Threading Building Blocks

Σύνοψη

- Γενικά για TBBs
- Tasks
- Γενικευμένος προγραμματισμός και templates
- parallel_for
- Εσωτερική λειτουργία βιβλιοθήκης

Σύνοψη

- **Γενικά για TBBs**
- Tasks
- Γενικευμένος προγραμματισμός και templates
- parallel_for
- Εσωτερική λειτουργία βιβλιοθήκης

Τι είναι τα TBBs;

- ***C++ template library*** *για αποδοτικό και εύκολο παράλληλο προγραμματισμό σε πλατφόρμες μοιραζόμενης μνήμης*

- Αναπτύσσεται από την Intel από το 2004 (open-source από το 2007)
- Δεν είναι καινούρια γλώσσα ή επέκταση
- Μεταφέρσιμη στους περισσότερους C++ compilers, λειτουργικά συστήματα και αρχιτεκτονικές

Βασικά χαρακτηριστικά

1. Ο προγραμματιστής ορίζει *tasks* αντί για threads
   – επικεντρώνεται στην ***έκφραση*** του παραλληλισμού στην εφαρμογή (σε υψηλότερο ή χαμηλότερο επίπεδο)
   – η βιβλιοθήκη είναι υπεύθυνη για την ***υλοποίησή*** του
      - διάσπαση συνολικής δουλειάς σε επιμέρους εργασίες
      - δρομολόγηση εργασιών στους επεξεργαστές
      - συγχρονισμός
      - ισοκατανομή φορτίου
      - διαχείριση πόρων συστήματος και εσωτερικών μηχανισμών

Βασικά χαρακτηριστικά

2. Σχεδιασμένη για *κλιμακωσιμότητα*
   - η συνολική δουλειά σπάει σε πολλά μικρά κομμάτια (tasks), συνήθως πολύ περισσότερα από τον αριθμό των επεξεργαστών («*parallel slack*»)
   - εξασφαλίζεται ότι θα υπάρχει πάντα διαθέσιμη δουλειά για κάθε επιπλέον επεξεργαστή που προστίθεται
   - ο μηχανισμός για load balancing εξασφαλίζει την κλιμακώσιμη απόδοση

Βασικά χαρακτηριστικά

3. Εκμεταλλεύεται τη δύναμη και την ευελιξία του *γενικευμένου προγραμματισμού* (*generic programming*)
   - παρέχει ένα πλούσιο σύνολο από παραμετροποιήσιμα (templated), «ready to use» παράλληλα αλγοριθμικά μοτίβα και δομές
     - αντίστοιχα με την C++ STL για τα σειριακά προγράμματα
   - δεν απαιτεί ειδική υποστήριξη από μεταγλωττιστή

TBB 4.0 Components

Generic Parallel Algorithms
parallel_for
parallel_reduce
parallel_scan
parallel_do
pipeline, parallel_pipeline,
parallel_sort
parallel_invoke

Synchronization primitives
atomic
mutex
recursive_mutex
spin_mutex, spin_rw_mutex
queuing_mutex, queuing_rw_mutex

Raw tasking
task
task_group
task_list
task_scheduler_observer

Flow Graph
graph
function_node
broadcast_node
…

Concurrent containers
concurrent_unordered_map,
concurrent_unordered_set,
concurrent_hash_map,
concurrent_queue,
concurrent_bounded_queue,
concurrent_priority_queue
concurrent_vector

Memory allocation
tbb_allocator
cache_aligned_allocator
scalable_allocator

TBB 4.0 Components

Generic Parallel Algorithms
parallel_for
parallel_reduce
parallel_scan
parallel_do
pipeline, parallel_pipeline,
parallel_sort
parallel_invoke

Concurrent containers
concurrent_unordered_map,
concurrent_unordered_set,
concurrent_hash_map,
concurrent_queue,
concurrent_bounded_queue,
concurrent_priority_queue
concurrent_vector

Synchronization primitives
atomic
mutex
recursive_mutex
spin_mutex, spin_rw_mutex
queuing_mutex, queuing_rw_mutex

Memory allocation
tbb_allocator
cache_aligned_allocator
scalable_allocator

Raw tasking
task
task_group
task_list
task_scheduler_observer

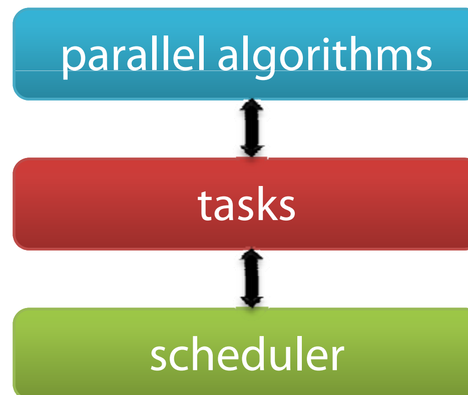
Flow Graph
graph
function_node
broadcast_node
…

Σύνοψη

- Γενικά για TBBs
- **Tasks**
- Γενικευμένος προγραμματισμός και templates
- parallel_for
- Εσωτερική λειτουργία βιβλιοθήκης

Tasks

- Εκφράζουν μια στοιχειώδη ανεξάρτητη εργασία στο πρόγραμμα του χρήστη
  - πολύ πιο lighweight από τα native threads του λειτουργικού



- Δυνατότητα άμεσης χρήσης των tasks από τον προγραμματιστή
  - δημιουργία αυθαίρετα πολύπλοκων γράφων εργασιών

Προγραμματιστικό μοντέλο

- Όπως και στη Cilk, δύο βασικές λειτουργίες για την περιγραφή ενός task graph
  - *spawn*: δημιουργία εργασίας
  - *wait*: συγχρονισμός εργασιών

Παράδειγμα

```
long SerialFib(long n) {
   if (n < 2)
      return n;
   else
      return SerialFib(n-1)
             + SerialFib(n-2);
}
```

```
long n, sum;

FibTask& r = *new (
  allocate_root())FibTask(n,&sum);

spawn_root_and_wait(r);

cout << sum;
```

```
class FibTask: public task {
   const long n;
   long *const sum;
   FibTask(long n_,long* sum_) {
      n=n_; sum=sum_;
   }

   task* execute() {
      if (n < cutOff)
         *sum = SerialFib(n);
      else {
         long x,y;
         FibTask& a = *new (
            allocate_child())FibTask(n-1,&x);
         FibTask& b = *new (
            allocate_child())FibTask(n-2,&y);

         set_ref_count(3);
         spawn(b);
         spawn(a);
         wait_for_all();
         *sum = x+y;
      }
      return NULL;
   }
};
```

Παράδειγμα

```
long SerialFib(long n) {
   if (n < 2)
      return n;
   else
      return SerialFib(n-1)
             + SerialFib(n-2);
}
```

```
long n, sum;

FibTask& r = *new (
  allocate_root())FibTask(n,&sum);

spawn_root_and_wait(r);

cout << sum;
```

```
class FibTask: public task {
   const long n;
   long *const    um;
   ...ong n_,long* sum ) {
               sum=sum_;
   }

   task* execute() {
      if (n < cutOff)
         *sum = SerialFib(n);
      else {
         long x,y;
         FibTask& a = *new (
            allocate_child())FibTask(n-1,&x);
         FibTask& b = *new (
            allocate_child())FibTask(n-2,&y);

         set_ref_count(3);
         spawn(b);
         spawn(a);
         wait_for_all();
         *sum = x+y;
      }
      return NULL;
   }
};
```

each user-defined task must extend tbb::task

and implement execute()

