

# Paralelni programski modeli

- ❖ pthreads
- ❖ MPI/OpenMP
- ❖ Cilk
- ❖ IPP, TBB – II deo

# Kontejneri

- ◆ Omogućavaju istovremen pristup za više niti
  - Zaštita STL kontejnera semaforima ograničava paralelizam, odnosno ubrzanje
  - TBB koristi dve metode
    - ◆ Fino zaključavanje (fine-grained locking) samo onih delova kontejnera koji se moraju zaključati
    - ◆ Tehnike bez zaključavanja – niti računavaju i popravljaju efekte drugih niti koje ih ometaju
  - Primitive kontejnera za paralelni pristup su duže nego kod klasičnih STL kontejnera
    - ◆ Treba ih koristiti samo ako se isplati dodatni paralelizam koji one omogućavaju



# Kontejneri

## concurrent\_hash\_map

- `concurrent_hash_map<Key, T, HashCompare >`
  - ◆ *HashCompare* je tip koji određuje kako se računa ključ i kako se dva elementa porede

**// Structure that defines hashing and comparison operations**

```
struct MyHashCompare {  
    static size_t hash( const string& x ) {  
        size_t h = 0;  
        for( const char* s = x.c_str(); *s; ++s )  
            h = (h*17)^*s;  
        return h;  
    }  
    // ! True if strings are equal  
    static bool equal( const string& x, const string& y ) {  
        return x==y;  
    }  
};
```



# Kontejneri

## concurrent\_vector

- `concurrent_vector< T>`
  - ♦ Metode: `push_back`, `grow_by`, `grow_to_at_least`
  - ♦ Nužno je sinhronizovati konstruisanje i pristup elementima
  - ♦ Operacije nad `concurrent_vector` su duže nego nad `vector`
- Čekanje elementa `v[i]`, koji dodaje druga nit
  - ♦ Čekaj dok je `i < v.size()`
  - ♦ Čekaj da `v[i]` bude konstruisan
    - „Zero allocator“ nov element postavlja na 0
    - Konstruktor postavlja zastavicu tipa **atomic** u elementu



# Kontejneri

## concurrent\_queue

- `concurrent_queue< T, Alloc >`, red vrednosti tipa T
  - ♦ Osnove operacije: `push` i `try_pop`
  - ♦ U programu sa jednom niti red je FIFO struktura
  - ♦ Za više niti, red odraža redosled po redosledu ubacivanja
  - ♦ Red je neograničen i neblokirajući
- `concurrent_bounded_queue< T, Alloc >`
  - ♦ Ograničen red: varijanta sa blokiranjem
  - ♦ Operacije: `push`, `pop`, `try_push`, `size`
  - ♦ Metoda `size` vraća broje nepreuzetih elemenata reda
  - ♦ Veličina se postavlja metodom `set_capacity`



# Kontejneri

## Nedostaci redova

- ◆ Pre upotrebe redova razmotriti `parallel_for_each` i protočnu obradu, koji su obično efikasniji
  - Red je usko grlo, zbog održavanja FIFO redosleda
  - Nit može čekati besposlena na redu
- ◆ Red je pasivna struktura; ubačen element se može "ohladiti" u baferu
  - ◆ Još gore, ako element preuzima nit u drugom jezgru, on i sve što referencira mora se prebaciti u bafer tog jezgra
  - ◆ Nasuprot tome, `parallel_for_each` optimizuje upotrebu niti radnika da stalno rade i da održavaju bafere vrućima

# Međusobno isključivanje niti

- ◆ Realizuje se pomoću *mutex* i *locks*
  - *mutex* je objekat, koji nit može zaključati, prethodno dobijenim ključem
  - Samo jedna nit može imati ključ, ostale moraju čekati
- ◆ Najjednostavniji mutex je *spin\_mutex*
  - Adekvatan ako se ključ drži samo nekoliko instrukcija
  - Primer: *FreeListMutex* štiti *FreeList*
  - Grupisanje iskaza u blok može izgledati neobično – uloga je da se skрати vreme života ključa



# Međusobno isključivanje niti

## Primer upotrebe spin\_mutex-a

```
Node* FreeList;  
typedef spin_mutex FreeListMutexType;  
FreeListMutexType FreeListMutex;  
  
Node* AllocateNode() {  
    Node* n;  
    {  
        // constructor locks the list  
        FreeListMutexType::scoped_lock lock(FreeListMutex);  
        n = FreeList;  
        if( n )  
            FreeList = n->next;  
    } // destructor unlocks the list  
    if( !n )  
        n = new Node();  
    return n;  
}
```





