Paralelno Programiranje

POGLAVLJE 18, UDŽBENIK

Cilj paralelnog programiranja

- Cilj paralelnog programiranja je da **skrati vreme obrade** podataka korišćenjem **paralelizma** koga nude računari sa više procesora opšte namene.
- Taj cilj se ostvaruje pronalaženjem **relativno nezavisnih delova ukupne obrade podataka** i zaduživanjem **posebnih niti** da ih obavljaju.
- Ostvarenje pomenutog cilja podrazumeva da su aktivnosti ovih niti **istovremene**, odnosno da su vezane za **različite procesore**.

Cilj paralelnog programiranja

- Način dekompozicije obrade podataka u relativno nezavisne delove zavisi od **karaktera obrade podataka**, ali i od oblika **raspoloživog paralelizma**.
- Zato se paralelni programi **specijalizuju** za pojedine oblike paralelizma.
- Paralelno programiranje se zasniva na **konkurentnom** programiranju, jer relativna nezavisnost delova obrade podataka znači da je, pre ili kasnije, **neizbežna saradnja** za njih zaduženih niti.

Cilj paralelnog programiranja

- Pored toga, zahvaljujući **konkurentnom programiranju**, paralelni programi se mogu razvijati i na **jednoprocesorskom** računaru, na kome, jasno, **nije moguće** demonstrirati skraćenje vremena obrade podataka koje paralelni programi nude.

Paralelno programiranje zasnovano na razmeni poruka

- Saradnju niti paralelnog programa je najbolje zasnovati na **razmeni poruka**, jer takav oblik saradnje osigurava najveću **nezavisnost** niti.
- Međutim, razmena poruka je **ograničena oblikom paralelizma** koji određuje između **kojih procesora** (i njima pridruženih niti) postoji mogućnost za **direktnu razmenu poruka**.

Paralelno programiranje zasnovano na razmeni poruka

- To znači da se paralelni program može izvršavati na **jednoprocesorskom računaru**, ako se obezbede **komunikacioni kanali**, saglasni sa oblikom paralelizma za koga je paralelni program specijalizovan.
- Zadatak ovih komunikacionih kanala je da podrže **direktnu razmenu poruka** samo između onih niti, između kojih je ta razmena moguća u pomenutom obliku paralelizma.
- Na taj način se niti vezuju, na primer, u **niz** (**array**), **matricu** (**mesh**) ili **potpuno** međusobno povezuju.

Zamisao paralelnog sumiranja niza brojeva

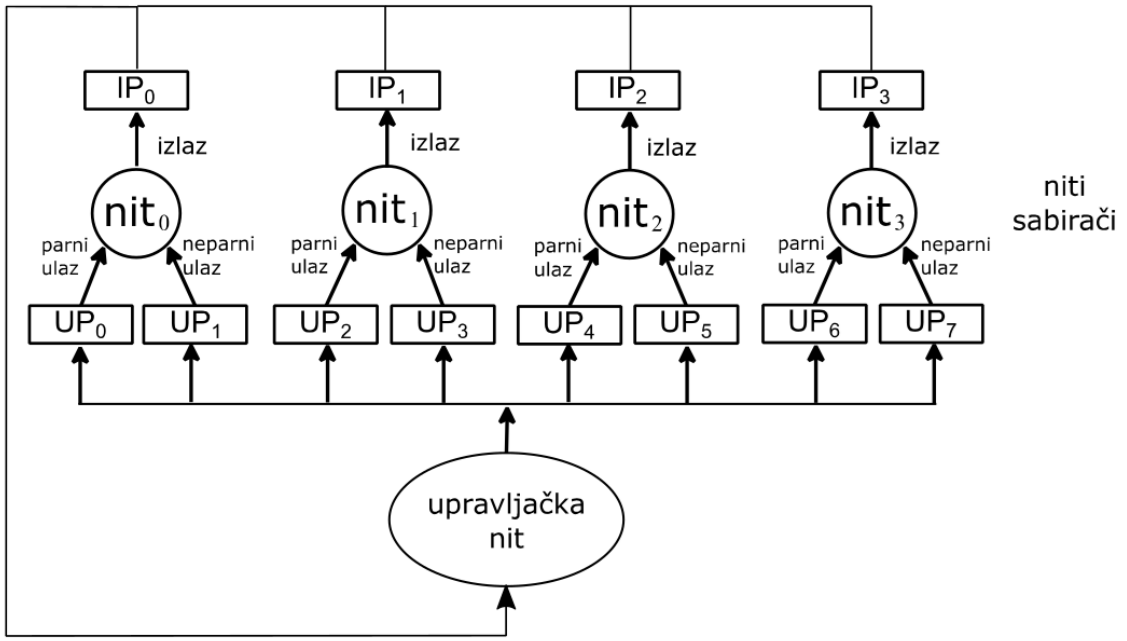
- Paralelno određivanje zbira niza brojeva započinje **istovremenim sabiranjem raznih parova** brojeva iz niza, a nastavlja se istovremenim sabiranjem **raznih parova prethodno paralelno određenih suma**, dok se ne dobije zbir svih brojeva iz niza.
- Pravilnost prethodnog postupka nalaže da dužina niza bude jednaka nekom **stepenu broja 2**.
- Na početku sumiranja angažovani broj **niti sabirača** je jednak **polovini dužine niza**.

Zamisao paralelnog sumiranja niza brojeva

- Na svakom **sledećem koraku** se angažuje **upola manje** niti sabirača, dok u **poslednjem** koraku ne preostane aktivna samo **jedna nit** sabirač.
- U svakom koraku aktivne niti sabirači sabiraju po **dva broja** koja su im dodeljena i njihov zbir prosleđuju dalje.
- Posebna **upravljačka nit** pre svakog koraka **upućuje** brojeve nitima sabiračima i nakon svakog koraka **preuzima** sume od angažovanih niti sabirača.

Komunikaciona osnova paralelnog sumiranja niza brojeva

- Svaka nit sabirač koristi **par** ulaznih pregradaka (**UPi** i **UPi+1**) za **preuzimanje** sumiranih brojeva i jedan izlazni pregradak (**IPi**) za **prosleđivaje** njihove sume:



Komunikaciona osnova paralelnog sumiranja niza brojeva

- Paralelno sumiranje se obavlja u **koracima**.
- U svakom koraku sumiranja upravljačka nit upućuje brojeve iz niza u ulazne pregratke niti sabirača i preuzima parcijalne sume iz izlaznih pregradaka niti sabirača.

Komunikaciona osnova paralelnog sumiranja niza brojeva

- Templejt klasa **Adder_boxes** omogućuje međusobno komunikaciono **povezivanje niti sabirača sa upravljačkom niti**.
- Dva parametra templejt klase **Adder_boxes** omogućuju definisanje **tipa poruke** i **broja niti** sabirača.
- Elementi polja **in_slots** odgovaraju ulaznim pregracima (**UPi**), a elementi polja **out_slots** odgovaraju izlaznim pregracima (**IPi**).

Komunikaciona osnova paralelnog sumiranja niza brojeva

- Podrazumeva se da se kao oznake niti sabirača koriste redni brojevi: **0, 1, 2, 3** i tako dalje.
- Operacije **send()** i **receive()** omogućuju **razmenu poruka**.
- **Prvi par** ovih operacija omogućuje **upravljačkoj niti** da razmenjuje poruke sa **nitima sabiračima** posredstvom njihovih **ulaznih** i **izlaznih** pregradaka.
- **Drugi par** ovih operacija omogućuje **nitima sabiračima** da razmenjuju poruke sa **upravljačkom niti** posredstvom svojih **ulaznih** i **izlaznih** pregradaka.
- Konstante **EVEN_IN** i **ODD_IN** označavaju parni i neparni ulazni pregradak.

Klasa Adder_boxes

```
#include "box.hh"

enum In_adder_boxes { EVEN_IN = 0, ODD_IN = 1 };

template <class MESSAGE, int THREADS>

class Adder_boxes {
        Message_box<MESSAGE> in_slots[THREADS * 2];
        Message_box<MESSAGE> out_slots[THREADS];
public:
        void send_in(int box, const MESSAGE message)
                { in_slots[box].send(&message); };
        MESSAGE receive_out(int box)
                { return out_slots[box].receive(); };
        void send_out(int sender, const MESSAGE message)
                { out_slots[sender].send(&message); };
        MESSAGE receive_in(int receiver, In_adder_boxes
source)
                { return in_slots[(receiver) * 2 +
source].receive(); };
};
```

Izvedba paralelnog sumiranja niza brojeva

- Klasa **Thread_identity** omogućuje jednoznačno numerisanje niti (sabirača):

```
class Thread_identity {
        mutex mx;
        int identity;
public:
        Thread_identity(int value = 0) : identity(value){};
        int get();
};

int Thread_identity::get()
{
        unique_lock<mutex> lock(mx);
        return identity++;
}
```

Izvedba paralelnog sumiranja niza brojeva

- Funkcija **thread_adder** opisuje ponašanje niti sabirača.
- Niti sabirači, nakon dobijanja svoje oznake, preuzimaju iz svojih **ulaznih pregradaka brojeve** za sabiranje, sabiraju ih i sumu prosleđuju u svoj **izlazni pregradak**.
- Niti sabirače stvara (korišćenjem bezimenih objekata klase thread) **upravljačka** nit, čiju aktivnost opisuje funkcija **thread_manager**.
- **Upravljačka nit** još **preuzima brojeve** za sabiranje, **upućuje ih** nitima **sabiračima** i zatim ponavlja **preuzimanje parcijalnih** suma **od** niti **sabirača** i upućivanje **ka njima** tih **parcijalnih suma**, dok ne preuzme i ne prikaže **traženi zbir**.

Izvedba paralelnog sumiranja niza brojeva

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "adder_boxes.hh"
#include "thread_identity.hh"

const int THREADS = 4;

Adder_boxes<int, THREADS> adder_boxes;
Thread_identity thread_identity;
```

Izvedba paralelnog sumiranja niza brojeva

```
void thread_adder()
{
        int tid = thread_identity.get();
        for(;;)
                adder_boxes.send_out(tid, adder_boxes.receive_in
(tid, EVEN_IN) + adder_boxes.receive_in(tid, ODD_IN));
}

void thread_manager()
{
        int number;
        int counter;
        int limit = THREADS;
        cout << endl << "PARALLEL SUMMATION"
                << endl << "input " << short(THREADS * 2)
                << " integers for summation" <<
          "(integer values from 1 to 100)";
        for(counter = 0; counter < limit; counter++) {
                thread (thread_adder).detach();
}
```

Izvedba paralelnog sumiranja niza brojeva

