Threading Building Blocks

Computer Science Department, University of Crete

Parallel Programming

Βασίζεται σε slides του Paul Guermonprez

Threading Building Blocks

- C++ template library για ανάπτυξη παράλληλου λογισμικού
- Η βιβλιοθήκη χρησιμοποιεί την έννοια των tasks για την έκφραση του παραλληλισμού
- Τα tasks ανατίθενται σε cores από τη βιβλιοθήκη δυναμικά
  - Διευκολύνει τον προγραμματισμό
  - Καλύτερη χρήση caches

Πλεονεκτήματα της TBB

- Λογικός παραλληλισμός αντί για threads
- Παραλληλισμός για ταχύτητα
- Συμβατότητα με άλλες μορφές threading
- Έμφαση σε data parallelism
- Generics (templates)
- Έτοιμα patterns παραλληλισμού

Μειονεκτήματα της TBB

- Όχι για παραλληλισμό I/O
- Όχι για real-time εφαρμογές
  - “unfair” εκτέλεση των διαθέσιμων tasks, για λόγους βελτιστοποίησης στις caches
- Βιβλιοθήκη – Θέματα compilation

Βασικές Ιδέες

- Ορισμός παραλληλισμού με tasks αντί threads
  - Το σύστημα αντιστοιχεί σε διαθέσιμα threads, υποστηρίζει nested tasks
- Στόχος η κλιμακωσιμότητα (scalability)
  - Χρησιμοποιεί έτοιμα αποδοτικά parallel patterns
  - Αυτόματο load-balancing με work-stealing
- Open Source: Linux, Windows, MacOS, FreeBSD, Solaris, XBox 360, ...

Δομές Παραλληλισμού

- Parallel Algorithms
  - `parallel_for`, `parallel_for_each`, `parallel_reduce`, `parallel_do`, ...
- Flow Graphs
  - data flow
  - functional nodes, buffering nodes, split/join nodes, ...
- Ranges, Partitioners
  - Περιγραφή κατανομής δεδομένων
- Tasks, Task groups
  - Γενικός tasking παραλληλισμός
- Task scheduler
  - Επιλογή προτεραιοτήτων, κλπ.
- Synchronization primitives
  - atomic operations, mutexes, rw_mutexes

Δομές Παραλληλισμού

- Thread Local Storage
  - `combinable`, `enumerable_thread_specific`
- Concurrent containers
  - `concurrent_hash_map`, `concurrent_queue`, `concurrent_bounded_queue`, `concurrent_priority_queue`, `concurrent_vector`, `concurrent_unordered_map`, `concurrent_unordered_set`
- Memory allocators
  - `tbb_allocator`, `cache_aligned_allocator`, `scalable_allocator`

Task-based Παραλληλισμός

- Τα tasks είναι παρόμοια με τα tasks στο OpenMP
  - Light-weight units of work
  - User-level
- TBB scheduler
  - Αντιστοιχεί tasks σε threads
  - Χειρίζεται το thread pool
- Ο TBB scheduler είναι *unfair*
  - Προτιμά tasks που έχουν καλό cache locality
- Αποφεύγει oversubscription και undersubscription με την τεχνική *work-stealing*
  - Το σύστημα βλέπει ποιοί cores είναι idle και “κλέβει” tasks από τους υπερφορτωμένους

Αλγόριθμοι TBB

High-level parallel patterns
Έτοιμα patterns σε generic κώδικα για τα συχνότερα είδη παραλληλισμού

- `parallel_for`, `parallel_for_each`:
  - Παράλληλη εκτέλεση βρόχων με ανεξάρτητα iterations
- `parallel_reduce`:
  - Παράλληλη εκτέλεση βρόχων με ανεξάρτητα iterations που στο τέλος συνδυάζουν τα αποτελέσματά τους
- `parallel_scan`:
  - Υπολογισμός του parallel prefix
- `parallel_pipeline`:
  - Παράλληλο pipeline που δημιουργείται από σειριακά και παράλληλα κομμάτια