# Međusobno isključivanje niti

## Alternativan primer upotrebe spin\_mutex-a

- Metode acquire i release
- OO sprega – otključavanje u slučaju izuzeća
- Mutex tipa *M* ima ključ tipa *M::scoped\_lock*

```
Node* AllocateNode() {  
    Node* n;  
    FreeListMutexType::scoped_lock lock;  
    lock.acquire(FreeListMutex);  
    n = FreeList;  
    if( n )  
        FreeList = n->next;  
    lock.release();  
    if( !n )  
        n = new Node();  
    return n;  
}
```



# Međusobno isključivanje niti

## Atributi mutex-a

### ◆ Atributi mutex-a:

- Skalabilni/Neskalabilni (Scalable/Non-scalable)
- Pravični/Nepravični (Fair/Unfair)
- Rekurzivni/Nerekurzivni (Recursive/Non-recursive)
- Sa-upošljenim-čekanjem/Sa-blokiranjem (Yield/Block)
  - ◆ Yield: nit prvo upošljeno čeka („vrti se“, eng. spins), a ako se čekanje oduži, privremeno prepušta procesor drugim nitima

# Međusobno isključivanje niti

## Vrste mutex-a

- Vrste mutex-a
  - ◆ spin\_mutex
    - Non-scalable, Unfair, Non-recursive, Yield
  - ◆ queuing\_mutex
    - Scalable, Fair, Non-recursive, Yield
  - ◆ spin\_rw\_mutex i queuing\_rw\_mutex
    - Kao predhodni + podrška za ključeve čitača (readers)
  - ◆ speculative\_spin\_mutex i speculative\_spin\_rw\_mutex
    - Kao predhodni + spekulativno izvršenje na procesorima sa HTM (kritične sekcije se izvršavaju kao transakcije)
  - ◆ mutex i recursive\_mutex su omotači oko OS primitiva
    - CRITICAL\_SECTION na OS Windows
  - ◆ null\_mutex i null\_rw\_mutex – ne rade ništa

# Međusobno isključivanje niti

## Mutex-i čitača-pisača

- Mutex-i su potrebni ako bar jedna nit piše u objekat
- Više čitača mogu dobiti ulaz u zaštićeni deo koda
- Mutex-i sa `_rw_` u imenu razlikuju ključeve
  - ◆ Ključevi za čitanje i upis
- Ključ za čitanje se može podići na nivo za pisanje
  - ◆ Metoda `upgrade_to_writer` može privremeno osloboditi ključ
  - ◆ inače bi moglo doći do međusobnog blokiranja (deadlock)



# Međusobno isključivanje niti

## Patologije ključeva

- Međusobno blokiranje (deadlock); nastaje ako
  - ♦ Postoji krug niti
  - ♦ Svaka nit drži bar jedan ključ i čeka na mutex, koji je zaključala druga nit
  - ♦ Ni jedna nit ne želi da oslobodi svoje ključeve
- Pravljenje konvoja (convoying)
  - ♦ OS prekine nit koja drži ključ
  - ♦ Druge niti moraju da sačekaju da prekinuta oslobodi ključ
  - ♦ Kod fair mutex-a, još gore, čekaju nastavak prekinute
- Izbegavanje konvoja
  - ♦ Min vreme držanja ključa
  - ♦ Koristiti atomske operacije umesto ključeva



# Atomske operacije

## Definicija (1/2)

- Druge niti vide atomske operacije kao trenutne
  - ♦ Brže su od operacija sa ključevima
  - ♦ Prednost: nema međusobnog zaključavanja niti konvoja
  - ♦ Nedostatak: rade ograničen skup operacija
  - ♦ Klasa `atomic<T>` implementira atomske operacije
- Primer: referentno brojanje: akcija ako `x==0`
  - ♦ Sekvencijalno: `--x; if(x==0) action()`
  - ♦ U sl. više niti, `x` postaje deljena promenljiva
  - ♦ Smanjenje `x` i provera da li je 0 mora se uraditi neprekidivo
  - ♦ `x` deklarirati kao `atomic<int>` i napisati `if(--x==0) action()`



