Cilk: Φιλοσοφία και Χρήση

Παράλληλα Συστήματα Επεξεργασίας
9ο εξάμηνο ΣΗΜΜΥ
ακ. έτος 2010-2011

`http://www.cslab.ece.ntua.gr/courses/pps`

Εργαστήριο Υπολογιστικών Συστημάτων
ΕΜΠ

Νοέμβριος 2010

Περιεχόμενα

- Εισαγωγή
- Cilk: Φιλοσοφία και Χρήση
- Cilk: Θέματα υλοποίησης

Το τζάμπα γεύμα

The free lunch

Το τζάμπα γεύμα:

- Εκθετική αύξηση της σειριακής επίδοσης των CPUs (frequency scaling, ILP exploitation)
- Εκθετική αύξηση στον αριθμό των transistors (νόμος Moore)

Το τζάμπα γεύμα **τελείωσε!**

The free lunch **is over!**

Το τζάμπα γεύμα:

- Εκθετική αύξηση της σειριακής επίδοσης των CPUs (frequency scaling, ILP exploitation)
- Εκθετική αύξηση στον αριθμό των transistors (νόμος Moore)

Τελείωσε!

- Οι αρχιτέκτονες έφθασαν σε φυσικά όρια
- Λύση: πολυπύρηνοι επεξεργαστές
- νόμος Moore $\leftrightarrow$ εκθετική αύξηση πυρήνων

the "Multicore Era"

όπου μόνο παράλληλα προγράμματα εκμεταλλεύονται το νέο υλικό

Ο παράλληλος προγραμματισμός είναι δύσκολος

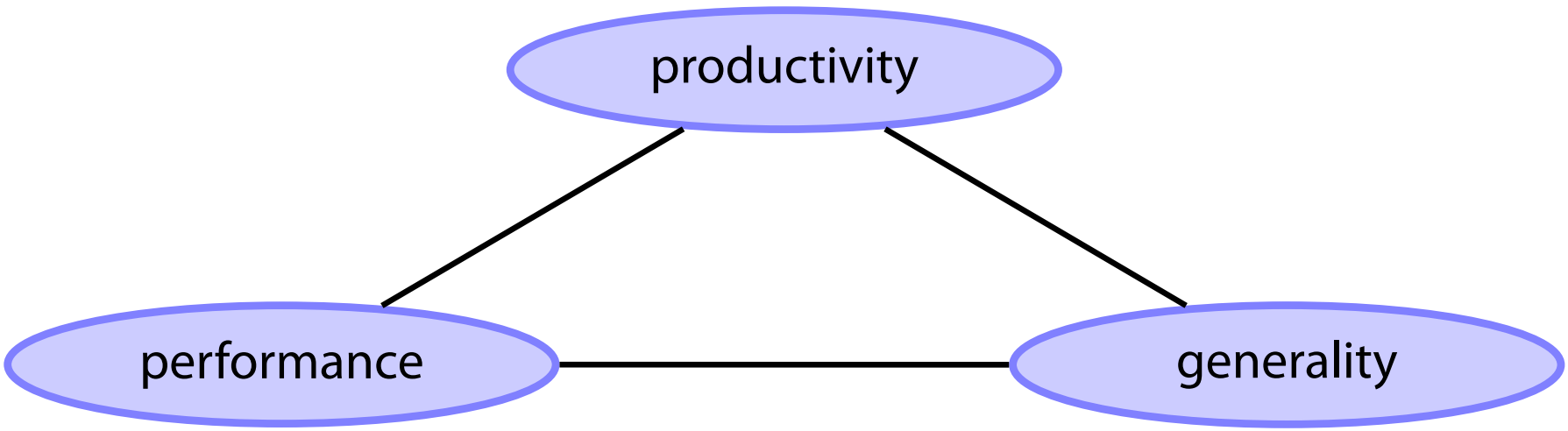
- Δύσκολο να επιχειρηματολογίσουμε για ορθότητα
  (π.χ., data races)
- Ο παράλληλος προγραμματισμός είναι "εσωστρεφής τέχνη"
- Έλλειψη εργασλείων
  (debuggers, profiles, languages, ...)

Οπότε τα τελευταία χρόνια:

- έμφαση στο να γίνει ο παράλληλος προγραμματισμός:
  - **εύκολος**
  - **λιγότερο επιρρεπής σε λάθη**
- Νέες γλώσσες, μοντέλα, κλπ

Παράλληλος προγραμματισμός

Is Parallel Programming Hard, And If So, Why? [McKenney et. al. '09]



- 2 από τα 3!

Μοντέλα έκφρασης παραλληλισμού

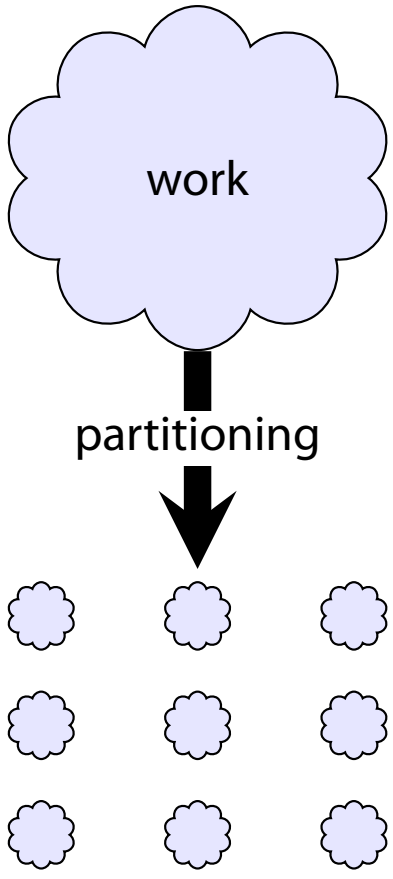
- **Data parallel**
  Μια πράξη εφαρμόζεται ταυτόχρονα σε συλλογικότητες δεδομένων (π.χ. πίνακες).
  (παραγωγικό, όχι γενικό)

- **Task parallel**
  Ο χρήστης ορίζει ρητά τα παράλληλα κομμάτια εργασίας (parallel tasks)
  (γενικό, όχι παραγωγικό)

Βήματα παράλληλης εκτέλεσης

διαμοιρασμός εργασίας (έκφραση παραλληλισμού)

Η συνολική **δουλεία** πρέπει να χωριστεί σε **παράλληλες εργασίες** (parallel tasks)



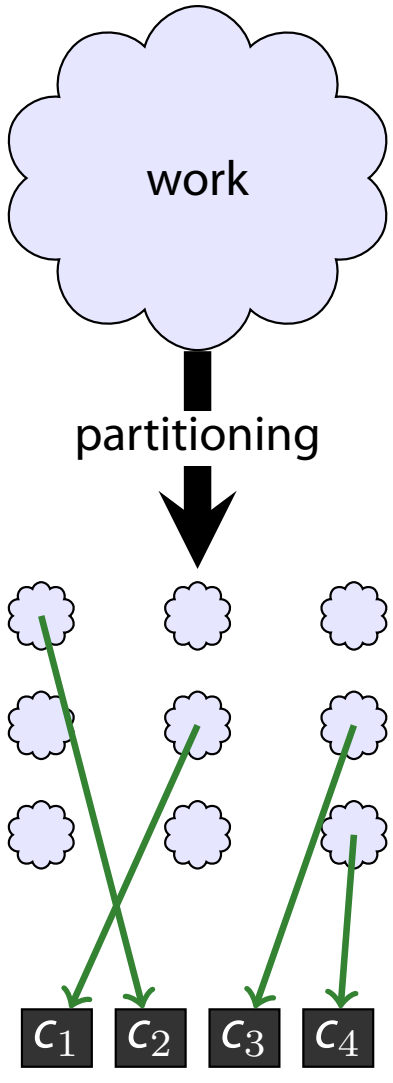
Βήματα παράλληλης εκτέλεσης

διαμοιρασμός εργασίας (έκφραση παραλληλισμού)

Η συνολική **δουλεία** πρέπει να χωριστεί σε **παράλληλες εργασίες** (parallel tasks)

χρονοδρομολόγηση (scheduling)

Οι εργασίες πρέπει να τοποθετηθούν σε επεξεργαστές



Βήματα παράλληλης εκτέλεσης

διαμοιρασμός εργασίας (έκφραση παραλληλισμού)

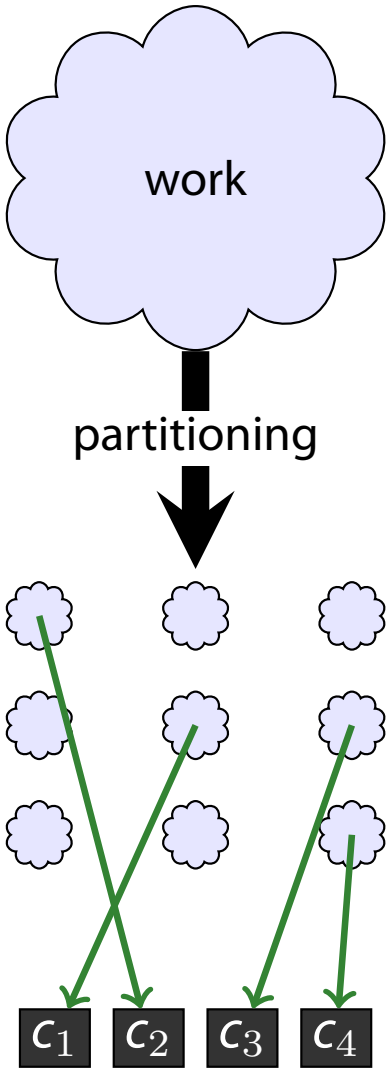
Η συνολική **δουλεία** πρέπει να χωριστεί σε **παράλληλες εργασίες** (parallel tasks)