```
        for(counter = 0; counter < limit * 2; counter++) {
                do {
                        cout << endl << ((short)counter) << ". integer:
";
                        cin >> number;
                } while((number < 1) || (number > 100));
                adder_boxes.send_in(counter, number);
        }

        while(limit > 1) {
                for(counter = 0; counter < limit; counter++)
                        adder_boxes.send_in(counter,

        adder_boxes.receive_out(counter));
                limit = limit / 2;
        }
        cout << endl << "total sum: " << adder_boxes.receive_out(0) <<
                        endl;
}

int main()
{
        thread manager(thread_manager);
        manager.join();
```

Izvedba paralelnog sumiranja niza brojeva

- Sadržaj izvorne datoteke **p06.cpp** predstavlja potpun konkurentni program. U toku njegovog izvršavanja nastane **pet** niti.
- Dužina sumiranja, opisanog prethodnim programom, je proporcionalna **logaritmu** (**baze dva**) **broja sumiranih brojeva**, ako se svakoj niti sabiraču dodeli **poseban procesor**.
- Ako se prethodni program izvršava na **jednoprocesorskom računaru**, tada je dužina sumiranja **proporcionalna broju sumiranih brojeva**.

Zamisao paralelnog sortiranja

- Zadatak sortiranja niza znakova je njihovo uređenje u **zadanom redosledu**.
- Sortiranje znakova započinje tako da se na **prvo mesto** niza dovede znak koji **prethodi svim preostalim** znakovima.
- Radi toga se **poredi** znak, koji je zatečen na **prvom** mestu niza, sa znakovima, koji su zatečeni na **preostalim** mestima nesortiranog niza.
- Cilj ovih poređenja je otkrivanje parova znakova čiji **redosled nije u skladu** sa željenim uređenjem i eventualna **zamena mesta** ovih znakova.

Zamisao paralelnog sortiranja

- Analognim postupkom se dovode na drugo i sva ostala mesta znakovi koji prethode svim preostalim znakovima, tako da su na kraju svi znakovi **poređani u zadanom redosledu**.
- Postupci dovođenja željenih znakova na pojedina mesta niza su **međusobno nezavisni** kada se ne odnose na **iste znakove**, pa se mogu odvijati **paralelno**.

Zamisao paralelnog sortiranja

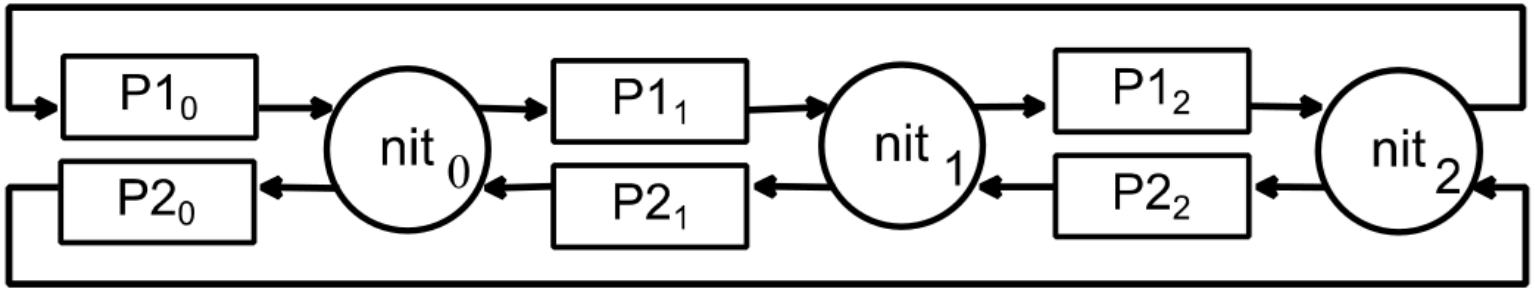
- Oni, znači, mogu biti predmet aktivnosti raznih niti, pod uslovom da ovakvih **sortnih niti ima koliko i sortiranih znakova**, kao i da su one uvezane u niz, odnosno da obrazuju protočnu strukturu.
- Tada znakovi, jedan za drugim, mogu da teku **od prve ka poslednjoj sortnoj niti**, tako da prva od sortnih niti može da izdvoji znak koji je na prvom mestu u uređenju, sortna nit iza nje znak koji je na drugom mestu u uređenju i tako dalje.

Zamisao paralelnog sortiranja

- Pri tome se podrazumeva da svaka sortna nit zadrži **prvi znak koji primi**.
- Ona zatim, **poredi** svaki novopridošli sa prethodno zadržanim znakom.
- Nakon poređenja, sortna nit **zadržava** uvek znak koji je **u skladu sa željenim uređenjem**, a šalje dalje preostali znak.
- Posebna **upravljačka nit** upućuje znakove na sortiranje, šaljući ih prvoj sortnoj niti i preuzima ih sa sortiranja, primajući ih prvo od prve, pa od druge i na kraju od poslednje sortne niti.

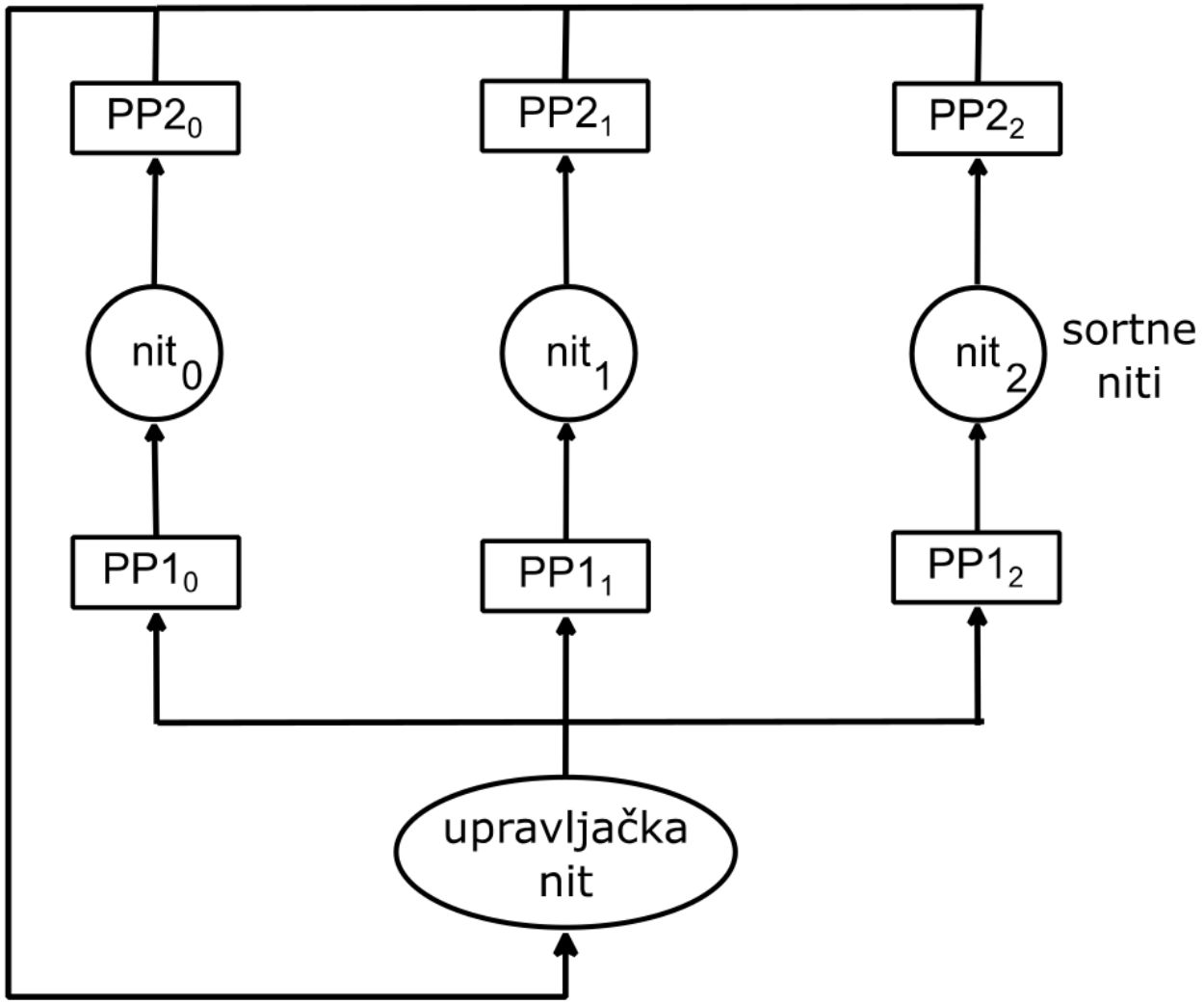
Komunikaciona osnova paralelnog sortiranja

- Vezivanje sortnih niti u niz je inspirisano idejom **jednodimenzionalne sistoličke arhitekture** (**array processors**).
- Svaka sortna nit iz ovog niza ima **levo** i **desno** od sebe po **dva pregratka** (**P1i**) i (**P2i**) za poruke:



Komunikaciona osnova paralelnog sortiranja

- Posmatrajući relativno, sa stanovišta svake sortne niti, pregraci **levo** uspostavljaju komunikacioni kanal **od prethodnika** i **ka prethodniku** iz niza.
- Slično pregraci **desno** uspostavljaju komunikacioni kanal **ka sledbeniku** i **od sledbenika**.
- Pošto je niz **cirkularan**, prva sortna nit je sledbenik poslednje sortne niti (odnosno poslednja sortna nit je prethodnik prve sortne niti).
- Podrazumeva se da uz svaku sortnu nit postoji i par posebnih pregradaka (**PP1i**) i (**PP2i**) za direktnu komunikaciju **sortnih niti sa upravljačkom niti**.



Komunikaciona osnova paralelnog sortiranja

Komunikaciona osnova paralelnog sortiranja

- Templejt klasa **Array** omogućuje uspostavljanje **komunikacionih kanala** koji su potrebni za uvezivanje sortnih niti u niz.
- Njena dva parametra omogućuju definisanje **tipa poruke** i **broja sortnih niti** u nizu.
- Elementi polja **slots1** i **slots2** odgovaraju, respektivno, pregracima (**P1i**) i (**P2i**), a elementi polja **special1** i **special2** odgovaraju posebnim pregracima (**PP1i**) i (**PP2i**), za direktnu komunikaciju sortnih niti sa upravljačkom niti.

Komunikaciona osnova paralelnog sortiranja

- Podrazumeva se da se kao oznake sortnih niti koriste brojevi: **0, 1, 2, 3** i tako dalje.
- Operacije **send()** i **receive()** omogućuju razmenu poruka.
- Prvi par ovih operacija omogućuje **upravljačkoj niti** da razmenjuje poruke sa **sortnim nitima** (koje su identifikovane svojim oznakama).

Komunikaciona osnova paralelnog sortiranja

- Drugi par ovih operacija omogućuje sortnim nitima da razmenjuju poruke sa svojim **prethodnikom** (koga označava konstanta **LEFT**), sa svojim **sledbenikom** (koga označava konstanta **RIGHT**) i sa **upravljačkom niti** (koju označava konstanta **SPECIAL**).
- Polje **threads** pokazuje koliko je angažovano **sortnih niti**.
- Njegovo postavljanje omogućuje operacija **threads_set()**.

Komunikaciona osnova paralelnog sortiranja

```
#include "box.hh"

enum Array_relative_position { LEFT, RIGHT, SPECIAL };

template<class MESSAGE, int THREADS>
class Array {
                Message_box<MESSAGE> slots1[THREADS];
                Message_box<MESSAGE> slots2[THREADS];
                Message_box<MESSAGE> special1[THREADS];
                Message_box<MESSAGE> special2[THREADS];
                int threads;
public:
                Array() : threads(THREADS) {};
                void send(int receiver, const MESSAGE* message)
                                                { special1[receiver].send(message); };
                MESSAGE receive(int sender) { return special2[sender].receive(); };
                void threads_set(int n) { threads = n; };
                void send(int sender, Array_relative_position relative_position,
                                                 const MESSAGE* message);
                MESSAGE receive(int receiver,
                                                Array_relative_position relative_position);
};
```

Komunikaciona osnova paralelnog sortiranja

```
template<class MESSAGE, int THREADS>
void Array<MESSAGE, THREADS>::send(int sender,
                                   Array_relative_position relative_position,
                                   const MESSAGE* message)
{
        Message_box<MESSAGE>* destination;
        switch(relative_position) {
                case LEFT:
                        destination = &(slots2[sender]);
                        break;
                case RIGHT:
                        if(sender < threads)
                                destination = &(slots1[sender+1]);
                        else
                                destination = &(slots1[0]);
                        break;
                case SPECIAL:
                        destination = &(special2[sender]);
                        break;
        }
        destination->send(message);
}
```

Komunikaciona osnova paralelnog sortiranja