Αλγόριθμοι TBB

- `parallel_do`:
  - Παράλληλη εκτέλεση βρόχων με ανεξάρτητα iterations όπου μπορεί να προκύψουν νέα iterations κατά την εκτέλεση
- `parallel_sort`:
  - Παράλληλη ταξινόμηση
- `parallel_invoke`:
  - Παράλληλη εκτέλεση γενικών συναρτήσεων μέσω function objects ή function pointers

Parallel For

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>

template <typename Range, typename Body>
void parallel_for(const Range& range, const Body& f
  [, task_group_context& context])
```

```
#include "tbb/parallel_for.h"

template <typename Index, typename Function>
void parallel_for(Index first, Index last
  [, Index step], const Function& f);
```

- Χωρίζει τα ranges σε subranges και μοιράζει τα subranges στα διαθέσιμα threads
  - Ισομοιρασμένοι υπολογισμοί
  - Καλή χρήση caches
  - Scalability

Range

- Το Range είναι generic κώδικας (template)
- Η βιβλιοθήκη περιέχει έτοιμα ranges για τις συχνότερες περιπτώσεις
  - `blocked_range`, `blocked_range2d`, `blocked_range3d`
- Το πρόγραμμα μπορεί να ορίσει custom ranges υλοποιώντας τις μεθόδους:

```
MyRange::MyRange (const MyRange&) // Copy constructor

MyRange::~MyRange() // Destructor

bool MyRange::is_empty() const // True if range is empty

// True if range can be partitioned
bool MyRange::is_divisible() const

// Splitting constructor; splits r into two subranges
MyRange::MyRange(MyRange& r, split)
```

Granularity

- Παρόμοιο με την παράμετρο grain/chunk του OpenMP
- Μέρος του `blocked_range`
- Το χρησιμοποιούν οι `parallel_for` και `parallel_reduce`, όχι άμεσα ο task scheduler
  - Σκοπός του grain size είναι το overhead amortization όχι το load-balancing
  - Το granularity μετρά loop iterations
- Τυπικά "good enough" επιλογή, σωστή τάξη μεγέθους
- Σωστή επιλογή granularity
  - Μέτρηση του single-processor performance
  - Καλύτερα λίγο μεγαλύτερο από το optimal
  - Αποφυγή μεγάλων overheads σε εκτελέσεις με 1 επεξεργαστή

Παράδειγμα parallel_for

```
const int N = 100000;

void change_array(float array, int M) {
  for (int i = 0; i < M; i++){
    array[i] *= 2;
  }
}

int main (){
  float A[N];
  initialize_array(A);
  change_array(A, N);
  return 0;
}
```

Παράδειγμα parallel_for

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>

using namespace tbb;

void parallel_change_array(float *array, size_t M) {
  parallel_for(blocked_range<size_t>(0, M, IdealGrainSize),
    [=](const blocked_range<size_t>& r) -> void {
      for(size_t i = r.begin(); i != r.end(); i++ )
        array[i] *= 2;
    }
  );
}
```

Task Scheduler

- Η βιβλιοθήκη δημιουργεί ένα task scheduler αυτόματα όταν χρειάζεται threads και τον καταστρέφει αυτόματα όταν δεν χρειάζονται πια
- Ο προγραμματιστής μπορεί να ελέγξει τη δημιουργία/καταστροφή του task scheduler (για αποφυγή overhead)

```
#include <tbb/task_scheduler_init.h>

using namespace tbb;

int main (){
  task_scheduler_init init; //threads creation
  float A[N];
  initialize_array(A);
  parallel_change_array(A, N);
  return 0;
} // out of scope -> threads destruction
```

