Paralelni programski modeli

- pthreads
- MPI/OpenMP
- Cilk
- IPP, TBB – I deo

IPP

- IPP = Integrated Performance Primitives
  - Visoko optimizovane primitive za paralelno programiranje
  - Mogu se bezbedno pozivati iz TBB niti i Cilk linija
- Širok skup primitiva specifičnih za razne domene
  - Obrada signala
  - Obrada slike i videa
  - Male matrice i realistično crtanje
  - Kriptografija

TBB (Threading Building Blocks)

- Teme
  - Paralelizacija jednostavnih petlji
  - Paralelizacija složenih petlje (petlje i protočne obrade)
  - Grafovi toka
  - Kontejneri
  - Međusobno isključivanje niti
  - Atomske operacije
  - Dodela memorije
  - Raspoređivač zadataka

Jenostavne petlje: primer (1/2)

- Neka je bezbedno paralelno primeniti funkciju Foo na sve elemente niza A
- Da bi paralelizovali sekvencijalno rešenje
  - Formiramo objekat tela
    - To je klasa čija metoda operator() radi nad delom iteracionog prostora
    - blocked_range<T> opisuje jednodimenzioni prostor
    - blocked_range2d opisuje dvodimenzioni prostor

```
void SerialApplyFoo( float a[], size_t n ) {
   for( size_t i=0; i != n; ++i )
      Foo(a[i]);
}
```

Jenostavne petlje: primer (2/2)

```
#include "oneapi/tbb.h"
using namespace oneapi::tbb;
class ApplyFoo {
    float *const my_a;
public:
    void operator()( const blocked_range<size_t>& r ) const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
            Foo(a[i]);
    }
    ApplyFoo( float a[] ) : my_a(a) {}
};
```

```
#include "oneapi/tbb.h"
void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a));
}
```

Jednostavne petlje
Kontrolisana podela iteracionog prostora

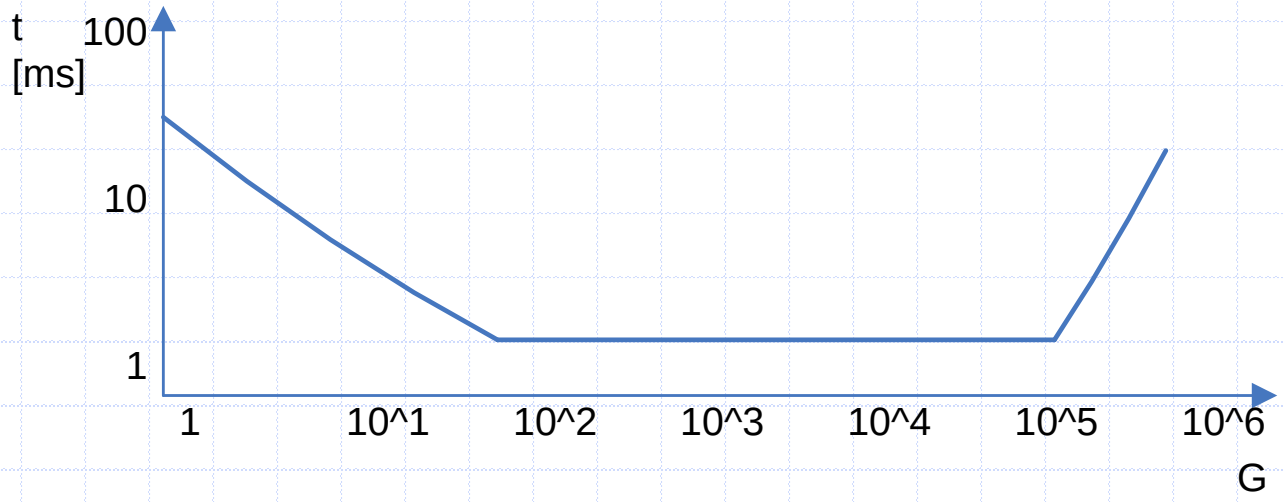
- Kontrola podele pomoću *partitioner* i *grainsize*
  - Zadati simple_partitioner() unutar parallel_for
  - blocked_range< *T*>(*begin,end,grainsize*)
    - Podrazumevani *grainsize* (G) je 1
    - simple_partitioner garantuje $\lceil G/2 \rceil \leq chunksize \leq G$
    - Postoje: {auto, simple, affinity, static}_partitioner

```
#pragma warning(disable: 588)

void ParallelApplyFoo( float a[], size_t n ) {
   parallel_for(blocked_range<size_t>(0,n,G), ApplyFoo(a),
                simple_partitioner());
}
```

Jednostavne petlje
Podešavanje kvanta podele (grainsize)

- Pravilo: grainsize iteracija traje 100.000 ciklusa
  - Eksperiment: od 100.000 polovljenjem do optimuma
  - Suviše mali grainsize može redukovati paralelizam
  - Bolje malo veći, nego malo manji grainsize
- Primer: a[i]=b[i]*c preko milion indeksa



Jednostavne petlje
Širina propusnog opsega i bliskost pri baferovanju

- Malo ubrzanje zbog nedovoljnog propusnog opsega između procesora i memorije
  - Prestruktuirati radi boljeg iskorišćenja bafera (cache)
  - Ili, iskoristiti affinity_partitioner, kada
    - Obradu čine par operacija po pristupu podatku
    - Podaci nad kojima petlja radi staju u bafer (cache)
    - Petlja ili slična petlja se izvodi nad istim podacima

Jednostavne petlje
Primer korišćenja affinity_partitioner (ap)

- ap živi preko iteracija petlje
  - To se postiže deklaracijom da je static
  - Ap dodeljuje iteraciju niti koja ju je ranije izvršavala