# Atomske operacije

## Definicija (2/2)

- `atomic< T>`, `T` int, enum ili pokazivač
- Pet osnovnih operacija nad `x` tipa `atomic< T>`
  - ♦ `= x`, pročitaj `x`
  - ♦ `x =`, upiši vredost u `x` i vrati tu vrednost
  - ♦ `x.fetch_and_store(y)`, uradi `x=y` i vrati staru vrednost od `x`
  - ♦ `x.fetch_and_add(y)`, uradi `x+=y` i vrati staru vrednost od `x`
  - ♦ `x.compare_and_swap(y,z)`, ako je `x==z`, uradi `x=y`. Uvek vrati staru vrednost od `x`

```
atomic<unsigned> counter;  
unsigned GetUniqueInteger() {  
    // return different int, until counter wraps around  
    return counter.fetch_and_add(1);  
}
```



# Dodela memorije

- `scalable_allocator<T>` rešava problem skalabilnosti
  - ♦ Konvencionalan allocator: ekskluzivan pristup mem. bazenu
  - ♦ `scalable_allocator<T>`: više niti istovremeno dobija objekte tipa T
- `cache_aligned_allocator<T>` rešava lažno deljenje linije bafera (cache)
  - ♦ Nastaje kad dve niti pristupaju različitim rečima u liniji bafera
  - ♦ Kad jedno jezgro promeni liniju, pa je promeni drugo jezgro, linija se iz prvog prebacuje u drugo – nekoliko stotina taktova
  - ♦ Rešenje: `cache_aligned_allocator<T>` - garantuje da objekti koje napravi neće deliti istu liniju bafera





# Raspoređivač zadataka

- ◆ Realizuje paralelne aptrakcije visokog niova
  - Nudi API, koji se može direktno koristiti
  - Moguće je relizovati svoje apstrakcije visokog nivoa
  - Treba koristiti logičke zadatke; prednosti:
    - ◆ Poklapanje paralelizma sa resursima, brže dizanje i spuštanje sistema, efikasniji redosled evaluacije, bolje uravnoteženje opterećenja, i razmišljanje na višem nivou
  - Programske niti se preslikavaju na fizička jezgra
    - ◆ Krajnjosti pretplate: nedovoljna i prevelika
    - ◆ Prevelika pretplata: dovodi do pada performanse zbog deljenja vremena (time slicing)
      - Raspoređivač pokušava da ima jednu nit na jednom jezgru



# Raspoređivač zadatka

## TBB zadaci

- Zadaci su puno “lakši” od niti (brže se pokreću)
  - ♦ Na OS Linux 18 puta, a na Windows više od 100 puta
  - ♦ Zadatak (task) u TBB je mala rutina (nema kontekst)
  - ♦ TTB zadaci jedan drugog ne istiskuju (preemption)
- TBB raspoređivač je efikasan zato što nije pravedan
  - ♦ On poseduje info o zadacima, tako da može da žrtvuje pravednost za efikasnost
  - ♦ Zadatak se pokreće tek kad može da učini koristan napredak
  - ♦ Raspoređivač uravnotežuje opterećenje (load balancing)



# Raspoređivač zadatka

## Mrešćenje (pokretanje) TBB zadataka

- TBB zadaci se mogu izmrestiti na dva načina:
  - ♦ tj. pomoću dve šablonklase: **task\_group** i **parallel\_invoke**
  - ♦ Oba načina omogućavaju upotrebu bilo lambda funkcije ili C++ funktora (čiju metodu operator treba implementirati)

```
struct Functor {  
    // Metoda koje se poziva kad se objekat ovog tipa prosledi metodi  
    // oneapi::tbb::task_group::run()  
    void operator()() const {}  
};
```

# Raspoređivač zadataka

## Prvi način mrešćenja zadataka

- Mrešćenje pojedinačnih zadataka:
  - ◆ Lambda funkcija ili objekat funktora se proslede kao argument metode *run* klase **task\_group**

```
#include <oneapi/tbb/task_group.h>
int main() {
    // Funktori ChildTask1 i ChildTask2 su negde definisani.
    oneapi::tbb::task_group tg;
    tg.run(ChildTask1{/* parametri konstruktora */});
    tg.run(ChildTask2{/* parametri konstruktora */});
    tg.wait();
}
```