Παράδειγμα

```
long SerialFib(long n) {
   if (n < 2)
      return n;
   else
      return SerialFib(n-1)
             + SerialFib(n-2);
}
```

allocate root task (has no parent)

```
long n, sum;

FibTask& r = *new (
  allocate_root())FibTask(n,&sum);

spawn_root_and_wait(r);

cout << sum;
```

spawn it, and wait here

```
class FibTask: public task {
   const long n;
   long *const sum;
   FibTask(long n_,long* sum_) {
      n=n_; sum=sum_;
   }

   task* execute() {
      if (n < cutOff)
         *sum = SerialFib(n);
      else {
         long x,y;
         FibTask& a = *new (
            allocate_child())FibTask(n-1,&x);
         FibTask& b = *new (
            allocate_child())FibTask(n-2,&y);

         set_ref_count(3);
         spawn(b);
         spawn(a);
         wait_for_all();
         *sum = x+y;
      }
      return NULL;
   }
};
```

Παράδειγμα

```
long SerialFib(long n) {
   if (n < 2)
      return n;
   else
      return Se
             +
}
```

if n small enough, execute task serially

otherwise create and run two tasks

```
long n, sum;

FibTask& r = *new (
  allocate_root())FibTask(n,&sum);

spawn_root_and_wait(r);

cout << sum;
```

```
class FibTask: public task {
   const long n;
   long *const sum;
   FibTask(long n_,long* sum_) {
      n=n_; sum=sum_;
   }

   task* execute() {
      if (n < cutOff)
         *sum = SerialFib(n);
      else {
         long x,y;
         FibTask& a = *new (
            allocate_child())FibTask(n-1,&x);
         FibTask& b = *new (
            allocate_child())FibTask(n-2,&y);

         set_ref_count(3);
         spawn(b);
         spawn(a);
         wait_for_all();
         *sum = x+y;
      }
      return NULL;
   }
};
```

Παράδειγμα

```
long SerialFib(long n) {
   if (n < 2)
      return n;
   else
      return SerialFib(n-1)
             + SerialFib(n-2);
}
```

```
long n, sum;

FibTask& r = *new (
  allocate_root())FibTa

spawn_root_and_wait(r);

cout << sum;
```

```
class FibTask: public task {
   const long n;
   long *const sum;
   FibTask(long n_,long* sum_) {
      n=n_; sum=sum_;
   }

   task* execute() {
      if (n < cutOff)
         *sum = SerialFib(n);
      else {
         long x,y;
         FibTask& a = *new (
            allocate_child())FibTask(n-1,&x);
         FibTask& b = *new (
            allocate_child())FibTask(n-2,&y);

         set_ref_count(3);
         spawn(b);
         spawn(a);
         wait_for_all();
         *sum = x+y;
      }
      return NULL;
   }
};
```

allocate child tasks

spawn tasks
(indicate them as
“ready to execute”)

how many children
should I wait for?
2 (+1 implicit...)

ok, now really wait
for children to
complete

merge their results
and store into *sum

Σύνοψη

- Γενικά για TBBs
- Tasks
- **Γενικευμένος προγραμματισμός και templates**
- parallel_for
- Εσωτερική λειτουργία βιβλιοθήκης

Γενικευμένος Προγραμματισμός

- «... *deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization*» [Czarnecki, Eisenecker – Generative Programming]

- «... *a style of computer programming in which algorithms are written in terms of* to-be-specified-later *types that are then* instantiated *when needed for specific types provided as parameters*» [wikipedia]

- Σκοπός η ανάπυξη λογισμικού ώστε να είναι επαναχρησιμοποιήσιμο με απλό και αποδοτικό τρόπο

Templates

- Επιτρέπουν την *παραμετροποίηση* τύπων σε *συναρτήσεις* και *κλάσεις*
- Παράδειγμα templated συνάρτησης

```
template<typename T>
void swap(T & x, T & y) {
   T tmp = x;
   x = y;
   y = tmp;
}
…
float f1,f2;
String s1,s2;

swap(f1,f2);  //template instantiation: swap floats
swap(s1,s2);  //template instantiation: swap strings
```

Ελάχιστες απαιτήσεις για τον T
1. copy constructor ***T(const T&)***
2. assignment operator
void T::operator=(const T&);
3. destructor ***~T()***

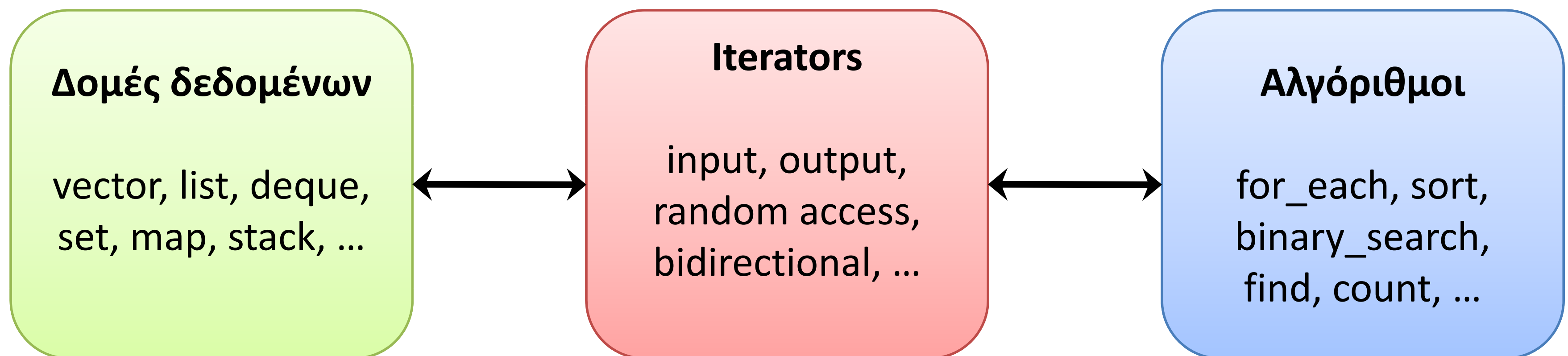
Templates

- Παράδειγμα templated κλάσης

```
template<typename T, typename U>
class pair {
public:
    T first;
    U second;

    pair( const T & x, const U & y ) : first(x), second(y) {}
};

…

//compiler instantiates pair with T=string and U=int
pair<string,int> x;
x.first = “abc”;
x.second = 42;
```

C++ Standard Template Library (STL)

- Ένα από τα πλέον επιτυχημένα παραδείγματα γενικευμένου προγραμματισμού
- Παρέχει ένα σύνολο από:



- οι καλύτεροι δυνατοί αλγόριθμοι και δομές γραμμένοι με τον πιο γενικό τρόπο
- instantiation ανάλογα με την περίπτωση κατά τη μεταγλώττιση

Παράδειγμα: πλήθος εμφανίσεων μιας τιμής σε ένα vector

```
vector<int> v;
vector<int>::iterator b = v.begin(), e = v.end();
long c = count( b, e, 4);

template<class Iter, class T>
long count(Iter first, Iter last, const T& value) {
   long ret=0;
   while ( first != last ) if (*first++ == value) ++ret;
   return ret;
}
```

- γενικευμένη υλοποίηση χωρίς να υστερεί σε απόδοση

Function objects (“Functors”)

- Κλάσεις στις οποίες έχει υπερφορτωθεί ο τελεστής ()
- Έχουν την λειτουργικότητα συναρτήσεων
  - στην ουσία είναι σαν δείκτες σε συναρτήσεις
  - επιπλέον πλεονεκτήματα (π.χ. αποθήκευση state)
- Σε αυτά στηρίζεται η λειτουργία των TBBs
  - παράλληλα αλγοριθμικά μοτίβα υλοποιημένα σαν template functions
  - κώδικας χρήστη υλοποιημένος σαν function object