(data parallel: σύστημα, task parallel: χρήστης)

χρονοδρομολόγηση (scheduling)

Οι εργασίες πρέπει να τοποθετηθούν σε επεξεργαστές

(σύστημα)



Βήματα παράλληλης εκτέλεσης

διαμοιρασμός εργασίας (έκφραση παραλληλισμού)

Η συνολική **δουλεία** πρέπει να χωριστεί σε **παράλληλες εργασίες** (parallel tasks)

(data parallel: σύστημα, task parallel: χρήστης)

χρονοδρομολόγηση (scheduling)

Οι εργασίες πρέπει να τοποθετηθούν σε επεξεργαστές

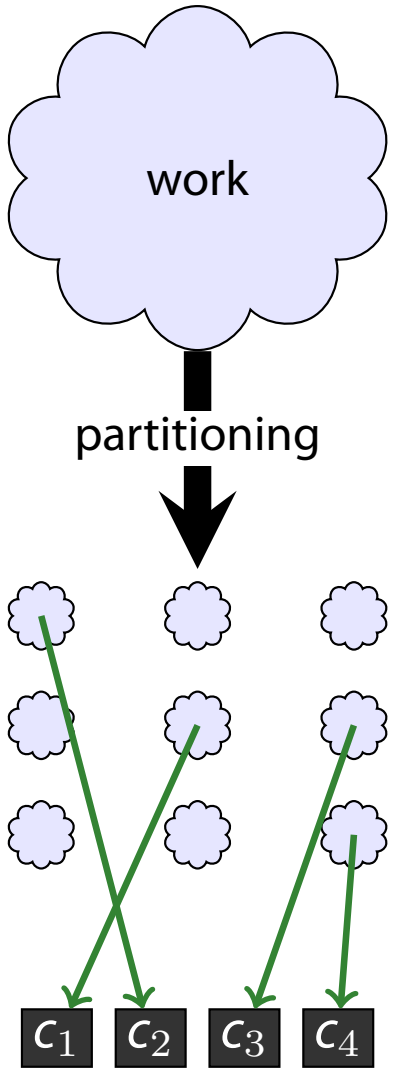
(σύστημα)

μέγεθος παράλληλων εργασιών (task granularity)

ποσότητα δουλείας ανά εργασία:

· μικρή $\longrightarrow$ μεγάλη επιβάρυνση

· μεγάλη $\longrightarrow$ περιορισμός παραλληλίας



Βασικό Θέμα παρουσίασης (και εργαστηριακής άσκησης)

- Data parallel: Η παραλληλοποίηση προκύπτει από τη δομή των δεδομένων (π.χ. πίνακες).

- Θα ασχοληθούμε με αλγόριθμους, όπου η παραλληλοποίηση προκύπτει από τη δομή του αλγόριθμου (π.χ. D&C).
- Θα χρησιμοποιήσουμε το Task parallel μοντέλο
  - Γενικό (*)

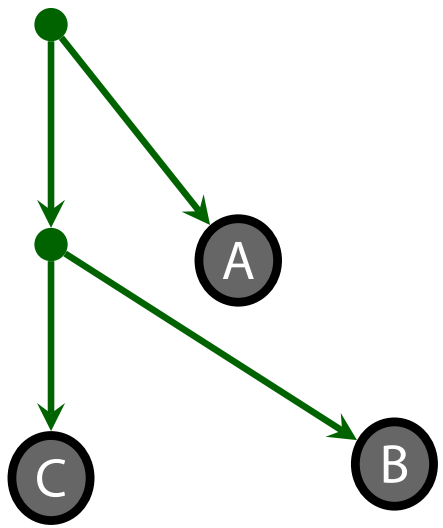
- Θα χρησιμοποιήσουμε τη γλώσσα Cilk
  - Επέκταση της C

- **σκοπός:** μεθοδολογία για παραλληλισμό με tasks
- **αλλά:** περιορίζουμε τη γενικότητα
  (συγκεκριμένες περιπτώσεις)

Task parallelism

- Ο προγραμματιστής δηλώνει ρητά τις παράλληλες εργασίες
  γράφος εργασιών (task graph)
- Ο χρήστης ορίζει
  - σημεία δημιουργίας εργασιών

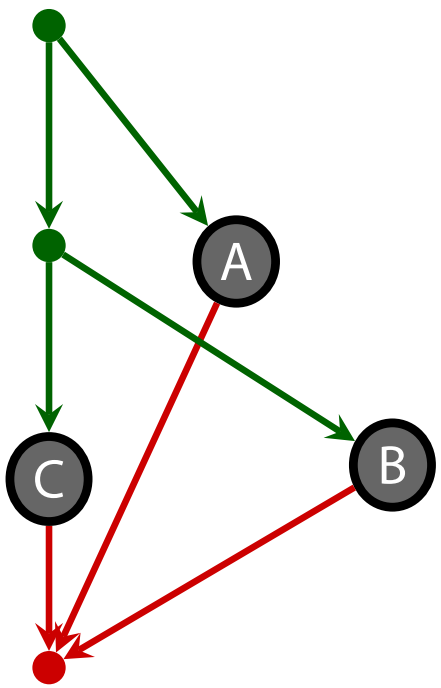
```
/* Cilk example */
x = spawn A();
y = spawn B();
C();
```



Task parallelism

- Ο προγραμματιστής δηλώνει ρητά τις παράλληλες εργασίες
  γράφος εργασιών (task graph)
- Ο χρήστης ορίζει
  - σημεία δημιουργίας εργασιών
  - σημεία συγχρονισμού διεργασιών

```
/* Cilk example */
x = spawn A();
y = spawn B();
C();
sync;
/* x,y are available */
```



Αλγόριθμοι Διαίρε και Βασίλευε

divide and conquer -- D&C

```
Divide and Conquer:
    if cant divide:
        return unitary solution  (stop recursion)
    divide problem in two
    solve first   (recursively)
    solve second  (recursively)
    combine solutions
```

- solve first/second μπορούν να γίνουν παράλληλα
- recursive splitting

Example: quicksort

divide & conquer is easily parallelized

```
def qsort(arr, low, high){
    if high == low
        return;
    pivotVal = findPivot();
    pivotLoc = partition(pivotVal);
    qsort(arr, low, pivotLoc-1);
    qsort(arr, pivotLoc+1, high);

    // no need for result combination
}
```

Example: quicksort

divide & conquer is easily parallelized

```
def qsort(arr, low, high){
    if high == low
        return;
    pivotVal = findPivot();
    pivotLoc = partition(pivotVal);
    spawn qsort(arr, low, pivotLoc-1);
    spawn qsort(arr, pivotLoc+1, high);
    sync;
    // no need for result combination
}
```

- recursive splitting

Τακτικές για παράλληλο προγραμματισμό

βλ. και ομιλία του Guy Steele στο ICFP '09

Μοντέλο συσσωρευτή (accumulator)

- Χωρισμός προβλήματος στην *αρχή* και στο *επόμενο*
- σταδιακή ανανέωση λύσης, καταναλώνοντας το *επόμενο*
- Χρησιμοποιείται στον σειριακό προγραμματισμό
- δεν μπορεί να παραλληλοποιηθεί

Μοντέλο διαίρε και βασίλευε (D&C)

- Χωρισμός του προβλήματος σε υπο-προβλήματα
- Αναδρομική λύση των υπο-προβλημάτων
- Συνδυασμός λύσεων *

* Συνήθως πιο πολύπλοκη διαδικασία από τη σταδιακή ανανέωση λύσης

Cilk

- Επέκταση της C με λίγα keywords
- Επικεφαλής της ομάδας ο καθ. Charles E. Leiserson (MIT)
- Ισχυρό θεωρητικό υπόβαθρο
  - φράγματα για χρόνο και χώρο παράληλης εκτέλεσης
- έμφαση στην **αποδοτική χρονοδρομολόγηση**

- **Ιστορία**

- 1994: Πρώτη έκδοση (Cilk-1)
- 1998: Implementation of the Cilk-5 Multithreaded Language paper
- 2006: Cilk Arts
- 2008: Cilk++ 1.0
- 2009: Αγορά Cilk arts από Intel

Cilk keywords

- cilk: Ορισμός συναρτήσεων που ορίζουν Cilk διεργασίες
- spawn: Δημιουργία Cilk διεργασίας
  Ο πατέρας και το παιδί μπορούν να εκτελεστούν παράλληλα
- sync: Αναμονή ολοκλήρωσης Cilk διεργασιών
  Τα αποτελέσματα των παιδιών είναι έτοιμα

```
cilk int fib(int n) {
     if (n < 2) return (n);
     x = spawn fib(n - 1);
     y = spawn fib(n - 2);
     sync;
     return (x + y);
}
```