```
template<class MESSAGE, int THREADS>
MESSAGE
Array<MESSAGE, THREADS>::receive(int receiver,
                                                    Array_relative_position relative_position)
{
        Message_box<MESSAGE>* source;
        switch(relative_position) {
                case LEFT:
                        source = &(slots1[receiver]);
                        break;
                case RIGHT:
                        if(receiver < threads)
                                source = &(slots2[receiver+1]);
                        else
                                source = &(slots2[0]);
                        break;
                case SPECIAL:
                        source = &(special1[receiver]);
                        break;
        }
        return source->receive();
}
```

Izvedba paralelnog sortiranja

- Ponašanje sortnih niti opisuje funkcija **thread_sorter()**.
- Sortne niti stvara (korišćenjem bezimenih objekata klase **thread**) **upravljačka nit**, čiju aktivnost opisuje funkcija **thread_manager()**.
- Upravljačka nit još preuzima **sa tastature** znakove za sortiranje, upućuje ih u **protočnu strukturu**, iz nje preuzima sortirane znakove i prikazuje ih na **ekranu**.
- Podrazumeva se da sortirani niz znakova na svom kraju sadrži **razmak** (**space**).

Izvedba paralelnog sortiranja

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "array.hh"
#include "thread_identity.hh"

const int THREADS = 10;
const char SPACE = ' ';

Array<char, THREADS> array;
Thread_identity thread_identity;
```

Izvedba paralelnog sortiranja

```
char receive(int receiver)
{
        char received;
        if(receiver == 0)
                received = array.receive(receiver, SPECIAL);
        else
                received = array.receive(receiver, LEFT);
        return received;
}
```

Izvedba paralelnog sortiranja

```
void thread_sorter()
{
        char old_character;
        char new_character;
        int tid = thread_identity.get();
        if((old_character = receive(tid)) > SPACE) {
                while((new_character = receive(tid)) > SPACE) {
                        if(new_character < old_character) {
                                array.send(tid, RIGHT,
&old_character);
                                old_character = new_character;
                        } else
                                array.send(tid, RIGHT,
&new_character);
                }
                array.send(tid, RIGHT, &new_character);
        }
        array.send(tid, SPECIAL, &old_character);
}
```

Izvedba paralelnog sortiranja

```
void thread_manager()
{
        int counter;
        char c;
        cout << endl << "PARALLEL SORTING"
                << endl << "unsorted array of characters "
                << "(enter " << THREADS-1 << " characters):" << endl;
        for(counter = 1; counter < THREADS; counter++) {
                thread (thread_sorter).detach();
                cin >> c;
                array.send(0, &c);
        }
        thread (thread_sorter).detach();
        array.send(0, &SPACE);
        cout << endl << "sorted array of characters (in
                                                        ascending
order)" << endl;
        for(counter = 1; counter <= THREADS; counter++)
                cout << array.receive(counter-1);
        cout << endl;
}
```

Izvedba paralelnog sortiranja

```
int main()
{
        thread manager(thread_manager);
        manager.join();
}
```

Izvedba paralelnog sortiranja

- Sadržaj izvorne datoteke **p07.cpp** predstavlja potpun konkurentni program.
- U toku njegovog izvršavanja nastane **jedanaest niti**.
- Dužina sortiranja zavisi od **broja poređenja**.
- Kod **sekvencijalnog** sortiranja po prethodnom algoritmu broj poređenja je: **(n-1)+(n-2)+...+1=n(n-1)/2** (za **prvi** element sortiranog niza napravi se **n-1** poređenja, za **drugi n-2** poređenja, a za **pretposlednji** element **1** poređenje).
- Znači dužina sortiranja je proporcionalna **kvadratu broja** sortiranih znakova.

Izvedba paralelnog sortiranja

- Paralelno sortiranje, opisano prethodnim programom, omogućuje **preklapanje poređenja** različitih parova znakova, ako se svakoj od sortnih niti dodeli **poseban procesor**.
- U tom slučaju **prva** sortna nit obavi **n-1** poređenja, **druga** sortna nit obavi **n-2** poređenja, ali sa kašnjenjem od **2** poređenja iza prve sortne niti.
- Do kašnjenja dolazi jer tek **nakon dva** poređenja druga sortna nit dobija od prve dva znaka i tek tada sama može započeti poređenja.

Izvedba paralelnog sortiranja

- To znači da se samo **poslednje poređenje** druge sortne niti **ne poklapa** sa poređenjima prve sortne niti.
- Isti odnos postoji između **druge** i **treće** sortne niti i tako redom.
- Prema tome, na dužinu sortiranja utiče **n-1** poređenje prve sortne niti i po **jedno** (poslednje) poređenje preostalih **n-2** sortnih niti.
- Sledi da je ukupan broj poređenja koja utiču na dužinu sortiranja: **(n-1)+(n-2)=2n-3**, pa je dužina sortiranja **linearno zavisna** od broja sortiranih znakova.

Zamisao paralelnog množenja matrica

- Izračunavanja elemenata matrice, koja je jednaka proizvodu druge dve matrice, su dovoljno međusobno nezavisna da mogu biti **paralelna**.
- To pokazuje primer množenja **dvodimenzionalnih matrica A i B,** čiji proizvod je jednak **dvodimenzionalnoj matrici C**

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{a_{11}b_{11}}+a_{12}b_{21} & a_{11}b_{12}+\mathbf{a_{12}b_{22}} \\ a_{21}b_{11}+\mathbf{a_{22}b_{21}} & \mathbf{a_{21}b_{12}}+a_{22}b_{22} \end{bmatrix}$$

Zamisao paralelnog množenja matrica

- Izračunavanju svakog od četiri elementa matrice C može se posveti posebna **nit množač.**
- Tada, svaka od njih, **nezavisno** jedna od druge (znači **paralelno**), može da izračuna proizvod elemenata matrica A i B, koje su označene **zadebljanim** slovima, ako poseduje potrebne elemente.
- To se može postići, ako se niti množači prostorno rasporede kao elementi matrice C i tako komunikaciono povežu da svaka od njih može da šalje poruke nitima množačima **desno** i **ispod** sebe, a prima poruke od niti množača **levo** i **iznad** sebe.

Zamisao paralelnog množenja matrica

- Podrazumeva se da na početku svaka nit množač primi od posebne **upravljačke niti** svoj element matrice A i svoj element matrice B.
- Pošto tada jedino nit množač **(1,1)** poseduje potreban par elemenata, ostale niti množači moraju da razmene svoje elemente.
- Tako niti množači **(2,1)** i **(2,2)** šalju **desno** raspoloživi element matrice A, a primaju s **leva** potrebni element iste matrice.
- Slično, niti množači **(1,2)** i **(2,2)** šalju **dole** raspoloživi element matrice B, a primaju **odozgo** potrebni element iste matrice.

Zamisao paralelnog množenja matrica

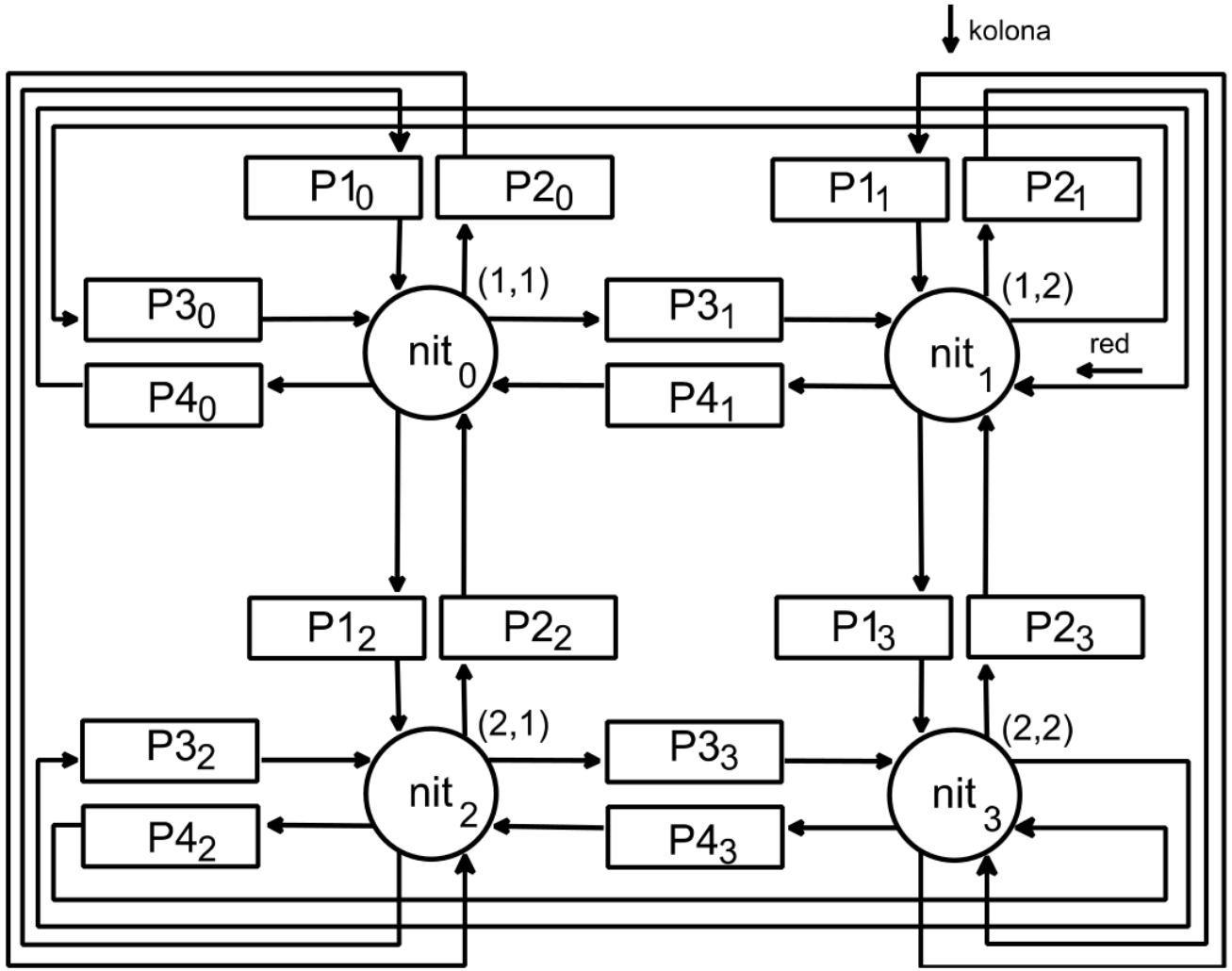
- Nakon toga sve niti množači raspolažu **potrebnim elementima** matrica A i B, pa mogu **istovremeno** da odrede njihov proizvod.
- Za računanje preostalog proizvoda, potrebno je da niti **množači međusobno razmene** raspoložive elemente matrica A i B.
- Međusobna razmena elemenata matrica A i B je **pravilna**, pa sve niti množači šalju raspoloživi element matrice A **desno**, a raspoloživi element matrice B **dole**, i zatim primaju novi element matrice A sa **leva**, a novi element matrice B **odozgo**.

Zamisao paralelnog množenja matrica

- Nakon ovakve razmene, sve niti množači raspolažu elementima neophodnim za računanje **drugog proizvoda**, pa mogu istovremeno da ga odrede.
- Po izračunavanju svog elementa matrice C, svaka nit ga šalje **upravljačkoj** niti.