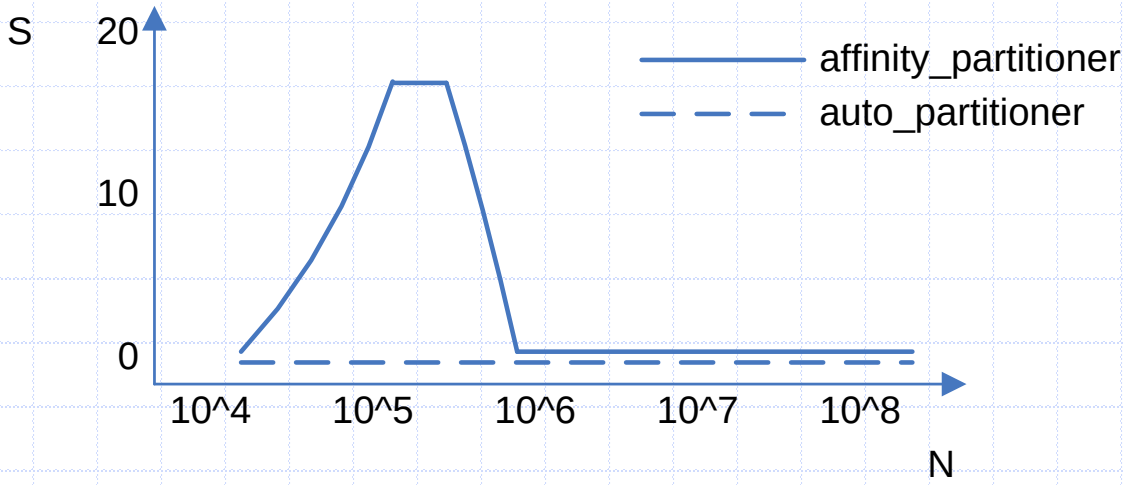
```
#include "tbb/tbb.h"

void ParallelApplyFoo( float a[], size_t n ) {
    static affinity_partitioner ap;
    parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a), ap);
}

void TimeStepFoo( float a[], size_t n, int steps ) {
    for( int t=0; t<steps; ++t )
        ParallelApplyFoo( a, n );
}
```

Jednostavne petlje
Korist od bliskosti pri baferovanju

- Zavisi od relativnog odnosa veličine skupa podataka i bafera
  - Primer: A[i]+=B[i] u opsegu [0,N)
    - Za malo N dominira paralelno raspoređivanje, što rezultuje u malom ubrzanju
    - Za preveliko N skup podatak je prevelik da bi mogao biti prenešen u baferu preko iteracija petlje



Šablonklase za podelu iteracionog prostora

- Sraćenice: G = *grainsize*, C = *chunksize*

Šablonklasa	Opis	Stvarni blok podele C
auto_partitioner	Automatski bira C.	G/2 ≤ C
simple_partitioner	Ograničava C sa G.	G/2 ≤ C ≤ G
affinity_partitioner	Automatski bira C, iskorišćuje skrivene memorije, i uniformno distribuira iteracije.	G/2 ≤ C
static_partitioner	Deterministički bira C, iskorišćuje skrivene memorije, i uniformno distribuira iteracije bez uravnoteženja opterećenja	max(G/3, P/R) ≤ C gde je P veličina problema, a R je broj resursa (proc. jezgara)

Jednostavne petlje
parallel_reduce (1/2)

```
float SerialSumFoo( float a[], size_t n ) {
    float sum = 0;
    for( size_t i=0; i!=n; ++i )
        sum += Foo(a[i]);
    return sum;
}
```

- Ako su iteracije nezavisne, gornja petlja se može paralelizovati sa parallel_reduce

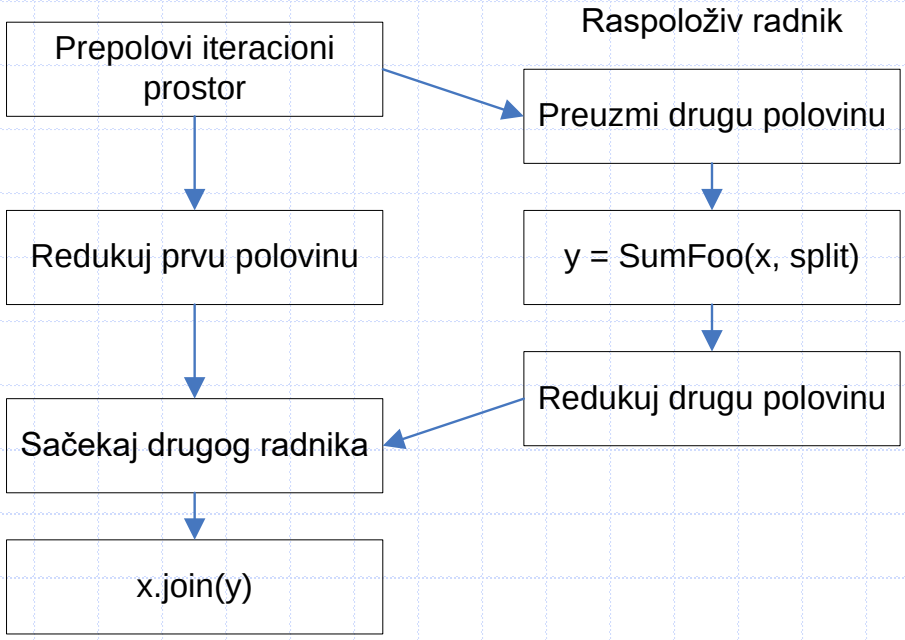
```
float ParallelSumFoo( const float a[], size_t n ) {
    SumFoo sf(a);
    parallel_reduce( blocked_range<size_t>(0,n), sf );
    return sf.my_sum;
}
```

Jednostavne petlje
parallel_reduce (2/2)

```
class SumFoo {
    float* my_a;
public:
    float my_sum;
    void operator()( const blocked_range<size_t>& r ) {
        float *a = my_a;
        float sum = my_sum;
        size_t end = r.end();
        for( size_t i=r.begin(); i!=end; ++i )
            sum += Foo(a[i]);
        my_sum = sum;
    }
    SumFoo( SumFoo& x, split ) : my_a(x.my_a), my_sum(0) {}
    void join( const SumFoo& y ) {my_sum+=y.my_sum;}
    SumFoo(float a[] ) :  my_a(a), my_sum(0)  {}
};
```

Jednostavne petlje
Izvršenje parallel_reduce-a

- Kada je nit radnik (worker) raspoloživa
  - Poziva se konstruktor razdvajanja
    - Pravi se podzadatak za nit radnika
    - Nakon završetka podzadatka poziva se join radi akumuliranja njegovog rezultata