- Παράμετρος στον constructor: maximum threads

Προγραμματισμός με generics ή lambda

Generics

```
class ChangeArrayBody {
  float *array;
public:
  ChangeArrayBody(float *a): array(a) {}
  void operator()( const blocked_range<size_t>& r ) const{
    for (size_t i = r.begin(); i != r.end(); i++ ){
      array[i] *= 2;
    }
  }
};

void parallel_change_array(float *array, size_t M) {
  parallel_for(
    blocked_range<int>(0, M, IdealGrainSize),
    ChangeArrayBody(array)
  );
}
```

Προγραμματισμός με generics ή lambda

Lambda

```
void parallel_change_array(float *array, size_t M) {
  parallel_for(blocked_range<size_t>(0, M, IdealGrainSize),
    [=](const blocked_range<size_t>& r) -> void {
      for(size_t i = r.begin(); i != r.end(); i++ )
        array[i] *= 2;
    }
  );
}
```

Generics και Lambda functions

- Και τα δύο δίνουν την ίδια ταχύτητα και overheads
- “Syntactic sugar” τρόπος για να εκφραστεί το ίδιο νόημα
- Κάποιες φορές χρειάζεται generic έκφραση, δε γίνεται με lambda
  - Παραδείγματα με lambda όταν γίνεται (μικρότερα)
- Lambda functions: υποστήριξη στη C++11, παράμετρος `-std=c++0x` στον compiler

Parallel Reduce

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_reduce.h>

template<typename Range, typename Body>
void parallel_reduce( const Range& range, Body& body );

template<typename Range, typename Value, typename RealBody, typena
Value parallel_reduce(
  const Range& range,
  const Value& identity,
  const RealBody& real_body,
  const Reduction& reduction
);
```

- Η δομή `parallel_reduce` χωρίζει το αρχικό range σε subranges όπως και η `parallel_for`
- Η συνάρτηση `RealBody` καλείται για τα subranges, και τα αποτελέσματά της "ενώνονται" από τη συνάρτηση `reductionFunc`

Παράδειγμα parallel_reduce

```
#include <limits>

// Find index of smallest element in a[0...n-1]
size_t serialMinIndex(const float a[], size_t n) {
  float value_of_min = numeric_limits<float>::max();
  size_t index_of_min = 0;
  for(size_t i = 0; i < n; ++i) {
    float value = a[i];
    if(value < value_of_min) {
      value_of_min = value;
      index_of_min = i;
    }
  }
  return index_of_min;
}
```

Παράδειγμα parallel_reduce

```
#include <limits>
#include <tbb/blocked_range.h>
#include <tbb/parallel_reduce.h>

size_t parallelMinIndex( const float a[], size_t n ) {
  return parallel_reduce(blocked_range<size_t>(0,n,10000), size_t(0),
    [=](blocked_range<size_t> &r, size_t index_of_min) -> size_t {
      float value_of_min = a[index_of_min];
      for(size_t i=r.begin();i!=r.end();++i) {
        float value = a[i];
        if( value < value_of_min ) { // accumulate result
          value_of_min = value;
          index_of_min = i;
        }
      }
      return index_of_min;
    },
    [=](size_t i1, size_t i2) {        // join
      return (a[i1]<a[i2])? i1:i2;
    }
  );
}
```

Parallel Sort

```
#include <tbb/parallel_sort.h>

template<typename RandomAccessIterator>
void parallel_sort(
  RandomAccessIterator begin,
  RandomAccessIterator end
);

template<typename RandomAccessIterator, typename Compare>
void parallel_sort(
  RandomAccessIterator begin,
  RandomAccessIterator end,
  const Compare& comp
);

// ...
```

- Η δομή `parallel_sort` χωρίζει το αρχικό range σε subranges όπως και η `parallel_for`
- Παράλληλη quicksort — Balance με work-stealing

Tasks

- Tasks: Units of work
- Διαχωρίζουν την έκφραση του παραλληλισμού από την εκτέλεσή του
- TBB task scheduler
  - Per thread
  - Unfair, non-preemptive
  - Χαμηλό overhead, αντιστοίχιση tasks/threads
  - Load-balance: work-stealing

Παράδειγμα: Fibonacci (πάλι)

- Toy benchmark
- Unbalanced υπολογισμός

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

parallel_invoke