Komunikaciona osnova paralelnog množenja matrica

- Vezivanje niti množača u matricu je inspirisano idejom **dvodimenzionalne sistoličke arhitekture** (**mesh**).
- Svaka nit množač iz ovog niza ima **levo**, **desno**, **iznad** i **ispod** sebe po **dva pregratka** za poruke



Komunikaciona osnova paralelnog množenja matrica

Komunikaciona osnova paralelnog množenja matrica

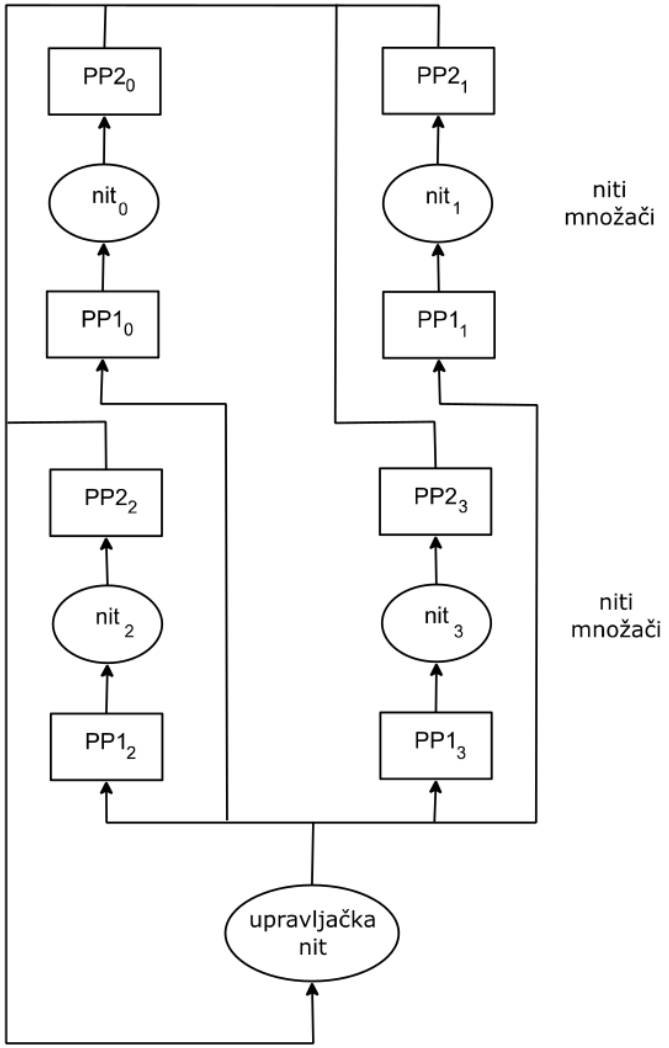
- Pregraci (**Pij**) **levo** uspostavljaju komunikacioni kanal od levog suseda i ka **levom** susedu iz matrice.
- Slično, pregraci **desno** uspostavljaju komunikacioni kanal od **desnog** suseda i ka **desnom** susedu iz matrice.
- Podrazumeva se da je nit množač sa **levog kraja** svakog reda matrice **desni sused** niti množača sa **desnog kraja** istog reda i **obrnuto**.

Komunikaciona osnova paralelnog množenja matrica

- Pregraci **iznad** uspostavljaju komunikacioni kanal od **gornjeg** suseda i ka **gornjem** susedu iz matrice.
- Slično, pregraci **ispod** uspostavljaju komunikacioni kanal ka **donjem** susedu i od **donjeg** suseda iz matrice.
- Podrazumeva se da je nit množač sa **gornjeg** kraja svake kolone matrice **donji** sused niti množača sa **donjeg** kraja iste kolone i **obrnuto**.

Komunikaciona osnova paralelnog množenja matrica

- Na ovaj način svaka nit množač može da razmenjuje poruke sa svojim **horizontalnim** i **vertikalnim** susedima.
- Podrazumeva se da uz svaku nit množač postoji **poseban par pregradaka** posredstvom koga niti množači mogu da direktno razmenjuju poruke sa **upravljačkom niti**.



Komunikaciona osnova paralelnog množenja matrica

Komunikaciona osnova paralelnog množenja matrica

- Templejt klasa **Mesh** omogućuje uspostavljanje komunikacionih kanala koji su potrebni za uvezivanje niti u matricu.
- Njena tri parametra omogućuju definisanje **tipa poruke** i **broja redova** i **kolona niti** u matrici.
- Elementi polja **slots1**, **slots2**, **slots3** i **slots4** odgovaraju, respektivno, pregracima: **P1i**, **P2i**, **P3i** i **P4i**.

Komunikaciona osnova paralelnog množenja matrica

- Elementi polja **special1** i **special2** odgovaraju posebnim pregracima **PP1i**, **PP2i**, za direktnu komunikaciju između niti množača i upravljačke niti.
- Operacije **my_row()** i **my_column()** omogućuju nitima da dobiju redni broj svog reda i kolone iz matrice.
- Podrazumeva se da se kao oznake niti množača koriste redni brojevi: **0, 1, 2, 3** i tako dalje.
- Operacije **send()** i **receive()** omogućuju razmenu poruka.

Komunikaciona osnova paralelnog množenja matrica

- Prvi par ovih operacija omogućuje **upravljačkoj** niti da komunicira sa svakom od niti **množača**, za šta je neophodno korišćenje **rednih brojeva** njihovih redova i kolona.
- Drugi par ovih operacija omogućuje nitima množačima da komuniciraju sa svojim **horizontalnim** i **vertikalnim** susedima koje označavaju pomoću konstanti **UPPER**, **DOWN**, **LEFT** i **RIGHT**, kao i sa upravljačkom niti koju označavaju pomoću konstante **SPECIAL**.
- Polja **rows** i **columns** pokazuju koliko je angažovano niti množača.
- Njihovo postavljanje omogućuju operacije **rows_set()** i **columns_set()**.

Komunikaciona osnova paralelnog množenja matrica

```
#include "box.hh"

enum Mesh_relative_position { UPPER, DOWN, LEFT, RIGHT, SPECIAL };
template<class MESSAGE, int ROWS, int COLUMNS>
class Mesh {
                Message_box<MESSAGE> slots1[ROWS * COLUMNS];
                Message_box<MESSAGE> slots2[ROWS * COLUMNS];
                Message_box<MESSAGE> slots3[ROWS * COLUMNS];
                Message_box<MESSAGE> slots4[ROWS * COLUMNS];
                Message_box<MESSAGE> special1[ROWS * COLUMNS];
                Message_box<MESSAGE> special2[ROWS * COLUMNS];
                int rows;
                int columns;
public:
                Mesh() : rows(ROWS), columns(COLUMNS) {};
                void send(int row, int column, const MESSAGE* message);
                MESSAGE receive(int row, int column);
                void rows_set(unsigned r) { rows = r; };
                void columns_set(unsigned c) { columns = c; };
                void send(int sender, Mesh_relative_position relative_position, const MESSAGE* message);
                MESSAGE receive(int receiver, Mesh_relative_position relative_position);
                int my_row(int thread_identity);
                int my_column(int thread_identity);
};
```

Komunikaciona osnova paralelnog množenja matrica

```
template<class MESSAGE, int ROWS, int COLUMNS>
void Mesh<MESSAGE, ROWS, COLUMNS>::send(int row, int column,
const MESSAGE* message)
{
        int index = (row - 1) * columns + column - 1;
        special1[index].send(message);
}

template<class MESSAGE, int ROWS, int COLUMNS>
MESSAGE
Mesh<MESSAGE, ROWS, COLUMNS>::receive(int row, int column)
{
        int index = (row - 1) * columns + column - 1;
        return special2[index].receive();
}
```

Komunikaciona osnova paralelnog množenja matrica

```
template<class MESSAGE, int ROWS, int COLUMNS>
void
Mesh<MESSAGE, ROWS, COLUMNS>::send(
        int sender,
        Mesh_relative_position
        relative_position,
        const MESSAGE* message)
{
        Message_box<MESSAGE>* destination;
        switch(relative_position) {
                case UPPER:
                        destination = &(slots2[sender]);
                        break;
                case DOWN:
                        sender += columns;
                        if(sender >= rows * columns)
                                sender -= rows * columns;
                        destination = &(slots1[sender]);
                        break;
```

Komunikaciona osnova paralelnog množenja matrica

```
            case LEFT:
                    destination = &(slots4[sender]);
                    break;
            case RIGHT:
                    if(((sender+1) % columns) == 0)
                            sender -= columns;
                    Destination = &(slots3[sender+1]);
                    break;
            case SPECIAL:
                    destination = &(special2[sender]);
                    break;
            }
            destination->send(message);
}
```

Komunikaciona osnova paralelnog množenja matrica

```
            case LEFT:
                    destination = &(slots4[sender]);
                    break;
            case RIGHT:
                    if(((sender+1) % columns) == 0)
                            sender -= columns;
                    Destination = &(slots3[sender+1]);
                    break;
            case SPECIAL:
                    destination = &(special2[sender]);
                    break;
            }
            destination->send(message);
}
```

Komunikaciona osnova paralelnog množenja matrica

```
template<class MESSAGE, int ROWS, int COLUMNS>
MESSAGE
Mesh<MESSAGE, ROWS, COLUMNS>::receive(int receiver,
Mesh_relative_position relative_position)
{
        Message_box<MESSAGE>* source;
        switch(relative_position) {
                case UPPER:
                        source = &(slots1[receiver]);
                        break;
                case DOWN:
                        receiver += columns ;
                        if(receiver >= rows * columns)
                                receiver -= rows *
                                                columns;
                        source = &(slots2[receiver]);
                        break;
                case LEFT:
                        source = &(slots3[receiver]);
                        break;
```

Komunikaciona osnova paralelnog množenja matrica

```
                case RIGHT:
                        if(((receiver+1) % columns) == 0)
                                receiver -= columns;
                                source =
&(slots4[receiver+1]);
                                break;
                case SPECIAL:
                        source = &(special1[receiver]);
                        break;
                }
                return source->receive();
}
```

Komunikaciona osnova paralelnog množenja matrica

```
            case RIGHT:
                if(((receiver+1) % columns) == 0)
                    receiver -= columns;
                    source =
                        &(slots4[receiver+1]);
                    break;
            case SPECIAL:
                source = &(special1[receiver]);
                break;
            }
            return source->receive();
}
```

Komunikaciona osnova paralelnog množenja matrica

```
template<class MESSAGE, int ROWS, int COLUMNS>
int
Mesh<MESSAGE, ROWS, COLUMNS>::my_row(int thread_identity)
{
        return(thread_identity / columns + 1);
}

template<class MESSAGE, int ROWS, int COLUMNS>
int
Mesh<MESSAGE, ROWS, COLUMNS>::my_column(int
thread_identity)
{
        return(thread_identity % columns + 1);
}
```

Izvedba paralelnog množenja matrica

- Ponašanje niti množača opisuje funkcija **thread_multiplier()**.
- Ona opisuje inicijalno **raspodeljivanje elemenata matrica A i B**, kao i **izračunavanje** pojedinih elemenata **matrice C**.

Izvedba paralelnog množenja matrica

- Funkcija **input_matrix()** omogućuje preuzimanje elemenata matrica **A i B sa tastature** i njihovo upućivanje pojedinim nitima uvezanim u matricu.
- Operacija **thread_manager()** opisuje aktivnost upravljačke niti, koja **stvara niti množače**, isporučuje im odgovarajuće elemente matrica **A i B, preuzima elemente matrice C i prikazuje ih**.