- inlet
- abort

Ο γράφος διεργασιών αναδιπλώνεται δυναμικά

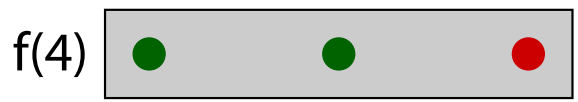
(στο χρόνο εκτέλεσης)

```
cilk int fib(int n) {
     if (n < 2) return (n);
     x = spawn fib(n - 1);
     y = spawn fib(n - 2);
     sync;
     return (x + y);
}
```

Ο γράφος διεργασιών αναδιπλώνεται δυναμικά

(στο χρόνο εκτέλεσης)

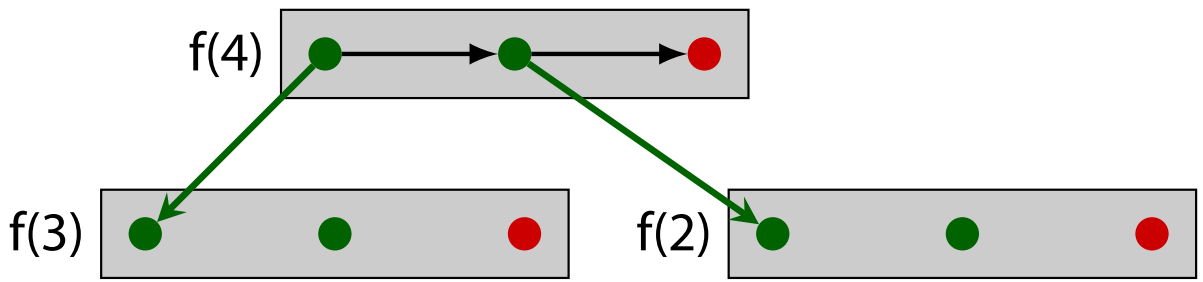
```
cilk int fib(int n) {
     if (n < 2) return (n);
     x = spawn fib(n - 1);
     y = spawn fib(n - 2);
     sync;
     return (x + y);
}
```



Ο γράφος διεργασιών αναδιπλώνεται δυναμικά

(στο χρόνο εκτέλεσης)

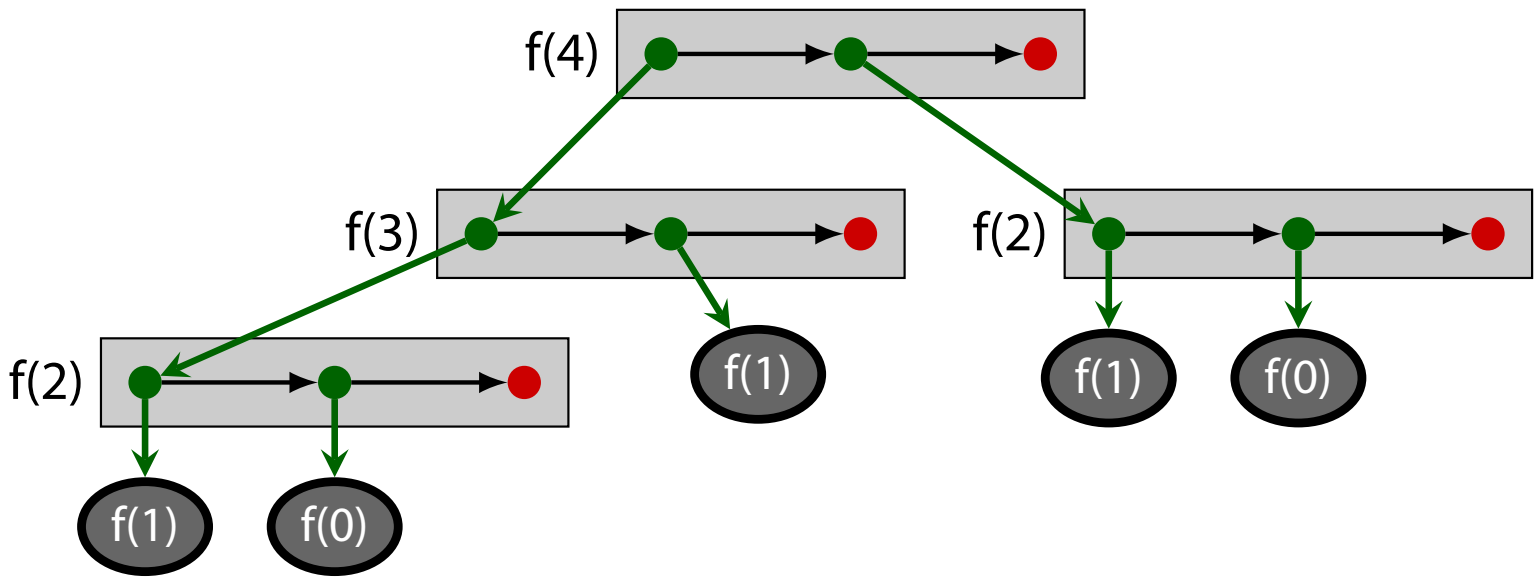
```
cilk int fib(int n) {
     if (n < 2) return (n);
     x = spawn fib(n - 1);
     y = spawn fib(n - 2);
     sync;
     return (x + y);
}
```



Ο γράφος διεργασιών αναδιπλώνεται δυναμικά

(στο χρόνο εκτέλεσης)

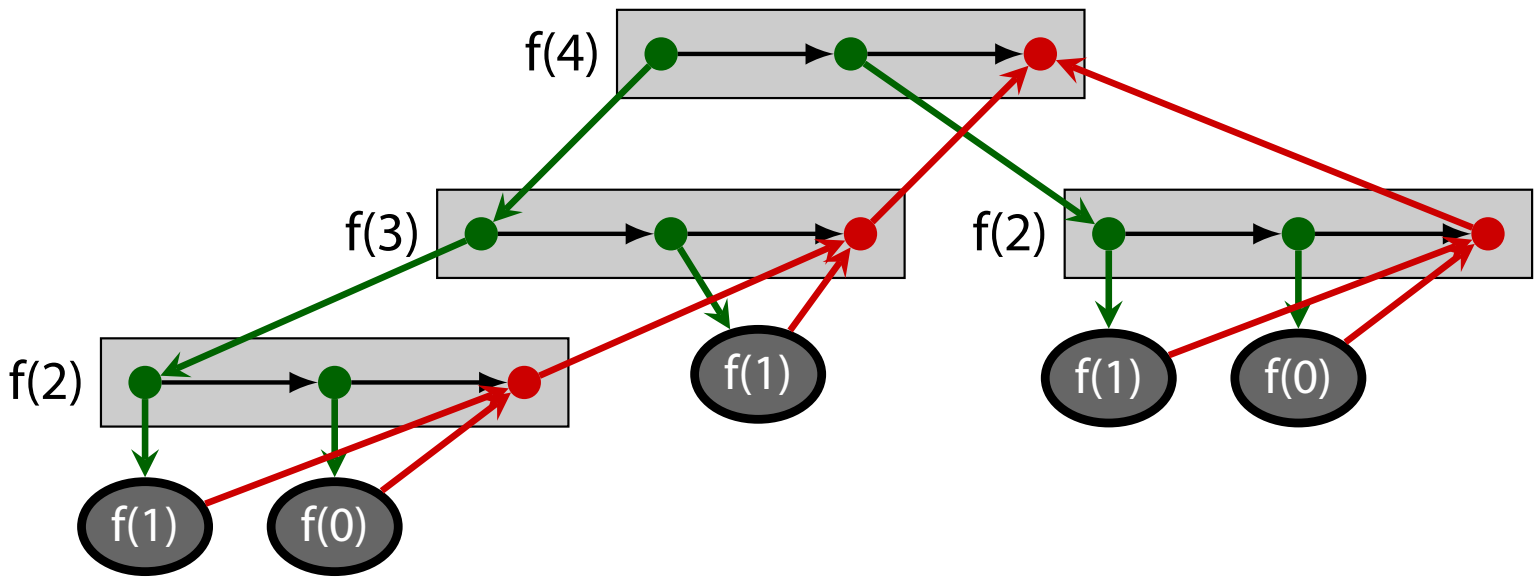
```
cilk int fib(int n) {
     if (n < 2) return (n);
     x = spawn fib(n - 1);
     y = spawn fib(n - 2);
     sync;
     return (x + y);
}
```



Ο γράφος διεργασιών αναδιπλώνεται δυναμικά

(στο χρόνο εκτέλεσης)

```
cilk int fib(int n) {
     if (n < 2) return (n);
     x = spawn fib(n - 1);
     y = spawn fib(n - 2);
     sync;
     return (x + y);
}
```



Παράδειγμα: πρόσθεση

Μετατροπή αλγορίθμου συσσωρευτή σε διαίρε και βασίλευε

$$\sum_{i=0}^{N} A_i$$

```
add(A,n){
  res = 0;
  for (i=0; i<N; i++)
    res += A[i];
  return res;
}
```