# Raspoređivač zadataka

## Drugi način mrešćenja zadataka

- Mrešćenje više paralelnih zadataka:
  - ◆ Lambda funkcije ili objekti funktora se proslede kao argumenti konstruktora klase **parallel\_invoke**

```
#include <oneapi/tbb/parallel_invoke.h>
int main() {
    // Funktori ChildTask1 i ChildTask2 su negde definisani.
    oneapi::tbb::parallel_invoke(
        ChildTask1{/* parametri konstruktora */},
        ChildTask2{/* parametri konstruktora */}
    );
}
```

# Raspoređivač zadatka

„Školski“ primer: Fibonačijev niz

- n-ti broj iz Fibonačijevog niza  $F_n$ :
  - ♦  $F_n = F_{n-1} + F_{n-2}$ ,  $n = 0, 1, 2, \dots$ ;  $F_0 = 0$ ,  $F_1 = 1$
  - ♦ Funkcija SerialFib sekvencijalno računa  $F_n$

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

# Raspoređivač zadatka

## Paralelizovano računanje $F_n$

- Funkcija ParallelFib paralelno računa  $F_n$ :
  - ♦ Primena dinamičkog PP, tj. mustre rekurzivni zadatak
  - ♦ Zadatak koji računa  $F_n$  izmresti zadatke za  $F_{n-1}$  i  $F_{n-2}$

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

```
#define CutOff 10
```

```
long ParallelFib(long n) {
```

```
    long x, y;
```

```
    tbb::task_group g;
```

```
    if(n < CutOff)
```

```
        return SerialFib(n);
```

```
    else
```

```
        g.run([&]{x = ParallelFib(n-1);}); // izmresti zadatak 1
```

```
        g.run([&]{y = ParallelFib(n-2);}); // izmresti zadatak 2
```

```
        g.wait(); // čekaj zadatke 1 i 2
```

```
        return x + y;
```

```
}
```

# Raspoređivač zadataka

## Graf zadataka

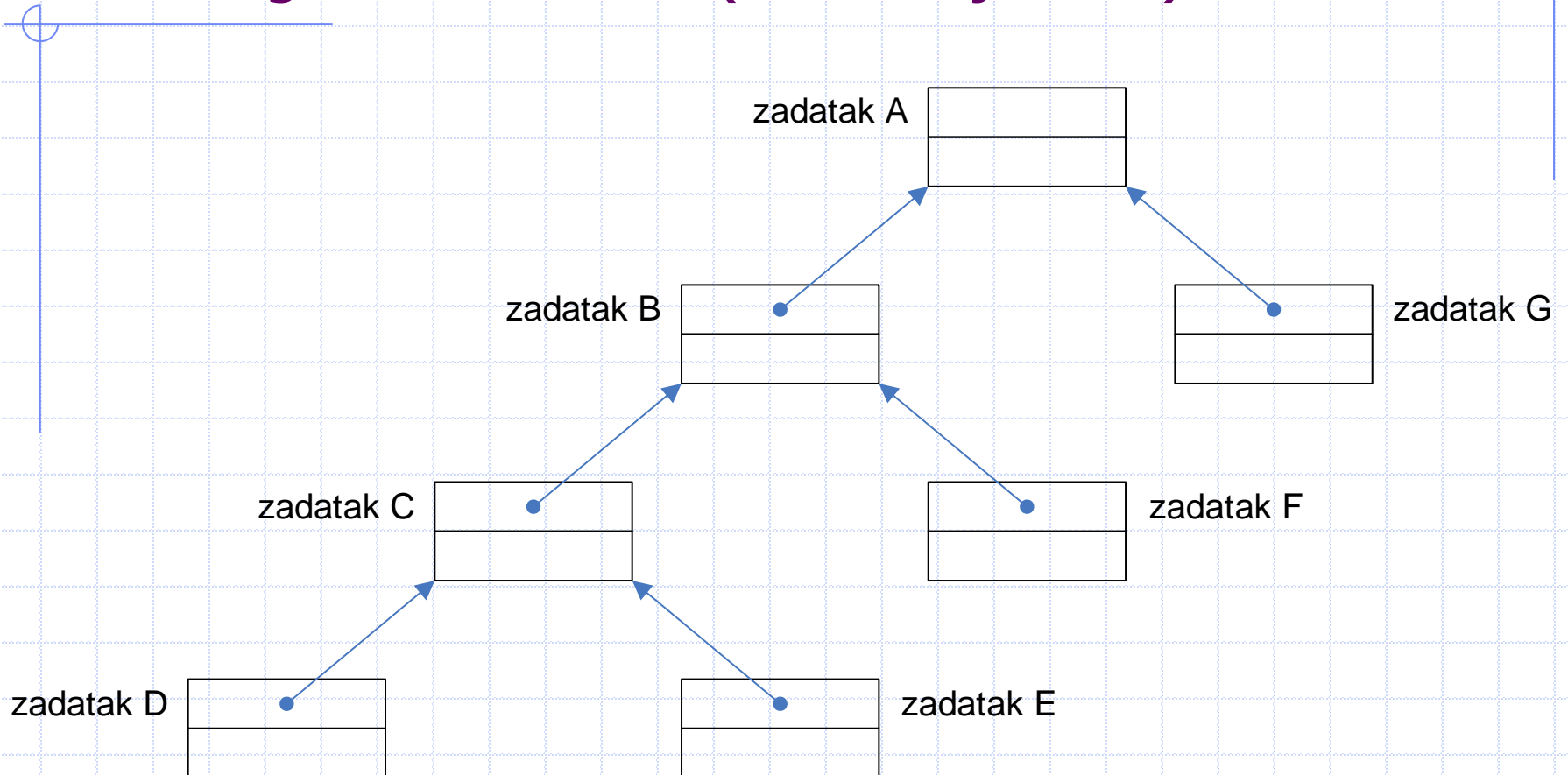
- Raspoređivač obrađuje graf zadataka
  - ♦ Usmeren graf, svaki čvor je zadatak
  - ♦ Svaki zadatak pokazuje na naslednika, koji ga čeka
  - ♦ Svaki zadatak ima atribut *refcount*: br. zadataka čiji je on naslednik
- Pimer grafa zadataka iz primera Fibonačijev niz
  - ♦ A, B i C su izmrestili svoje potomke i čekaju ih
  - ♦ D se izvršava, još nije izmrestio potomke
  - ♦ E, F i G čekaju na izvršenje