Izvedba paralelnog množenja matrica

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "mesh.hh"
#include "thread_identity.hh"

const int SQUARE_MATRIX_ORDER = 2;
const int THREAD_ROWS = SQUARE_MATRIX_ORDER;
const int THREAD_COLUMNS = SQUARE_MATRIX_ORDER;
Mesh<int, THREAD_ROWS, THREAD_COLUMNS> mesh;
```

Izvedba paralelnog množenja matrica

```
Thread_identity thread_identity;

void thread_multiplier()
{
        int counter;
        int matrix_a_element;
        int matrix_b_element;
        int matrix_c_element;
        int tid = thread_identity.get();
        matrix_a_element = mesh.receive(tid, SPECIAL);
        matrix_b_element = mesh.receive(tid, SPECIAL);
        counter = (THREAD_ROWS - mesh.my_row(tid) + 1) % THREAD_ROWS;
        while((counter--) > 0) {
                mesh.send(tid, RIGHT, &matrix_a_element);
                matrix_a_element = mesh.receive(tid, LEFT);
        }
        counter = (THREAD_COLUMNS - mesh.my_column(tid) + 1) % THREAD_COLUMNS;
        while((counter--) > 0){
                mesh.send(tid, DOWN, &matrix_b_element);
                matrix_b_element = mesh.receive(tid, UPPER);
        }
```

Izvedba paralelnog množenja matrica

```
            counter = SQUARE_MATRIX_ORDER;
            matrix_c_element = 0;
            while((counter--) > 0) {
                        matrix_c_element += matrix_a_element *
matrix_b_element;
                        mesh.send(tid, RIGHT, &matrix_a_element);
                        mesh.send(tid, DOWN, &matrix_b_element);
                        matrix_a_element = mesh.receive(tid, LEFT);
                        matrix_b_element = mesh.receive(tid, UPPER);
            }
            mesh.send(tid, SPECIAL, &matrix_c_element);
}
```

Izvedba paralelnog množenja matrica

```
void input_matrix(char name)
{
        int matrix_element;
        cout << endl << "matrix " << name
                << " input (elements values from -100 to 100)";
        for(int row = 1; row <= THREAD_ROWS; row++)
                for(int column = 1; column <= THREAD_COLUMNS;
column++)       {
                        do {
                                cout << endl << name << '('
                                        << (short)row << ','
        << (short)column << "):";
                                cin >> matrix_element;
                        }
                        while((matrix_element < -100) ||
                                (matrix_element > 100));
                        mesh.send(row, column, &matrix_element);
                }
}
```

Izvedba paralelnog množenja matrica

```
void thread_manager()
{
        int row;
        int column;
        cout << endl << "PARALLEL MATRIX-BY-MATRIX MULTIPLICATION";
        for(row = 1; row <= THREAD_ROWS; row++)
                for(column = 1; column <= THREAD_COLUMNS; column++) {
                        thread (thread_multiplier).detach();
                }
        input_matrix('A');
        input_matrix('B');
        cout << endl << "matrix C=A*B";
        for(row = 1; row <= THREAD_ROWS; row++) {
                cout << endl;
                for(column = 1; column <= THREAD_COLUMNS; column++) {
                        cout << mesh.receive(row, column) << " ";
                }
        }
        cout << endl;
}
```

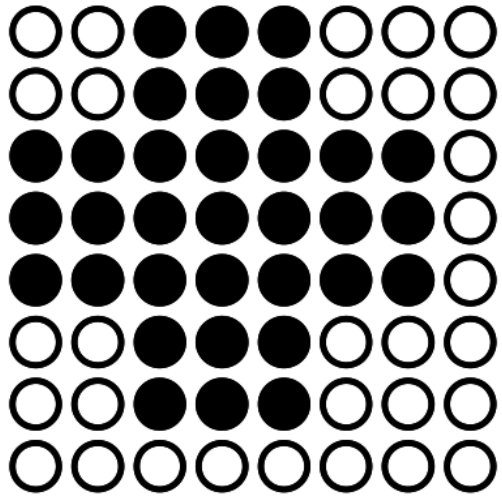
Izvedba paralelnog množenja matrica

- Računanje svih elemenata matrice C, opisano u prethodnom programu, traje koliko i računanje **samo jednog** od ovih elemenata, ako se svakoj od niti, zaduženih za računanje po jednog elementa matrice C, dodeli **poseban procesor**.
- Ali, ako se prethodni program izvršava na jednoprocesorskom računaru, tada je dužina njegovog izvršavanja **proporcionalna broju elemenata** matrice C.

```
int main()
{
        thread manager(thread_manager);
        manager.join();
}
```

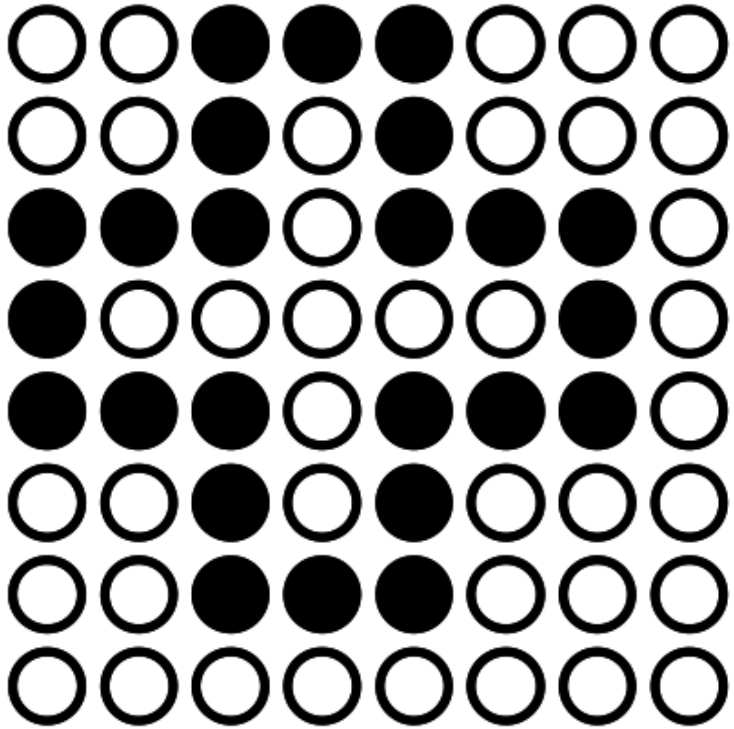
Zamisao paralelnog izdvajanja konture

- Liku, predstavljenom crnim tačkama na crno–beloj slici:



- odgovara kontura koju obrazuju crne tačke, u čijem susedstvu je bar jedna bela tačka:

Zamisao paralelnog izdvajanja konture



Zamisao paralelnog izdvajanja konture

- Izdvajanje konture ovakvog lika se zasniva na **uparivanju** boje svake tačke lika sa bojom tačaka iz njenog susedstva.
- Podrazumeva se da su svakoj tački **p** susedne tačke koje su za nju **iznad** (**up: u**), **iznad desno** (**up right: ur**), **iznad levo** (**up left: ul**), **ispod** (**down: d**), **ispod desno** (**down right: dr**), **ispod levo** (**down left: dl**), **desno** (**right: r**) i **levo** (**left: l**).
- Ovo važi i za **rubne** tačke slike, jer se smatra da su **suprotne ivice** slike **spojene**.

Zamisao paralelnog izdvajanja konture

- Ako crnoj tački odgovara vrednost 1, a beloj vrednost 0, tada iskaz:

  - **c = p&(~u)|p&(~ur)|p&(~ul)|p&(~d)|p&(~dr)|p&(~dl)|p&(~r)|p&(~l);**

- opisuje proveru da li je tačka p konturna.

Zamisao paralelnog izdvajanja konture

- Za razne tačke, uparivanja njihovih boja su **nezavisna**, pa se mogu odvijati **paralelno**.
- One, znači, mogu biti predmet aktivnosti **raznih niti**.
- Pri tome, svaka od ovih konturnih niti proverava za po jednu tačku slike da li dotična tačka pripada **konturi lika**.
- Podrazumeva se da su konturne niti raspoređene u prostoru kao i tačke koje su im dodeljene, znači uvezane u **matricu**.

Zamisao paralelnog izdvajanja konture

- Za susedne konturne niti je važno da budu **komunikaciono povezane**, radi razmena boja tačaka koje su im dodeljene.
- Dovoljno je da svaka konturna nit bude komunikaciono povezana sa konturnim nitima **iznad**, **ispod**, **levo** i **desno** od sebe, jer tada, posredstvom svojih vertikalnih i horizontalnih suseda, ona može razmeniti boje tačaka i sa svojim **dijagonalnim** susedima.
- Podrazumeva se da posebna **upravljačka** nit saopštava konturnim nitima boju njihove tačke i od njih prima podatak da li je njihova tačka konturna.

Komunikaciona osnova paralelnog izdvajanja konture

- Templejt klasa **Mesh** omogućuje komunikaciono povezivanje konturnih niti između sebe, kao i njihovu direktnu komunikaciju sa upravljačkom niti.

Izvedba paralelnog izdvajanja konture

- Funkcija **`thread_contour()`** opisuje aktivnost konturnih niti.
- Svaka od njih raspolaže bojom svoje tačke (**`my_pixel`**), preuzima boju susednih tačaka (**`foreign_pixel`**) i izračunava da li je njena tačka konturna (**`contour_pixel`**).
- Stvaranje ovih niti (korišćenjem bezimenih objekata klase **`thread`**), uvezanih u matricu, je u nadležnosti upravljačke niti, čiju aktivnost opisuje operacija **`thread_manager()`**.
- Upravljačka nit šalje konturnim nitima tačke slike (**`picture`**) i preuzima i prikazuje konturu.

Izvedba paralelnog izdvajanja konture

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "mesh.hh"
#include "thread_identity.hh"

const int SQUARE_MATRIX_ORDER = 8;

const int THREAD_ROWS = SQUARE_MATRIX_ORDER;

const int THREAD_COLUMNS = SQUARE_MATRIX_ORDER;

Mesh<char, THREAD_ROWS, THREAD_COLUMNS> mesh;

Thread_identity thread_identity;
```

Izvedba paralelnog izdvajanja konture

```
void thread_contour()
{
        char my_pixel;
        char foreign_pixel;
        char contour_pixel;
        int tid = thread_identity.get();
        my_pixel = mesh.receive(tid, SPECIAL);
        mesh.send(tid, DOWN, &my_pixel);
        contour_pixel = my_pixel &(~(foreign_pixel = mesh.receive(tid, UPPER)));
        mesh.send(tid, LEFT, &foreign_pixel);
        contour_pixel |= my_pixel &(~(mesh.receive(tid, RIGHT)));
        mesh.send(tid, RIGHT, &foreign_pixel);
        contour_pixel |= my_pixel &(~(mesh.receive(tid, LEFT)));
        mesh.send(tid, UPPER, &my_pixel);
        contour_pixel |= my_pixel &(~(foreign_pixel = mesh.receive(tid, DOWN)));
        mesh.send(tid, LEFT, &foreign_pixel);
        contour_pixel |= my_pixel &(~(mesh.receive(tid, RIGHT)));
        mesh.send(tid, RIGHT, &foreign_pixel);
        contour_pixel |= my_pixel &(~(mesh.receive(tid, LEFT)));
        mesh.send(tid, LEFT, &my_pixel);
        contour_pixel |= my_pixel &(~(mesh.receive(tid, RIGHT)));
        mesh.send(tid, RIGHT, &my_pixel);
        contour_pixel |= my_pixel &(~(mesh.receive(tid, LEFT)));
        mesh.send(tid, SPECIAL, &contour_pixel);
}
```