Παράδειγμα: πρόσθεση

Μετατροπή αλγορίθμου συσσωρευτή σε διαίρε και βασίλευε

$$\sum_{i=0}^{N} A_i$$

```
add(A,n){
  res = 0;
  for (i=0; i<N; i++)
    res += A[i];
  return res;
}
```

```
add_r(A, n){
  if (n == 1)
    return A[0];
  x1 = spawn add_r(A,n/2);
  x2 = spawn add_r(A+n/2,n-n/2);
  sync;
  return (x1+x2);
}
```

Παράδειγμα: πρόσθεση

Μετατροπή αλγορίθμου συσσωρευτή σε διαίρε και βασίλευε

$$\sum_{i=0}^{N} A_i$$

```
add(A,n){
  res = 0;
  for (i=0; i<N; i++)
    res += A[i];
  return res;
}
```

```
add_r(A, n){
  if (n == 1)
    return A[0];
  x1 = spawn add_r(A,n/2);
  x2 = spawn add_r(A+n/2,n-n/2);
  sync;
  return (x1+x2);
}
```

- βελτίωση ταχύτητας;

Παράδειγμα: πρόσθεση

Μετατροπή αλγορίθμου συσσωρευτή σε διαίρε και βασίλευε

$$\sum_{i=0}^{N} A_i$$

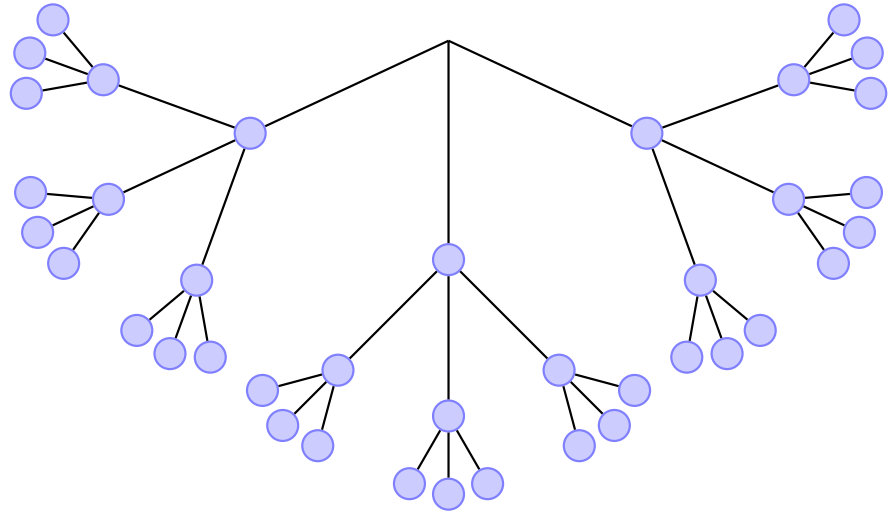
```
add(A,n){
  res = 0;
  for (i=0; i<N; i++)
    res += A[i];
  return res;
}
```

```
add_r(A, n){
  if (n < K)
    return add(A,n);
  x1 = spawn add_r(A,n/2);
  x2 = spawn add_r(A+n/2,n-n/2);
  sync;
  return (x1+x2);
}
```

- βελτίωση ταχύτητας;

Παράδειγμα: Αναζήτηση

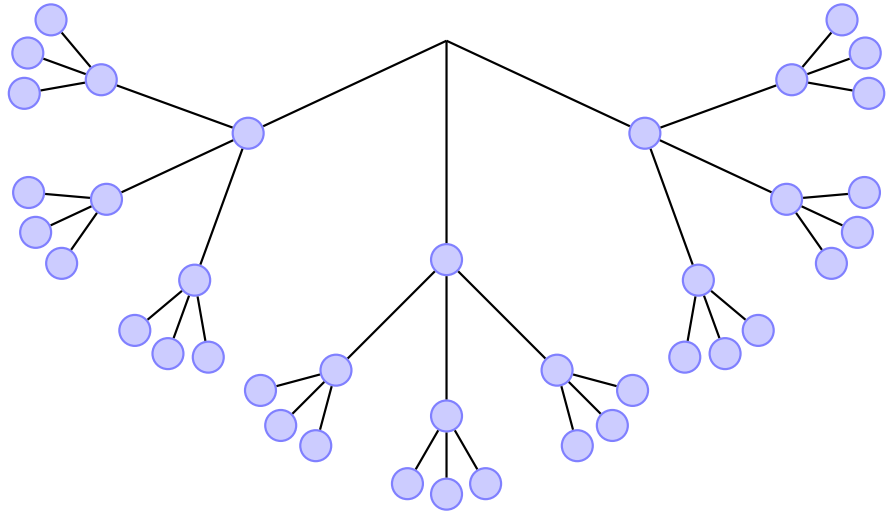
- Χώρος λύσεων (πχ AI)
- Δομή δεδομένων



```
DF_Search(node):
    for c in node.get_children():
            DF_Search(c)
```

Παράδειγμα: Αναζήτηση

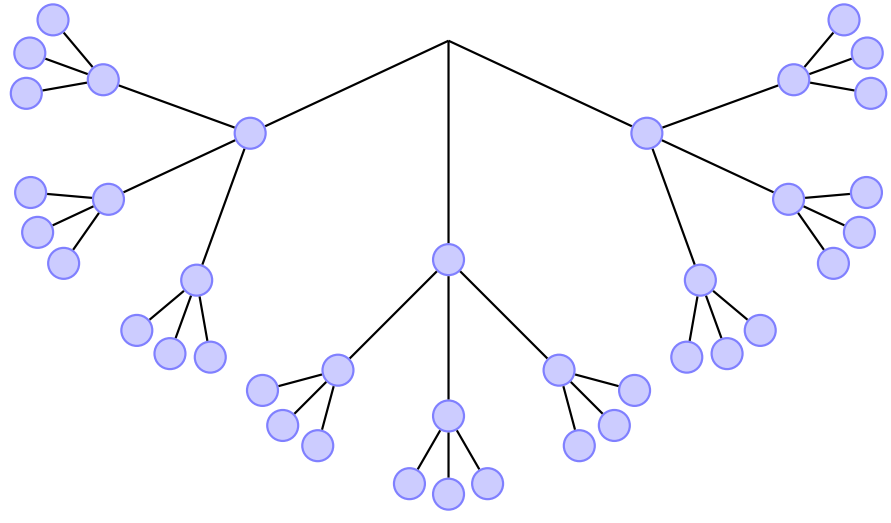
- Χώρος λύσεων (πχ AI)
- Δομή δεδομένων



```
DF_Search(node):
    for c in node.get_children():
        spawn DF_Search(c)
    sync
```

Παράδειγμα: Αναζήτηση

- Χώρος λύσεων (πχ AI)
- Δομή δεδομένων



```
DF_Search(node):
    for c in node.get_children():
        spawn DF_Search(c)
    sync
```

- Αν βρω αυτό που ψάχνω;

  (ή ανακαλύψω ότι η αναζήτηση σε ένα τμήμα του χώρου είναι ανευ ουσίας;)

Παράδειγμα: πολλαπλασιασμός

$$\prod_{i=0}^{N} A_i$$

```
mul(A,n){
  res = 1;
  for (i=0; i<N; i++)
    res *= A[i];
  return res;
}
```

```
mul_r(A, n){
  if (n == 1)
    return A[0];
  x1 = spawn mul_r(A,n/2);
  x2 = spawn mul_r(A+n/2,n-n/2);
  sync;
  return (x1*x2);
}
```

- Τι (αλγοριθμική) βελτίωση μπορώ να κάνω;

Παράδειγμα: πολλαπλασιασμός

$$\prod_{i=0}^{N} A_i$$

```
mul(A,n){
  res = 1;
  for (i=0; i<N; i++)
    res *= A[i];
  return res;
}
```

```
mul_r(A, n){
  if (n == 1)
    return A[0];
  x1 = spawn mul_r(A,n/2);
  x2 = spawn mul_r(A+n/2,n-n/2);
  sync;
  return (x1*x2);
}
```

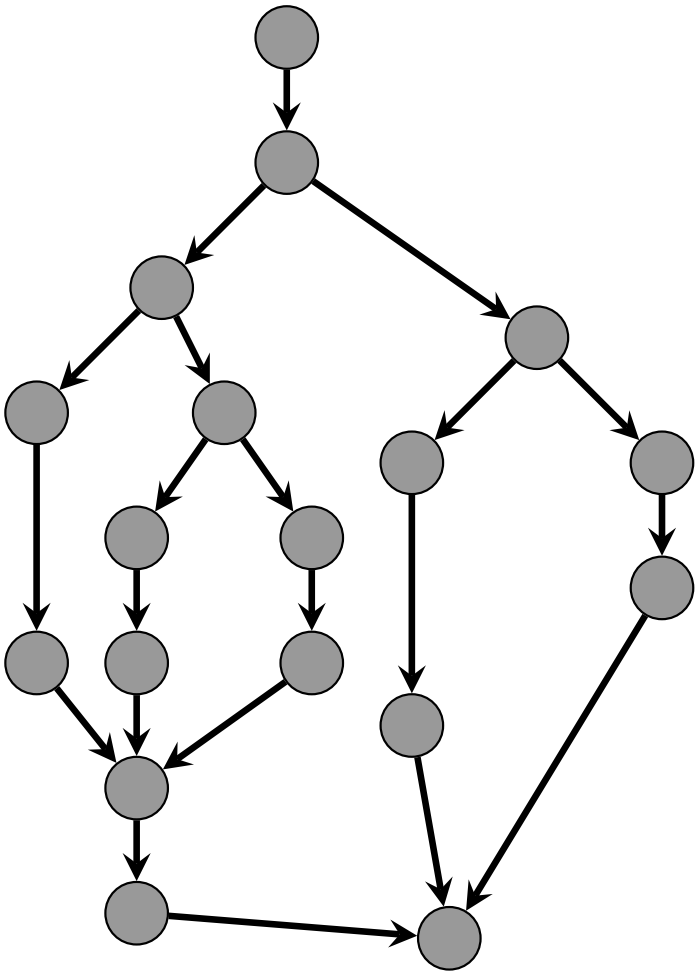
- Τι (αλγοριθμική) βελτίωση μπορώ να κάνω;
  - Όταν συναντήσω το 0, μπορώ να σταματήσω!