```
void parallel_fib(int n, long &sum) {
  if (n < 2)
    sum = n;
  else if (n < 1000)
    sum = serial_fib(n);
  else {
    long x, y;
    tbb::parallel_invoke(
      [&] () { parallel_fib(n-1, x); },
      [&] () { parallel_fib(n-2, y); }
    );
    sum = x + y;
  }
}
```

parallel_invoke

- Οι συναρτήσεις που δέχεται η `parallel_invoke` δεν μπορούν να δέχονται ή να επιστρέφουν τιμές.
  - Εύκολη λύση με lambda εκφράσεις, capture μεταβλητών by reference
- Εκτελούνται ως χωριστά tasks από το διαθέσιμο παραλληλισμό
- Στο τέλος της `parallel_invoke` έχουν εκτελεστεί όλα

Task Groups

- Ένας τρόπος να ομαδοποιηθούν tasks παράλληλα μεταξύ τους
  - Διαφορετικά task groups μπορεί να μην είναι παράλληλα (ή να είναι)
- Μπορεί να μεγαλώσει δυναμικά
- Για πάρα πολλά tasks στο ίδιο task_group μπορεί να γίνει αργό
  - Task spawn: σειριακό, synchronization operation

Task Groups Fibonacci

```
#include <tbb/task_group.h>

using namespace tbb;

int Fib(int n) {
  if( n<2 ) {
    return n;
  } else {
    int x, y;
    task_group g;
    g.run([&]{x=Fib(n-1);}); // spawn a task
    g.run([&]{y=Fib(n-2);}); // spawn another task
    g.wait();                //  wait for both tasks to complete
    return x+y;
  }
}
```

Tasks χωρίς lambda

- “Παραδοσιακό” API
  - Task class
  - Allocate, Construct
- Recursive

```
long parallel_fib(long n) {
  long sum;
  FibTask& a = *new(Task::allocate_root()) FibTask(n, &sum);
  Task::spawn_root_and_wait(a);
  return sum;
}
```

Tasks χωρίς lambda

```
public:
  const long n;
  long *const sum;
  FibTask(long n_, long *sum_) : n(n_), sum(sum_) {} // constructor

  task* execute() { // override virtual execute for task body
    if( n < 1000 ) {
      *sum = serial_fib(n);
    } else {
      long x, y;
      FibTask& a = *new(allocate_child()) FibTask(n-1, &x);
      FibTask& b = *new(allocate_child()) FibTask(n-2, &x);
      // keep count of children tasks (count +1 for waiting on 2 children)
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      sum = x + y;
    }
    return NULL;
  }
};
```

Παράλληλα Containers

- Η βιβλιοθήκη TBB περιέχει παράλληλες υλοποιήσεις χρήσιμων containers
  - Τα C++ STL containers δεν είναι γραμμένα για χρήση με παραλληλισμό
  - Η χρήση STL containers παράλληλα μπορεί να τα κάνει corrupt
  - Συνήθης χρήση: wrap με lock
    - Σειριοποίηση
    - Δεν κλιμακώνεται, bottleneck, congestion
- TBB Containers
  - Fine-grain συγχρονισμός
  - Χαμηλότερη απόδοση σε σειριακή χρήση
  - Καλύτερη κλιμακωσιμότητα σε παράλληλο κώδικα
  - Δεν χρειάζονται τον TBB scheduler
  - Συμβατά με pthreads, OpenMP

Container API για παραλληλισμό

- Η C++ STL περιέχει containers με APIs που απαιτούν σειριακή εκτέλεση
- Παράδειγμα