Izvedba paralelnog izdvajanja konture

```
const char picture[THREAD_ROWS][THREAD_COLUMNS] =
{ {0, 0, 1, 1, 1, 0, 0, 0},
  {0, 0, 1, 1, 1, 0, 0, 0},
  {1, 1, 1, 1, 1, 1, 1, 0},
  {1, 1, 1, 1, 1, 1, 1, 0},
  {1, 1, 1, 1, 1, 1, 1, 0},
  {0, 0, 1, 1, 1, 0, 0, 0},
  {0, 0, 1, 1, 1, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0}
};
```

Izvedba paralelnog izdvajanja konture

```
void thread_manager()
{
		int row;
		int column;
		cout << endl << "PARALLEL CONTOUR FINDING";
		for(row = 1; row <= THREAD_ROWS; row++) {
				cout << endl;
				for(column = 1; column <= THREAD_COLUMNS; column++) {
						cout << ((picture[row - 1][column - 1] == 0) ? ' ' : '.');
						thread (thread_contour).detach();
						mesh.send(row, column, &picture[row - 1][column - 1]);
				}
		}
		for(row = 1; row <= THREAD_ROWS; row++) {
				cout << endl;
				for(column = 1; column <= THREAD_COLUMNS; column++)
						cout << ((mesh.receive(row, column) == 0) ? ' ' : '.');
		}
}

int main()
{
		thread manager(thread_manager);
		manager.join();
}
```

Izvedba paralelnog izdvajanja konture

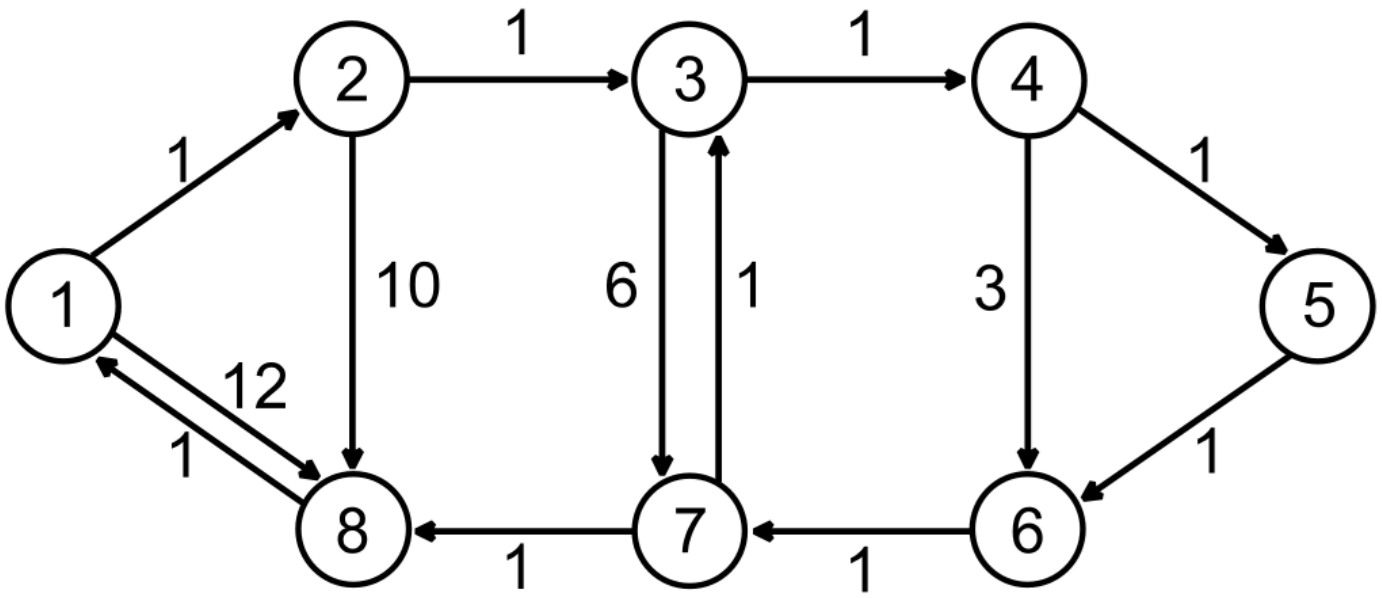
- Sadržaj izvorne datoteke p09.cpp predstavlja potpun konkurentni program. U toku njegovog izvršavanja nastane **65** niti.
- Izdvajanje konture, opisano u prethodnom programu, traje koliko i provera za **jednu tačku** da li je konturna, ako se svakoj od niti, zaduženoj za po jednu ovakvu proveru, dodeli **poseban procesor**.
- Ali, ako prethodni program izvršava jednoprocesorski računar, tada je vreme, potrebno za izdvajanje konture, **proporcionalno broju tačaka** slike.

Izvedba paralelnog izdvajanja konture

- Pristup, primenjen u prethodnom programu za izdvajanje konture, se može primeniti i za **čišćenje slike** od posledica **šuma** (koji se javlja pri telekomunikacionom prenosu slike).
- Ovaj šum izaziva **izmenu boja** pojedinih tačaka, pa se javljaju **crne tačke na beloj pozadini i obrnuto**.
- Čišćenje šuma podrazumeva **izmenu boje** svake tačke koja je **okružena tačkama suprotne boje**.

Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Usmereni graf se sastoji od numerisanih čvorova, povezanih usmerenim spojnicama raznih **nenultih dužina:**



Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Određivanje **najkraće udaljenosti** čvorova usmerenog grafa od zadanog čvora se svodi na **sabiranje dužina** spojnica, koje vode od zadanog do pojedinih od preostalih čvorova.
- Za svaki od preostalih čvorova udaljenost se određuje **relativno u odnosu na prethodnika**, sabiranjem udaljenosti prethodnika i dužine spojnice koja dolazi od prethodnika.

Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Ako od zadanog do nekog čvora vodi **k** spojnica, odnosno, ako se između ovih čvorova nalazi **k-1** drugih čvorova, tada se udaljenost posmatranog čvora od zadanog čvora može odrediti tek nakon **k** koraka, pošto se za svaki od **k-1** čvorova, koji prethode pomenutom čvoru, odrede njihove udaljenosti od zadanog čvora.
- Kada od zadanog do nekog od preostalih čvorova vodi **više puteva**, koji se razlikuju bar po jednoj spojnici, tada udaljenost pomenutog čvora od zadanog čvora, određena u nekom od koraka, može biti **izmenjena** u nekom od narednih koraka, ako je put preko više spojnica **kraći**.

Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Određivanja najkraće udaljenosti čvorova koji ne leže na istom putu, posmatrano od zadanog čvora, su međusobno **nezavisna**, pa se mogu odvijati **paralelno**.
- Ona, znači, mogu biti predmet aktivnosti **raznih** niti.
- Pri tome se svakoj od ovih **čvornih niti** dodeljuje po jedan čvor, a čvorne niti su raspoređene u prostoru kao čvorovi koji su im dodeljeni.

Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Za čvorne niti je važno da budu međusobno **komunikaciono povezane** kao i pridruženi im čvorovi.
- To je potrebno da bi čvorne niti mogle međusobno sarađivati, radi slanja **sledbenicima** njihovih udaljenosti i radi prijema svojih udaljenosti od **prethodnika**.
- U opštem slučaju ovo podrazumeva **potpuno međusobno povezivanje** čvornih niti.

Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Na početku određivanja udaljenosti, zadanom čvoru se dodeli udaljenost **0**, a svim ostalim čvorovima **beskonačna** udaljenost.
- Saradnja čvornih niti se odvija u **koracima**.
- Svaki od njih obuhvata **dve faze**.
- U **prvoj** fazi, svaka čvorna nit **šalje** svojim **sledbenicima** njihovu **udaljenost**, koja je jednaka ili **izračunatoj** ili vrednosti **beskonačno**, zavisno od toga da li je čvorna nit **primila** ili **ne** svoju udaljenost od prethodnika.

Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- U **drugoj** fazi, svaka čvorna nit **prima** od svojih **prethodnika** svoju udaljenost.
- Primljena udaljenost **zamenjuje** postojeću samo ako je **manja** od nje.
- Nakon svakog koraka sledi provera da li se izmenila ijedna od udaljenosti.
- Prvi korak, u kome **izostanu izmene** određivanih udaljenosti, označava kraj postupka, jer su tada određene **najkraće udaljenosti** čvorova usmerenog grafa od zadanog čvora, pa nove izmene nisu moguće.

Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Provera kraja postupka nastupa kada **sve čvorne niti završe započeti korak**.
- Prelazak čvorne niti na **novi korak** ima smisla tek nakon što se u pomenutoj proveri ustanovi da postupak **nije završen**.
- Zato aktivnosti čvornih niti moraju biti usklađene sa pomenutom proverom.

Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Ovakvo usklađivanje aktivnosti niti, u kome, pre prelaska ijedne niti na novi korak, **sve niti moraju završiti započeti korak**, se naziva **barijerna sinhronizacija** (**barrier synchronization**).
- Barijerna sinhronizacija se uspostavlja zahvaljujući **posredovanju upravljačke niti**.

Zamisao paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Radi ostvarenja barijerne sinhronizacije, svaka od čvornih niti, po završetku započetog koraka, **šalje upravljačkoj niti**, u okviru jedne poruke, **broj svoga čvora** i **svoju važeću udaljenost**.
- Nastavak aktivnosti čvornih niti zavisi od **broja izmenjenih udaljenosti**, koji je primila upravljačka nit.
- Ako je ovaj broj **0**, tada **niti završavaju svoju aktivnost**, jer je postupak **završen**.

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Pravilan karakter saradnje niti opravdava korišćenje **posebnog pregratka** za **svaki smer** razmene poruka između bilo koje dve komunikaciono povezane niti.
- Ali, ako je saradnja niti **sporadična**, kao u slučaju paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa, tada postoji **neizvesnost** posredstvom **kog** pregratka dolazi poruka, pa je moguće da nit očekuje poruku iz **jednog** pregratka, a da poruka u međuvremenu pristigne u **drugi** pregradak.

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Zato je bolje, u situaciji kao što je prethodna, da **svaka nit** prima sve poruke posredstvom samo **jednog** pregratka.
- Pri tome je važno da pregradak sadrži **više odeljaka** da bi mogao da primi **sve poruke**, koje su upućene niti kojoj je pregradak dodeljen.
- U suprotnom slučaju, neizbežna je **mrtva petlja**, ako se **svi pregraci napune**, a sve niti **nastave sa slanjem poruka**.

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Pregratke sa više odeljaka opisuje **nova verzija** templejt klase **Message_box**.
- Njeno polje slots sadrži **SLOTS** odeljaka.
- Svaki od njih prima po **jednu poruku**.
- Polja **first_empty_slot** i **first_full_slot** ukazuju na **prvi prazan**, odnosno na **prvi pun** odeljak.
- Njihove vrednosti se menjaju po **modulo** aritmetici, zbog **ciklične** organizacije odeljaka.

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Polje **full_slot_number** sadrži broj napunjenih pregradaka.
- Polja **not_full** i **not_empty** određuju uslove, od čijeg ispunjenja zavise aktivnosti niti **primalaca** i **pošiljalaca** poruka.
- Operacije **send()** i **receive()** omogućuju **slanje** i **prijem** poruka.