Παράδειγμα: πολλαπλασιασμός

inlet/abort: έλεγχος για 0

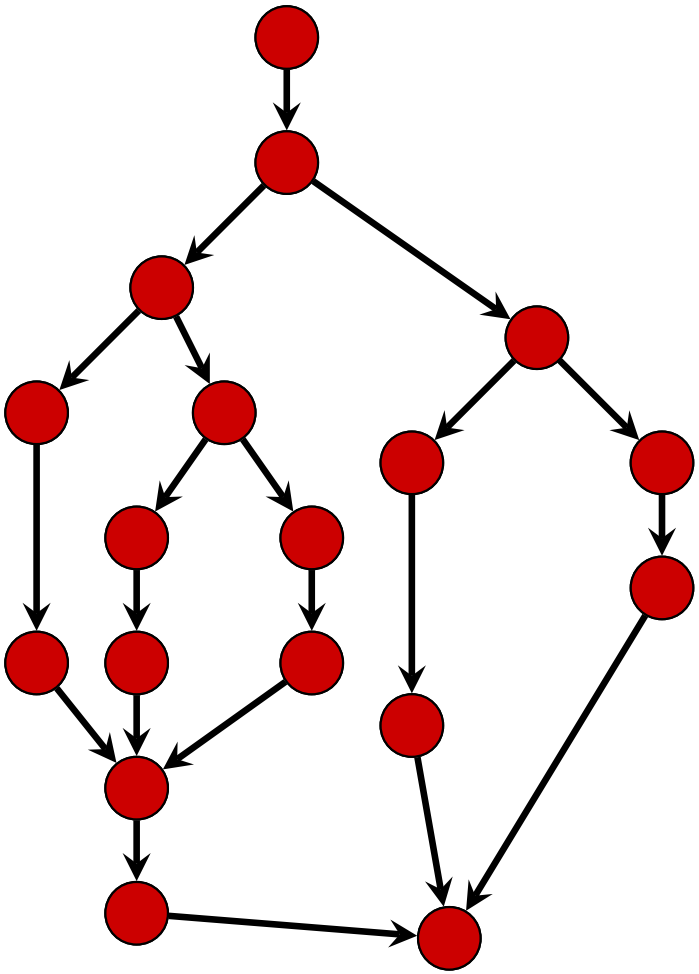
```
cilk int product(int *A, int n) {
    int p=1;
    inlet void mult(int x) {
        p *= x;
        if (p == 0)
            abort;
        return;
    }
    if (n == 1)
        return A[0];
    mult( spawn product(A, n/2) );
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
}
```

Μοντέλο Επίδοσης Cilk



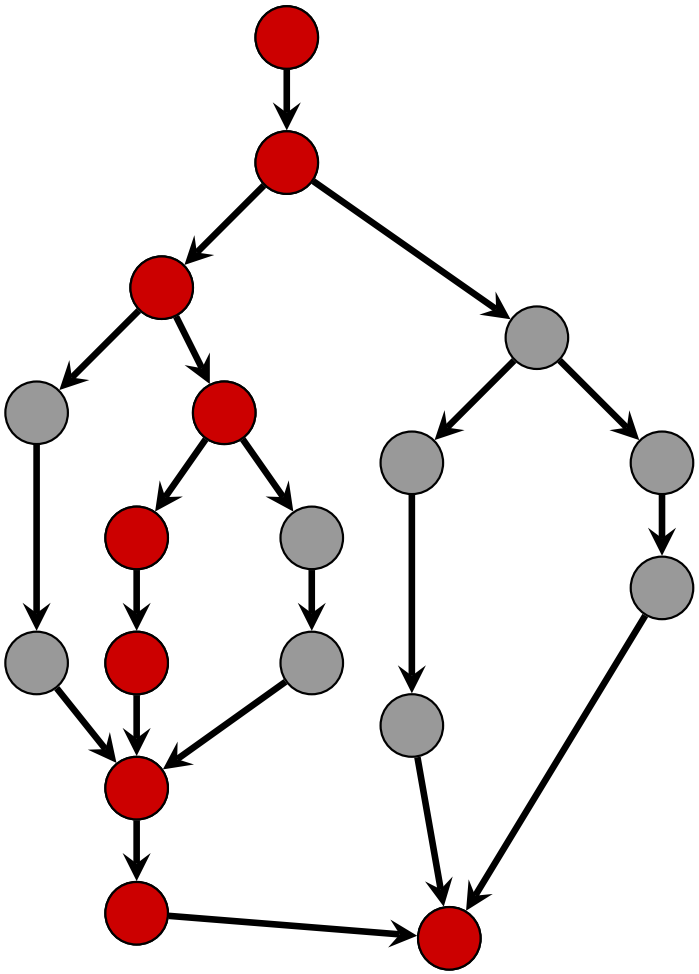
- T_p: χρόνος εκτέλεσης σε P επεξ.

Μοντέλο Επίδοσης Cilk



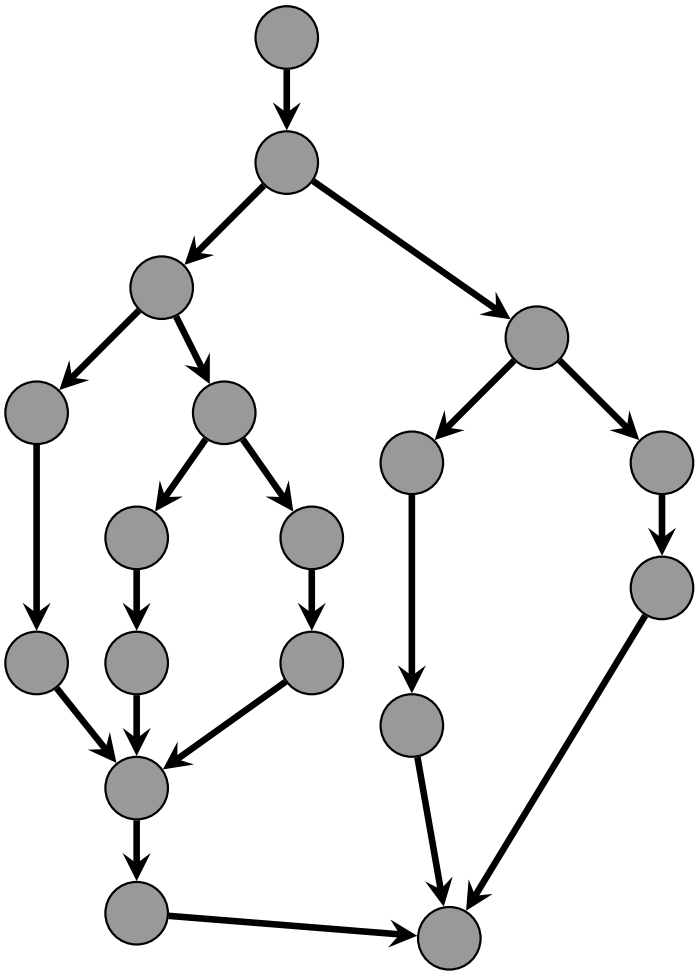
- T_p: χρόνος εκτέλσης σε P επεξ.
- T_1: **work**
  (κόστος εκτέλεσης όλων των κόμβων)

