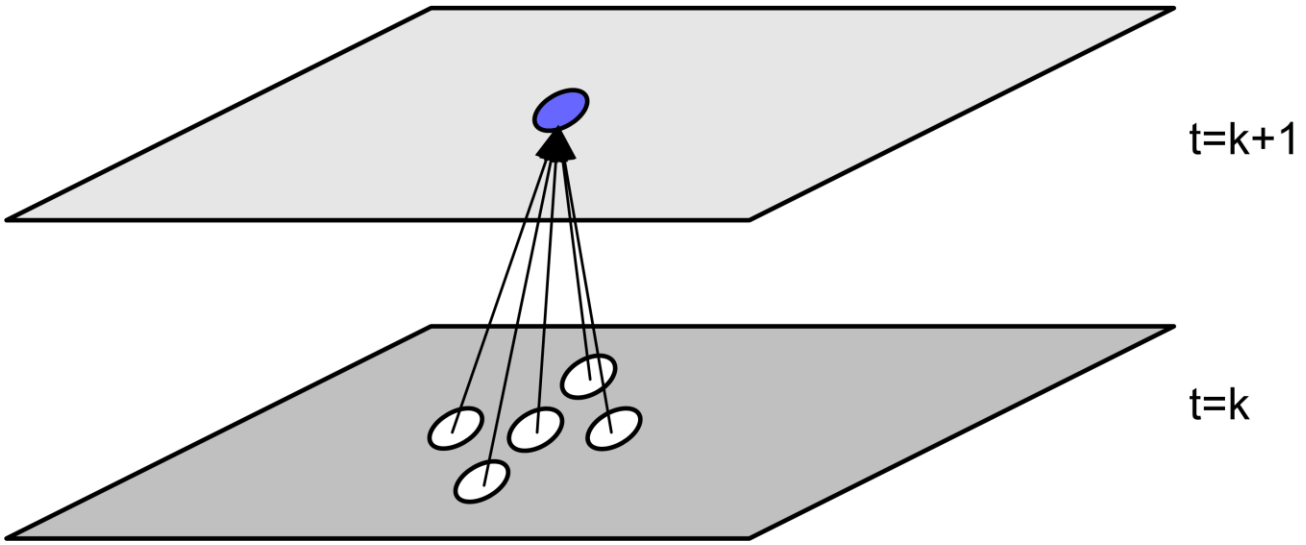
Task graphs: βασικές ιδιότητες



- **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:
  - **$T_1 / p \leq T_p$** (υποθέτει ότι δεν υπάρχει superlinear speedup)
  - **$T_\infty \leq T_p$**
  - **$T_p \leq T_\infty + (T_1 - T_\infty) / p$** *(Brent's law – υποθέτει PRAM)*
    - **or $T_p \leq T_\infty + T_1 / p$**
  - **Μέγιστο speedup T_1 / T_∞**

Εξαρτήσεις στην εξίσωση θερμότητας

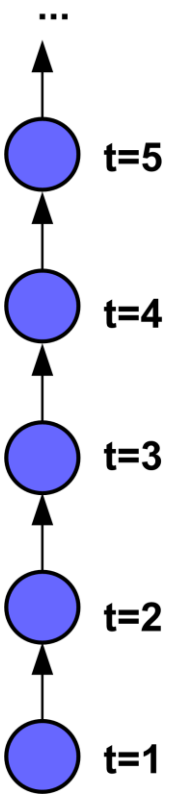
- 1 task = υπολογισμός θερμότητας για κάθε σημείο του χωρίου




```
A[step+1][i][j] = 1/5 (A[step][i][j] + A[step][i-1][j] + A[step][i+1][j]
   + A[step][i][j-1] + A[step][i][j+1])
```

Εξαρτήσεις στην εξίσωση θερμότητας

- 1 task = υπολογισμός θερμότητας για κάθε χρονικό βήμα

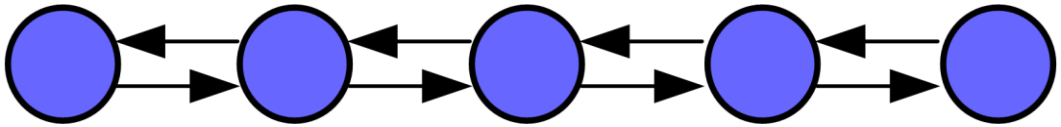


Δεν υπάρχουν ταυτόχρονα tasks!

```
A[step+1][i][j] = 1/5 (A[step][i][j] + A[step][i-1][j] + A[step][i+1][j]
   + A[step][i][j-1] + A[step][i][j+1])
```

Εξαρτήσεις στην εξίσωση θερμότητας

- 1 task = υπολογισμός θερμότητας για μία γραμμή του δισδιάστατου πλέγματος για όλες τις χρονικές στιγμές

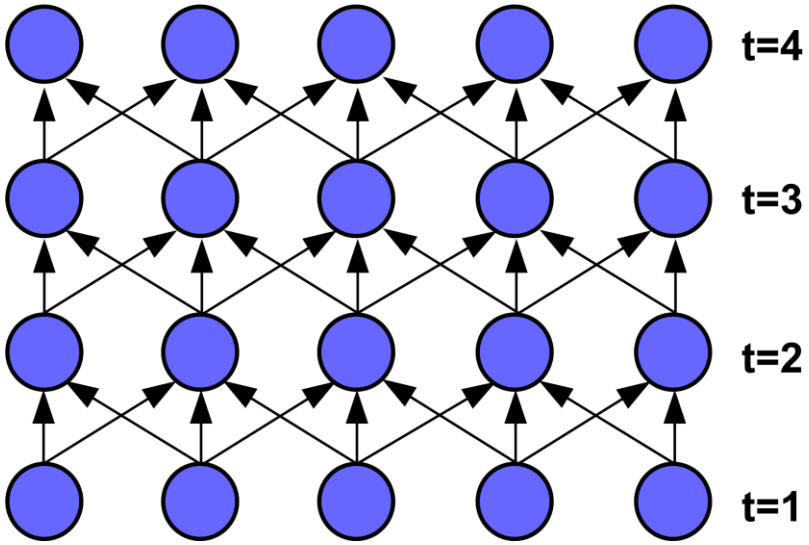


Δεν υπάρχει έγκυρη παράλληλη εκτέλεση!

```
A[step+1][i][j] = 1/5 (A[step][i][j] + A[step][i-1][j] + A[step][i+1][j]
   + A[step][i][j-1] + A[step][i][j+1])
```

Εξαρτήσεις στην εξίσωση θερμότητας

- 1 task = υπολογισμός θερμότητας για μία γραμμή του δισδιάστατου πλέγματος για μία χρονική στιγμή




```
A[step+1][i][j] = 1/5 (A[step][i][j] + A[step][i-1][j] + A[step][i+1][j]
   + A[step][i][j-1] + A[step][i][j+1])
```

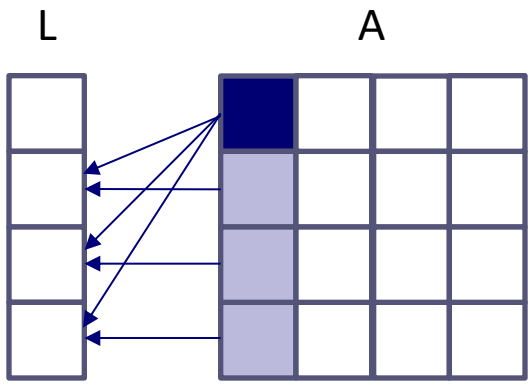
Παράδειγμα: Task graph για LU decomposition

```
//LU decomposition kernel
for (k = 0; k < N-1; k++)
   for(i = k+1; i < N; i++){
      L[i] = A[i][k] / A[k][k];
      for(j = k+1; j < N; j++)
         A[i][j] = A[i][j] - L[i] * A[k][j];
   }
```

- 1 task = μία στοιχειώδης αλγεβρική πράξη

Παράδειγμα: Task graph για LU decomposition

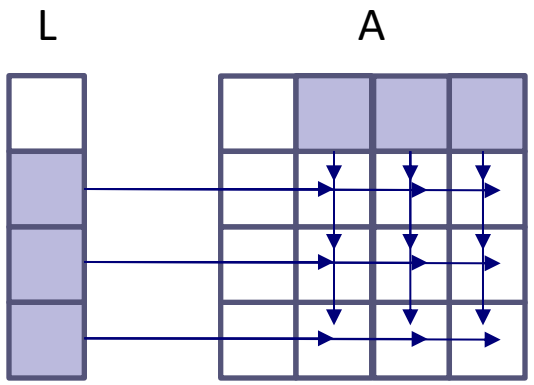
```
//LU decomposition kernel
for (k = 0; k < N-1; k++)
   for(i = k+1; i < N; i++){
      L[i] = A[i][k] / A[k][k];
      for(j = k+1; j < N; j++)
         A[i][j] = A[i][j] - L[i] * A[k][j];
   }
```



k = 0

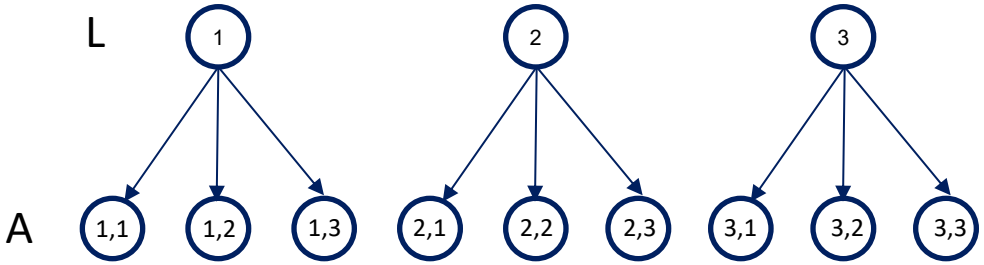
Παράδειγμα: Task graph για LU decomposition

```
//LU decomposition kernel
for (k = 0; k < N-1; k++)
   for(i = k+1; i < N; i++){
      L[i] = A[i][k] / A[k][k];
      for(j = k+1; j < N; j++)
         A[i][j] = A[i][j] - L[i] * A[k][j];
   }
```