```
extern std:queue q;
if (!q.empty()) {
  // race: first thread that pop()s will make queue empty
  item = q.front();
  q.pop();
}
```

- Λύση: `concurrent_queue`
  - Νέο API: `pop_if_present()`

Concurrent Queue

- `concurrent_queue<T>`
- Διατηρεί την τοπική FIFO σειρά
  - Αν ένα thread εισάγει δύο τιμές και ένα άλλο thread βγάλει, η μεταξύ τους σειρά θα είναι ίδια
- Η μέθοδος `push(const T&)` εισάγει αντίγραφο του αντικειμένου στο τέλος της ουράς
- Δύο είδη pop:
  - `pop(T&)`: Blocking, περιμένει
  - `pop_if_present(T&)`: Non-Blocking, δεν περιμένει
- Η μέθοδος `size()` επιστρέφει ακέραιο αριθμό *με πρόσημο*, signed
  - Αρνητικό αποτέλεσμα μετρά πόσα `pop` περιμένουν

Καλή χρήση Concurrent Queue

- Κάθε queue είναι ουσιαστικά bottleneck
  - Πρέπει να διατηρεί την first-in-first-out σειρά
- Κάθε thread που προσπαθεί να πάρει μια τιμή μπορεί να χρειαστεί να περιμένει μέχρι να υπάρξει μια τιμή
- Αν ένα thread εισάγει μια τιμή και ένα άλλο thread την βγάλει, τα δεδομένα πρέπει να μεταφερθούν στον πυρήνα όπου εκτελείται το 2ο
- Συνήθως τα queues αφήνουν τα δεδομένα να φύγουν από την cache πριν χρησιμοποιηθούν (από ένα μέγεθος και πάνω)
- Use wisely
  - Ίσως χρειάζεται rewrite με χρήση `parallel_pipeline`

Παράδειγμα Concurrent Queue

```
#include<iostream>
#include<tbb/concurrent_queue.h>

using namespace tbb;

int main() {
  concurrent_queue<int> queue;
  int j;

  for (int i = 0; i < 10; i++)
    queue.push(i);

  while(!queue.empty()) {
    queue.pop(&j);
    cout << "From queue:" << j << endl;
  }

  return 0;
}
```

Concurrent Vector

- `concurrent_vector<T>`
- Λειτουργεί όπως ένας πίνακας από `T` που μπορεί να μεγαλώσει δυναμικά
  - Μέθοδος `grow_by(size_type)` προσθέτει στοιχεία στο τέλος
  - Μέθοδος `grow_to_at_least(size_type)` προσθέτει στοιχεία μέχρι το ζητούμενο μέγεθος
  - Μέθοδος `size()` επιστρέφει τον αριθμό στοιχείων που περιέχει το vector
  - Μέθοδος `empty()` επιστρέφει `size() == 0`
- Τα στοιχεία δεν μετακινούνται ποτέ μέχρι να αφαιρεθούν
  - Μπορεί να γίνεται πρόσβαση παράλληλα με grow
  - Η μέθοδος `clear()` δεν είναι thread-safe για παράλληλη εκτέλεση με resize

Παράδειγμα Concurrent Vector