Μοντέλο Επίδοσης Cilk



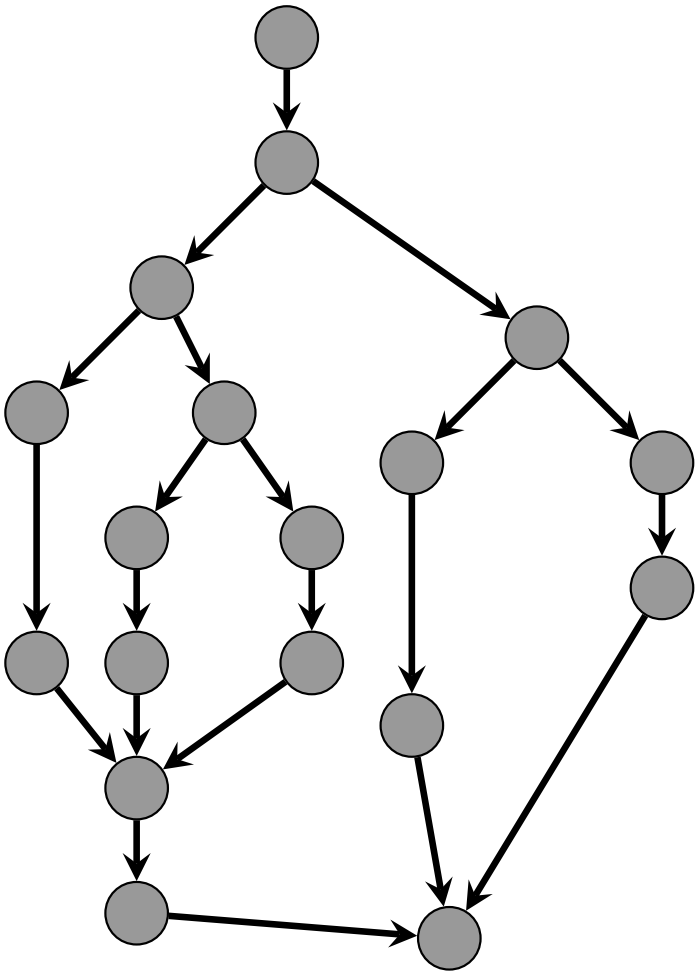
- T_p: χρόνος εκτέλσης σε P επεξ.
- T_1: **work**
  (κόστος εκτέλεσης όλων των κόμβων)
- T_∞: **span**
  (κόστος εκτέλεσης για ∞ επεξ.)

Μοντέλο Επίδοσης Cilk



- T_p: χρόνος εκτέλσης σε P επεξ.
- T_1: **work**
  (κόστος εκτέλεσης όλων των κόμβων)
- T_∞: **span**
  (κόστος εκτέλεσης για ∞ επεξ.)
- Κάτω όρια: $T_p \geq T_1/P$, $T_p \geq T_\infty$
- Μέγιστο speedup: T_1/T_∞

Μοντέλο Επίδοσης Cilk



- T_p: χρόνος εκτέλσης σε P επεξ.
- T_1: **work**
  (κόστος εκτέλεσης όλων των κόμβων)
- T_∞: **span**
  (κόστος εκτέλεσης για ∞ επεξ.)
- Κάτω όρια: $T_p \geq T_1/P$, $T_p \geq T_\infty$
- Μέγιστο speedup: T_1/T_∞
- Ο χρόνοδρομολογητής της Cilk
  - επιτυγχάνει $T_p = T_1/P + \mathcal{O}(T_\infty)$
    (γραμμικό speedup εάν $P \ll T_1/T_\infty$)
  - απαιτεί $S_p \leq PS_1$ χώρο στη στοίβα
  - έχει καλή επίδοση και στην πράξη

Parallel slack (task grain)

- $P \ll T_1/T_\infty$
- parallel slack: $(T_1/T_\infty)\,/P$
  - Παραλληλισμός προγράμματος T_1/T_∞
  - Παραλληλισμός μηχανήματος P

- Χρειάζεται τάξης μεγέθους περισσότερος παραλληλισμός στο πρόγραμμα από ότι στο πραγματικό σύστημα.

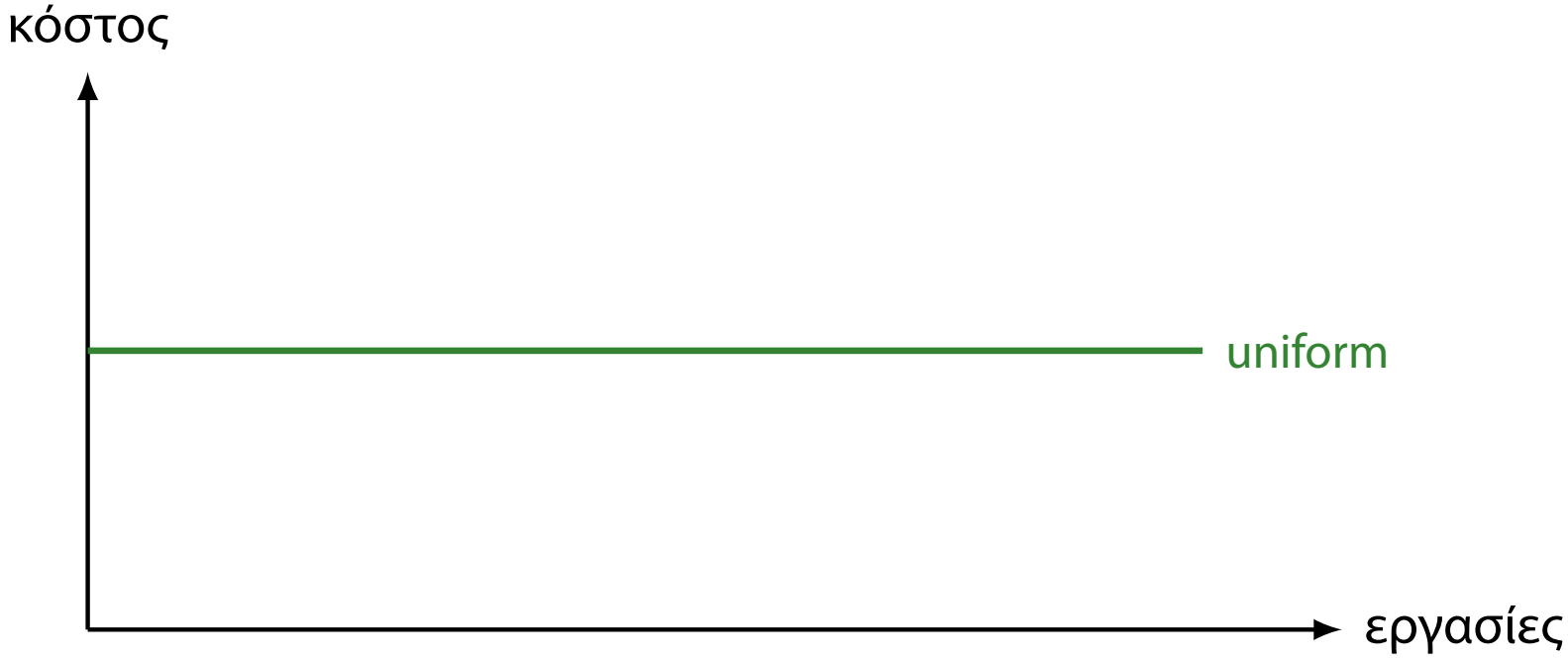
  Διαισθητικά:

  + Πρόγραμμα δεν εξαρτάται από αριθμό επεξεργαστών
  + Συνεισφέρει στην αποδοτική χρονοδρομολόγηση
  + Επιτρέπει καλύτερη κατανομή φόρτου (load balancing)

  - **αλλά:**
  - Αν το μέγεθος δουλειάς της κάθε εργασίας μικρό, τότε οι επιβαρύνσεις του συστήματος επηρεάζουν σημαντικά