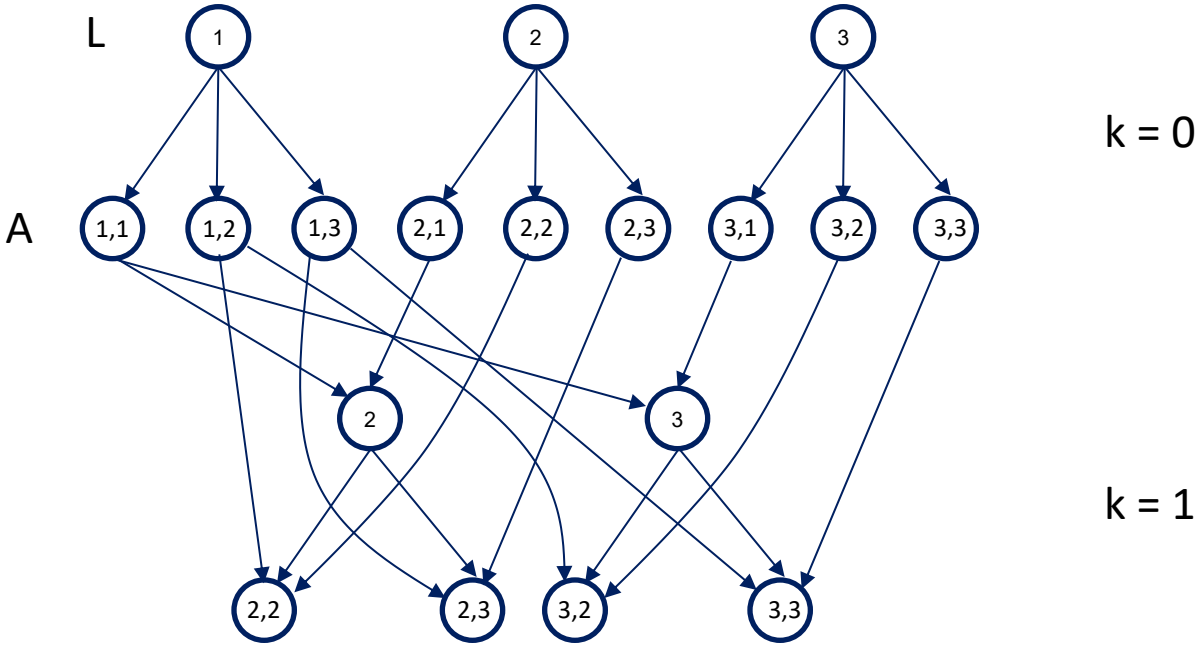
k = 0

Παράδειγμα: Task graph για LU decomposition

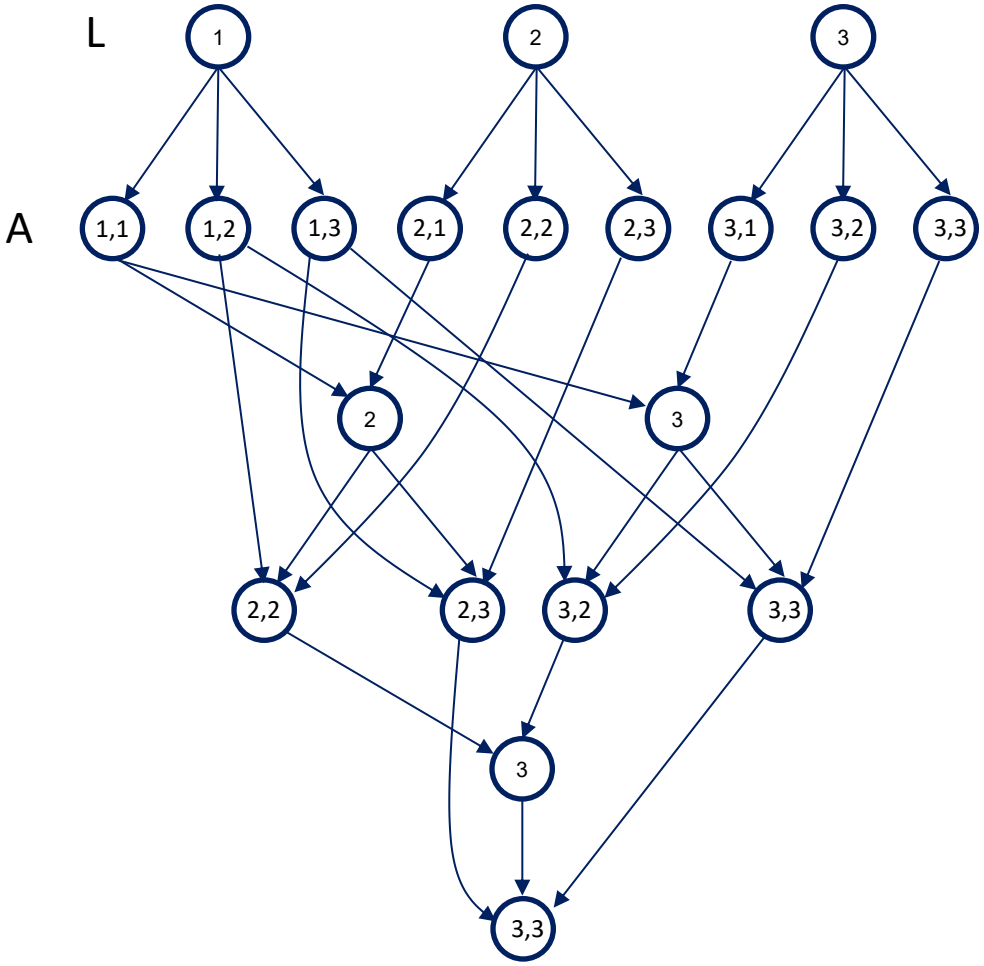


k = 0

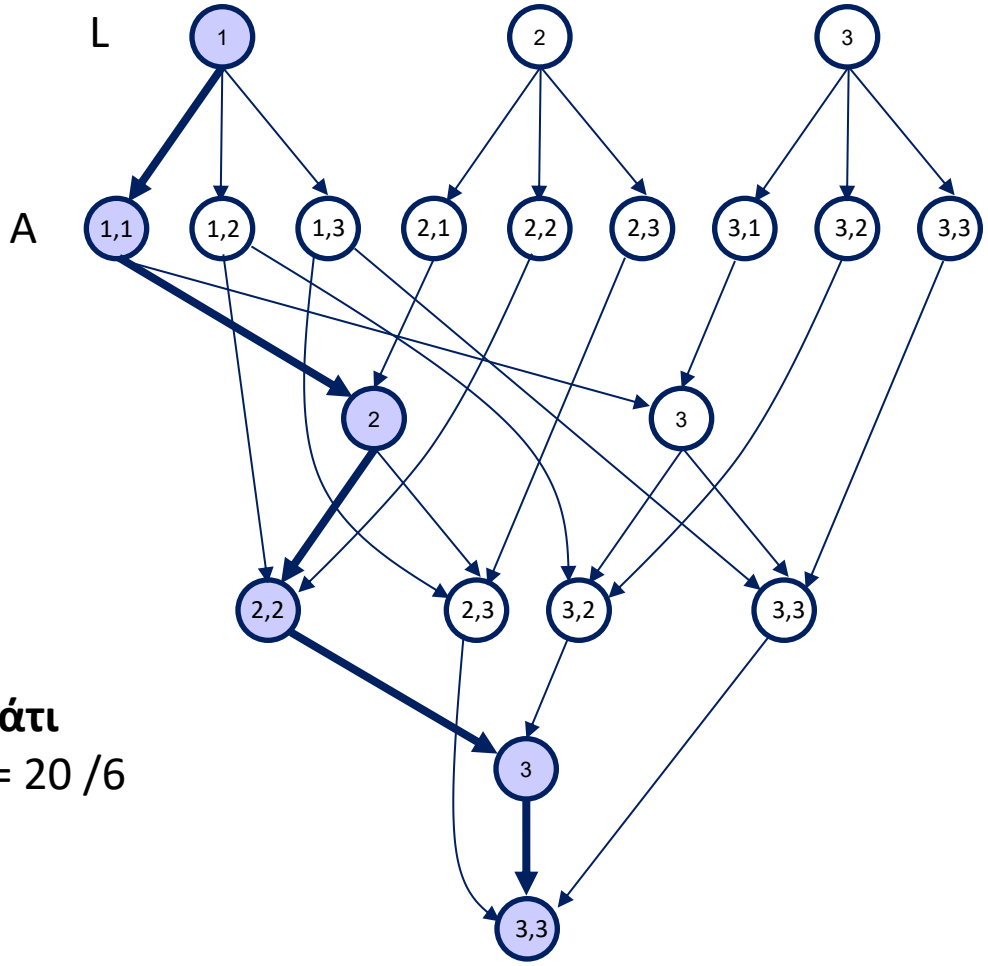
Παράδειγμα: Task graph για LU decomposition



Παράδειγμα: Task graph για LU decomposition



Παράδειγμα: Task graph για LU decomposition



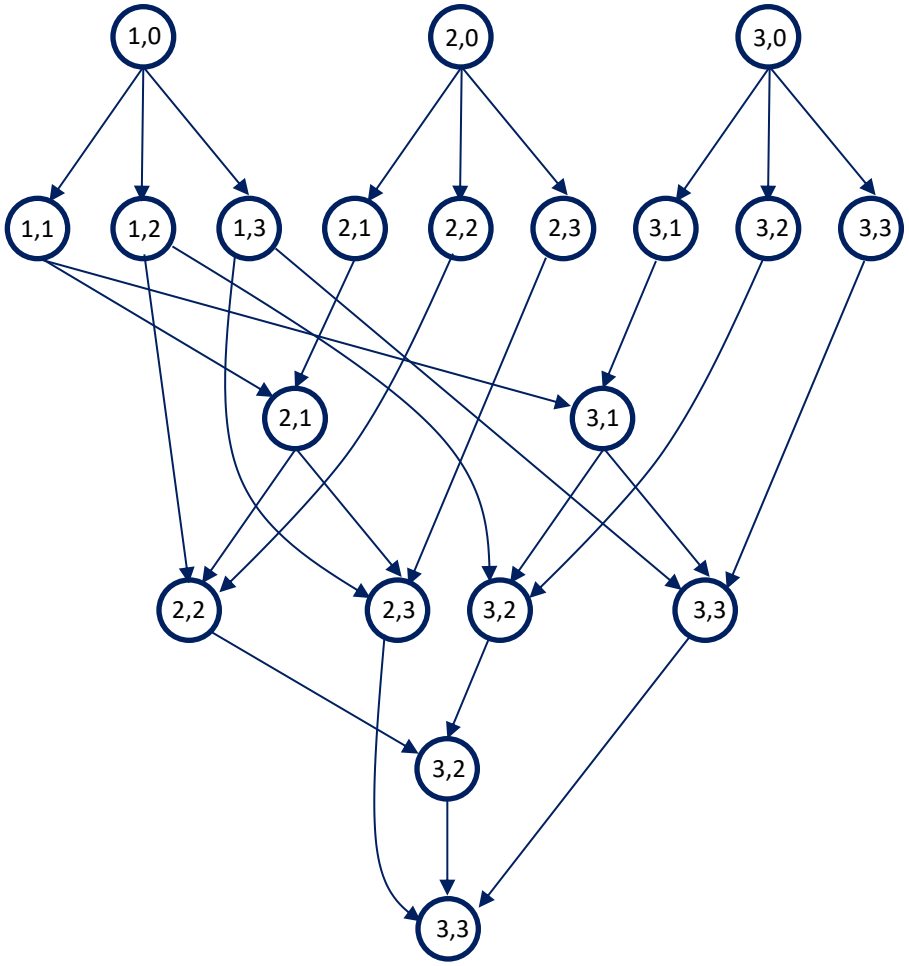
Κρίσιμο μονοπάτι

Μέγιστο speedup = 20 /6

Βήμα 3: Χαρακτηρισμός δεδομένων

- Μοιραζόμενα δεδομένα (κοινός χώρος διευθύνσεων)
  - Ο γράφος εξαρτήσεων υποδηλώνει data flow (μία εργασία υπολογίζει κάποιο δεδομένο που χρειάζεται κάποια άλλη). Απαιτείται συγχρονισμός (**σειριοποίηση**) για την ορθή σειρά εκτέλεσης των εργασιών
  - Απαιτείται μελέτη για το ενδεχόμενο ύπαρξης race conditions. Αν εντοπιστούν, χρειάζεται εισαγωγή κατάλληλων λειτουργιών συγχρονισμού (**critical section**), π.χ. κλειδώματα.
- Κατανεμημένα δεδομένα
  - Χρειάζεται ανταλλαγή μηνυμάτων ανάμεσα στις εργασίες για την αποστολή / λήψη των δεδομένων λόγων των ακμών στο γράφο εξαρτήσεων
  - Πιθανή ανάγκη επικοινωνίας για άλλα δεδομένα (π.χ. read only)
- Αντιγραμμένα δεδομένα
  - Αν γίνονται εγγραφές σε αντιγραμμένα δεδομένα απαιτείται επέκταση του γράφου για να ενσωματωθούν οι λειτουργίες ενημέρωσης (reduction)

Βήμα 4: Ανάθεση εργασιών σε οντότητες εκτέλεσης



k = 0

k = 1

k = 2

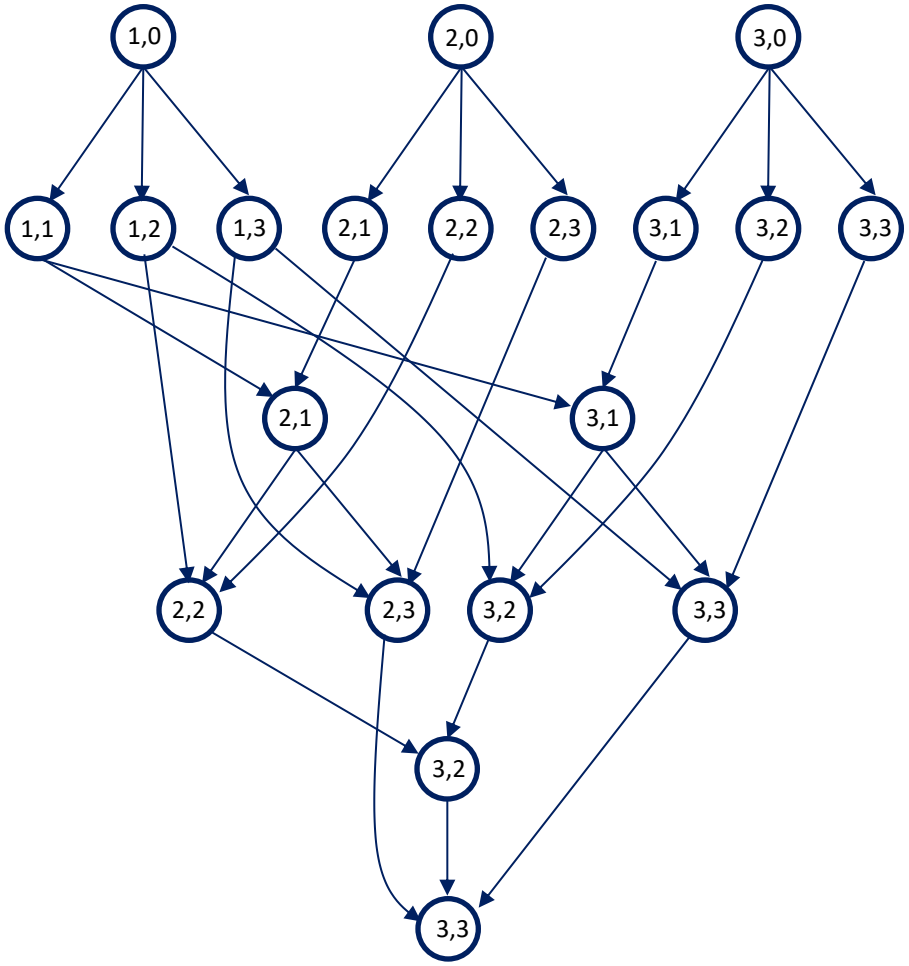
Γενικές παρατηρήσεις

- Τα tasks γενικά είναι περισσότερα από τους εργάτες. Πώς θα ομαδοποιήσουμε tasks σε εργάτες;
  - **Παρατήρηση:** Μερικά tasks έχουν εξαρτήσεις (ανήκουν σε ένα μονοπάτι του γράφου) και επομένως μπορούν να ανατεθούν στον ίδιο εργάτη