Složene petlje
parallel_for_each (1/2)

- parallel_for_each se koristi ako iteracioni prostor nije poznat unapred
  - Primer: povezana lista
    - U principu je bolje koristiti dinamičke nizove; liste su serijske
    - Ima smisla ako obrada jednog elementa liste uključuje nekoliko hiljada instrukcija
    - Recimo da hoćemo da paralelizujmo ovaj sekvencijalan kod

```
void SerialApplyFooToList( const std::list<Item>& list ) {
   for( std::list<Item>::const_iterator i=list.begin() i!=list.end();
          ++i )
      Foo(*i);
}
```

Složene petlje
parallel_for_each (2/2)

```
class ApplyFoo {
public:
    void operator()( Item& item ) const {
        Foo(item);
    }
};

void ParallelApplyFooToList( const std::list<Item>& list ) {
    parallel_for_each( list.begin(), list.end(), ApplyFoo() );
}
```

- Nikada dve niti ne konkurišu za isti iterator
  - Dva načina da se skalabilno dođe do više posla
    - Korišćenjem iteratora sa slučajnim pristupom
    - Dodavanjem posla sa *feeder*.add(*item*)

Složene petlje
protočna obrada (1/5)

- TBB realizuje mustru protočne obrade
  - Apstrakcije: pipeline i filter
  - Primer: kvadriranje brojeva u datoteci
    - Stepeni protočne obrade: čitaj iz dat., kvadriraj, piši u dat.
    - Blok podataka (4000 znakova) predstavljen klasom TextSlice

```
class TextSlice {
    char* logical_end, physical_end;
    static TextSlice* allocate( size_t max_size ) {
        // +1 leaves room for a terminating null character.
        TextSlice* t = (TextSlice*)tbb::tbb_allocator<char>().allocate(
                                                sizeof(TextSlice)+max_size+1 );
        ....
        return t;
    } ...
```

Složene petlje
protočna obrada (2/5)

- Kod za pravljenje i izvršenje protočne obrade
  - TextSlice objekti se prosleđuju putem pokazivača

```
void RunPipeline( int ntoken, FILE* input_file, FILE* output_file ) {
    tbb::parallel_pipeline(
        ntoken,
        tbb::make_filter<void,TextSlice*>(
            tbb::filter::serial_in_order, MyInputFunc(input_file) )
    &
        tbb::make_filter<TextSlice*,TextSlice*>(
            tbb::filter::parallel, MyTransformFunc() )
    &
        tbb::make_filter<TextSlice*,void>(
            tbb::filter::serial_in_order, MyOutputFunc(output_file) ) );
}
```

Složene petlje
protočna obrada (3/5)

- Parametar *ntoken* kotroliše nivo paralelizma
  - *ntoken* specificira max br. žetona koji su u protočnoj obradi
  - 1 žeton = 1 TextSlice objekat, tzv. stavka (item) obrade
  - Potencijalan problem: gde čuvati objekte u srednjem stepenu
- Drugi parametar specificira niz filtara
  - make_filter<*inputType,outputType*>(*mode,functor*)
  - Filtri se spajaju pomoću operatora &
  - serial_in_order filter: serijska obrada
  - parallel filter: paralelna obrada
  - serial_out_of_order filter: ne održava originalni redosled

Složene petlje
protočna obrada (4/5)

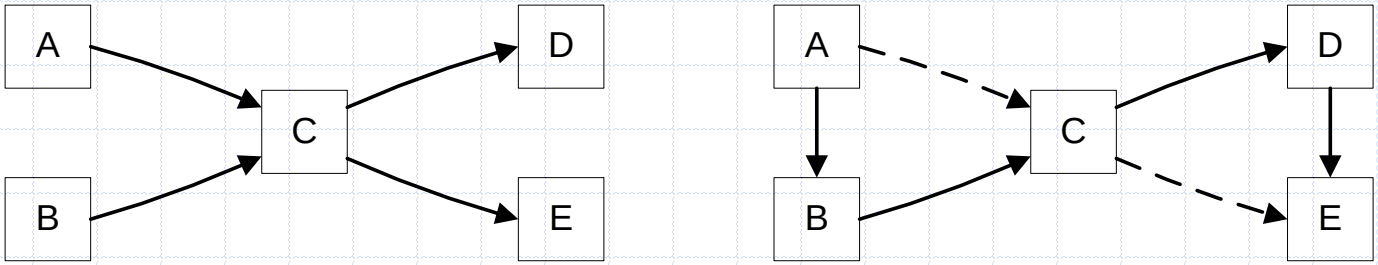
- Kružni baferi (KB):
  - KB se koriste da bi se minimizirala režija dodele i oslobađanja stavki koje se prenose između filtara
  - Dodela i oslobađanje stavki ide preko KB veličine *ntoken* stavki
  - Ako su prvi i zadnji filtar serial_in_order, kad se zauzme zadnja pozicija, prva je sigurno slobodna
  - Ako oba filtra nisu serial_in_order, mora se voditi evidencija o pozicijama koje su trenutno u upotrebi

Složene petlje
protočna obrada (5/5)

- Propusnost protočne obrade ograničavaju:
  - Broj žetona N koji se istovremeno obrađuju
    - Malo N, mali paralelizam; preveliki N, previše resursa
  - Najsporiji filtar je usko grlo protočne obrade
    - U TextSlice primeru U-I radnje ograničavaju ubrzanje
  - Veličina prozora = veličina podataka za 1 žeton
    - Prevelik prozor – podaci ne mogu da stanu u bafer (cache)

Složene petlje
nelinearne protočne obrade

- Mogu se transformisati u linearne
  - Propusnost ostaje ista, jer je usko grlo ostalo isto
  - Kašnjenje se povećava
    - Npr.: kašnjenje 3 filtra se povećava na kašnjenje 5 filtara
    - Dobar kompromis između jednostavnosti i performanse