# Raspoređivač zadataka

## Primer grafa zadataka (Fibonačijev niz)



# Raspoređivač zadataka

## Algoritam raspoređivača (1/3)

- Minimizira memorijske zahteve i komunikaciju niti
- Balansira izvršenje u dubinu i u širinu
- Izvršenje u dubinu je najbolje za serijsko izvršenje
  - ♦ Radi dok je bafer vruć. D je najnoviji, zatim C, pa B i A.
  - ♦ Minimizira prostor. U širinu, eksp. br. zadataka. U dubinu, linearan, jer se ostali smeštaju na stek (E, F i G iz primera).
- Iako izvršenje u širinu ima problem sa potrošnjom memorije, ono maksimizira paralelizam
  - ♦ Koristiti dovoljno paralelizma, da jezgra budu upošljena



# Raspoređivač zadatka

## Algoritam raspoređivača (2/3)

- Hibrid izvršenja u dubinu i u širinu
  - ♦ Svako jezgro ima svoj red sa dva kraja (deque)
  - ♦ Npr. kad zadatak F izmresti E, gurne ga na dno reda

zadatak G
zadatak F
zadatak E

- Jezgro bira sledeći zadatak za izvršenje:
  - ♦ Onaj sa dna svog reda, ako nije prazan
  - ♦ Krade zadatak sa vrha reda drugog slučajno izabranog jezgra

# Raspoređivač zadatka

## Algoritam raspoređivača (3/3)

- Dva uslova da jezgro gurne zadatak na dno reda:
  - ♦ Eksplicitno izmrešćen zadatak
  - ♦ Nakon izvršenja zadnjeg prethodnika refcount naslednika postaje 0 i jezgro gura naslednika na dno svog reda
- Strategija: "Radi u dubinu, preuzimaj u širinu"
  - ♦ "breadth-first theft and depth-first work"
  - ♦ Izvršenje u širinu ide samo do potrebnog paralelizma
  - ♦ Izvršenje u dubinu održava operativnu efikasnost, jednom kada ima dovoljno posla