Γενικές παρατηρήσεις

- Τα tasks γενικά είναι περισσότερα από τους εργάτες. Πώς θα ομαδοποιήσουμε tasks σε εργάτες;
  - **Παρατήρηση:** Μερικά tasks έχουν εξαρτήσεις (ανήκουν σε ένα μονοπάτι του γράφου) και επομένως μπορούν να ανατεθούν στον ίδιο εργάτη
  - **Παρατήρηση:** Ακόμα κι έτσι σε δεδομένη «χρονική στιγμή» (π.χ. επίπεδο του γράφου εξαρτήσεων) ο παραλληλισμός (= ο αριθμός των εργασιών που μπορούν να εκτελεστούν παράλληλα) μπορεί είναι μεγαλύτερος από τον διαθέσιμο αριθμό εργατών.

Ανάθεση εργασιών σε οντότητες εκτέλεσης

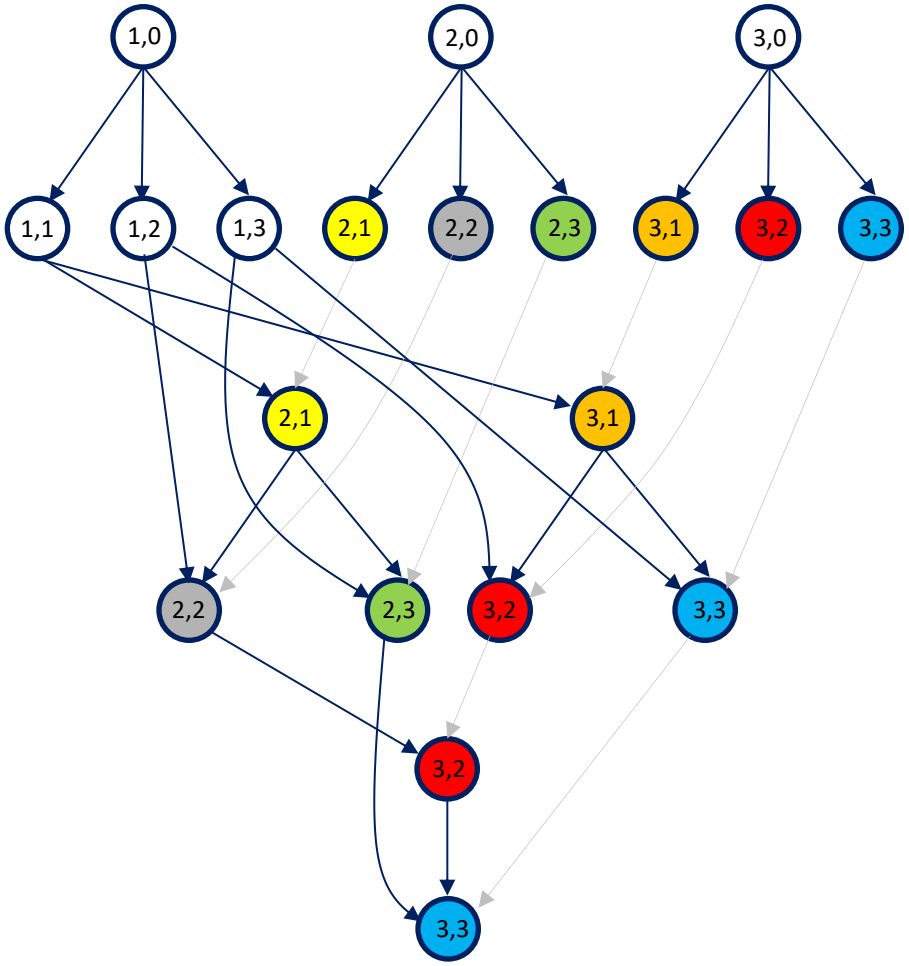


k = 0

k = 1

k = 2

Ανάθεση εργασιών σε οντότητες εκτέλεσης



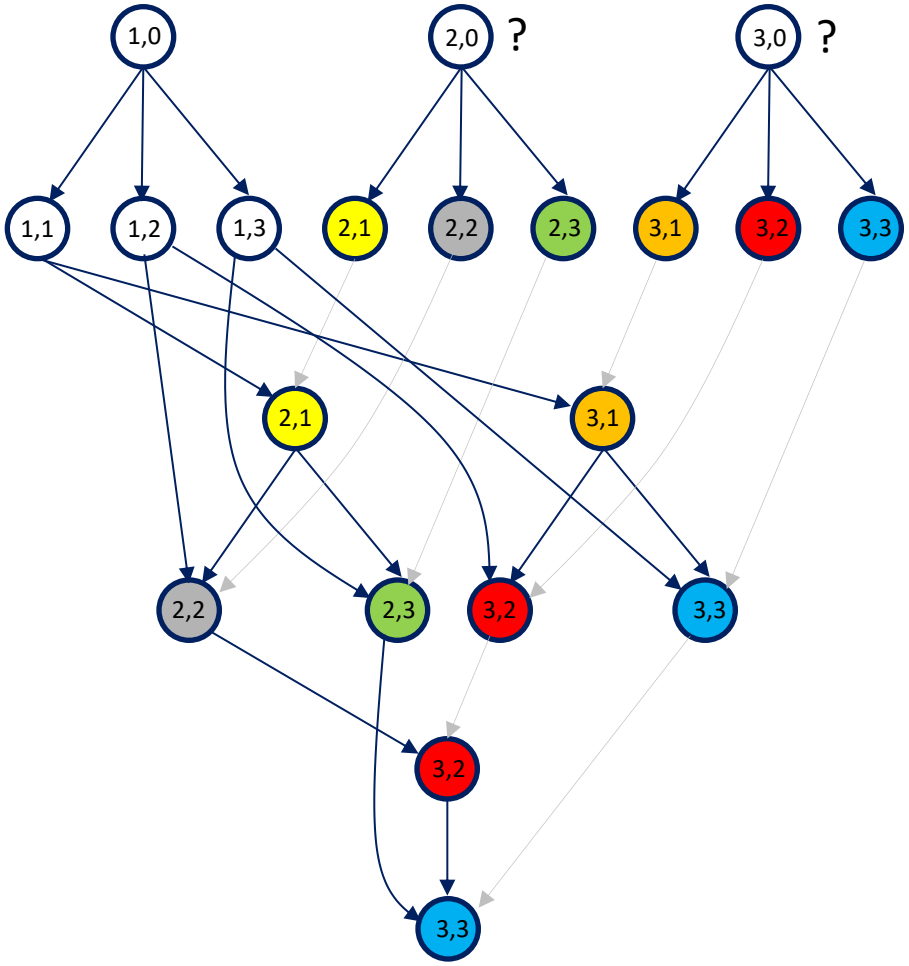
k = 0

k = 1

k = 2

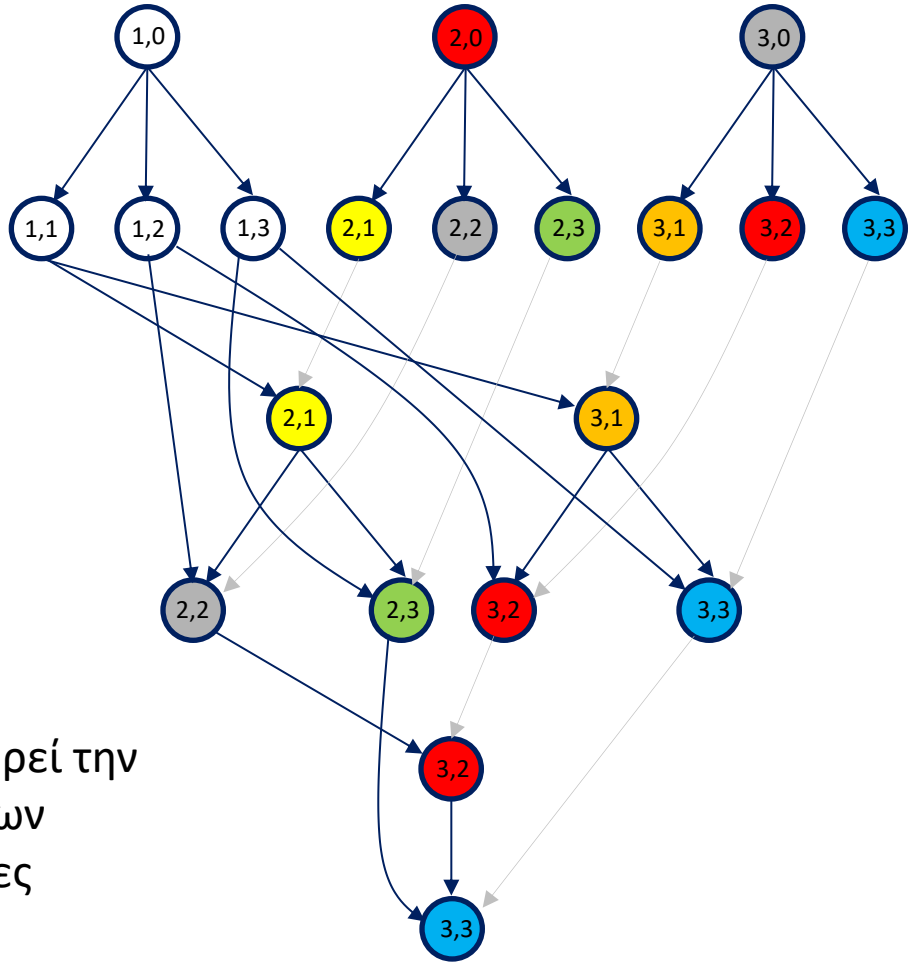
Τοπικότητα / μείωση επικοινωνίας

Ανάθεση εργασιών σε οντότητες εκτέλεσης



Τοπικότητα / μείωση επικοινωνίας

Ανάθεση εργασιών σε οντότητες εκτέλεσης



k = 0

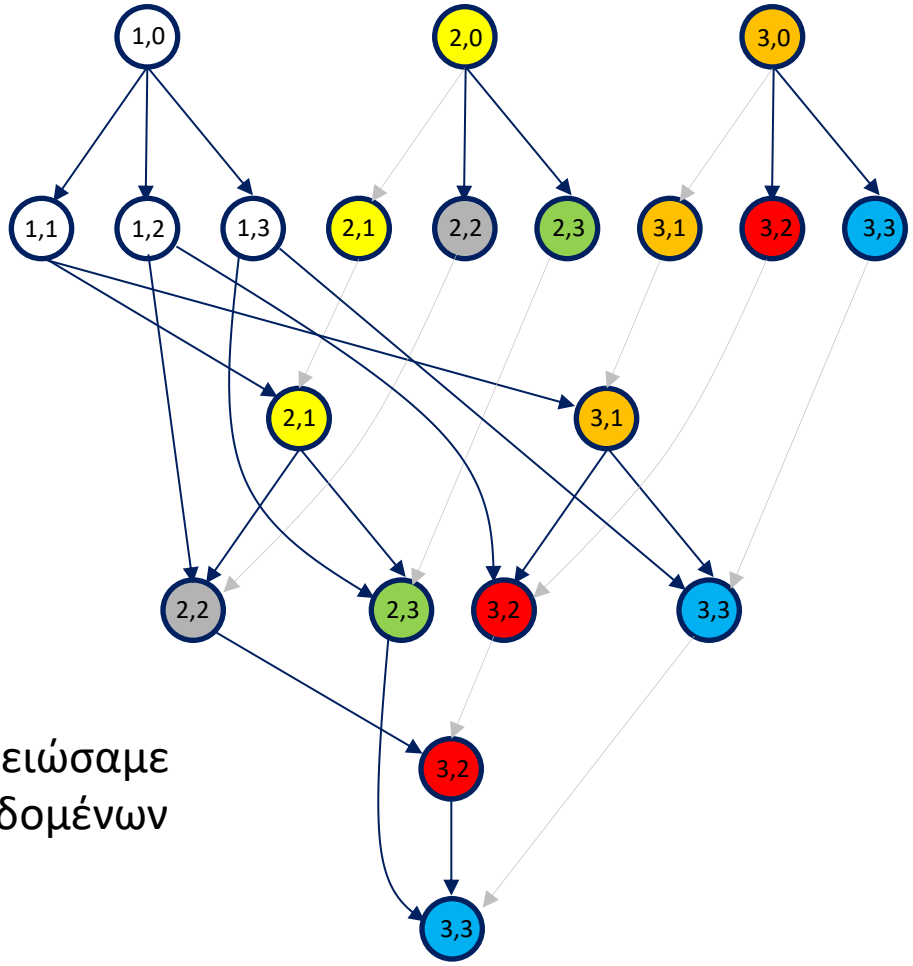
k = 1

k = 2

Τοπικότητα / μείωση επικοινωνίας

Ατυχής επιλογή: διατηρεί την ανάγκη ροής δεδομένων ανάμεσα στους εργάτες