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
template<class MESSAGE, int SLOTS>
class Slot_box {
        mutex mx;
        MESSAGE slots[SLOTS];
        unsigned int first_empty_slot;
        unsigned int first_full_slot;
        unsigned int full_slot_number;
        condition_variable not_full;
        condition_variable not_empty;
public:
        Slot_box() : first_empty_slot(0), first_full_slot(0),
        full_slot_number(0) {};
        void send(const MESSAGE* message);
        MESSAGE receive();
};
```

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
template<class MESSAGE, int SLOTS>
void
Slot_box<MESSAGE, SLOTS>::send(const MESSAGE* message)
{
        unique_lock<mutex> lock(mx);
        while(full_slot_number == SLOTS)
                not_full.wait(lock);
        slots[first_empty_slot] = *message;
        full_slot_number++;
        if(++first_empty_slot == SLOTS)
                first_empty_slot = 0;
        not_empty.notify_one();
}
```

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
template<class MESSAGE, int SLOTS>
MESSAGE
Slot_box<MESSAGE, SLOTS>::receive()
{
        unsigned int return_index;
        unique_lock<mutex> lock(mx);
        while(full_slot_number == 0)
                not_empty.wait(lock);
        return_index = first_full_slot;
        full_slot_number--;
        if(++first_full_slot == SLOTS)
                first_full_slot = 0;
        not_full.notify_one();
        return slots[return_index];
}
```

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Paralelno određivanje najkraćih međusobnih udaljenosti čvorova usmerenog grafa zahteva mogućnost **direktne razmene** poruka između **svih** čvornih niti.
- U tom slučaju, potrebno je komunikaciono **potpuno međusobno povezati sve čvorne niti**.
- To se ostvaruje ako se uz svaku čvornu nit veže **pregradak sa više odeljaka** i ako se **svim** čvornim nitima dozvoli da **šalju poruke u pregratke preostalih čvornih niti**.

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Templejt klasa **Any2any** omogućuje uspostavljanje komunikacionih kanala koji su potrebni za **potpuno međusobno povezivanje** čvornih niti.
- Njena tri parametra omogućuju definisanje **tipa poruke**, **broja pregradaka** i **broja odeljaka** u **svakom pregratku**.

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Pojedinim **pregracima** odgovaraju elementi polja **boxes**.
- Operacije **send()** i **receive()** omogućuju razmenu poruka.
- Podrazumeva se da **pregracima** i **oznakama čvornih niti** odgovaraju isti redni brojevi: **1, 2, 3 i tako dalje**.
- Takođe se podrazumeva da pregradak sa rednim brojem **0** (**boxes[0]**) odgovara **upravljačkoj niti**.

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
#include "slot_box.hh"

template<class MESSAGE, int NODES, int SLOTS>
class Any2any {
        Slot_box<MESSAGE, SLOTS> boxes[NODES + 1];
public:
        Any2any() {};
        void send(int node, const MESSAGE* message);
        MESSAGE receive(int node);
};
```

Komunikaciona osnova paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
template<class MESSAGE, int NODES, int SLOTS>
void
Any2any<MESSAGE, NODES, SLOTS>::send(int node, const
MESSAGE* message)
{
        boxes[node].send(message);
}

template<class MESSAGE, int NODES, int SLOTS>
MESSAGE
Any2any<MESSAGE, NODES, SLOTS>::receive(int node)
{
        return boxes[node].receive();
}
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- U posmatranom primeru graf ima **8** čvorova i predstavljen je nizom **graph**.
- Njegovih **prvih 8** elemenata sadrži **dužine spojnica** usmerenih od **prvog čvora** prema svim čvorovima, njegovih **drugih 8** elemenata sadrži dužine spojnica usmerenih od **drugog čvora** prema svim čvorovima i tako dalje.
- Promenljiva **barrier** omogućuje ostvarenja barijerne sinhronizacije.
- Konstanta **TOP** simbolizuje beskonačnu udaljenost.

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Funkcija **thread_node()** opisuje ponašanje čvornih niti.
- Njene lokalne promenljive su:
  - **successors** - omogućuje smeštanje **broja sledbenika čvora**
  - **links** - omogućuje smeštanje **oznaka sledbenika** i **udaljenosti čvora od svojih sledbenika**
  - **predecessors** - omogućuje **smeštanje broja prethodnika čvora**
  - **message** - omogućuje smeštanje **poruke koja se šalje sledbenicima** čvora
  - **new_distance** - omogućuje smeštanje **nove najmanje udaljenosti** čvora od zadatog čvora
  - **distance** - omogućuje smeštanje **aktuelne najmanje udaljnosti čvora od zadanog čvora**.

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Funkcija **thread_manager()** opisuje ponašanje **upravljačke niti**.
- Ona na početku, svim čvornim nitima šalje njihovu **početnu udaljenost**, **broj sledbenika**, **oznake** i **udaljenosti** tih sledbenika, kao i **broj prethodnika**, a zatim **stvori** i **pokrene** pomenute čvorne niti (korišćenjem bezimenih objekata klase **thread**), da bi nakon toga **pratila izmene udaljenosti** i na osnovu toga utvrdila kada su određene **najkraće udaljenosti**.
- Ova funkcija sadrži lokalnu promenljivu **distances** za smeštanje **trenutnih udaljenosti svih čvorova** od zadatog čvora, kao i lokalnu promenljivu **change_counter** u kojoj se čuva **broj izmena udaljenosti** u svakom koraku.

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
#include<thread>
#include<iostream>

using namespace std;
using namespace chrono;
using namespace this_thread;

#include "box.hh"
#include "any2any.hh"
#include "thread_identity.hh"

struct Package {
        int node;
        int value;
};

const int THREAD_NODES = 8;
const int BOX_SLOTS = THREAD_NODES * 2;
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
const int TOP = 127;

Message_box<int> barrier[THREAD_NODES];

Any2any<Package, THREAD_NODES, BOX_SLOTS> any2any;

Thread_identity thread_identity(1);
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
void thread_node()
{
        int successors;
        int i;
        Package links[THREAD_NODES - 1];
        int predecessors;
        Package message;
        int new_distance;
        int distance;
        int tid = thread_identity.get();
        distance = any2any.receive(tid).value;
        successors = any2any.receive(tid).value;
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
        for(i = 0; i < successors; i++)
                links[i] = any2any.receive(tid);
        predecessors = any2any.receive(tid).value;
        message.node = tid;
        do {
                for(i = 0; i < successors; i++) {
                        message.value = (distance < TOP)
                                ? (distance + links[i].value) : (TOP);
                        any2any.send(links[i].node, &message);
                }
                for(i = 0; i < predecessors; i++) {
                        new_distance = any2any.receive(tid).value;
                        if(distance > new_distance)
                                distance = new_distance;
                }
                message.value = distance;
                any2any.send(0, &message);
        }
        while(barrier[tid-1].receive() > 0);
}
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
const short
graph[THREAD_NODES][THREAD_NODES] = { {0, 1, 0, 0, 0, 0, 0, 12},
                                     {0, 0, 1, 0, 0, 0, 0, 10},
                                     {0, 0, 0, 1, 0, 0, 6, 0},
                                     {0, 0, 0, 0, 1, 3, 0, 0},
                                     {0, 0, 0, 0, 0, 1, 0, 0},
                                     {0, 0, 0, 0, 0, 0, 1, 0},
                                     {0, 0, 1, 0, 0, 0, 0, 1},
                                     {1, 0, 0, 0, 0, 0, 0, 0}
};
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
void
thread_manager()
{
        int i;
        short distances[THREAD_NODES];
        int j;
        Package message;
        int change_counter;
        cout << endl << "PARALLEL FINDING OF SHORTEST PATH IN
"
                << "DIRECTED AND WEIGHTED GRAPH" << endl
                << "WEIGHT          MATRIX:";
        for(i = 0; i < THREAD_NODES; i++) {
                message.node = 0;
                message.value = ((i == 0) ? 0 : TOP);
                any2any.send(i + 1, &message);
        }
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
for(i = 0; i < THREAD_NODES; i++) {
        distances[i] = 0;
        message.node = 0;
        message.value = 0;
        cout << endl;
        for(j = 0; j < THREAD_NODES; j++) {
                cout << graph[i][j] << ' ';
                if(graph[i][j] > 0)
                        message.value++;
        }
        any2any.send(i + 1, &message);
        for(j = 0; j < THREAD_NODES; j++)
                if(graph[i][j] > 0) {
                        message.node = j + 1;
                        message.value = graph[i][j];
                        any2any.send(i + 1, &message);
                }
        }
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
for(i = 0; i < THREAD_NODES; i++) {
        message.node = 0;
        message.value = 0;
        for(j = 0; j < THREAD_NODES; j++)
                if(graph[j][i] > 0)
                        message.value++;
        any2any.send(i + 1, &message);
}
for(i = 0; i < THREAD_NODES; i++) {
        thread (thread_node).detach();
}
cout << endl << "DISTANCES FROM NODE 1 TO" << endl;
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
do {
        change_counter = 0;
        for(i = 0; i < THREAD_NODES; i++) {
                message = any2any.receive(0);
                if(distances[message.node - 1] != message.value) {
                        change_counter++;
                        distances[message.node - 1] =
message.value;
                }
        }
        if(change_counter == 0)
                for(i = 1; i < THREAD_NODES; i++) {
                        cout << "NODE " << i+1 << ": ";
                        if(distances[i] == TOP)
                                cout << " ";
                        else
                                cout << distances[i] << endl;
                }
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

```
                    message.node = 0;
                    message.value = change_counter;
                    for(i = 0; i < THREAD_NODES; i++)
                            barrier[i].send(&change_counter);
            }
            while(change_counter > 0);
}

int main()
{
            thread manager(thread_manager);
            manager.join();
}
```

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Sadržaj izvorne datoteke p10.cpp predstavlja potpun konkurentni program.
- U toku njegovog izvršavanja nastane 9 niti.
- Paralelno određivanje najkraće udaljenosti čvorova usmerenog grafa od zadanog čvora, opisano u prethodnom programu, je, u najgorem slučaju, **proporcionalno proizvodu najvećeg broja spojnica**, koje **dolaze** u neki čvor i **odlaze** iz njega, i **najvećeg broja koraka**, ako se svakoj od čvornih niti dodeli poseban procesor.

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

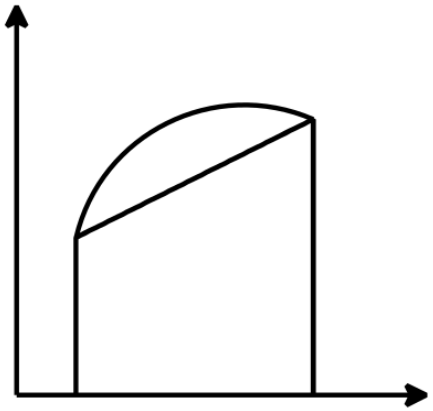
- Ali, ako prethodni program izvršava **jednoprocesorski** računar, tada je vreme, potrebno za određivanje pomenutih udaljenosti, **proporcionalno proizvodu najvećeg broja spojnica**, koje **dolaze** u neki čvor i **odlaze** iz njega, **najvećeg broja koraka** i **ukupnog broja čvorova**.

Izvedba paralelnog određivanja najkraćih međusobnih udaljenosti čvorova usmerenog grafa

- Pristup primenjen za paralelno određivanje najkraće udaljenosti čvorova usmerenog grafa od zadanog čvora predstavlja prirodnu osnovu za paralelno određivanje najkraće udaljenosti svih parova čvorova usmerenog grafa.
- Potrebno je samo uočiti da se određivanje najkraće udaljenosti svih parova čvorova usmerenog grafa može razložiti na **nezavisna** određivanja najkraćih udaljenosti njegovih čvorova od različitih zadanih čvorova.