```
void append( concurrent_vector<char>& V, const char* str) {
  size_type n = strlen(str) + 1;
  memcpy( &V[V.grow_by(n)], str, n+1 );
}
```

- Προσθήκη ενός str στον `concurrent_vector`
- Μεγαλώνει τον vector ώστε να χωρά το νέο αλφαριθμητικό
  - Η `grow_by` επιστρέφει το προηγούμενο μέγεθος
  - Εκεί που μπαίνουν τα νέα στοιχεία
  - Γίνεται copy το string στο νέο χώρο
  - Ταυτόχρονα επιτρέπονται νέα grow

Concurrent Hash Map

- `concurrent_hash_map<Key, T, HashCompare>`
- Αντιστοιχεί το `Key` στο στοιχείο τύπου `T`
- Μπορεί να γενικεύσει τη σύγκριση κλειδιών με την κλάση `HashCompare`
  - Μέθοδος `hash()` αντιστοιχεί το `Key` σε κάποιο `size_t`
  - Μέθοδος `equal()` επιστρέφει true αν δυο κλειδιά είναι ίδια
- Επιτρέπει ταυτόχρονα `count()`, `find()`, `insert()`, και `erase()`
  - Οι `find()` και `insert()` χρησιμοποιούν "smart pointer" που λειτουργεί ως lock για το κάθε στοιχείο
    - Ως `accessor` δίνει read-write πρόσβαση
    - Ως `const_accessor` δίνει read-only πρόσβαση
  - Το lock γίνεται release όταν καταστραφεί ο smart pointer

Παράδειγμα Concurrent Hash Map

```
typedef concurrent_hash_map<string, int> myMap;
myMap table;
string newstring;
int place = 0;

//  ...

  while ( getNextString(&newString)) {
    myMap::accessor a;
    if  (table.insert(a, newString)) // new string inserted
      a->second = ++place;
  }
```

- Αν η `insert` επιστρέψει true, έγινε εισαγωγή του νέου αλφαριθμητικού
  - Το περιεχόμενο του hash map για το αλφαριθμητικό είναι η σειρά της λέξης από την `getNextString`
- Αν η `insert` επιστρέψει false, υπάρχει ήδη

Παράδειγμα Concurrent Hash Map

```
myMap table;
string s1, s2;
int p1, p2
//  ...
{
  myMap::const_accessor a; // read lock
  myMap::const_accessor b;
  if (table.find(a, s1) && table.find(b, s2)) {
    p1 = a->second; p2 = b->second;
    if (p1 < p2) cout << s1 << " seen before " << s2 << endl;
    else cout << s2 << " seen before " << s1 << endl;
  }
  else cout << "One or both strings not seen before" << endl;
}
```

- Αν η `find` επιστρέψει true, το αλφαριθμητικό υπάρχει στο hash table
- Το πεδίο `second` του pair που περιέχει ο accessor είναι ο αριθμός σειράς του αλφαριθμητικού

Scalable Memory Allocator

- Η δυναμική διαχείριση μνήμης μπορεί να γίνει bottleneck
  - Τα threads δεσμεύουν μνήμη στο heap με mutual exclusion
  - Μόνο ένα malloc() ανά πάσα στιγμή
- False sharing: Πάνω από ένα threads γράφουν διαφορετικά σημεία του ίδιου cache line
  - Ping-pong του cache line στις διάφορες caches, καθυστερεί όλους
- Η βιβλιοθήκη TBB περιέχει δύο allocators
  - Παρόμοια με τον STL `std::allocator`
  - `scalable_allocator`
    - Κλιμακώσιμο, αλλά χωρίς προστασία από false sharing
    - Η μνήμη δεσμεύεται για κάθε thread από διαφορετικό memory pool
  - `cache_aligned_allocator`
    - Κλιμακωσιμότητα και προστασία από false sharing

API - scalable allocator

```
#include <tbb/scalable_allocator.h>
template<typename T> class scalable_allocator;

void *scalable_malloc( size_t size );
void scalable_free( void *ptr );
void *scalable_realloc( void *ptr, size_t size );
void *scalable_calloc( size_t nobj, size_t size );

T* A::allocate( size_type n, void* hint=0 ) // Allocate space for n values
void A::deallocate( T* p, size_t n ) // Deallocate n values from p
void A::construct( T* p, const T& value )
void A::destroy( T* p )
```

Παράδειγμα χρήσης

```
#include <tbb/scalable_allocator.h>
typedef char _Elem;
typedef std::basic_string<_Elem,
                          std::char_traits<_Elem>,
                          tbb::scalable_allocator<_Elem>> MyString;
//  ...