Ανάθεση εργασιών σε οντότητες εκτέλεσης



k = 0

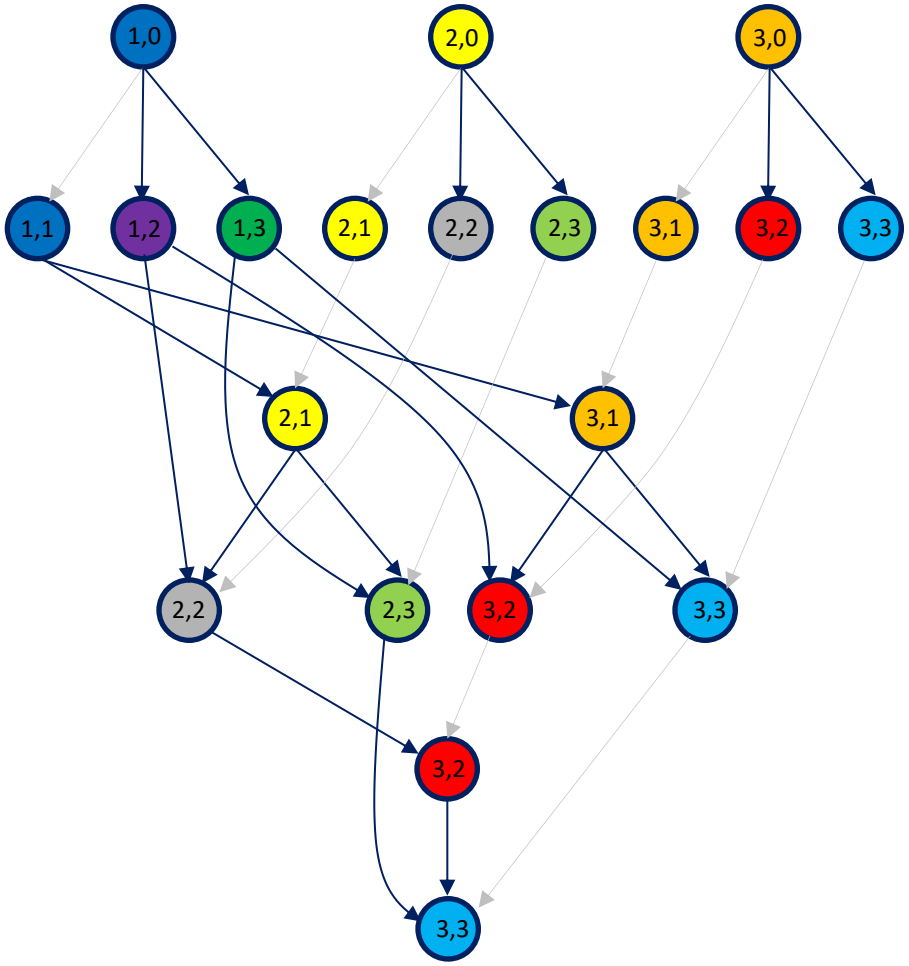
k = 1

k = 2

Τοπικότητα / μείωση επικοινωνίας

Προφανής επιλογή: Μειώσαμε περαιτέρω την ροή δεδομένων

Ανάθεση εργασιών σε οντότητες εκτέλεσης



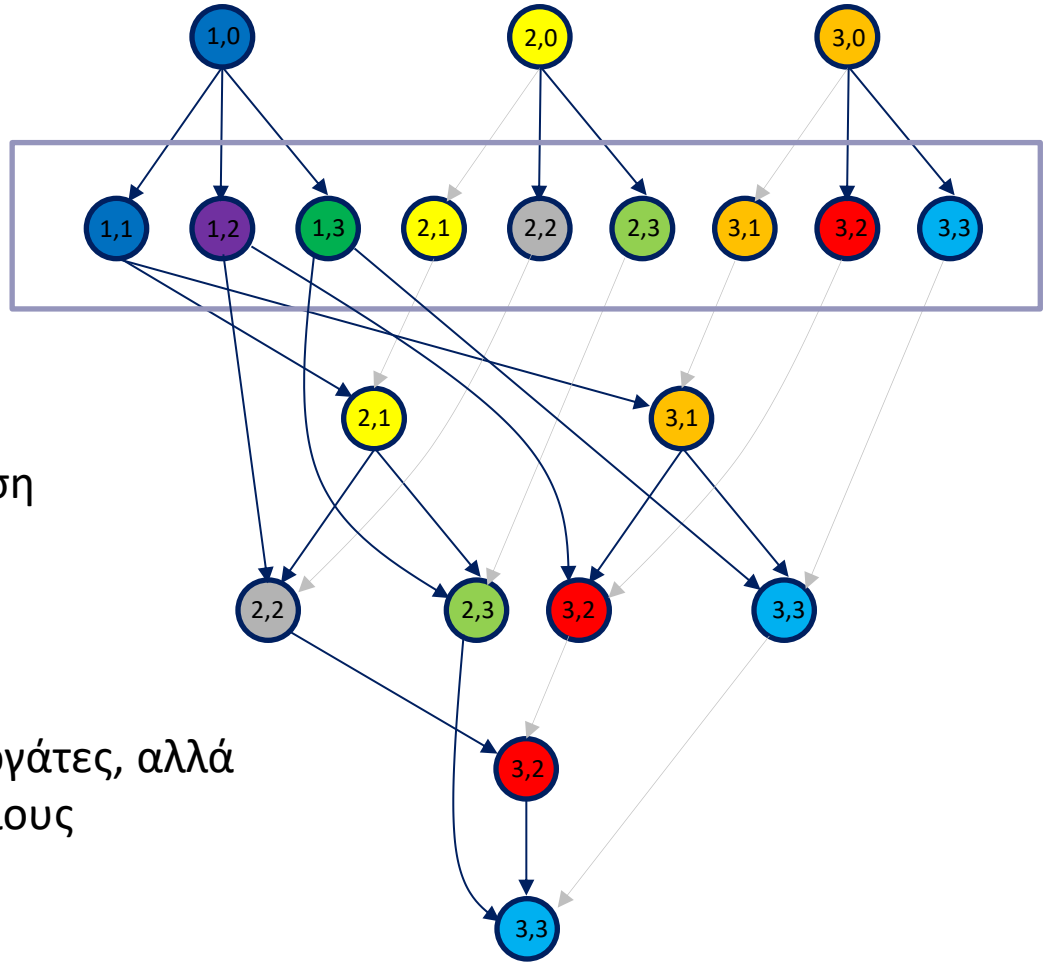
k = 0

k = 1

k = 2

Τοπικότητα / μείωση επικοινωνίας

Ανάθεση εργασιών σε οντότητες εκτέλεσης



k = 0

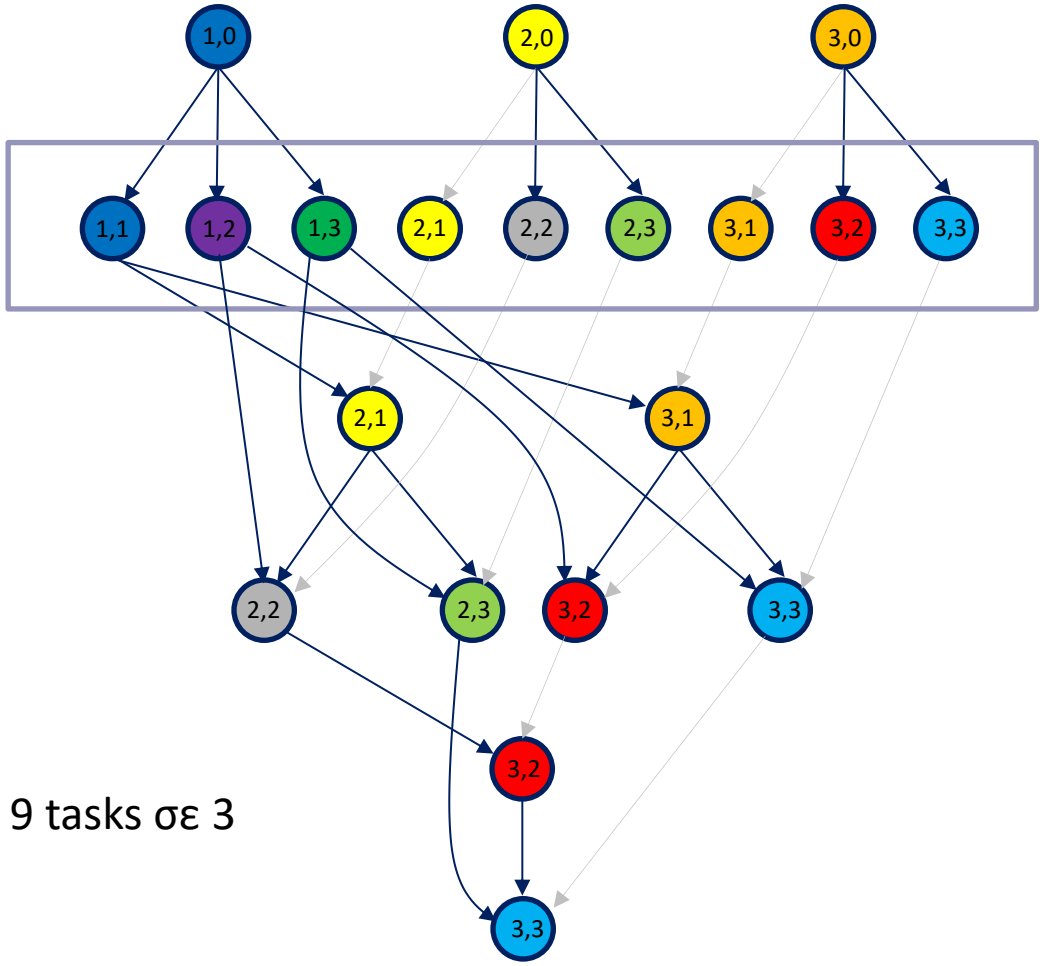
k = 1

k = 2

Τοπικότητα / μείωση επικοινωνίας

Χρειαζόμαστε 9 εργάτες, αλλά έχουμε 3 διαθέσιμους

Ανάθεση εργασιών σε οντότητες εκτέλεσης



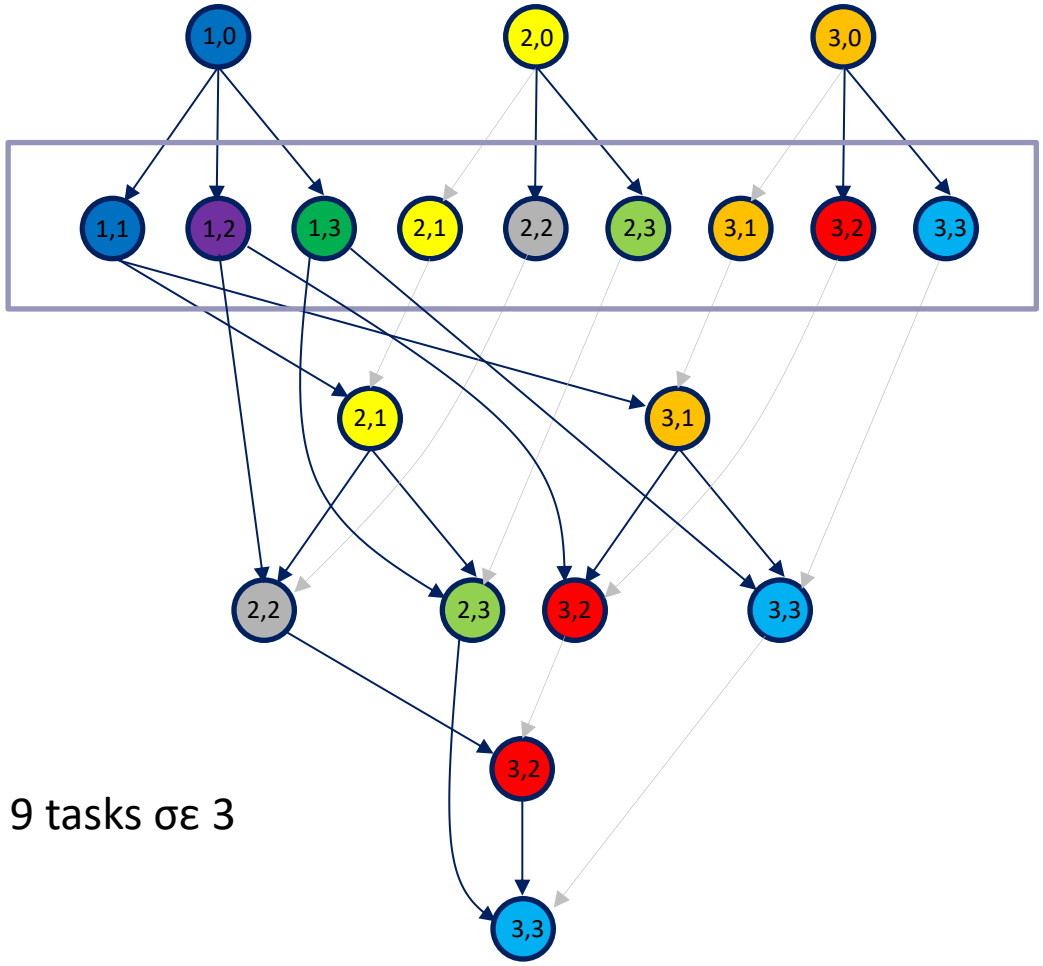
k = 0

k = 1

k = 2

Πώς θα αναθέσω (ομαδοποιήσω) τα 9 tasks σε 3 εργάτες;

Ανάθεση εργασιών σε οντότητες εκτέλεσης



k = 0

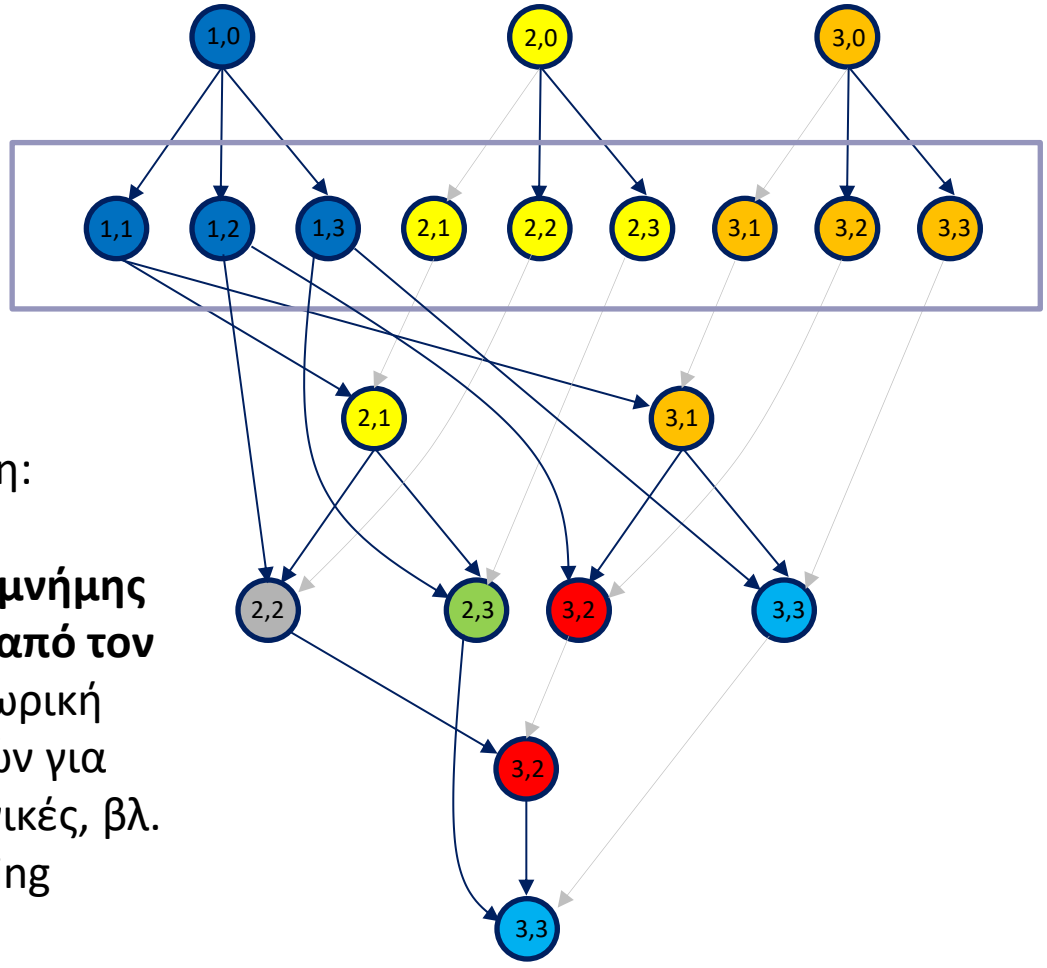
k = 1

k = 2

Πώς θα αναθέσω (ομαδοποιήσω) τα 9 tasks σε 3 εργάτες;

Αλληλεπίδραση με το υλικό

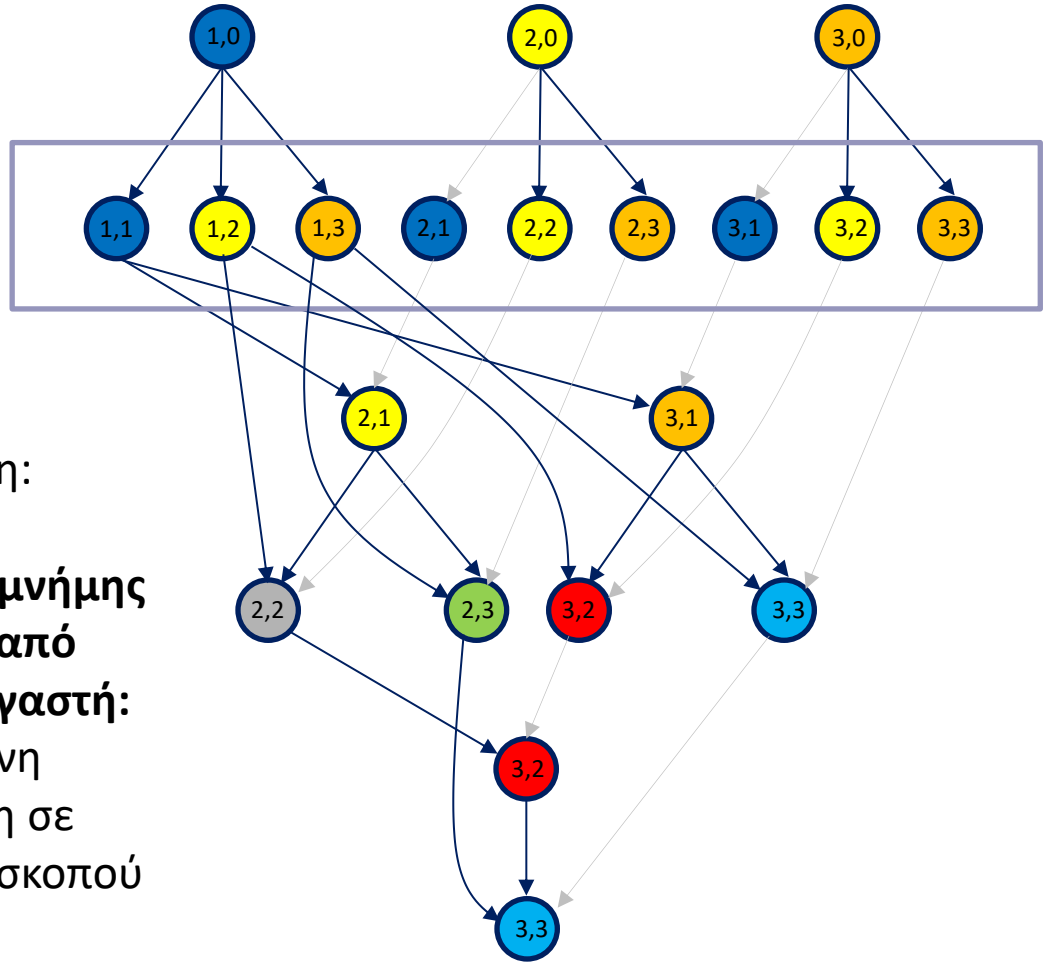
Ανάθεση εργασιών σε οντότητες εκτέλεσης



Πρόσβαση στη μνήμη:

Συνεχόμενες θέσεις μνήμης θα προσπελαστούν από τον ίδιο επεξεργαστή: χωρική τοπικότητα αναφορών για multicore αρχιτεκτονικές, βλ. cache lines, prefetching