Σύνοψη

- Γενικά για TBBs
- Tasks
- Γενικευμένος προγραμματισμός και templates
- **parallel_for**
- Εσωτερική λειτουργία βιβλιοθήκης

TBB: Αρχικοποίηση

- Για την χρήση οποιουδήποτε παράλληλου αλγόριθμου της βιβλιοθήκης, απαιτείται η δημιουργία ενός αντικειμένου task_scheduler_init

```
#include <tbb/task_scheduler_init.h>
#define NPROCS 4

int main()
{
   tbb::task_scheduler_init init(NPROCS);
   …
}
```

Παραλληλοποίηση for-loop

- Υπόθεση: εφαρμογή συνάρτησης Foo() σε κάθε στοιχείο ενός πίνακα

- Σειριακός κώδικας

```
float a[100];
for ( int i=0; i!=100; ++i )
    Foo(a[i]);
```

Παραλληλοποίηση for-loop

- tbb::parallel_for
  - χωρίζει τον αρχικό χώρο επαναλήψεων σε μικρότερα κομμάτια και τα εκτελεί παράλληλα
  - template function
- 1ο βήμα: χρειάζεται να δώσουμε μια περιγραφή για το τι δουλειά θα γίνεται σε έναν οποιονδήποτε υποχώρο επαναλήψεων του loop
  - γράφουμε το σώμα του loop στον operator() ενός function object
  - ο operator() είναι παραμετροποιημένος με βάση έναν υποχώρο

```
class ApplyFoo {
   float *const my_a;
public:
   ApplyFoo( float *a ) : my_a(a) {}
   void operator()( const blocked_range<int>& r ) const {
      float *a = my_a;
      for ( int i=r.begin(); i!=r.end(); ++i )
         Foo(a[i]);
}
```

blocked_range<T>: κλάση που εκφράζει 1D γραμμικό range πάνω στον τύπο T

Παραλληλοποίηση for-loop

- 2ο βήμα: κλήση parallel_for

```
float a[100];
parallel_for( blocked_range<int>(0,100),
              ApplyFoo(a)
            );
```

δημιουργία (ανώνυμου) αντικειμένου για την περιγραφή του αρχικού χώρου επαναλήψεων

δημιουργία function object

- η parallel_for αναλαμβάνει:
  - να διασπάσει το αρχικό range σε πολλά μικρότερα
  - να εφαρμόσει παράλληλα το function object σε καθένα από αυτά

Δήλωση parallel_for

```
template <typename Range, typename Body>
void parallel_for(const Range& R,
                  const Body& B );
```

- Απαιτήσεις για το Body B

B::B(const F&)	Copy constructor
B::~B()	Destructor
void B::**operator()** (**Range**& *subrange*) const	Apply B to subrange

- Απαιτήσεις για το Range R

R(const R&)	Copy a range
R::~R()	Destroy a range
bool R::**empty**() const	Is range empty?
bool R::**is_divisible**() const	Can range be split?
R::**R (R& r, split)**	Split r into two subranges

η βιβλιοθήκη παρέχει τις κλάσεις blocked_range, blocked_range2d, blocked_range3d

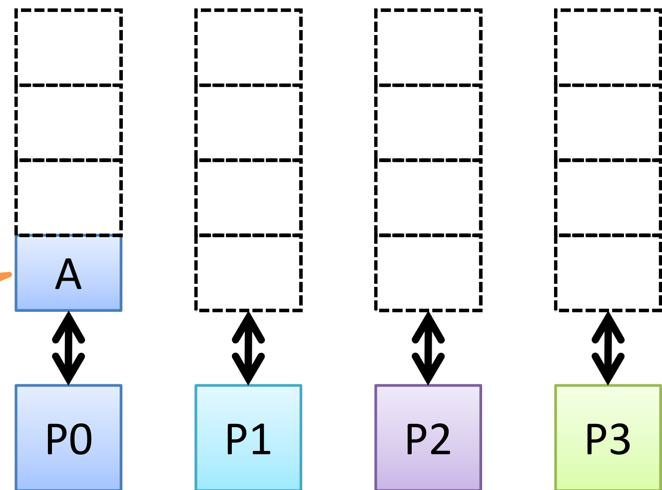
Σύνοψη

- Γενικά για TBBs
- Tasks
- Γενικευμένος προγραμματισμός και templates
- parallel_for
- **Εσωτερική λειτουργία βιβλιοθήκης**

Λειτουργία parallel_for

A

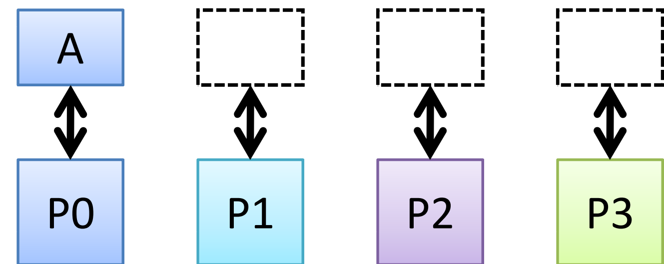
αναδρομική διάσπαση του range, μέχρι να γίνει ≤ ***GrainSize***



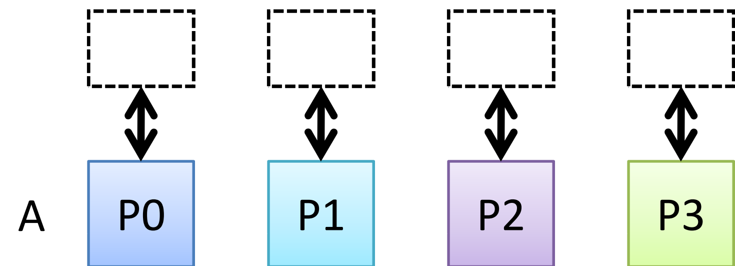
worker threads με double-ended queues

- σε κάθε εκτέλεση της αναδρομής ένα range διασπάται σε 2 subranges
- δημιουργούνται 2 νέα tasks που τοποθετούνται στη βάση της ουράς
- κάθε worker παίρνει το task από τη βάση της τοπικής του ουράς και το ***εκτελεί***
- αν δεν βρει, τότε ***κλέβει*** κάποιο από την κορυφή της ουράς ενός τυχαίου worker