# Raspoređivač zadatka

## Korisne tehnike

### ◆ Tehnike:

- Rekurzivan lanac reakcija (Recursive Chain Reaction)
- Dodavanje posla (Adding More Work)
- Ponovna upotreba (Recycling)
- Opšti aciklični grafovi zadatka

# Raspoređivač zadataka

## Rekurzivan lanac reakcija

- Rekurzivan lanac reakcija
  - ◆ Najbolja performansa za grafove u obliku stabala zadataka
  - ◆ Direktno stvaranje  $N$  zadataka  $O(N)$ , kroz stablo  $O(\lg(N))$
  - ◆ Iteracioni prostor petlje se sa `parallel_for` može preslikati na binarno stablo zadataka

# Raspoređivač zadataka

## Dodavanje posla

- Dodavanje posla:

- ◆ Zadatak može izmrestiti (tj. pokrenuti) novi zadatak ako se pojavio novi posao koji treba obaviti

```
int main() {  
    std::vector<int> items = { 0, 1, 2, 3, 4, 5, 6, 7 };  
    oneapi::tbb::task_group tg;  
    for (std::size_t i = 0; i < items.size(); ++i) {  
        tg.run([&i = items[i], &tg] {  
            // ... uradi neki posao za i-tu stavku ...  
            if (pojavio_se_nov_posao)  
                // Funktor OtherWork je negde definisan.  
                tg.run(OtherWork{});  
        });  
    }  
    tg.wait();  
}
```

# Raspoređivač zadatka

## Ponovna upotreba zadatka

- Ponorva upotreba zadatka:
  - ◆ Zadatak može samog sebe ponovo pokrenuti tako što pozove metodu run klase task\_group i prosledi joj argument \*this

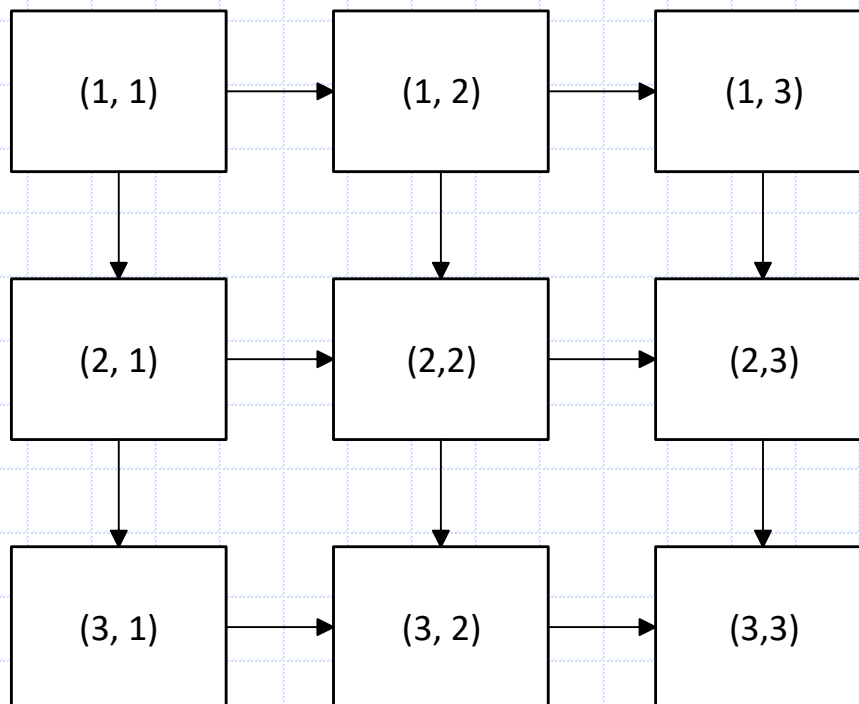
```
#include <memory>
#include <oneapi/tbb/task_group.h>
struct SharedStateFunctor {
    std::shared_ptr<Data> m_shared_data;
    oneapi::tbb::task_group& m_task_group;
    void operator>() const {
        // uradi neki posao obradom m_shared_data
        if (ima_dodatnog_posla)
            m_task_group.run(*this);
        // odavde instance imaju konkurentan pristup m_shared_data
    }
};
```



# Raspoređivač zadataka

## Opšti aciklični grafovi zadataka (1/4)

- Opšti aciklični grafovi zadataka:
  - ◆ Do sada su razmatrana isključivo stabla zadataka
  - ◆ Moguće su složenije strukture, npr. 2D rešetka zadataka