Ανάθεση εργασιών σε οντότητες εκτέλεσης

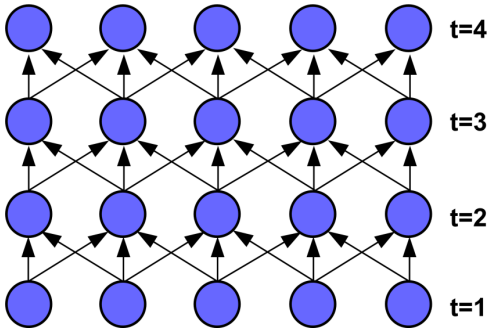


Πρόσβαση στη μνήμη:

Συνεχόμενες θέσεις μνήμης θα προσπελαστούν από διαφορετικό επεξεργαστή: ευνοεί την ταυτόχρονη πρόσβαση στη μνήμη σε επιταχυντές ειδικού σκοπού

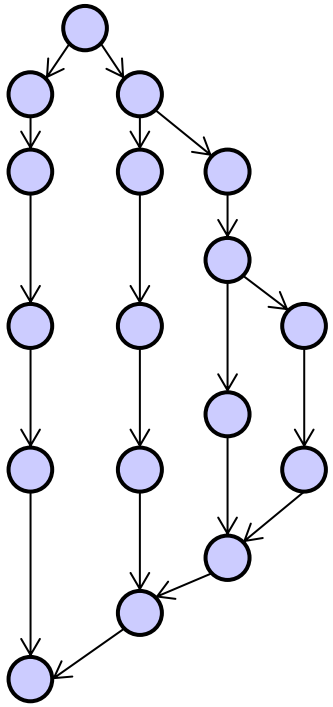
Στατική vs. Δυναμική ανάθεση εργασιών

- **Στατική απεικόνιση:**
  - Διαμοιρασμός των tasks σε threads πριν την έναρξη της εκτέλεσης
  - Καλή πρακτική: Αν η εφαρμογή το επιτρέπει, συνηθίζουμε να έχουμε 1 task / thread σε δεδομένη χρονική στιγμή (προσαρμόζοντας κατάλληλα το σχεδιασμό μας και στο Βήμα 1)
  - Κατάλληλη στρατηγική για εφαρμογές με:
    - ομοιόμορφη και γνωστή εξαρχής κατανομή φορτίου (π.χ. regular task graph)
    - ανομοιόμορφη αλλά προβλέψιμη κατανομή φορτίου
  - Πλεονεκτήματα :
    - Απλή στην υλοποίηση
    - Καλή επίδοση για «κανονικές εφαρμογές»
    - Μηδενικό overhead
  - Μειονεκτήματα:
    - Κακή επίδοση για δυναμικές και ακανόνιστες εφαρμογές
  - **Παράδειγμα:** default, static parallel for στο OpenMP