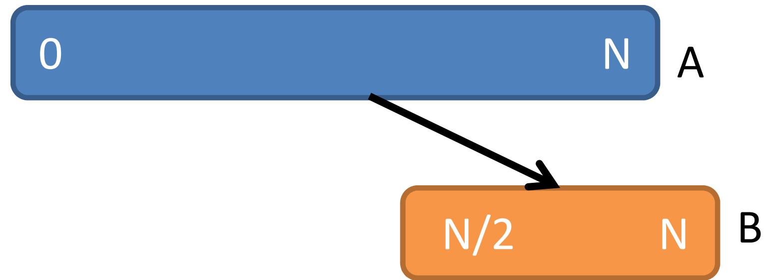
Λειτουργία parallel_for

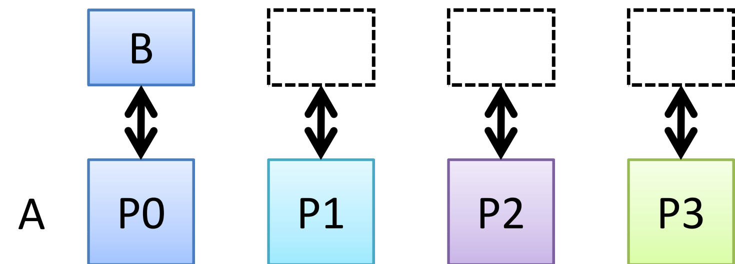


Λειτουργία parallel_for

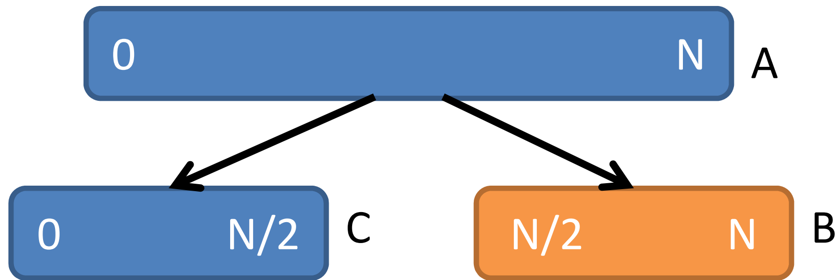


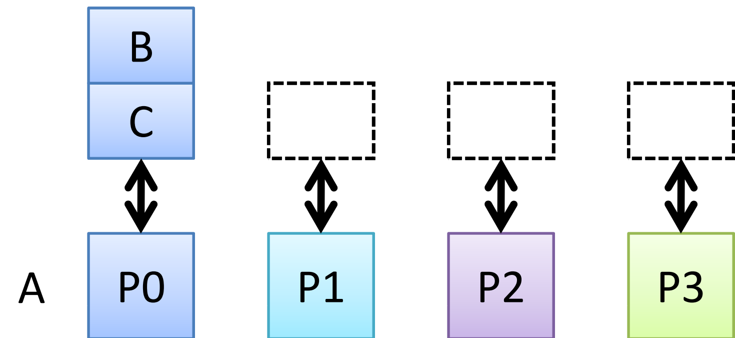
Λειτουργία parallel_for



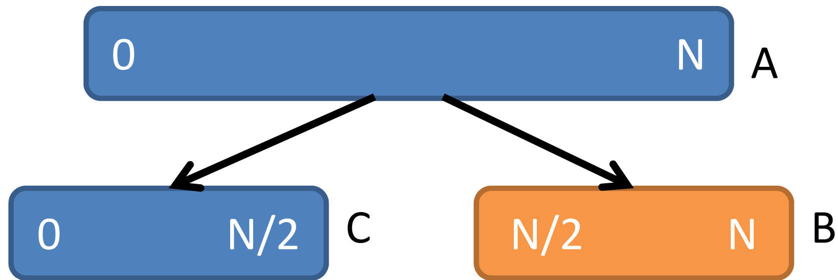


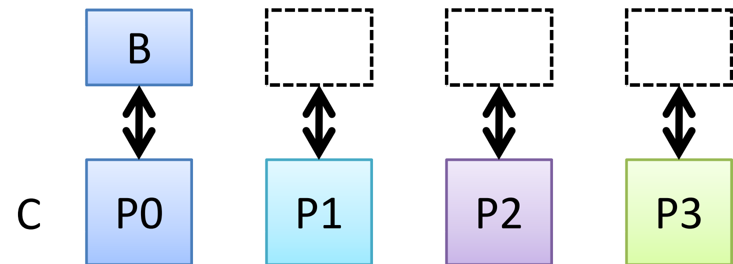
Λειτουργία parallel_for



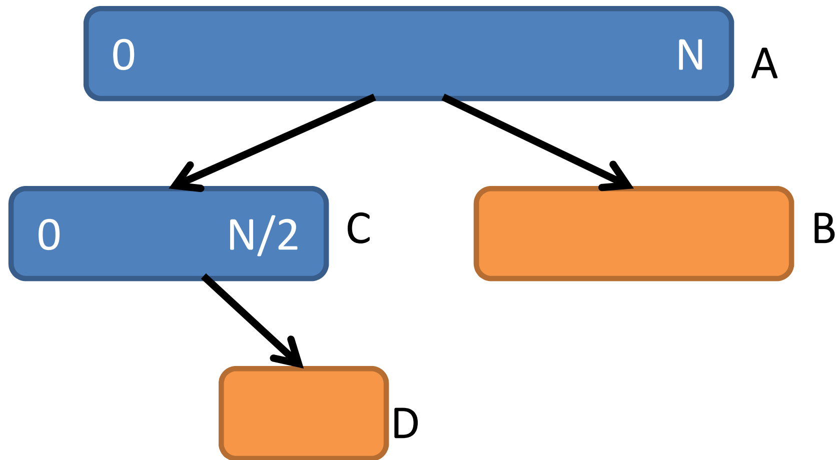


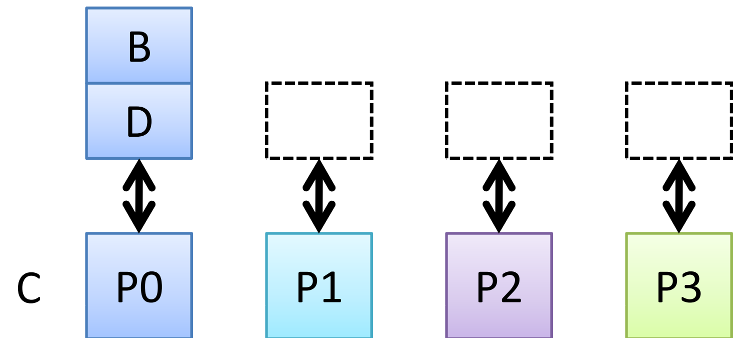