◆ Zadatak zavisi od zadatka od gore (sa severa) i s leva (sa zapada)

◆ Kod aplikacija koje koriste šablon pod nazivom fronttalaša (wavefront)

# Raspoređivač zadatka

## Opšti aciklični grafovi zadatka (2/4)

- Šablon fronttalisa se koristi u naučnim aplikacijama:
  - ◆ Podaci su distribuirani na višedimenzionalnim rešetkama
  - ◆ Primer 2D fronttalisa je obrada koju izvodi funkcija wave

```
const int n = 4;  
const int A_size = n * n;  
std::vector<double> A(A_size);
```

```
// Sekvencijalan (serijski) fronttalsa  
// Fronttalis ide dijagonalno od elem. (1,1) do (n-1,n-1)  
void wave(std::vector<double> &A, int n) {  
    for(int i=1; i<n; i++)  
        for(int j=1; j<n; j++)  
            A[i*n+j] = foo(A[i*n+j], A[(i-1)*n+j], A[i*n+(j-1)]);  
}
```

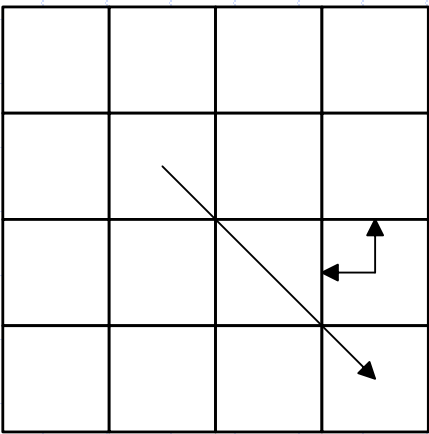
# Raspoređivač zadatka

## Opšti aciklični grafovi zadatka (3/4)

- Ideja paralelizacije 2D fronttalasa:

- ♦ Fronttalas prebrisava matricu  $M$  po njenoj glavnoj dijagonali, a na antidijagonali se pojavljuju nezavisni zadaci
- ♦ Matrica atomskih brojača sadrži broj zavisnosti zadatka od drugih zadatak (atom. brojača, zbog konkurentnog pristupa)

$i \backslash j$	0	1	2	3
0				
1				
2				
3				



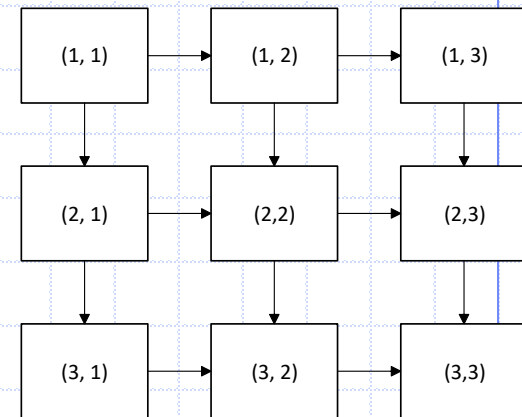
a) Dijagonalno prebrisavanje matrice podataka  $M$

$i \backslash j$	0	1	2	3
0				
1		0	1	1
2		1	2	2
3		1	2	2

b) Matrica atomskih brojača

# Raspoređivač zadatka

## Opšti aciklični grafovi zadatka (4/4)



- Paralelizovan 2D fronttala:

- ♦ Matrica atomskih brojača omogućava dinamičko pokretanje zadatka
- ♦ Zadatak (i, j) dekrementira brojače zavisnosti zadatka na jugu (i+1, j) i na istoku (i, j+1), ako oni postoje, i pokreće zadatak čiji brojač padne na 0

// Paralelni fronttala

```
void wave_p(std::vector<double> &A, int n, int i, int j) {  
    tbb::task_group g;  
    A[i*n+j] = foo(gs, A[i*n+j], A[(i-1)*n+j], A[i*n+(j-1)]);  
    if(j<n-1 && --counters[i*n+(j+1)]==0)    // istočni zadatak?  
        g.run([&]{wave_p(A, n, i, j+1);});    // izmresti istočni zadatak  
    if(i<n-1 && --counters[(i+1)*n+j]==0)    // južni zadatak?  
        g.run([&]{wave_p(A, n, i+1, j);});    // izmresti južni zadatak  
    g.wait();    // čekaj zadatke u grupi g  
}
```