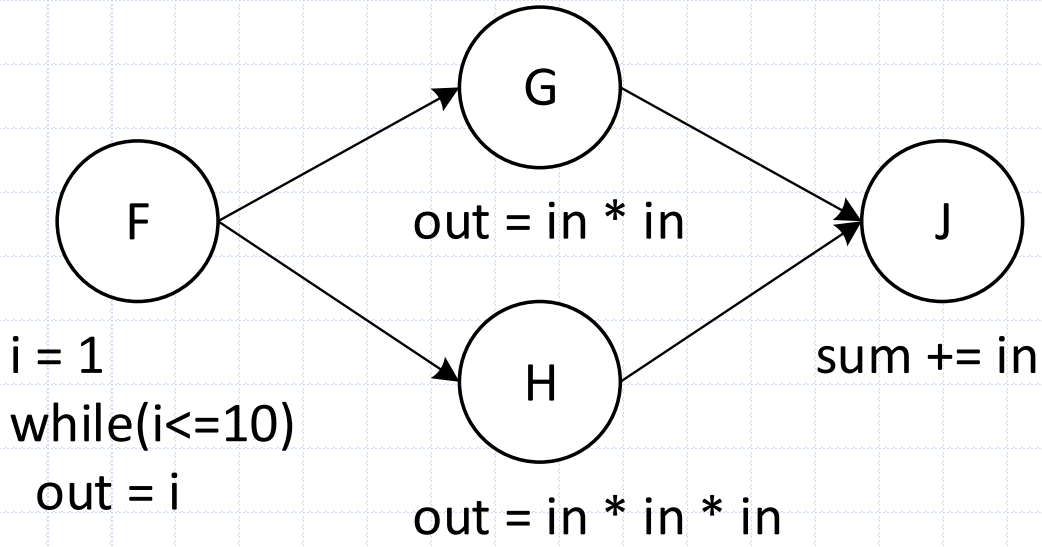


Grafovi toka
uvod

- Postoje dve vrste grafova toka:
  - grafovi toka podataka
  - grafovi zavisnosti
- Elementi grafova toka:
  - Čvorovi predstavljaju računanja
  - Grane predstavljaju komunikacione kanale
- Po prijemu poruke, čvor izmresti zadatak
  - koji izvrši objekat tela nad dolaznom porukom

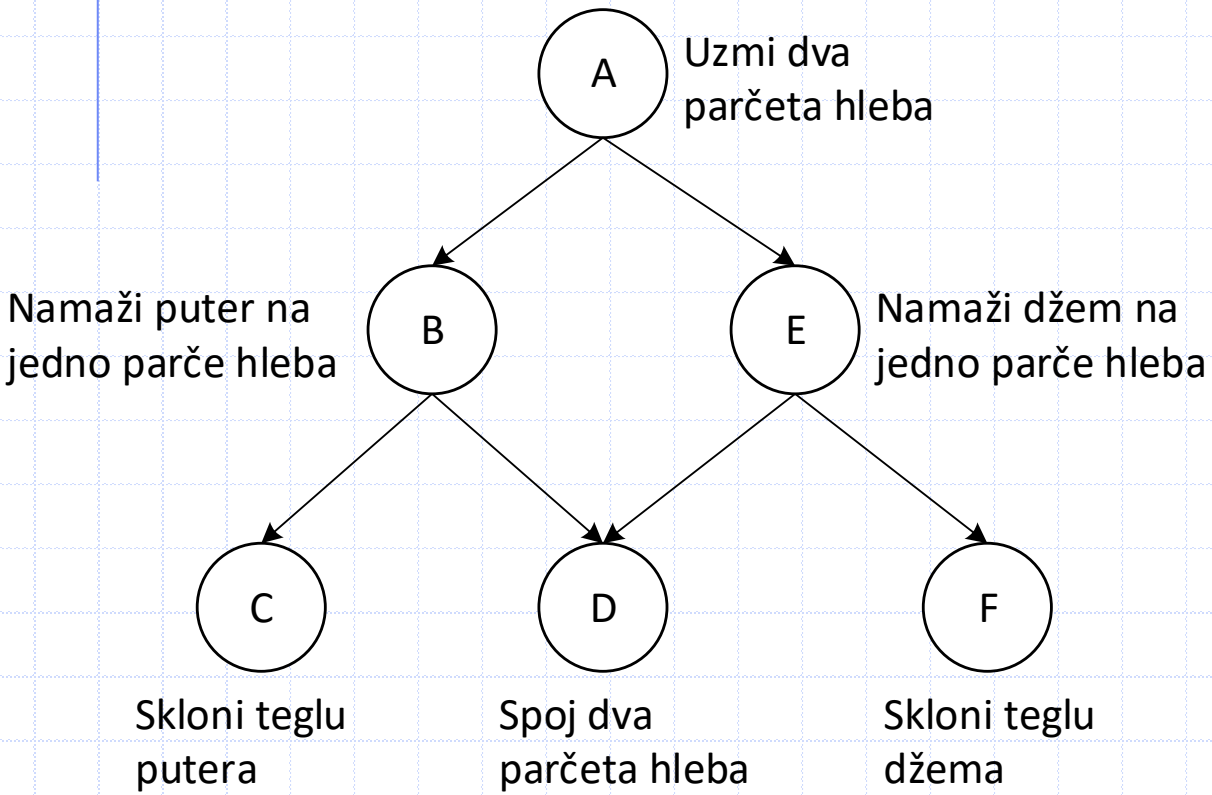
Grafovi toka
grafovi toka podataka

- Grafovi toka podataka:
  - su aplikacije za obradu toka podataka (striming)
  - Podaci se obrađuju tokom prolaska kroz čvorove



Grafovi toka
grafovi zavisnosti

- Grafovi zavisnosti:
  - su aplikacije u kojima postoji delimično uređenje između računanja koja obavljaju čvorovi



- Naznačen je samo zahtevani redosled koraka, npr. A mora pre B itd.
- Delimično uređenje je zato što postoje međusobno nezavisni koraci, npr. B i E, čiji redosled je proizvoljan

Grafovi toka
objekat grafa

- Graf toka je kolekcija čvorova i grana
  - Svaki čvor pripada tačno jednom grafu
  - Grane se prave samo između čvorova u istom grafu
- Objekat grafa predstavlja tu kolekciju:
  - Služi za pozivanje svih operacija grafa, kao što su:
  - Čekanje svih zadatak, reset i otkazivanje svih čvorova
- Primer:

```
graph g;
g.wait_for_all();
```