Λειτουργία parallel_for



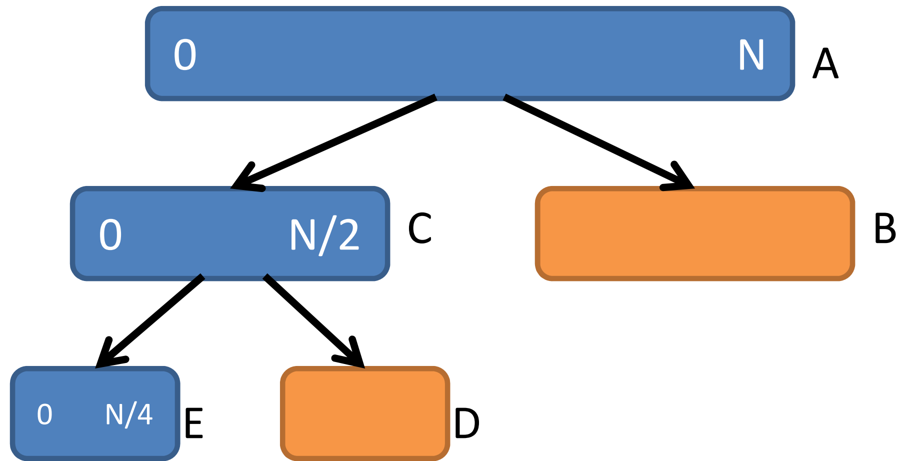


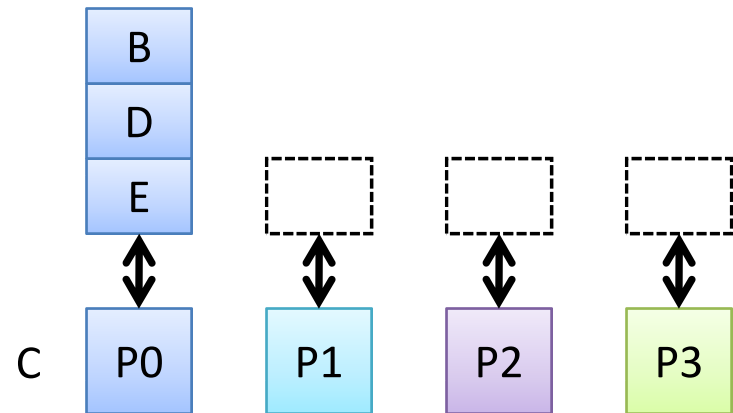
Λειτουργία parallel_for



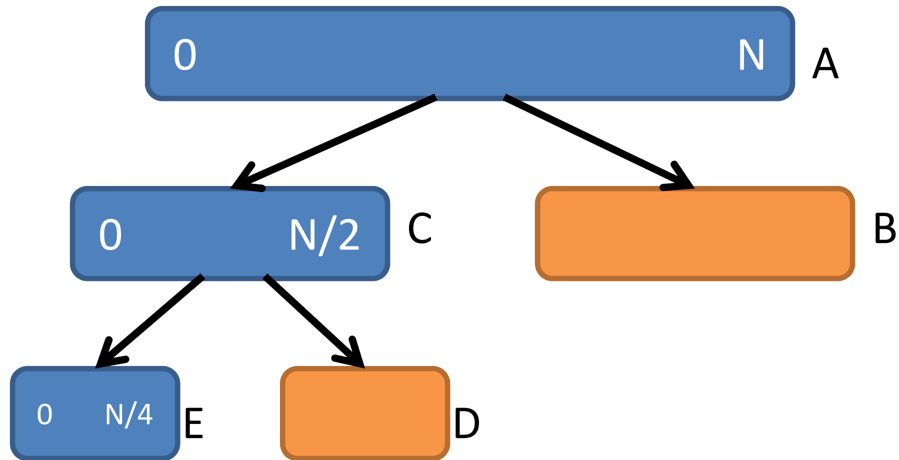


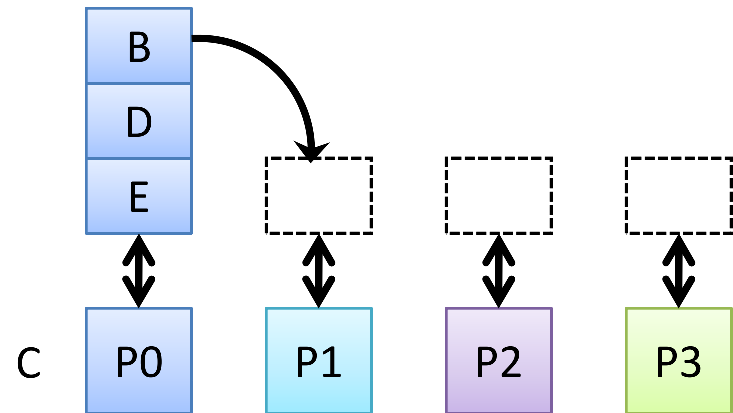
Λειτουργία parallel_for



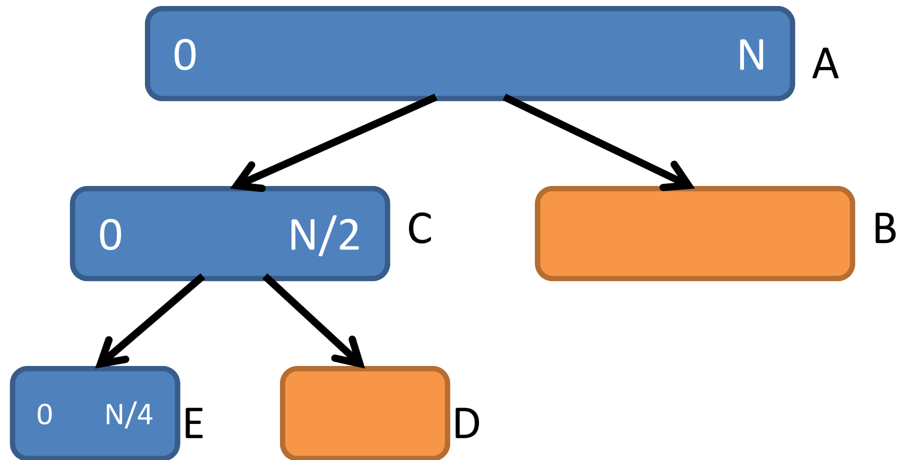


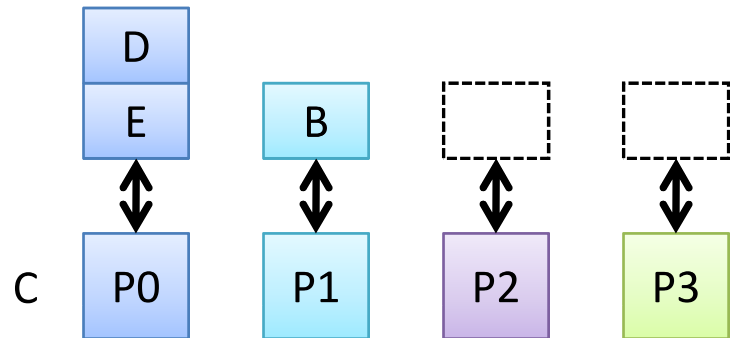
Λειτουργία parallel_for



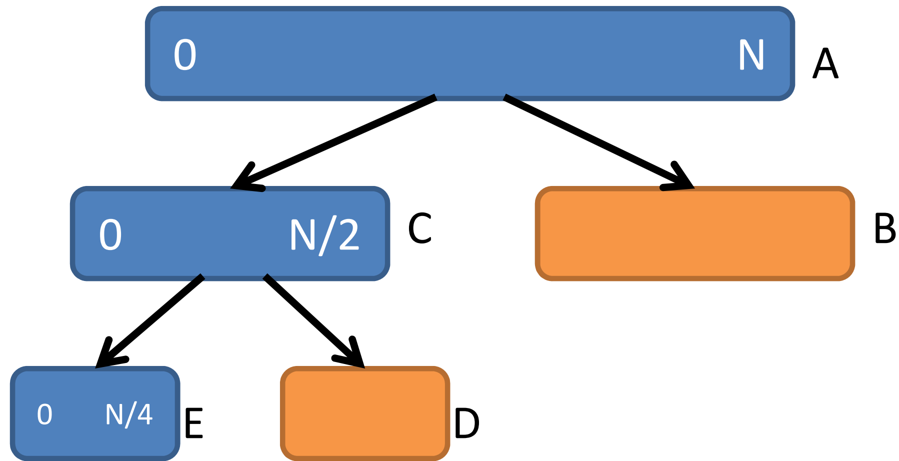