int *p;
MyString str1 = "qwertyuiopasdfghjkl";
MyString str2 = "asdfghjklasdfghjkl";
p = tbb::scalable_allocator<int>().allocate(24);
```

Flow Graph

- Data Flow Graph pattern
  - Μερικές εφαρμογές εκφράζονται ως κόμβοι γράφου που ανταλλάζουν μηνύματα
  - Reactive προγραμματισμός: event $\rightarrow$ response
  - Task graph, πολύπλοκες σχέσεις μεταξύ tasks
  - Εφαρμογές actor-based, κόμβοι actors που αντιδρούν στο περιβάλλον
  - κλπ

Είδη κόμβων του Flow Graph

- Functional: εκτελούν μια λειτουργία
  - `source_node`: Εκτελεί μια συνάρτηση και παράγει το αποτέλεσμα
  - `continue_node`: Παράγει έξοδο όταν λάβει είσοδο από όλες τις εισόδους
  - `function_node`: Εκτελεί μια συνάρτηση πάνω στην είσοδο και επιστρέφει το αποτέλεσμά της σε όλες τις εξόδους
  - `multifunction_node`: Εκτελεί μια συνάρτηση πάνω στην είσοδο και επιστρέφει πολλά αποτελέσματα σε πολλές εξόδους
- Buffering: λειτουργία αποθήκευσης δεδομένων
  - `buffer_node`: Buffer μηνυμάτων, έξοδος σε έναν-έναν, όχι απαραίτητα με τη σειρά (παράλληλα)
  - `queue_node`: FIFO buffer, έξοδος με τη σειρά
  - `priority_queue_node`: Queue με προτεραιότητα
  - `sequencer_node`: Priority με βάση το sequence που παράγεται με custom συνάρτηση

Είδη κόμβων του Flow Graph

- Split/Join: συνδυασμός αποτελεσμάτων διαφορετικών μονοπατιών στο γράφο, ή δημιουργία μονοπατιών
  - `queuing_join`: Είσοδος από FIFO queues, όταν όλες έχουν στοιχείο, έξοδος tuple
  - `reserving_join`: Κράτηση των εισόδων και verification όταν είναι όλες διαθέσιμες, αλλιώς release
  - `tag_matching_join`: Υπολογισμός tag/key με custom hash() και επιλογή με βάση tag
  - `split_node`: Είσοδος από tuple και έξοδος σε χωριστά κομμάτια
  - `indexer_node`: Αναμετάδοση κάθε εισόδου από κάθε θύρα εισόδου σε όλες τις εξόδους ως union

Είδη κόμβων του Flow Graph

- Other
  - broadcast_node: Αναμετάδοση κάθε εισόδου σε όλες τις εξόδους
  - write_once_node: Buffer μεγέθους 1, δεν μπαίνει 2ο περιεχόμενο αν δεν αδειάσει το πρώτο
  - overwrite_node: Buffer μεγέθους 1 που κρατά μόνο το τελευταίο push
  - limiter_node: Όπως το broadcast_node αλλά επιλέγει N μηνύματα, μετά από N μηνύματα σταματά να δέχεται είσοδο

Παράδειγμα Flow Graph