Στατική vs. Δυναμική ανάθεση εργασιών

- **Δυναμική απεικόνιση:**
  - Διαμοιρασμός των tasks σε threads κατά την εκτέλεση
  - Κατάλληλη στρατηγική για εφαρμογές με:
    - Ακανόνιστο γράφο εργασιών
    - μη προβλέψιμο φορτίο
    - δυναμικά δημιουργούμενα tasks
  - Πλεονεκτήματα:
    - Εξισορρόπηση φορτίου σε δυναμικές και ακανόνιστες εφαρμογές
  - Μειονεκτήματα:
    - Δύσκολη στην υλοποίηση (συνήθως το αναλαμβάνει το run-time σύστημα)
    - Μπορεί να δημιουργήσει bottleneck σε περίπτωση υλοποίησης με ένα κεντρικό scheduling thread (απαιτούνται κατανεμημένοι αλγόριθμοι)
    - Δεν υπάρχει εύκολος έλεγχος στον τρόπο πρόσβασης στα δεδομένα (απώλεια τοπικότητας αναφορών)
  - **Παράδειγμα:** OpenMP parallel for με dynamic scheduling, ή task-based υλοποίηση



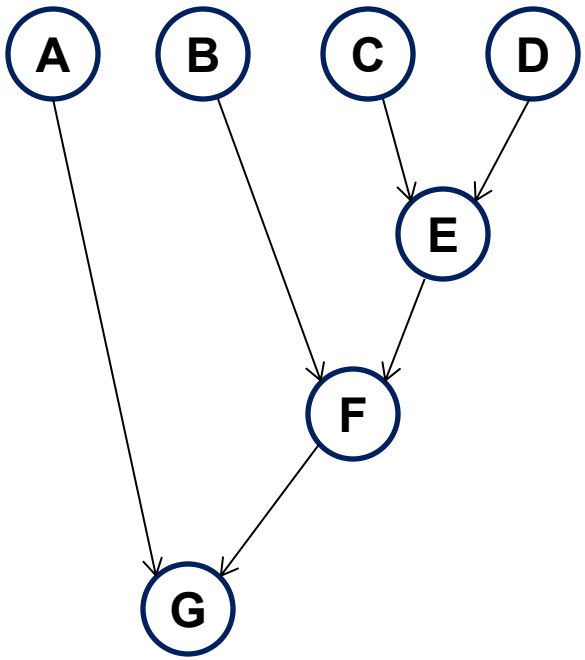
Ζητήματα υλοποίησης

- Υλοποίηση σε προγραμματιστικά μοντέλα **κοινού χώρου διευθύνσεων**
  - **Βήμα 1 και Βήμα 2:** Πως περιγράφω το γράφο εξαρτήσεων
    - fork / join
    - task graph
  - **Βήμα 3:** Χαρακτηρισμός δεδομένων (κοινά δεδομένα)
  - **Βήμα 4:** Ανάθεση εργασιών σε οντότητες εκτέλεσης
    - Ζητήματα Συστημάτων Χρόνου Εκτέλεσης (ΣΧΕ), Work stealing
- Υλοποίηση σε προγραμματιστικά μοντέλα ανταλλαγής μηνυμάτων
  - Συζήτηση

fork / join

- Αφορά εφαρμογές με δυναμική ανάγκη για δημιουργία / τερματισμό tasks
- Τα tasks δημιουργούνται (fork) και τερματίζονται (join) δυναμικά
- Π.χ. αλγόριθμος σχεδιασμένος με την Divide and Conquer στρατηγική
- OpenMP tasks
  - `#pragma omp task`
  - `#pragma omp taskwait`
  - `#pragma omp taskgroup`
- Cilk
  - `spawn`
  - `sync`

Παράδειγμα fork - join




```
#pragma omp parallel
{
   #pragma omp single
   {
      #pragma omp task A();
      #pragma omp task if (0)
      {
             #pragma omp task B();
             #pragma omp task if (0)
             {
                #pragma omp task C();
                D();
                #pragma omp taskwait
                E();
             }
             #pragma omp taskwait
             F();
      }
      #pragma omp taskwait
      G();
    }
}
```

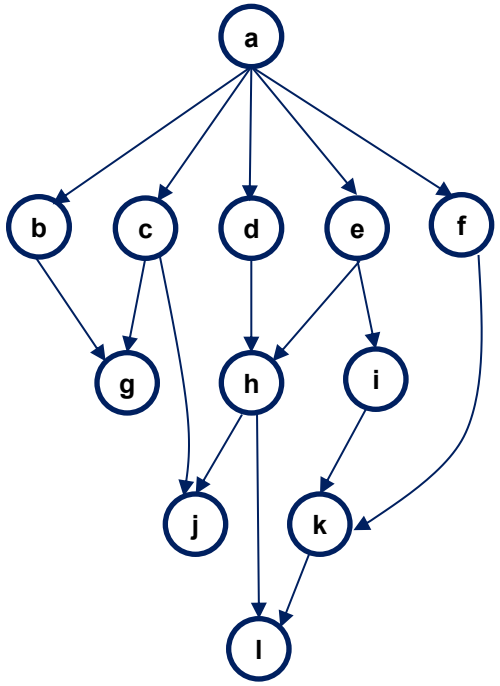
Παράδειγμα: LU decomposition με tasks

```
//LU decomposition kernel
for (k = 0; k < N-1; k++) {
    for(i = k+1; i < N; i++)
      task {
         L[i] = A[i][k] / A[k][k];
         for(j = k+1; j < N; j++)
         A[i][j] = A[i][j] - A[i][k] * A[k][j];
      }
    taskwait
}
```

Ποιος γράφος εκτελείται σε αυτή την περίπτωση;

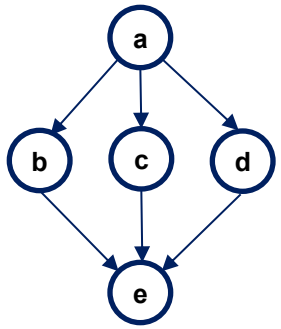
Task graphs

- Η fork-join δομή δεν μπορεί να περιγράψει όλα τα task graphs. Π.χ. η υλοποίηση του παρακάτω γράφου με fork-join οδηγεί σε πιο περιοριστική εκτέλεση.



Περιορισμοί του μοντέλου fork-join

- Η fork-join δομή δεν είναι κατάλληλη για την περιγραφή οποιουδήποτε task graph
- Επιβάλλει να υπάρχει ένα σημείο συγχρονισμό για κάθε task (ή ομάδα tasks)

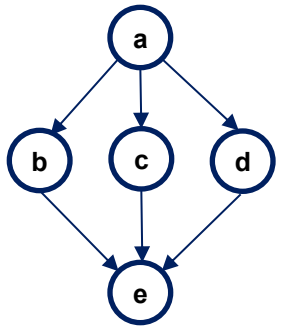


Περιορισμοί του μοντέλου fork-join

- Η fork-join δομή δεν είναι κατάλληλη για την περιγραφή οποιουδήποτε task graph
- Επιβάλλει να υπάρχει ένα σημείο συγχρονισμού για κάθε task (ή ομάδα tasks)

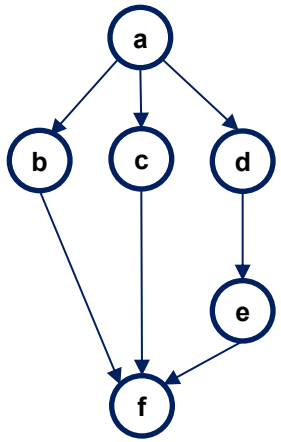
```
a();
task b();
task c();
task d();
taskwait();
task d();
```

OK



Περιορισμοί του μοντέλου fork-join

- Η fork-join δομή δεν είναι κατάλληλη για την περιγραφή οποιουδήποτε task graph
- Επιβάλλει να υπάρχει ένα σημείο συγχρονισμό για κάθε task (ή ομάδα tasks)

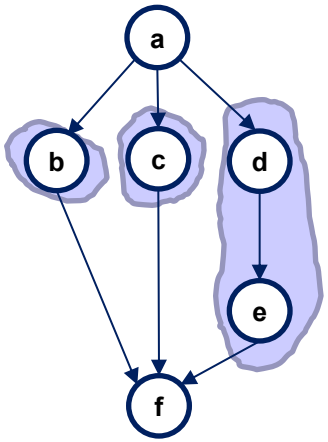


Περιορισμοί του μοντέλου fork-join

- Η fork-join δομή δεν είναι κατάλληλη για την περιγραφή οποιουδήποτε task graph
- Επιβάλλει να υπάρχει ένα σημείο συγχρονισμό για κάθε task (ή ομάδα tasks)

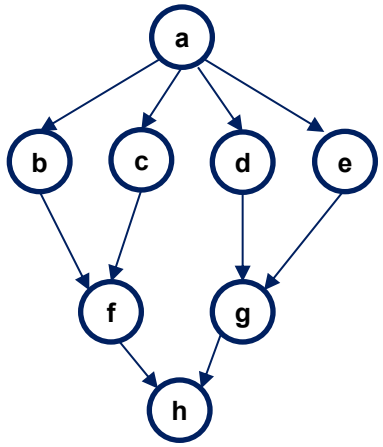
```
a();
task b();
task c();
task {d(); e()}
taskwait();
task f();
```

OK



Περιορισμοί του μοντέλου fork-join

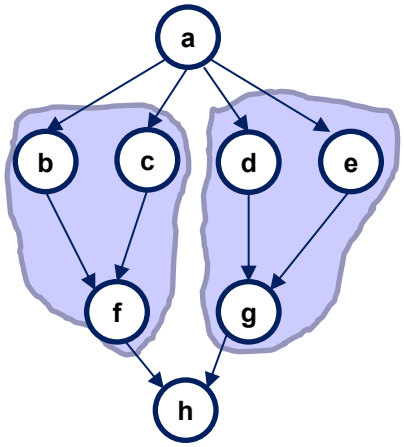
- Η fork-join δομή δεν είναι κατάλληλη για την περιγραφή οποιουδήποτε task graph
- Επιβάλλει να υπάρχει ένα σημείο συγχρονισμό για κάθε task (ή ομάδα tasks)



Περιορισμοί του μοντέλου fork-join

- Η fork-join δομή δεν είναι κατάλληλη για την περιγραφή οποιουδήποτε task graph
- Επιβάλλει να υπάρχει ένα σημείο συγχρονισμό για κάθε task (ή ομάδα tasks)

```
a();
task {
  task b();
  task c();
  taskwait;
  f();
}
task {
  task d();
  task e();
  taskwait;
  g();
}
taskwait;
h();
```

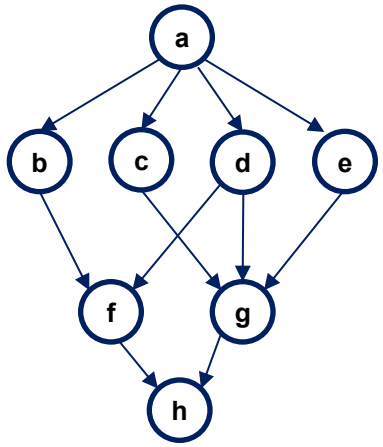


OK

Περιορισμοί του μοντέλου fork-join

- Η fork-join δομή δεν είναι κατάλληλη για την περιγραφή οποιουδήποτε task graph
- Επιβάλλει να υπάρχει ένα σημείο συγχρονισμό για κάθε task (ή ομάδα tasks)

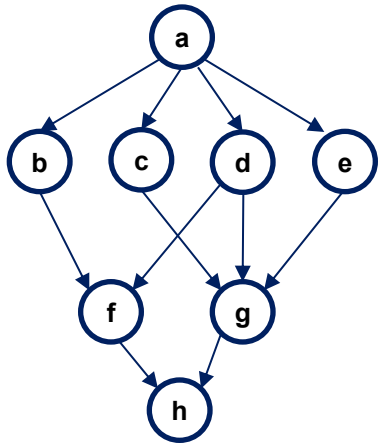
?