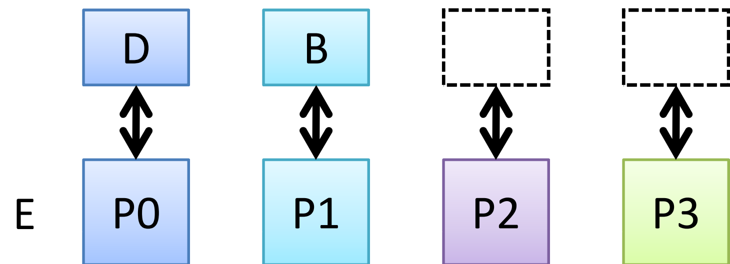
Λειτουργία parallel_for



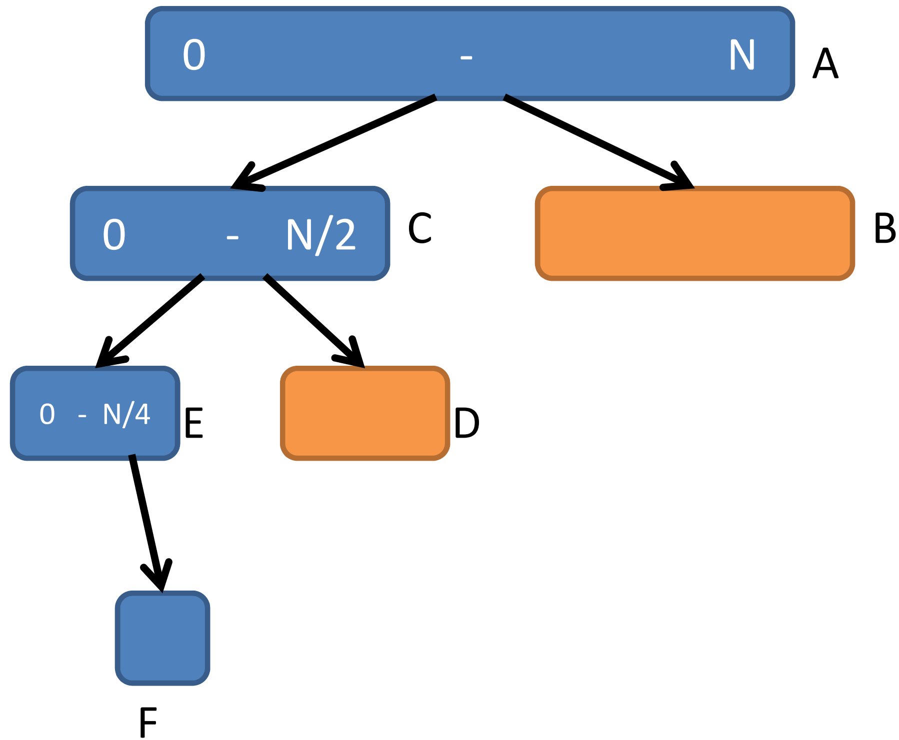


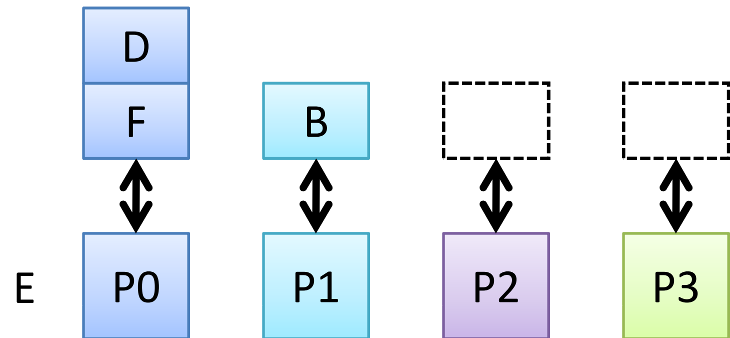
Λειτουργία parallel_for



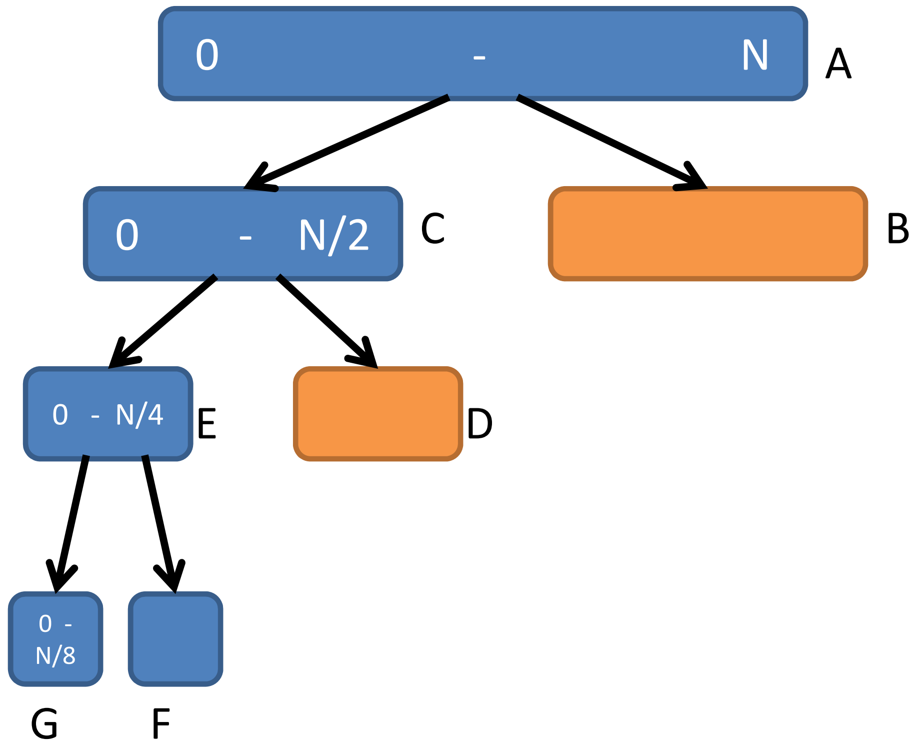


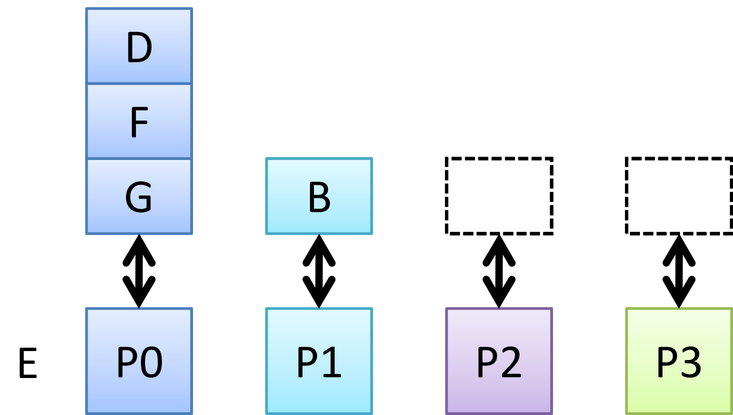
Λειτουργία parallel_for



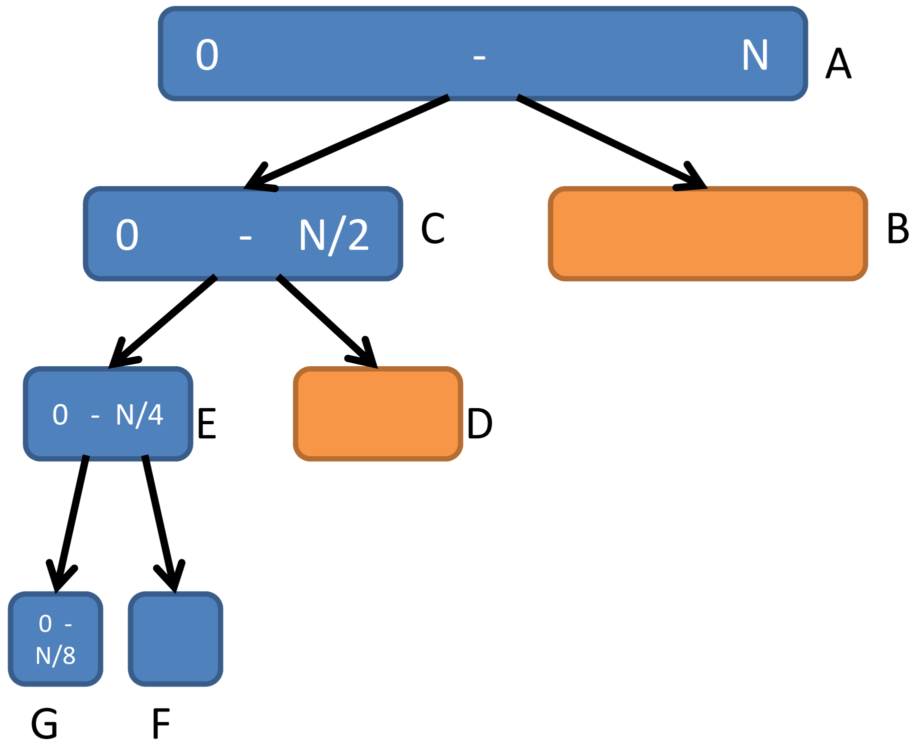


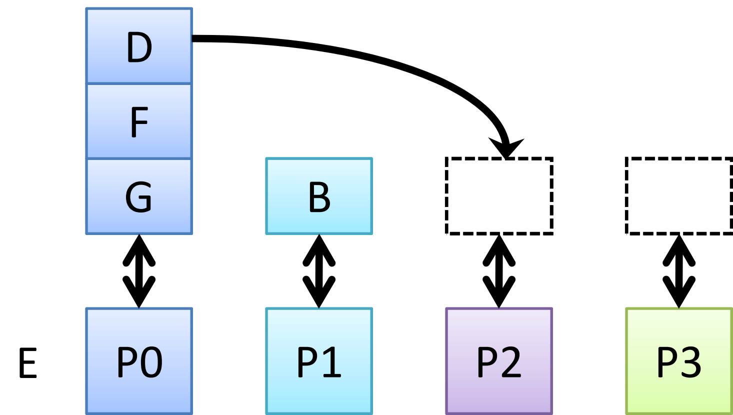
Λειτουργία parallel_for



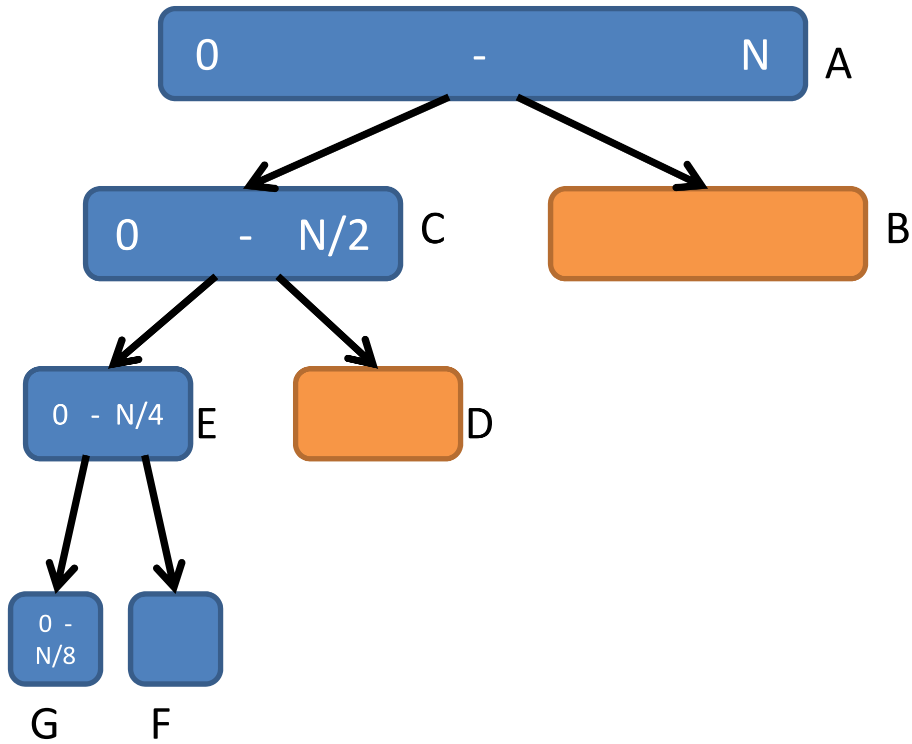