Ισοκατανομή φόρτου - μέγεθος εργασίας

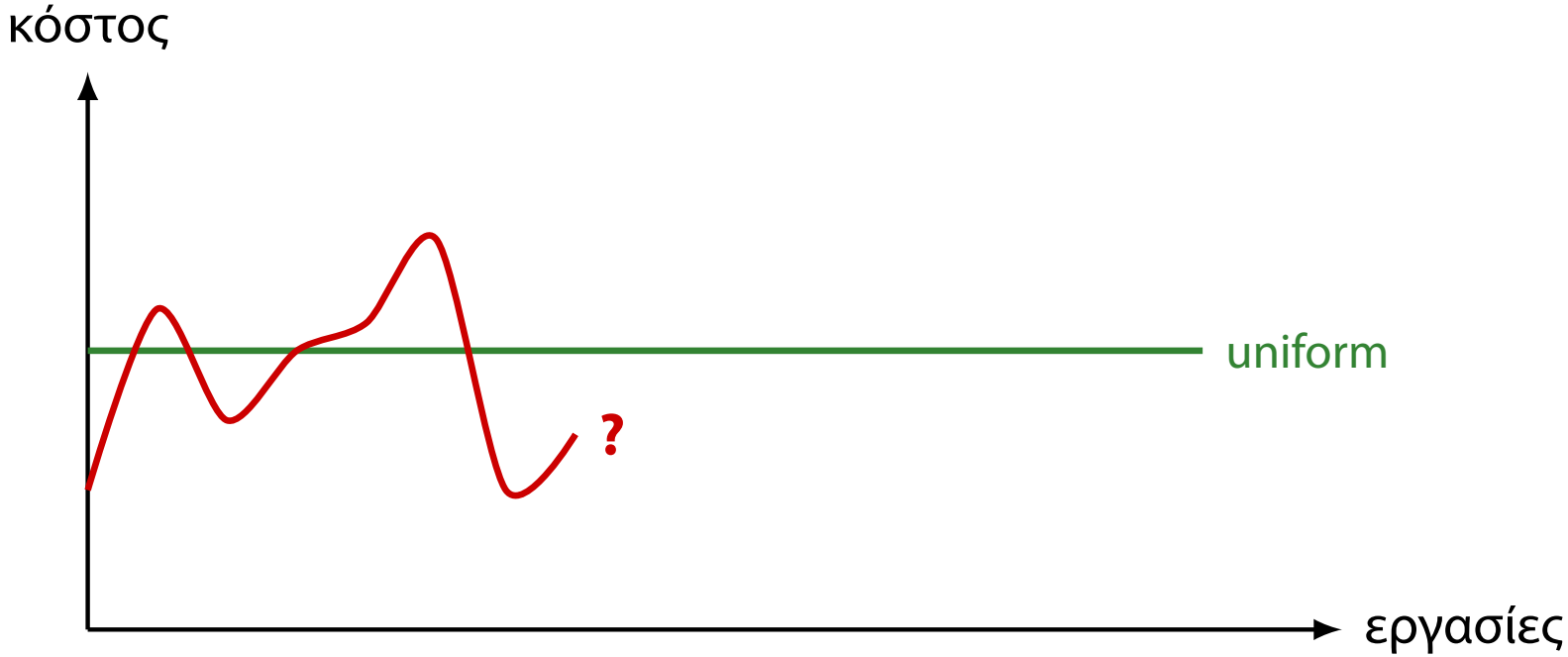
load balancing - task grain



- Σταθερό κόστος εκτέλεσης:
  - 1 εργασία / πραγματικό επεξεργαστή

Ισοκατανομή φόρτου - μέγεθος εργασίας

load balancing - task grain



- Σταθερό κόστος εκτέλεσης:
  - 1 εργασία / πραγματικό επεξεργαστή
- Το κόστος εκτέλεσης είναι μη-σταθερό και άγνωστο
  - Μεγάλο parallel slack

Χρονοδρομολόγηση στο χώρο χρήστη

user-level scheduling

- Αριθμός εργασιών *T*
- *P* επεξεργαστές
  (kernel threads)
- Η κάθε εργασία μπορεί να παράγει άλλες
- ... να περιμένει την ολοκλήρωση των παιδιών της.

Στόχοι

- Ισοκατανομή φόρτου εργασίας
- Αποδοτική χρήση χώρου
- Μικρή επιβάρυνση (ανεξάρτητη του *T*)

- Πότε εκτελείται ο ΧΔ;

Χρονοδρομολόγηση στο χώρο χρήστη

user-level scheduling

- Αριθμός εργασιών *T*
- *P* επεξεργαστές
  (kernel threads)
- Η κάθε εργασία μπορεί να παράγει άλλες
- ... να περιμένει την ολοκλήρωση των παιδιών της.

Στόχοι

- Ισοκατανομή φόρτου εργασίας
- Αποδοτική χρήση χώρου
- Μικρή επιβάρυνση (ανεξάρτητη του *T*)

- Πότε εκτελείται ο ΧΔ;
  - Εισάγονται κατάλληλες κλήσεις στις spawn/sync/κλπ

Τακτικές χρονοδρομολόγησης

- **work sharing:** Όταν δημιουργούνται νέες εργασίες ο ΧΔ προσπαθεί να τις "στείλει" σε ανενεργούς επεξεργαστές.
- **work stealing:** Οι ανενεργοί επεξεργαστές προσπαθούν να "κλέψουν" εργασίες.

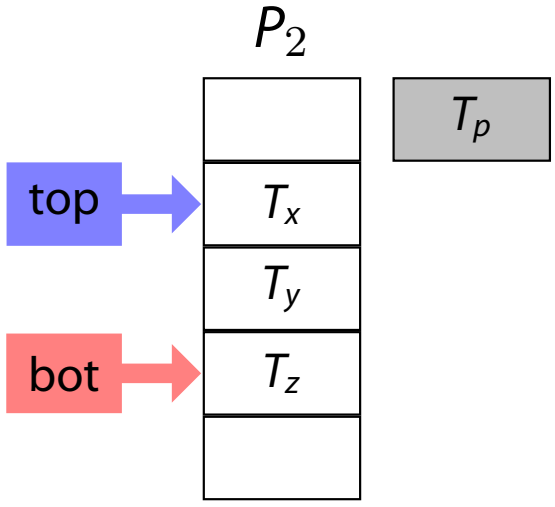
Γενικά προτιμάται η τακτική work stealing
(στη Cilk και όχι μόνο)

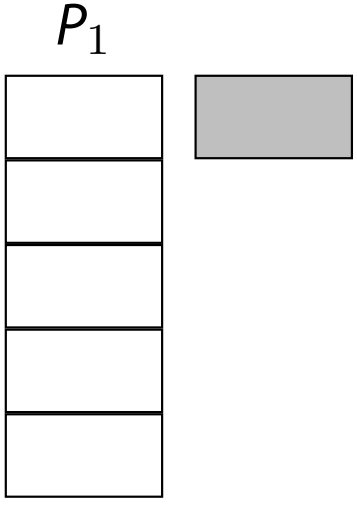
- καλύτερο locality
- μικρότερη επιβάρυνση συγχρονισμού
- Βέλτιστα θεωρητικά όρια ως προς χρόνο, χώρο
  [Blumofe and Leiserson '99]

work stealing

Χρονοδρομολόγηση στη Cilk

- Μια ουρά ΧΔ ανά P
  deque (double-ended queue)
  - `pushBot`
  - `popBot`
  - `popTop`

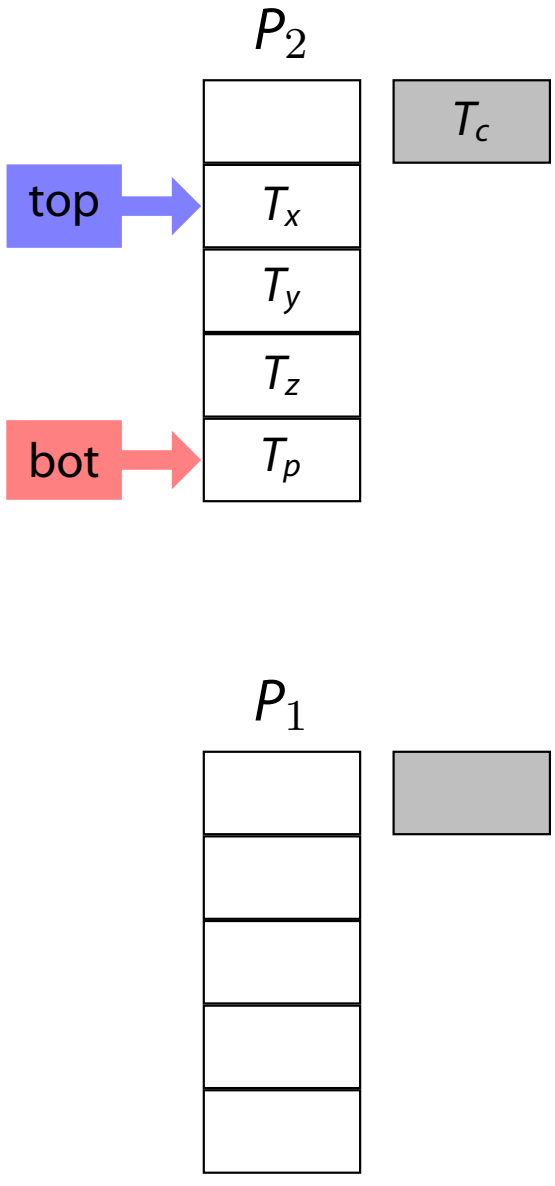




work stealing

Χρονοδρομολόγηση στη Cilk

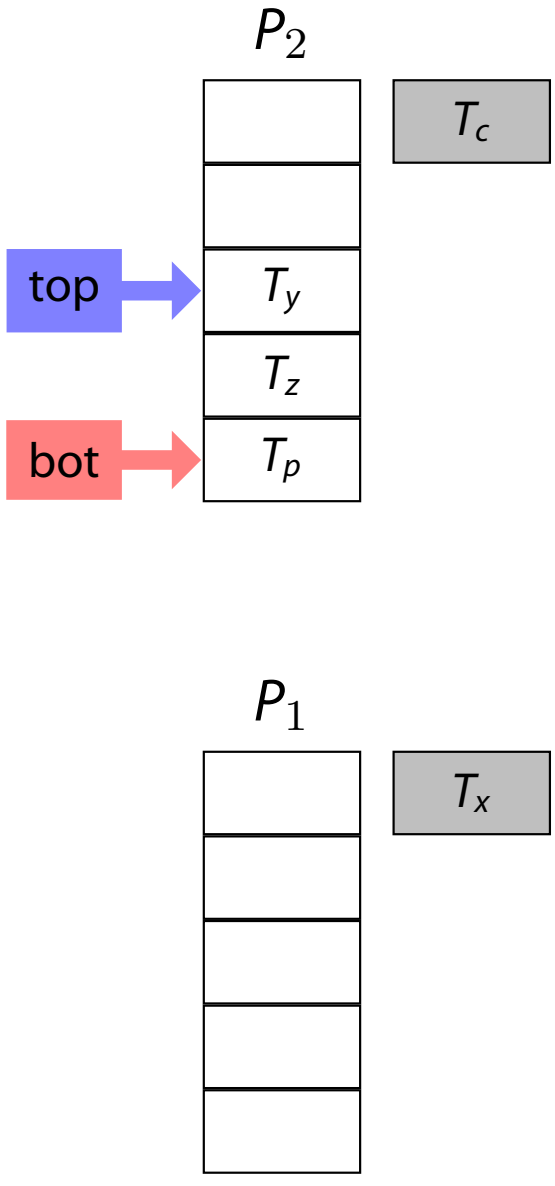
- Μια ουρά ΧΔ ανά P
  deque (double-ended queue)
  - `pushBot`
  - `popBot`
  - `popTop`
- $T_p \rightarrow$ spawn task T_c
  - `pushBot(`T_p`)`
  - `execute(`T_c`)` (FAST!)