Grafovi toka
čvorovi grafa

- Čvor je klasa koja nasleđuje graph_node
  - i obično nasleđuje sender< *T*> i/ili receiver< *T*>
- Čvor obavlja neku operaciju,
  - obično na dolaznim porukama (jednoj ili više),
  - i može generisati ni jednu ili više izlaznih poruka.
  - Ugrađeni tipovi čvorova su definisani u flow_graph.h
- Npr. function_node predstavlja funkciju sa jednim ulazom i jednim izlazom

Grafovi toka
konstruktor tipa function_node

template< typename *Body*> function_node(graph &*g*, size_t *concurrency*, *Body body*)

- *Body* je tip objekta tela
- *g* je graf kome čvor pripada
- *concurrency* je granica konkurencije, tj. dozvoljeni br. izmrešćenih zadataka (od 1 do beskonačno)
- *body* je objekat tela

```
graph g;
function_node< int, int > n( g, 1, []( int v ) -> int {
   cout << v; spin_for( v ); return v;
} );
n.try_put( 1 ); n.try_put( 2 ); n.try_put( 3 );
g.wait_for_all();
```

Grafovi toka

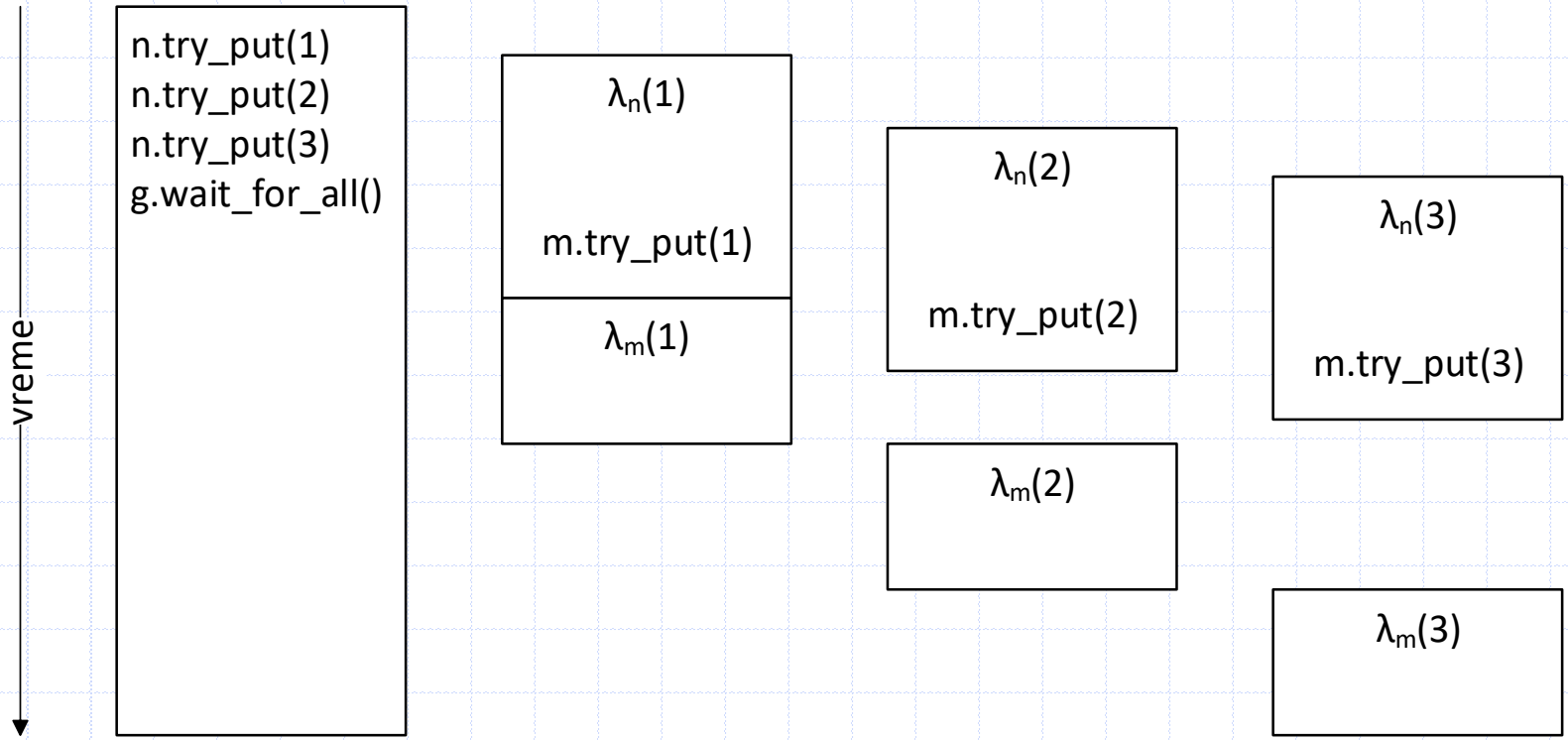
grane grafa i primer sa 2 function_node-a

- Grane su usmereni kanali za prenos poruka
  - Prave se pozivanjem funkcije *make_edge(p, s)*
  - *p* je čvor predhodnik a *s* je čvor sledbenik

```
graph g;
function_node< int, int > n( g, unlimited, []( int v ) -> int {
   cout << v; spin_for( v ); return v;
} );
function_node< int, int > m( g, 1, []( int v ) -> int {
   v *= v; cout << v; spin_for( v ); return v;
} );
make_edge( n, m );
n.try_put( 1 ); n.try_put( 2 ); n.try_put( 3 );
g.wait_for_all();
```