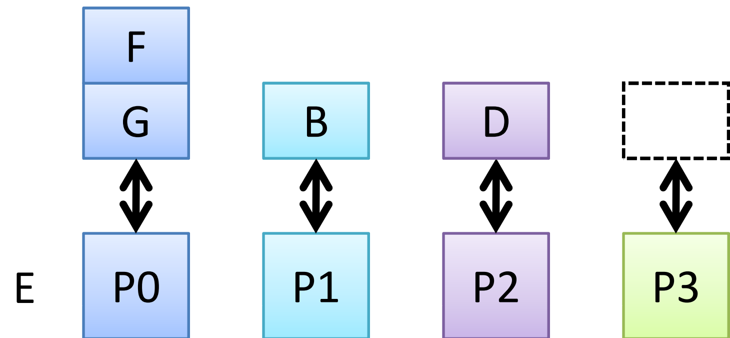
Λειτουργία parallel_for





Λειτουργία parallel_for

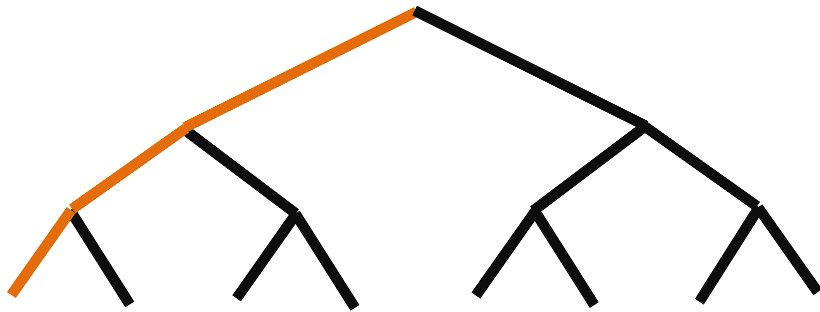


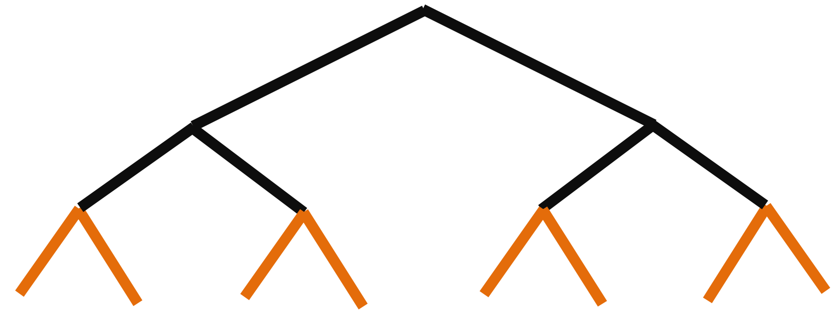


Βασικοί μηχανισμοί

- Work stealing
  - εξασφαλίζει ισοκατανομή φορτίου
- Recursive splitting
  - επιτρέπει την επεξεργασία πάνω σε αυθαίρετα μικρά κομμάτια και τη βέλτιστη εκμετάλλευση της cache («cache-oblivious algorithms»)