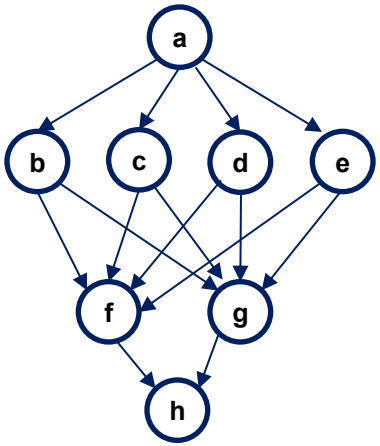
Περιορισμοί του μοντέλου fork-join

- Η fork-join δομή δεν είναι κατάλληλη για την περιγραφή οποιουδήποτε task graph
- Επιβάλλει να υπάρχει ένα σημείο συγχρονισμό για κάθε task (ή ομάδα tasks)

```
a();
task b();
task c();
task d();
task e();
taskwait;
task f();
task g();
taskwait;
h();
```



Αρχικός γράφος



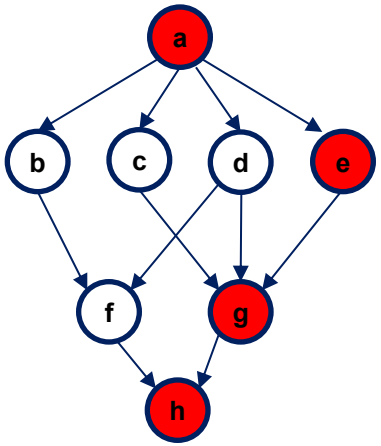
Γράφος fork/join

Περιορισμοί του μοντέλου fork-join

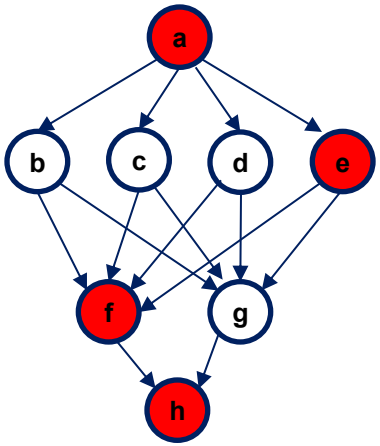
- Η fork-join δομή δεν είναι κατάλληλη για την περιγραφή οποιουδήποτε task graph
- Επιβάλλει να υπάρχει ένα σημείο συγχρονισμό για κάθε task (ή ομάδα tasks)

```
a();
task b();
task c();
task d();
task e();
taskwait;
task f();
task g();
taskwait;
h();
```

Αν a = b = c = d = g = 1,
e = f = 100



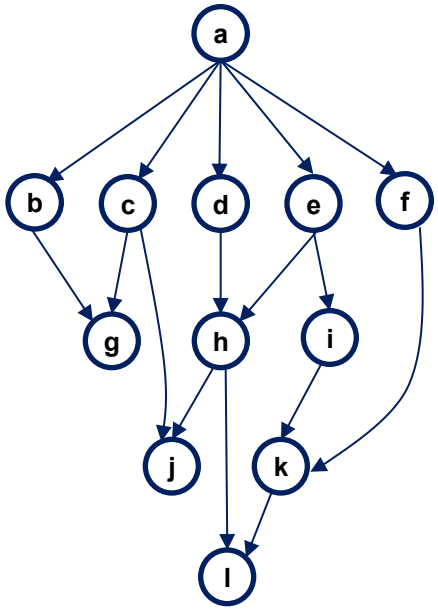
Αρχικός γράφος
Κρίσιμο μονοπάτι: **103**



Γράφος fork/join
Κρίσιμο μονοπάτι: **202**

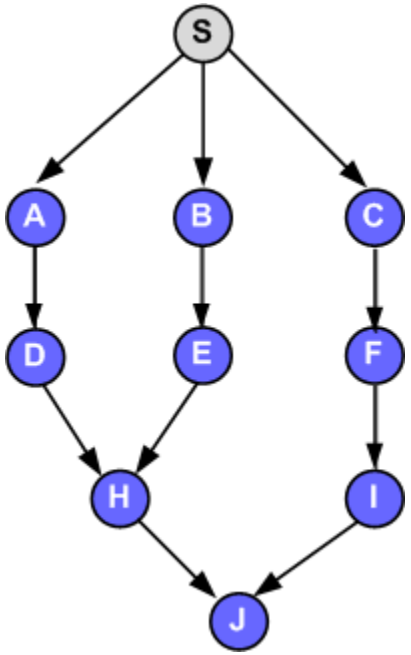
Περιορισμοί του μοντέλου fork-join

- Υπάρχουν πολλές περιπτώσεις γράφων με σύνθετες εξαρτήσεις, όπου το fork-join μοντέλο οδηγεί σε περιοριστική εκτέλεση
- Υπάρχουν εργαλεία παράλληλου προγραμματισμού που επιτρέπουν την έκφραση task graphs με δήλωση του γράφου
  - Τα tasks δηλώνονται ως κορυφές του γράφου
  - Οι εξαρτήσεις μεταξύ των tasks δηλώνονται ως ακμές του γράφου
- Η βιβλιοθήκη των Threading Building Blocks της C++ προσφέρει τη διεπαφή *flow graph* για την παραλληλοποίηση οποιουδήποτε task graph



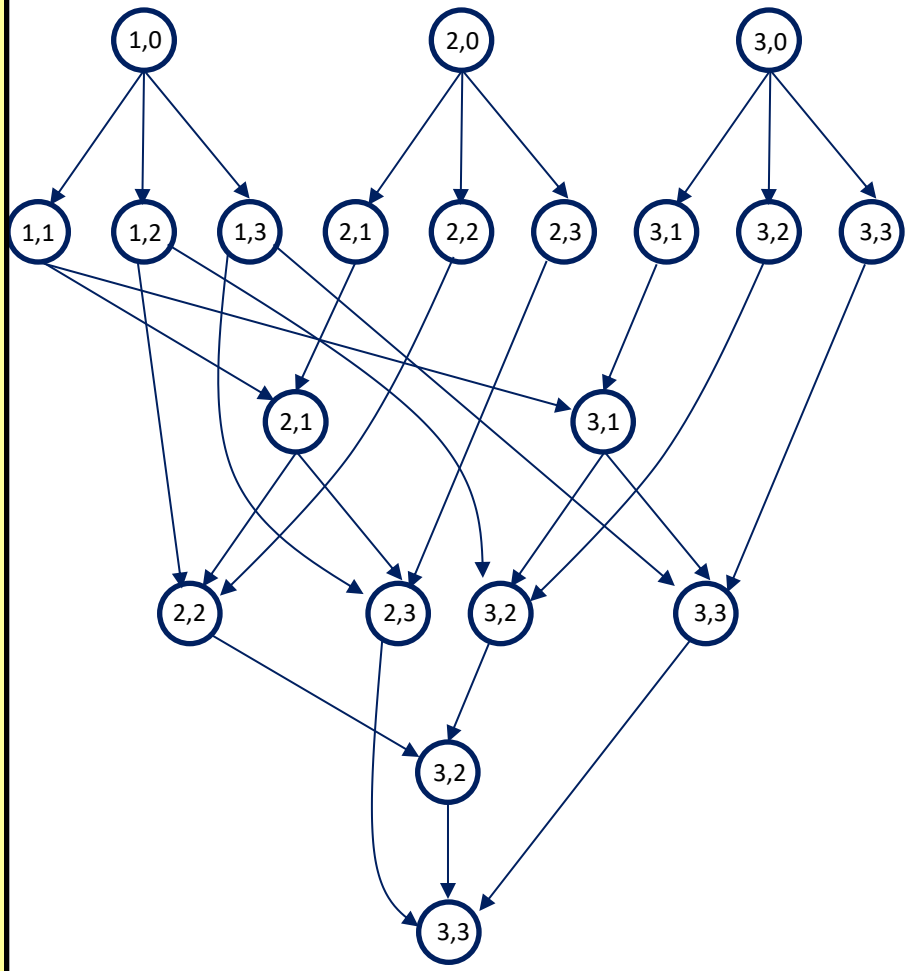
```
graph g;
source_node s;  //source node

//each node executes a different function body
node a( g, body_A() );
node b( g, body_B() );
node c( g, body_C() );
node d( g, body_D() );
node e( g, body_E() );
node f( g, body_F() );
node h( g, body_H() );
node i( g, body_I() );
node j( g, body_J() );

//create edges
make_edge( s, a );
make_edge( s, b );
make_edge( s, c );
make_edge( a, d );
make_edge( b, e );
make_edge( c, f );
make_edge( d, h );
make_edge( e, h );
make_edge( f, i );
make_edge( h, j );
make_edge( i, j );

s.start(); // start parallel execution of task graph
g.wait_for_all();  //wait for all to complete
```

Παράδειγμα task graph



Παράδειγμα: Task graph για LU decomposition με Threading Building Blocks (ψευδοκώδικας)

```
namespace tbb::flow
graph g;
broadcast_node s;  //source node
                   //to start execution
for (k = 0 ; k < N - 1 ; k++)
  for (i = k+1 ; i < N ; i++) {
      node(k,i,j) = new continue_node {g,
          A[i][k] = A[i][k]/A[k][k];
      }
      if (k == 0)
         make_edge(s, node(k,i,j));
      else {
         make_edge(node(k-1,i,k), node(k,i,k));
         make_edge(node(k-1,k,k), node(k,i,k));
      }
      for(j = k+1 ; j < N ; j++) {
         node(k,i,j) = new continue_node {g,
            A[i][j] -= A[i][k]*A[k][j];
         }
         make_edge(node(k,i,k), node(k,i,j)
         if (k > 0)
            make_edge(node(k-1,i,j), node(k,i,j));
            make_edge(node(k-1,k,j), node(k,i,j));
      }
    }
 s.try_put();         //fire!
 g.wait_for_all();  //and wait for everything
```



Ζητήματα ΣΧΕ:
Ανάθεση εργασιών σε οντότητες εκτέλεσης

- Το στάδιο αυτό αναλαμβάνει να αναθέσει εργασίες (tasks) σε «εργάτες» ή 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
- Στην τυπική περίπτωση οι οντότητες εκτέλεσης είναι συγκεκριμένες και σταθερές για ένα σύστημα, π.χ. ίσες με τον αριθμό των πυρήνων (cores) του επεξεργαστή.
- Άρα το πρόβλημα είναι πως θα ομαδοποιήσουμε/απεικονίσουμε τις εργασίες ενός γράφου εξαρτήσεων σε διεργασίες ή νήματα εκτέλεσης (ποιος εργάτης θα πάρει ποια δουλειά)

Χρονοδρομολόγηση εργασιών

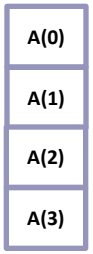
- Αριθμός εργασιών *T*
- Αριθμός διεργασιών / νημάτων *P*
- Κάθε εργασία μπορεί:
  - Να παράξει άλλες εργασίες
  - Να περιμένει τα παιδιά της
- Στόχοι χρονοδρομολογητή:
  - Ισοκατανομή φορτίου
  - Αποδοτική χρήση των πόρων
  - Μικρή επιβάρυνση

Χρονοδρομολόγηση εργασιών

- Δύο βασικές στρατηγικές:
  - **Work sharing:** Όταν δημιουργούνται νέες εργασίες ο ΧΔ προσπαθεί να τις "στείλει" σε ανενεργούς επεξεργαστές.
  - **Work stealing:** Οι ανενεργοί επεξεργαστές προσπαθούν να "κλέψουν" εργασίες.
- Γενικά προτιμάται η τακτική work stealing
  - καλύτερο locality
  - μικρότερη επιβάρυνση συγχρονισμού
  - βέλτιστα θεωρητικά όρια ως προς χρόνο, χώρο [Blumofe and Leiserson '99]

work stealing

P1 P2 P3 P4

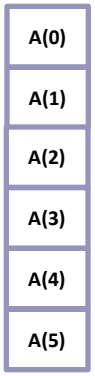



```
#pragma omp parallel
{
   #pragma omp single
   {
      for (i = 0; i < n; i++)
      #pragma omp task A(i);
      #pragma omp taskwait
   }
}
```

task creation

work stealing

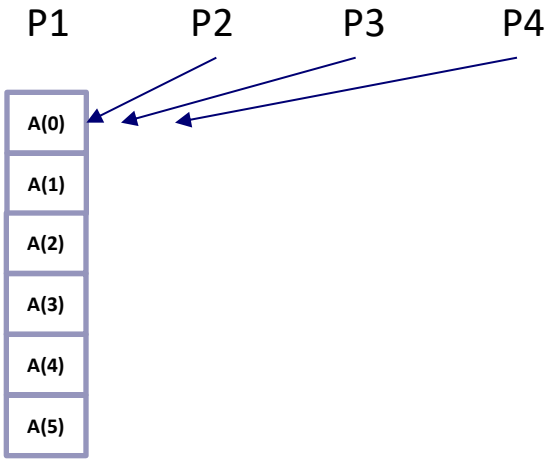
P1 P2 P3 P4



task creation

```
#pragma omp parallel
{
   #pragma omp single
   {
      for (i = 0; i < n; i++)
      #pragma omp task A(i);
      #pragma omp taskwait
   }
}
```