Grafovi toka

jedan moguć raspored izvršenja čvorova *n* i *m*



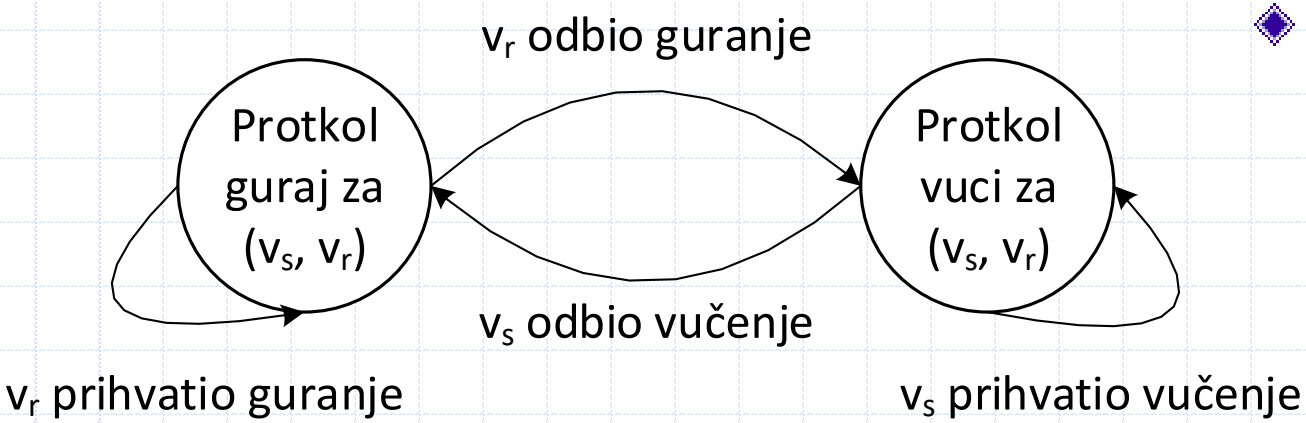
- Tela čvorova *n* i *m* su označena sa λ_n i λ_m
- Tela λ_n se izvršavaju konkurentno (jer je *concurrency* za $n = \infty$)
- Tela λ_m se izvršavaju serijski (jer je *concurrency* za $m = 1$)

Grafovi toka
Protokol za prenos poruka (1/2)

- Problem:
  - Ponekad neki čvor neće moći da primi i obradi poruku
- Rešenje koje omogućava efikasnost grafa:
  - Grana između ovih čvorova može promeniti svoje stanje u stanje vučenja (pull).
  - Kad sledbenik bude mogao da obradi poruku, on pita predhodnika da li je poruka raspoloživa.
    - Ako predhodnik ima poruku, čvor sledbenik će je obraditi i grana će ostati u stanju vučenja
    - Ako predhodnik nema poruku, grana će se prebaciti iz stanja vučenja u stanje guranja (push)

Grafovi toka
Protokol za prenos poruka (2/2)

- Protokol za prenos poruka preko grana:
  - se kratko zove PROTOKOL GURAJ-VUCI (Push-Pull)
  - Koristi dva podprotokola: prot. guraj i prot. vuci
  - Sledi dijagram stanja protokola guraj-vuci



- v_s i v_r su čvor predhodnika (predajnika) i čvor sledbenika (prijemnika)

Grafovi toka
pojedinačno-guranje i guranje-svima

- Postoje dve politike guranja poruka:
  - pojedinačno-guranje: bez obzira koliko ima sledbenika koji mogu da prihvate poruku, svaka poruka se šalje samo jednom sledbeniku (slučajno izabranom)
  - guranje-svima: poruka se gura svakom sledbeniku koji je povezan sa predhodnikom granom u stanju guranja i koji prihvata poruku
- Primene:
  - pojedinačno-guranje: primenjuju samo čvorovi koji baferuju poruke, npr. buffer_node
  - guranje-svima: primenjuju svi drugi tipovi čvorova, npr. broadcast_node

Grafovi toka
zajednički deo za primere dve politike guranja por.

- Slede primeri za dve politike guranja poruka, ispod je zajednički deo za oba primera

```
using namespace oneapi::tbb::flow;
std::atomic<size_t> g_cnt;
struct fn_body1 {
   std::atomic<size_t> &body_cnt;
   fn_body1(std::atomic<size_t> &b_cnt) : body_cnt(b_cnt) {}
   continue_msg operator()( continue_msg /*dont_care*/) {
      ++g_cnt;
      ++body_cnt;
      return continue_msg();
   }
};
void run_example1() {
   graph g;
   std::atomic<size_t> b1;
   std::atomic<size_t> b2;
   function_node<continue_msg> f1(g,serial,fn_body1(b1));
   function_node<continue_msg> f2(g,serial,fn_body1(b2));
```

Grafovi toka
primer primene politike pojedinačno-guranje

```
buffer_node<continue_msg> buf1(g);
g_cnt = b1 = b2 = 0;
make_edge(buf1,f1);
make_edge(buf1,f2);
buf1.try_put(continue_msg());
buf1.try_put(continue_msg());
g.wait_for_all();
printf("g_cnt == %d, b1==%d, b2==%d\n",
        (int)g_cnt, (int)b1, (int)b2);
```

- Tip buffer_node primenjuje pojedinačno-guranje
- Rezultat: g_cnt == 2, b1==2, b2==0

Grafovi toka
primer primene politike guranje-svima

```
broadcast_node<continue_msg> bn(g);
g_cnt = b1 = b2 = 0;
make_edge(bn,f1);
make_edge(bn,f2);
bn.try_put(continue_msg());
bn.try_put(continue_msg());
g.wait_for_all();
printf("g_cnt == %d, b1==%d, b2==%d \n",
        (int)g_cnt, (int)b1, (int)b2);
```

- Tip broadcast_node primenjuje guranje-svima
- Rezultat: g_cnt == 4, b1==2, b2==2

Grafovi toka
baferovanje i prosleđivanje poruka

- Problem:
  - Ponekad čvor ne može uspešno da gurne poruku ni jednom sledbeniku
- Tada postoje dve mogućnosti:
  - čvor će sačuvati poruku (da bi je kasnije prosledio)
  - čvor će odbaciti poruku
    - Može se izbeći dodavanjem baferišućeg čvora
- Dva načina da se sačuvana poruka preda dalje:
  - Sledbenik je može povući sa try_get ili try_reserve
  - Sledbenik može biti dinamički povezan sa make_edge

Grafovi toka
rezervacija poruka (1/3)