work stealing

Χρονοδρομολόγηση στη Cilk

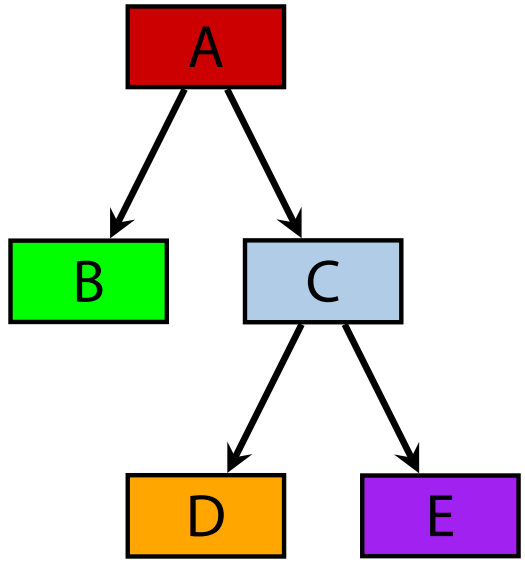
- Μια ουρά ΧΔ ανά P
  deque (double-ended queue)
  - `pushBot`
  - `popBot`
  - `popTop`
- $T_p \rightarrow$ spawn task T_c
  - `pushBot(`T_p`)`
  - `execute(`T_c`)` (FAST!)
- P_1 ανενεργός
  - **Τυχαία** επιλογή επεξεργαστή *p*
  - *p*`->popTop()`
  - Εκτέλεση task που προέκυψε (SLOW!)



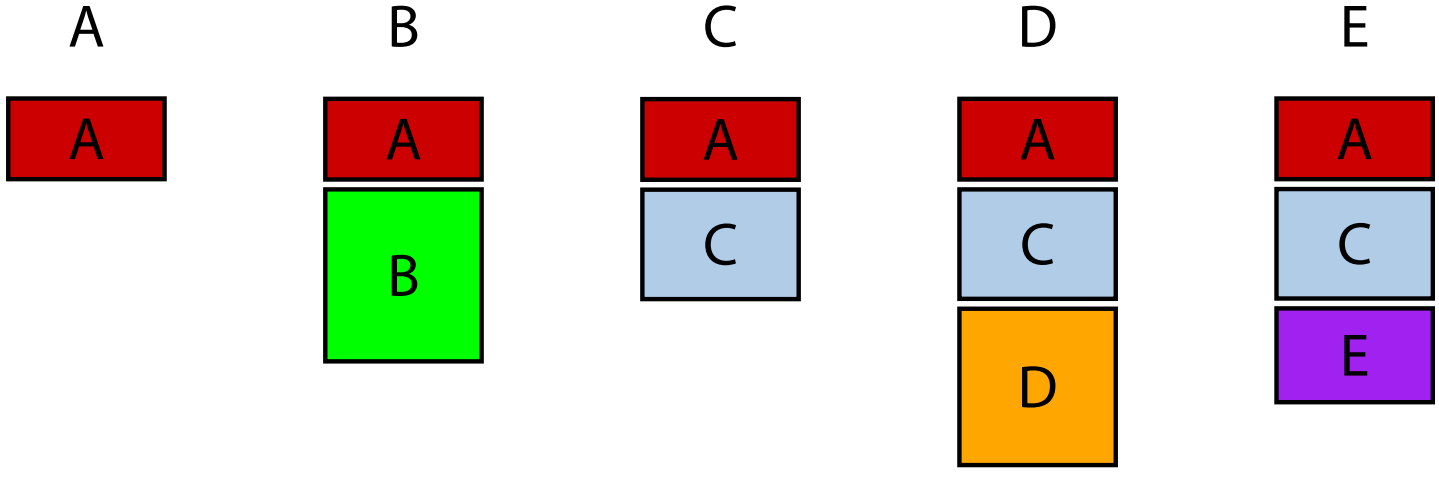
Υλοποίηση ΧΔ χώρου χρήστη

- Για να υποστηρίζεται η κλοπή εργασίων πρέπει η κάθε εργασία να μπορεί να μεταφερθεί σε διαφορετικό επεξεργαστή
- Τι χρείαζεται η κάθε διεργασία για να εκτελεστεί;
  - Κώδικα (πχ δείκτη σε συνάρτηση)
  - Θέση στον κώδικα
  - Δεδομένα:
    - Στοίβα
    - Σωρός
- Η Cilk υλοποιεί cactus stack

Cactus stack



Εικόνα της κάθε συνάρτησης:



Μεταγλώττιση και εκτέλεση προγραμμάτων Cilk

```
$ cilkc fib.cilk -o fib
```

Μεταγλώττιση και εκτέλεση προγραμμάτων Cilk

```
$ cilkc fib.cilk -o fib
$ ./fib --nproc 2 --stats 1 -- 35
Result: 9227465

RUNTIME SYSTEM STATISTICS:

Wall-clock running time on 2 processors: 2.114348 s
```

Μεταγλώττιση και εκτέλεση προγραμμάτων Cilk

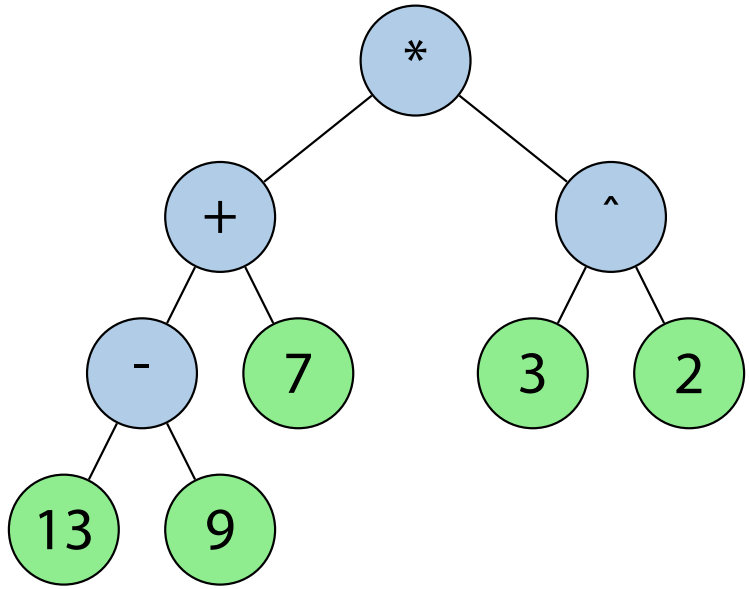
```
$ cilkc fib.cilk -o fib
$ ./fib --nproc 2 --stats 1 -- 35
Result: 9227465

RUNTIME SYSTEM STATISTICS:

Wall-clock running time on 2 processors: 2.114348 s
$ ./fib --nproc 8 --stats 1 -- 35
Result: 9227465

RUNTIME SYSTEM STATISTICS:

Wall-clock running time on 8 processors: 538.614000 ms
```

Μεταγλώττιση και εκτέλεση προγραμμάτων Cilk

```
$ cilkc fib.cilk -o fib
$ ./fib --nproc 2 --stats 1 -- 35
Result: 9227465

RUNTIME SYSTEM STATISTICS:

Wall-clock running time on 2 processors: 2.114348 s
$ ./fib --nproc 8 --stats 1 -- 35
Result: 9227465

RUNTIME SYSTEM STATISTICS:

Wall-clock running time on 8 processors: 538.614000 ms
$ ./fib --help
Cilk options:
...
```

Παράλληλη αποτίμηση έκφρασης

(εργαστηριακή άσκηση)



- Δένδρο αριθμητικής έκφρασης
  - Φύλλα: αριθμητικές τιμές
  - Ενδιάμεσοι κόμβοι: δυαδικοί τελεστές
- Ζητείται η παράλληλη αποτίμηση της έκφρασης

Run-length encoding

(εργαστηριακή άσκηση)

```
def rle(xs):
    ret = []      # return list
    curr = xs[0] # current element
    freq = 1      # current element's frquency
    for item in xs[1:]:
        if item == curr:
            freq += 1
        else:
            ret.append((curr,freq))
            curr,freq = (item,1)
    ret.append((curr,freq))
    return ret
```

- Ζήτείται παράλληλη έκδοση του αλγόριθμου κωδικοποίησης χρησιμοποιώντας D&C.