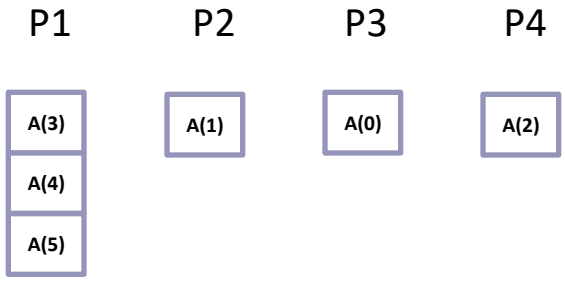
work stealing




```
#pragma omp parallel
{
    #pragma omp single
    {
        for (i = 0; i < n; i++)
        #pragma omp task A(i);
        #pragma omp taskwait
    }
}
```

work stealing: τυχαία επιλογή μη κενής ουράς (εδώ υπάρχει μόνο μία)
ανάγκη συγχρονισμού για ορθή αφαίρεση
αφαίρεση από την κορυφή (παλαιότερες εργασίες –σεβασμός τοπικότητας της διεργασίας)

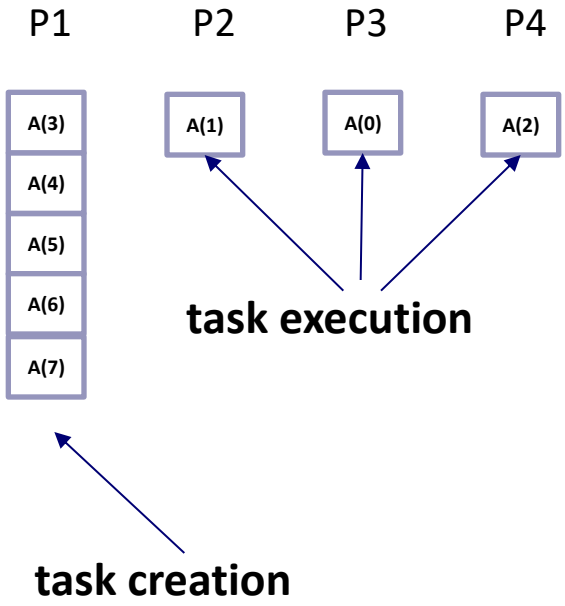
work stealing




```
#pragma omp parallel
{
   #pragma omp single
   {
      for (i = 0; i < n; i++)
      #pragma omp task A(i);
      #pragma omp taskwait
   }
}
```

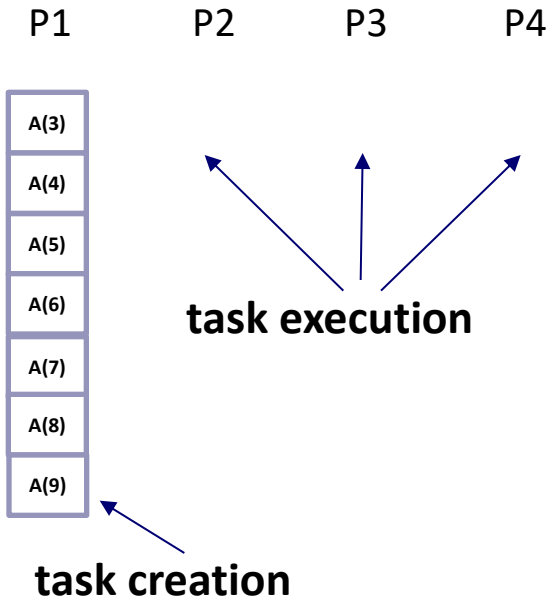
work stealing: τυχαία επιλογή μη κενής ουράς (εδώ υπάρχει μόνο μία)
ανάγκη συγχρονισμού για ορθή αφαίρεση
αφαίρεση από την κορυφή (παλαιότερες εργασίες –σεβασμός τοπικότητας της διεργασίας)

work stealing



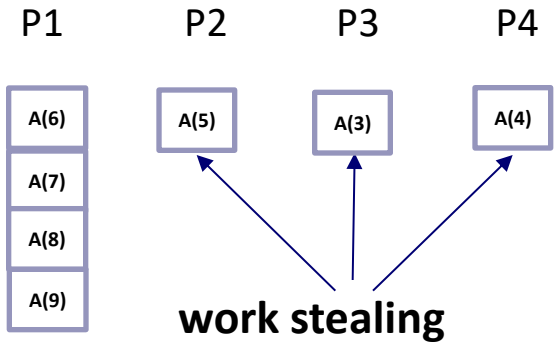

```
#pragma omp parallel
{
   #pragma omp single
   {
      for (i = 0; i < n; i++)
      #pragma omp task A(i);
      #pragma omp taskwait
   }
}
```

work stealing



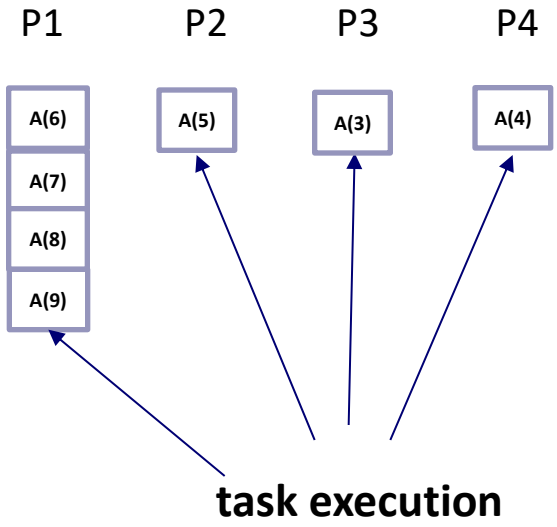

```
#pragma omp parallel
{
    #pragma omp single
    {
        for (i = 0; i < n; i++)
        #pragma omp task A(i);
        #pragma omp taskwait
    }
}
```

work stealing



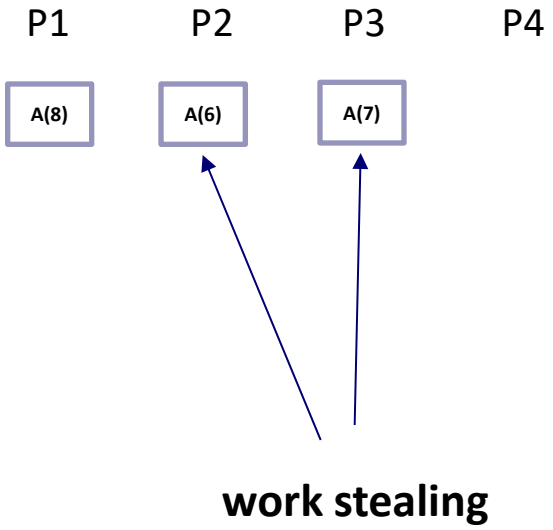

```
#pragma omp parallel
{
   #pragma omp single
   {
      for (i = 0; i < n; i++)
      #pragma omp task A(i);
      #pragma omp taskwait
   }
}
```

work stealing



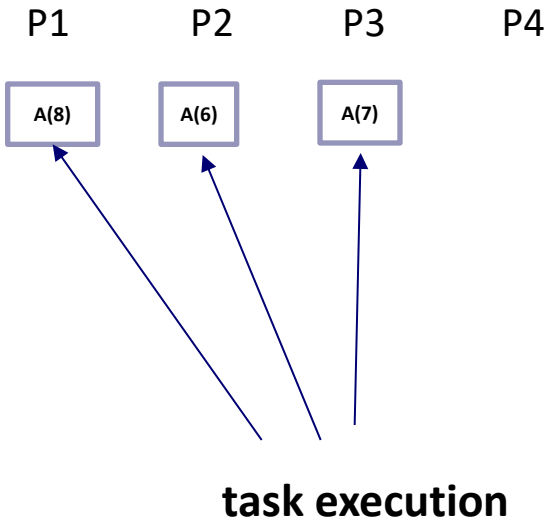

```
#pragma omp parallel
{
   #pragma omp single
   {
      for (i = 0; i < n; i++)
      #pragma omp task A(i);
      #pragma omp taskwait
   }
}
```

work stealing



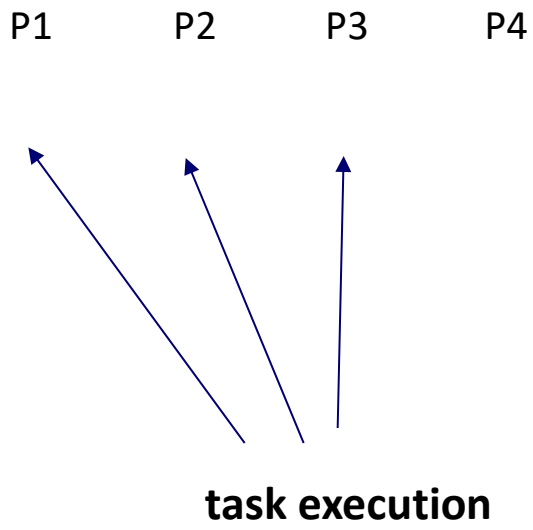

```
#pragma omp parallel
{
    #pragma omp single
    {
        for (i = 0; i < n; i++)
        #pragma omp task A(i);
        #pragma omp taskwait
    }
}
```

work stealing




```
#pragma omp parallel
{
   #pragma omp single
   {
      for (i = 0; i < n; i++)
      #pragma omp task A(i);
      #pragma omp taskwait
   }
}
```

work stealing




```
#pragma omp parallel
{
    #pragma omp single
    {
        for (i = 0; i < n; i++)
        #pragma omp task A(i);
        #pragma omp taskwait
    }
}
```

work stealing

Παράλληλη δημιουργία tasks

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < n; i++)
        #pragma omp task A(i);
    #pragma omp taskwait
}
```

Υλοποίηση task parallelism σε προγραμματιστικά μοντέλα ανταλλαγής μηνυμάτων

- Ο κοινός χώρος διευθύνσεων που υποθέσαμε μέχρι στιγμής διευκόλυνε την υλοποίηση γιατί:
  - Δεν ασχολούμαστε με τη ροή δεδομένων ανάμεσα σε εξαρτώμενα tasks (αρκεί να επιβάλουμε την κατάλληλη σειριοποίηση)
  - Δεν ασχολούμαστε με το που βρίσκονται τα read-only δεδομένα
  - **Σημείωση:** Χρειάζεται βέβαια προσοχή για τα race conditions
  - Το ΣΧΕ έχει επίσης κοινό χώρο διευθύνσεων → ένας «κυβερνήτης» για όλη την εκτέλεση (βλ. εύκολη ισοκατανομή φορτίου με work stealing)

Υλοποίηση task parallelism σε προγραμματιστικά μοντέλα ανταλλαγής μηνυμάτων

- Σε προγραμματιστικό μοντέλο ανταλλαγής μηνυμάτων τα δεδομένα του προβλήματος είναι είτε κατανεμημένα είτε αντιγραμμένα
  - Απαιτείται αποστολή/λήψη μηνύματος για τις ακμές του γράφου εξαρτήσεων αν οι εργασίες πηγής και προορισμού έχουν ανατεθεί σε διαφορετική διεργασία
  - Απαιτείται αποστολή/λήψη μηνυμάτων ανάμεσα στο χρήστη και τον ιδιοκτήτη read-only δεδομένων αν αυτά δεν μπορούν να αντιγραφούν σε όλες τις διεργασίες
  - Απαιτείται ειδική μέριμνα όταν χρειάζεται εγγραφή από πολλές διεργασίες σε κατανεμημένα δεδομένα (π.χ. επιπλέον reduction)
  - Το ΣΧΕ ΔΕΝ έχει κοινό χώρο διευθύνσεων → πολλοί «κυβερνήτες» για όλη την εκτέλεση (βλ. πιο δύσκολη υλοποίηση τεχνικών όπως το work stealing)
- Για τους παραπάνω λόγους τόσο η υλοποίηση ενός ΣΧΕ όσο και η υλοποίηση ενός προγράμματος με τη χρήση του ΣΧΕ είναι αρκετά πολύπλοκες (βλ. παράδειγμα Charm++ https://charmplusplus.org/)

Ερωτήσεις;