- Čvor tipa join_node ima četiri moguće politike:
  - ulančavanje, rezervisanje, poklapanje-ključa, i poklapanje-oznake
- Potrebna je poruka na svakom ulazu da bi se proizvela poruka na izlazu
- Rezervišući join_node nema baferovanje:
  - Da bi napravio izlaznu poruku, on privremeno rezerviše poruku na svakom ulaznom priključku
  - Ako uspe, povlači sve rez. poruke i pravi izl. poruku
  - Ako ne uspe, ne povlači ni jednu poruku

Grafovi toka
rezervacija poruka (2/3)

- Neki tipovi čvorova podržavaju rezervaciju
- Procedura rezervisanja:
  - Rezervišući join_node uvek odbija guranje od čvora predhodnika, i grana se prebacuje u stanje vučenja.
  - Rezervišući ul. priključci pozivaju try_reserve na svakoj grani u stanju vučenja. Ako try_reserve ne uspe, grana se prebacuje u stanje guranja.
    - Sve dok je predhodnik ulaznog priključka u stanju rezervisan, ni jedan drugi čvor ne može da dobije rezervisanu poruku.
  - ... (nastavak je na sledećem slajdu)

Grafovi toka
rezervacija poruka (3/3)

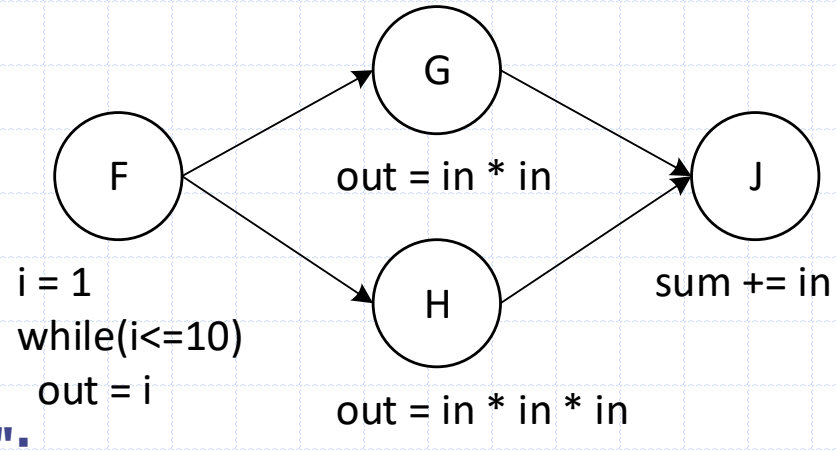
- Ako rezervisanje uspe, join_node pravi izl. poruku i pokušava da je gurne bilo kom sledbeniku.
- Ako je izl. poruka uspešno gurnuta, predhodnicima se signalizira sa try_consume, da su poruke iskorišćene.
  - Te poruke će biti odbačene u čvorovima predhodnicima.
- Ako izl. poruka nije bila uspešno gurnuta, napravljene rezervacije se otkazuju sa try_release.
- Napomena: Da bi mogla da se proizvede izl. poruka, bar jedan predhodnik na svakom ul. priključku mora da može biti rezervisan.

- DOMAĆI: Proučiti primer na slici 2.66

Grafovi toka

primeri implementacije grafa toka podataka

```
int sum = 0;
graph g;
function_node< int, int > squarer( g, unlimited, [](const int &v) {
    return v*v; } );
function_node< int, int > cuber( g, unlimited, [](const int &v) {
    return v*v*v; } );
function_node< int, int > summer( g, 1, [&](const int &v ) -> int {
    return sum += v; } );
make_edge( squarer, summer );
make_edge( cuber, summer );
for ( int i = 1; i <= 10; ++i ) {
  squarer.try_put(i);
  cuber.try_put(i);
}
g.wait_for_all();
cout << "Sum is " << sum << "\n";
```