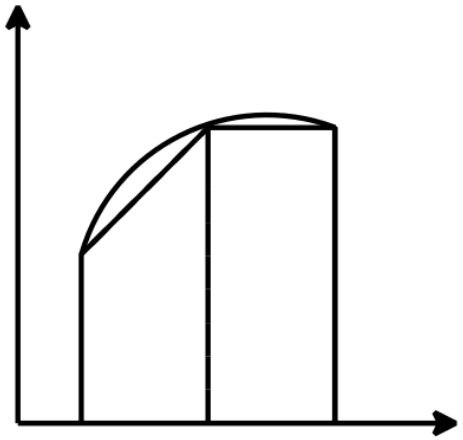
Zamisao paralelnog izračunavanja površine ispod polukruga

- Površina ispod grafa funkcije na intervalu, na kome je ona **neprekidna**, **nenegativna** i **neperiodična**, se određuje **približno**, kao površina **trapeza**, čija visina se podudara sa posmatranim intervalom, a čije baze su jednake **vrednostima funkcije u krajnjim tačkama** ovog intervala



- Površina ovog trapeza predstavlja tra[illegible]e **dovoljno bliska** sumi površina trapeza, konstruisanih na isti način iznad **leve** i **desne** polovine posmatranog intervala

Zamisao paralelnog izračunavanja površine ispod polukruga



- Ako površina **prvog** trapeza **nije bliska** sumi površina **druga dva trapeza**, tada se za svaku od **polovina** posmatranog intervala na prethodni način određuje površina ispod grafa krive.
- Postupak se završava tek kada se **za svaki** od podintervala odredi površina.
- **Suma** površina **svih** podintervala daje traženu površinu.

Zamisao paralelnog izračunavanja površine ispod polukruga

- Određivanja površina za pojedine podintervale su međusobno nezavisna, pa mogu da budu predmet aktivnosti raznih **niti iteratora**.
- Svaka od njih, pri podeli [pod]intervala na dve polovine, odlaže podatke o **jednoj polovini u skladište**, a zatim nastavlja da se bavi **drugom polovinom**.
- Podatke o [pod]intervalima **iz skladišta** preuzimaju **besposlene niti iteratori**.

Zamisao paralelnog izračunavanja površine ispod polukruga

- Ako na početku određivanja tražene površine **upravljačka nit** u skladište **smesti podatke** o posmatranom intervalu, tada je skladište **prazno** po uspešnom određivanju tražene površine, pa **poslednja aktivna** nit iterator, nakon neuspešnog preuzimanja podataka iz **praznog skladišta**, obaveštava upravljačku nit da je tražena površina **određena**.
- S druge strane, nakon **neuspešnog pokušaja** da odloži podatke u **puno skladište**, poslednja aktivna nit iterator obaveštava upravljačku nit da tražena površina **nije određena (kako ovo moze da se desi?)**.

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

- Rukovanje skladištem opisuje templejt klasa **Pool**.
- Njena tri parametra omogućuju saopštavanje **tipa podataka** koji se odlažu u **skladište**, **broja niti iteratora** koje odlažu podatke u **skladište** (**CLIENTS**) i **broja odeljaka** za ove podatke **po svakoj od ovih niti**.

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

- Ova templejt klasa sadrži i polja **not_full** i **not_empty**, koja određuju uslove od čije ispunjenosti zavisi aktivnost niti iteratora koje **odlažu**, odnosno **preuzimaju** podatke.
- Aktivnost svake od njih se **zaustavlja**: pri **odlaganju** podataka, ako je skladište **puno**, i pri preuzimanju **podataka**, ako je skladište **prazno**.

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

- Ovakvo **neselektivno** zaustavljanje aktivnosti uzrokuje **mrtvu petlju**, ako **svaka od niti iteratora** proba: da **odloži podatke u puno skladište**, odnosno, da **preuzme podatke iz praznog skladišta**.
- Zato, radi **sprečavanja** ovakve mrtve petlje, templejt klasa **Pool** sadrži polja **not_full_await_count**, odnosno **not_empty_await_count**, sa **brojem niti iteratora zaustavljenih pri odlaganju**, odnosno, **pri preuzimanju podataka**.

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

- Na osnovu poređenja vrednosti parametra **CLIENTS** sa poljem **not_full_await_count**, odnosno sa poljem **not_empty_await_count**, moguće je ustanoviti kada **poslednja nit** iterator pokuša da **odloži** podatke u **puno skladište**, odnosno da **preuzme** podatke iz **praznog skladišta**.
- Tada se **poslednjoj niti** iteratoru, umesto zaustavljanja njene aktivnosti, vraća vrednost **false**, koja označava **neuspešno odlaganje**, odnosno, **neuspešno preuzimanje** podataka.

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

- Preostale niti iteratori se bude uz vraćanje vrednosti **false**, koja takođe označava **neuspešno odlaganje**, odnosno, **neuspešno preuzimanje** podataka.
- To omogućuju polja **end**, **pool_ending_state** i **exit**, kao i operacija **blocked()**, namenjena upravljačkoj niti.

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

- Broj slobodnih odeljaka u skladištu za prijem podataka sadrži polje **free_slots_count** templejt klase **Pool**.
- Podrazumeva se da su ovi odeljci **kružno raspoređeni** (tako da iza poslednjeg odeljka sledi prvi).
- Indekse **prvog praznog**, odnosno **prvog punog** odeljka sadrže polja **first_empty_slot**, odnosno **first_full_slot**.
- **Odlaganje** podataka u skladište omogućuje operacija **insert()**, a **preuzimanje** podataka iz skladišta operacija **extract()**.

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

```
enum
Pool_ending_states { NOT_END, EMPTY, FULL };

template<class ITEM, int CLIENTS, int SLOTS_PER_CLIENT>
class Pool {
        mutex mx;
        ITEM slots[CLIENTS * SLOTS_PER_CLIENT];
        int free_slots_count;
        int first_empty_slot;
        int first_full_slot;
        condition_variable not_full;
        condition_variable not_empty;
        int not_full_wait_count;
        int not_empty_wait_count;
        condition_variable end;
        Pool_ending_states pool_ending_state;
        bool exit;
```

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

```
public:
        Pool() : free_slots_count(CLIENTS * SLOTS_PER_CLIENT),
                first_empty_slot(0), first_full_slot(0),
                not_full_wait_count(0),
not_empty_wait_count(0),
                pool_ending_state (NOT_END), exit(false) {};
        Pool_ending_states blocked();
        bool insert(ITEM* item);
        bool extract(ITEM * item);
};
```

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

```
template<class ITEM, int CLIENTS, int SLOTS_PER_CLIENT>
Pool_ending_states
Pool<ITEM, CLIENTS, SLOTS_PER_CLIENT>::blocked()
{
        unique_lock<mutex> lock(mx);
        do { end.wait(lock); } while(pool_ending_state ==
NOT_END);
        exit = true;
        if(pool_ending_state == FULL)
                not_full.notify_one();
        else
                not_empty.notify_one();
        return pool_ending_state;
}
```

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

```
template<class ITEM, int CLIENTS, int SLOTS_PER_CLIENT>
bool
Pool<ITEM, CLIENTS, SLOTS_PER_CLIENT>::insert(ITEM* item)
{
        unique_lock<mutex> lock(mx);
        if( (free_slots_count == 0) && (not_full_wait_count == (CLIENTS - 1)) )
        {
                pool_ending_state = FULL;
                end.notify_one();
                return false;
        }
        if(free_slots_count == 0) {
                not_full_wait_count++;
                do { not_full.wait(lock); }
                while( (free_slots_count == 0) && !exit );
                if(exit) {
                        not_full.notify_one();
                        return false;
                }
                not_full_wait_count--;
        }
```

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

```
        slots[first_empty_slot] = *item;
        if((++first_empty_slot) == CLIENTS * SLOTS_PER_CLIENT)
                first_empty_slot = 0;
        free_slots_count--;
        not_empty.notify_one();
        return true;
}
```

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

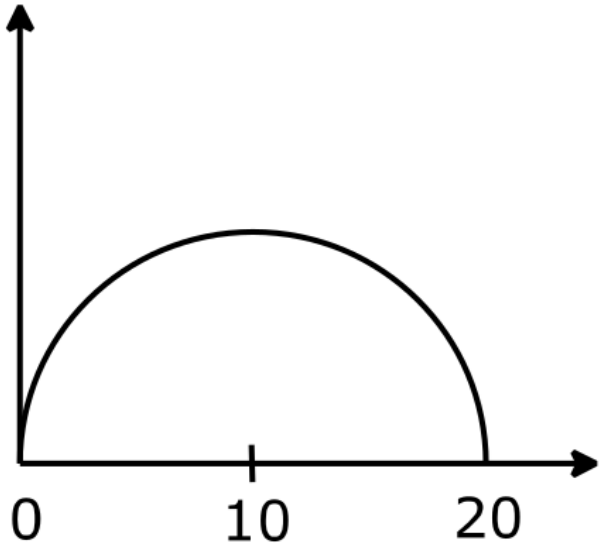
```
template<class ITEM, int CLIENTS, int SLOTS_PER_CLIENT>
bool
Pool<ITEM, CLIENTS, SLOTS_PER_CLIENT>::extract(ITEM* item)
{
        unique_lock<mutex> lock(mx);
        if((free_slots_count == (CLIENTS * SLOTS_PER_CLIENT)) &&
        (not_empty_wait_count == (CLIENTS - 1))) {
                pool_ending_state = EMPTY;
                end.notify_one();
                return false;
        }
        if(free_slots_count == CLIENTS * SLOTS_PER_CLIENT) {
                not_empty_wait_count++;
                do { not_empty.wait(lock); }
                while( (free_slots_count == CLIENTS * SLOTS_PER_CLIENT) && !exit
);
                if(exit) {
                        not_empty.notify_one();
                        return false;
                }
                not_empty_wait_count--;
        }
```

Komunikaciona osnova paralelnog izračunavanja površine ispod polukruga

```
        *item = slots[first_full_slot];
        if((++first_full_slot) == CLIENTS * SLOTS_PER_CLIENT)
                first_full_slot = 0;
        free_slots_count++;
        not_full.notify_one();
        return true;
}
```

Izvedba paralelnog izračunavanja površine ispod polukruga

- U ovom primeru površina se određuje ispod polukruga



Izvedba paralelnog izračunavanja površine ispod polukruga

- Funkcija **semicircle()** omogućuje izračunavanje ordinata tačaka ovog polukruga, a funkcija **square_root()** omogućuje računanje kvadratnog korena.
- On se uvek nalazi između **nule** i **kvadrata**, odnosno između **nule** i **jedinice**, pa se računa primenom postupka **polovljenja intervala**.
- Funkcija **trapezoid()** omogućuje računanje površine trapeza.
- Tip **Segment** opisuje podatke koji se odlažu u skladište, odnosno preuzimaju iz skladišta.

Izvedba paralelnog izračunavanja površine ispod polukruga

- Funkcija **thread_numerical_integrator()** opisuje ponašanje niti iteratora.
- Ove niti akumuliraju površine za pojedine [pod]intervale u polju **total** isključive promenljive **area**.

Izvedba paralelnog izračunavanja površine ispod polukruga

- Funkcija **thread_manager()** određuje aktivnost upravljačke niti, koja preuzima podatke o preciznosti određivanja površine i o granicama posmatranog intervala, priprema podatke o ovom intervalu, odlaže ih u skladište (isključiva promenjiva **pool**), stvara i pokreće niti iteratore (korišćenjem bezimenih objekata klase **thread**) i na kraju čeka ishod određivanja tražene površine.

Izvedba paralelnog izračunavanja površine ispod polukruga

- Podatak o preciznosti određivanja površine se koristi kod računanja kvadratnog korena za određivanje kraja iteracije, ali i kod izračunavanja površine za utvrđivanje da li je površina trapeza sa celog [pod]intervala dovoljno bliska površinama trapeza sa polovina [pod]intervala.