Πιθανές σειρές εκτέλεσης των tasks





Depth-first

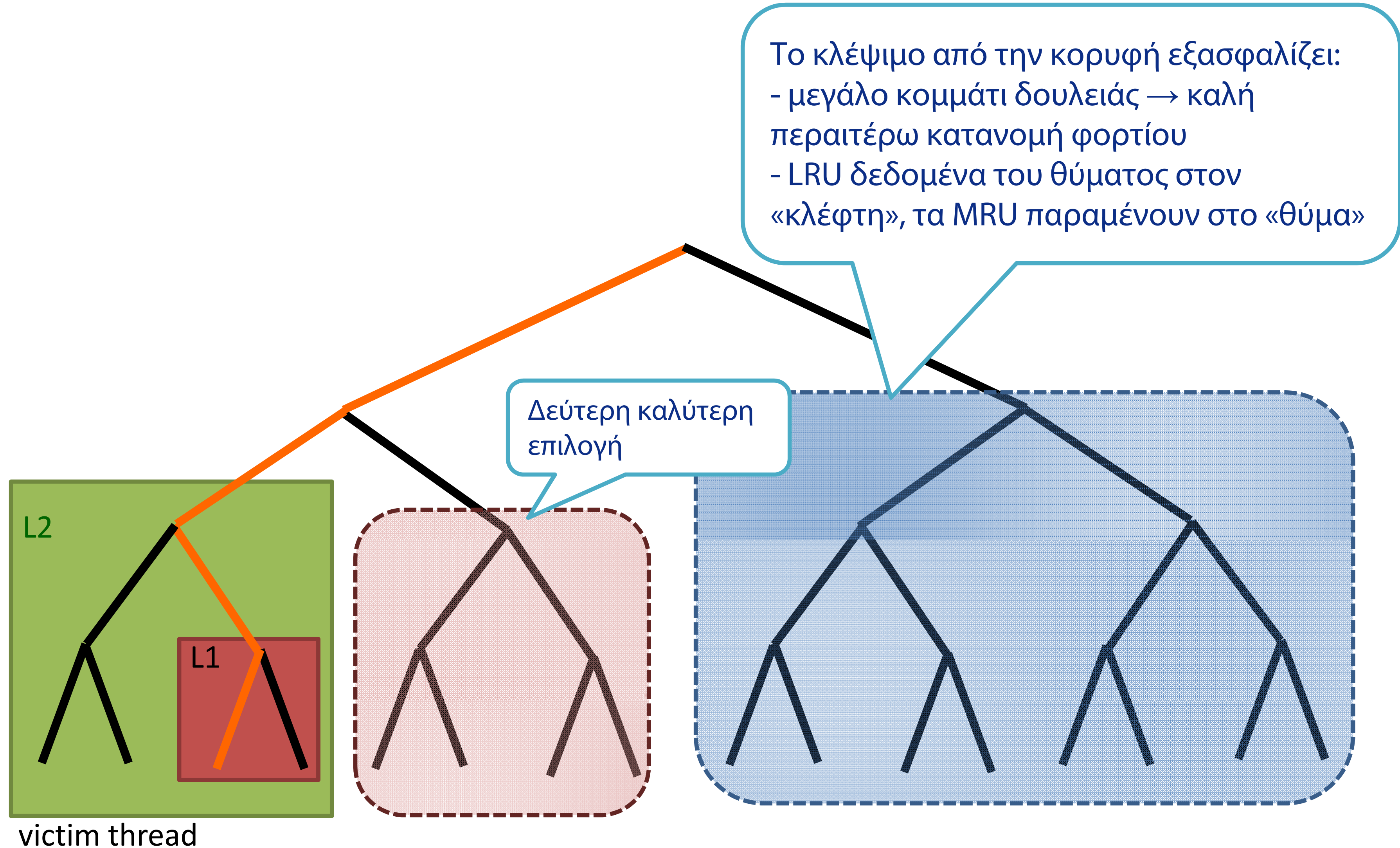
- Απαιτεί λίγο χώρο
- Καλή τοπικότητα αναφορών
- Μηδενικός παραλληλισμός

Breadth-first

- Απαιτεί πολύ χώρο
- Κακή τοπικότητα αναφορών
- Μέγιστος παραλληλισμός

Work depth-first, steal breadth-first

- Εξισορροπεί καλύτερα τις προηγούμενες απαιτήσεις



ΤΕΛΟΣ

Extra slides

Lambda Expressions

- *“C++11 feels like a new language”* [B. Stroustrup]
- Δυνατότητα “in-place” ορισμού συναρτήσεων στο σημείο που χρησιμοποιούνται
  - αντί των function objects
  - ο compiler δημιουργεί μοναδικό, ανώνυμο function object για κάθε lambda expression

```
char s[]="Hello World!";
int nup = 0; //modified by the lambda
for_each( s, s+sizeof(s),
          [&nup] (char c) {
             if (isupper(c)) nup++;
          }
);
cout << nup << " uppercase letters in: "<< s << endl;
```

- gcc 4.5 or newer

Lambda Syntax

- [*capture_mode*] (*formal_parameters*) -> *return_type* {*body*}

[&] ⇒ by-reference
[=] ⇒ by-value
[] ⇒ no capture

Can omit if there are no parameters *and* return type is implicit.

Can omit if return type is void or *code* is "return *expr*;"

Examples

```
[&](float x) {sum+=x;}

[&]{return *p++;}

[=](float x) {return a*x+b;}

[]{return rand();}

[](float x, float y)->float {
   if(x<y) return x;
   else return y;
}
```

Fibonacci, revisited

```
long ParallelFib(long n)
{
    if ( n < 16 )
        return SerialFib(n);
    else {
        int x, y;
        tbb::task_group g;
        g.run( [&]{ x = ParallelFib(n-1);} );
        g.run( [&]{ y = ParallelFib(n-2);} );
        g.wait();
        return x+y;
    }
}
```

parallel_for, revisited

```
tbb::parallel_for(
           tbb::blocked_range<size_t>(0,n),
           [=](const tbb::blocked_range<size_t>& r) {
               for ( size_t i = r.begin(); i != r.end(); ++i )
                   Foo(a[i]);
           }
           );
```

Chunking και loop partitioners

```
parallel_for( blocked_range<size_t>(0,n,G),
              ApplyFoo(a)
              ,some_partitioner())
```

- Chunking: το μέγεθος των ranges στο οποίο σταματά η αναδρομική διάσπαση
  - optional argument στον constructor του blocked_range

- Partitioners
  - optional argument στην parallel_for

1. simple_partitioner
   - recursive binary splitting, εγγυάται ότι $\lceil G/2 \rceil \leq$ chunksize $\leq$ G
2. affinity_partitioner
   - αναθέτει τα ranges με τρόπο ώστε να μεγιστοποιείται το cache locality
3. auto_partitioner (*default*)
   - επιλέγει αυτόματα το grainsize με βάση ευριστική μέθοδο
   - προσπαθεί να ελαχιστοποιήσει το range splitting σε σημείο που να εξασφαλίζεται καλό load balancing

Resources

- Home
  - http://threadingbuildingblocks.org/
- Latest stable release (4.0):
  - https://threadingbuildingblocks.org/file.php?fid=77
  - use sources
- Documentation:
  - https://threadingbuildingblocks.org/documentation.php
  - Getting Started
  - Tutorial
  - Reference
- Intel Software Network blogs:
  - http://software.intel.com/en-us/blogs/tag/tbb/
- Forum:
  - http://software.intel.com/en-us/forums/intel-threading-building-blocks/

Παράδειγμα functor

Template function for iteration

```
template<typename I, typename Functor>
void ForEach( I lower, I upper, const Functor& f ) {
    for ( I i=lower; i<upper; ++i )
        f(i);
}
```

Functor

```
class Accumulate {
    float& acc;
    float* src;
public:
    Accumulate(float& acc_, float* src_) : acc(acc_), src(src_) (0) {}
    void operator()( int i ) { acc += src[i];}
};
```

Pass functor to template function.

```
float a[4] = {1,3,9,27};
float sum = 0.0;
Accumulate A(sum,a);
ForEach( 0, 4, A );
cout << sum;
```