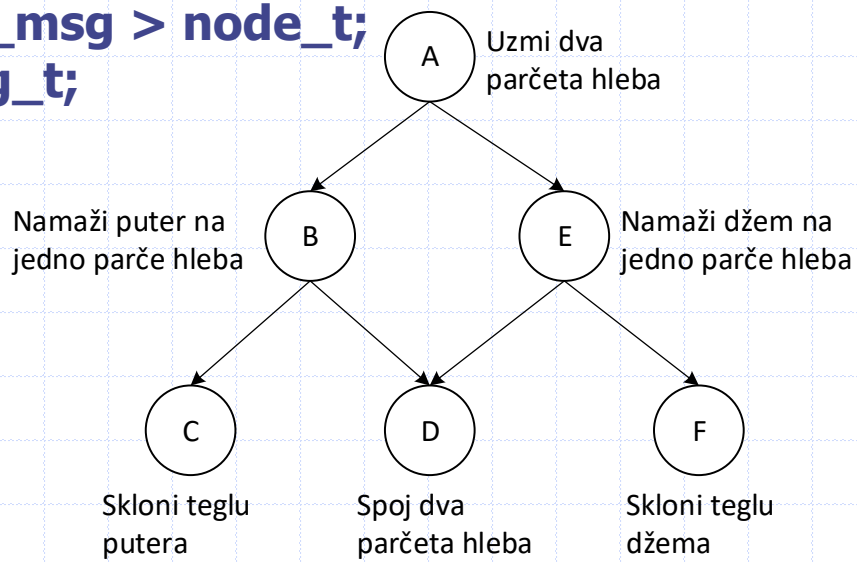


- Ovaj je Slika 2.74 u knjizi, Slike 2.75 i 2.76 su za domaći

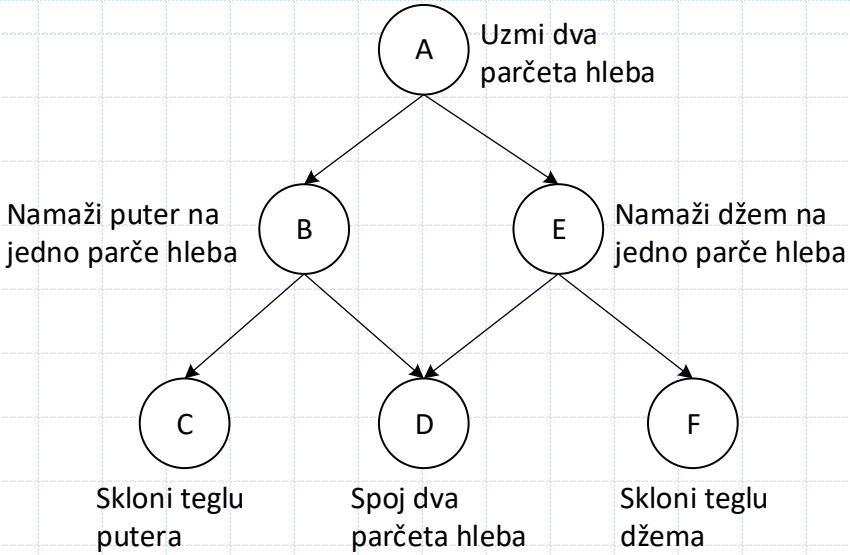
Grafovi toka

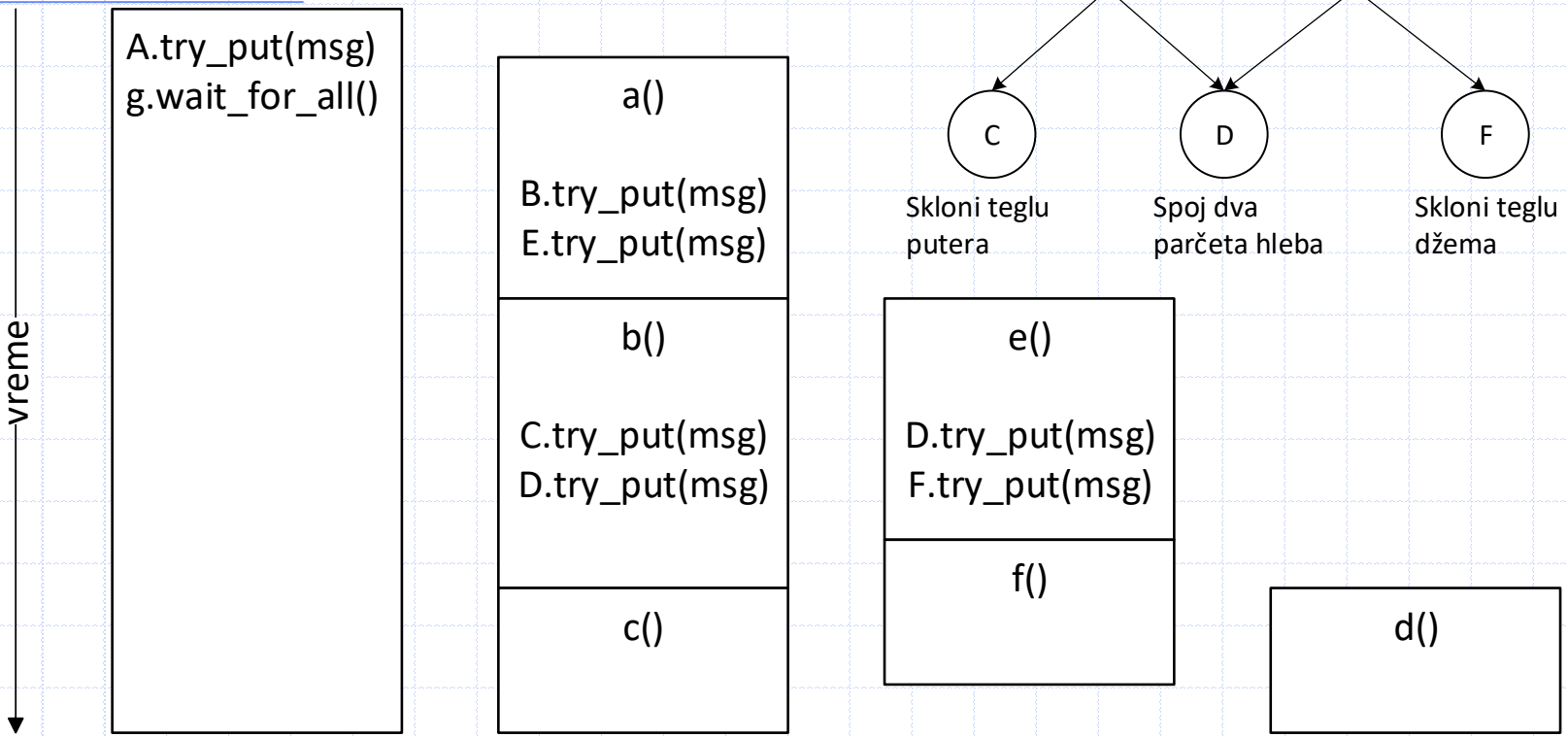
primer implementacije grafa zavisnosti

```
typedef continue_node< continue_msg > node_t;
typedef const continue_msg & msg_t;
int main() {
  oneapi::tbb::flow::graph g;
  node_t A(g, [](msg_t){ a(); } );
  node_t B(g, [](msg_t){ b(); } );
  node_t C(g, [](msg_t){ c(); } );
  node_t D(g, [](msg_t){ d(); } );
  node_t E(g, [](msg_t){ e(); } );
  node_t F(g, [](msg_t){ f(); } );
  make_edge(A, B); make_edge(B, C); make_edge(B, D);
  make_edge(A, E); make_edge(E, D); make_edge(E, F);
  A.try_put( continue_msg() );
  g.wait_for_all();
  return 0;
}
```



Grafovi toka
raspored izvršenja





- Izvršenje je usklađeno sa delimičnim uređenjem koje graf definiše
- Npr. izvršenje D ne započinje sve dok se i B i E ne završe
- Ovo je raspored zadataka u slučaju kad ima dovoljno niti

Grafovi toka
ugrađeni tipovi čvorova

- Postoji 17 ugrađenih tipova:
  - input_node, function_node, continue_node, multifunction_node, broadcast_node, buffer_node, queue_node, priority_queue_node, sequencer_node, join_node, split_node, write_once_node, overwrite_node, limiter_node, indexer_node, composite_node, async_node
- DOMAĆI: Proučiti tabelu tipova u knjizi