```
#include <iostream>
#include <tbb/flow_graph.h>
using namespace std;
using namespace tbb::flow;
int main() {
  graph g;
  continue_node<continue_msg> h(g,
    [](const continue_msg&) { std::cout<<"Hello "; }
  );
  continue_node<continue_msg> w(g,
    [](const continue_msg&) { std::cout<<"Flow Graph World\n"; }
  );
  make_edge(h, w);
  h.try_put(continue_msg());
  g.wait_for_all();
}
```

Συγχρονισμός

- Μερικές φορές υπάρχει ανάγκη παράλληλα tasks να έχουν πρόσβαση σε shared data
  - Mutual exclusion, αποφυγή data races
- Η TBB παρέχει high-level abstraction για hardware synchronization
  - Atomic update μιας μόνο μεταβλητής

Primitives Συγχρονισμού

- Scoped locks
  - Lock range που εξαρτάται από το lifetime του αντικειμένου (scope)
  - Έξοδος από το scope καλεί τον destructor, exception-safe
  - Ελαχιστοποίηση του lock lifetime: λιγότερο congestion
  - Πολλά πιθανά patterns
    - Spin-locks, queue-locks
    - Writer, reader/writer locks
    - Scoped wrapper της native συνάρτησης

Atomic<T>

- `atomic<T>`
- Είτε τύπος δείκτη, είτε βασικός τύπος
- 8, 16, 32, 64-bit integers
- type-safe
- `= x` και `x =`: read/write
- `x.fetch_and_store(y)`: αντικατάσταση τιμής, επιστροφή της προηγούμενης
- `x.fetch_and_add(y)`: αύξηση τιμής, επιστροφή της προηγούμενης
- `x.compare_and_swap(y, p)`: conditional αντικατάσταση με νέα τιμή, επιστροφή της προηγούμενης

```
atomic<int> i;
int z = i.fetch_and_add(2);
```

TBB Mutex

- Αντικείμενα C++ που βασίζονται σε scoped locking
- Χρησιμοποιούνται για τη διαχείριση locks, παρέχουν mutual exclusion

```
M()                                  // Construct unlocked mutex
~M()                                 // Destroy unlocked mutex
typename M::scoped_lock              // Corresponding scoped_lock type
M::scoped_lock ()                    // Construct lock w/out acquiring a mutex
M::scoped_lock (M&)                  // Construct lock and acquire lock on mutex
M::~scoped_lock ()                   // Release lock if acquired
M::scoped_lock::acquire (M&)         // Acquire lock on mutex
M::scoped_lock::release ()           // Release lock
```

Είδη mutex

- Fair
  - Τα threads εκτελούν το critical region με τη σειρά που το φτάνουν
  - Unfair mutexes επιτρέπουν λιγότερα context switch αφήνοντας threads που τρέχουν ήδη να εκτελέσουν πρώτα το region
- Reentrant
  - Το thread που έχει κλειδώσει το mutex μπορεί να το ξανακλειδώσει
  - Χρήσιμο σε αναδρομικό κώδικα
- Spin
  - Busy-wait χωρίς context switch
  - Καλύτερο για μικρή αναμονή
  - Κατανάλωση cpu resources
  - Χειρότερο σε congestion

Είδη mutex

- `spin_mutex`
  - Non-reentrant, unfair, spin
  - Πολύ γρήγορο για συγχρονισμό που χρειάζεται σπάνια, low-congestion, πολύ μικρά critical sections
- `queuing_mutex`
  - Non-reentrant, fair, spin
  - Χρήσιμο όταν χρειάζεται κλιμακωσιμότητα και fairness
- `queuing_rw_mutex`
  - Non-reentrant, fair, spin
- `spin_rw_mutex`
  - Non-reentrant, unfair, spin
  - Χρήση για το `ReaderWriterMutex` pattern
- `mutex`
  - Wrapper για το OS sync
  - Στο linux είναι `pthread_mutex`
- `recursive_mutex`
  - Όπως το `mutex`, αλλά reentrant

Παράδειγμα Mutex

```
#include <tbb/spin_rw_mutex.h>
using namespace tbb;

spin_rw_mutex MyMutex;

int foo() {
  /* Construction of 'lock' acquires 'MyMutex' */
  spin_rw_mutex::scoped_lock lock(MyMutex, /*is_writer*/ false);

  read_shared_data (data);

  if (!lock.upgrade_to_writer()) {
    /* lock was released to upgrade;
       may be unsafe to access data, recheck status before use */
  } else {
    /* lock was not released; no other writer was given access */
  }

  return 0;
  /* Destructor of 'lock' releases 'MyMutex' */
}
```