SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 407

# Primjena i ispitivanje učinkovitosti biblioteke “Intel Threading Building Blocks”

Anthony Lipovac

Zagreb, siječanj 2009.

SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 407

# Primjena i ispitivanje učinkovitosti biblioteke “Intel Threading Building Blocks”

Anthony Lipovac

Zagreb, siječanj 2009.

## Sadržaj

# 1. Uvod

Prije samo deset godina bilo je sasvim normalno da se brzina procesora označavala u MHZ. U današnje vrijeme procesor čija se brzina ne označava uz GHz jednostavno ne postoji na tržištu. Još davne 1965. godine, Gordon E. Moore koji je bio jedan od utemeljitelja Intela, izjavio je da će se svakih 24 mjeseci broj tranzistora u integriranom sklopu u odnosu na cijeni povećati dva puta. To njegovo predviđanje se i ostvarilo i danas je poznato kao Mooreov zakon. Iako se su Mooreva predviđanje ostvarila, u zadnjih nekoliko godina došlo je do usporavanja u rastu broja tranzistora unutar procesora. Glavni krivci ove promjene su fizičke granice koje onemogućuju proizvođače procesora da i dalje nude procesore sa sve većim brzinama. Rješenje ovog problema su višejezgreni (*engl. multi-core*) procesori.

Višejezgreni procesori su danas postali standard u velikoj većini računala, iako su se nekad samo koristili u sklopu vrlo skupih i specijaliziranih računala za industriju i akademsku zajednicu. Dok su veliki proizvođači procesora poput Intel ili AMD uspjeli riješiti svoj problem usporavanjem procesorske brzine, istovremeno su se stvorili novi problemi. Računalni programi koji su bili pisani prije dolaska višejezgrenih procesora na tržišta, bili su pisani za procesore koji su imali jednu procesorsku jezgru. Ti programi zapravo ne iskorištavaju maksimalne mogućnosti svih jezgri. Rješenje tom problemu postoji u pisanju paralelnih programa koje je također problematično zbog složenosti programiranja. Najveći problem u paralelizaciji je u tome što je dosta puta teško programerima odrediti koji dijelovi problema nisu međusobno zavisni tako da bi se mogli izvršavati paralelno. Nakon što se i odrede ti dijelovi, programeru dalje ostaje problem rada sa dretvama koji zahtjeva da se odredi količina dretvi koji će se koristiti, što će se na njima izvršavati, te njihova sinkronizacija. Često se svi ovi parametri moraju eksperimentalno određivati što znači da je za paralelizaciju kompleksnih aplikacija potrebno dosta vremena i truda.

Zbog svih ovih problema u 2006. godini, Intel je ponudio je svoje rješenje i to samo godinu nakon što je izbacio svoj prvi višejezgreni procesor (Pentium D). To rješenje se zove Intel Threading Building Blocks koja je C++ biblioteka

napisana za iskorištavanje prednosti višejezgrenih procesora u programskim rješenjima. Korištenjem ugrađenih algoritama i podatkovnih struktura, biblioteka omogućava programerima korištenje paralelizacije unutar svojih programa bez potrebe za ručno stvaranje i sinkronizaciju dretvi.

## 2. Osnove TBB biblioteke

Umjesto stvaranja i sinkroniziranja dretvi koji se koriste tijekom direktnog rada sa dretvama, u TBB biblioteci pristup paralelizaciji je nešto drugačiji. TBB pretvara operacije koje se trebaju paralelizirati u zadatke (*engl. tasks*) koje onda raspoređuje po jezgrama procesora. Korištenjem zadataka potpuno se izbjegava direktni rad korisnika sa dretvama, te se korisnik može koncentrirati na paralelizaciju aplikacije. Da bi se iskoristile maksimalne mogućnosti procesora, TBB koristi metodu koja se zove krađa zadataka (*engl. task stealing*). Kada se program koji je napisan sa TBB bibliotekom pokrene, početna količina posla se ravnomjerno rasporedi po jezgrama procesora. Ako u nekom trenutku jedna jezgra završi sav svoj posao dok druge jezgre još rade i imaju dosta posla, TBB prebaci dio posla na jezgru koja je slobodna.

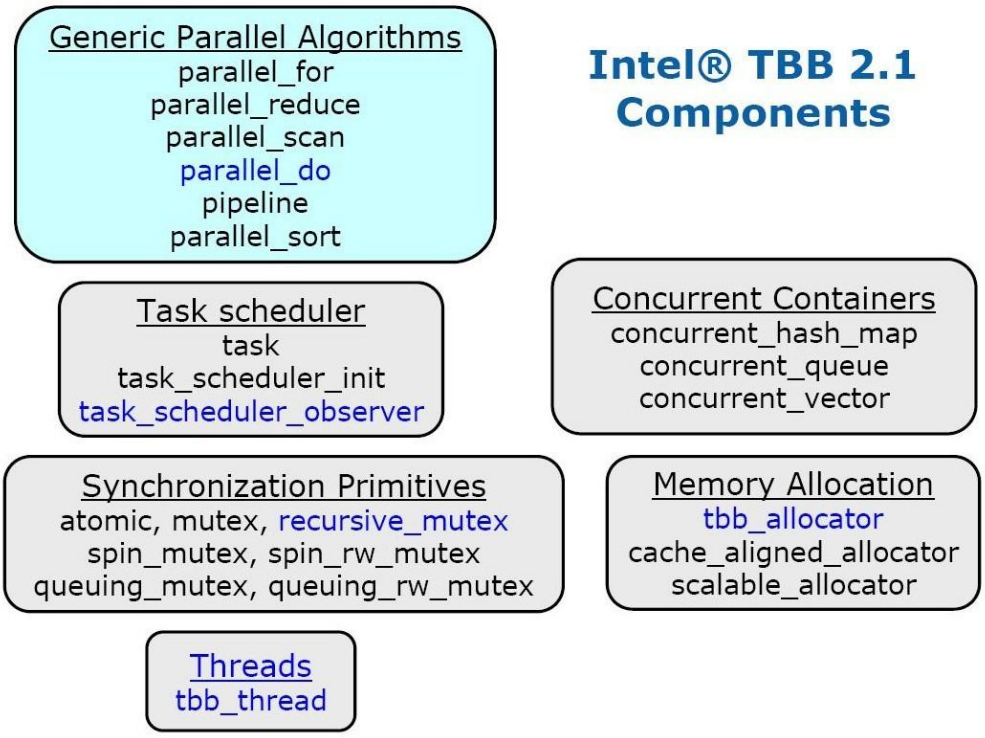
Za razliku od programa koji koriste ručno stvaranja dretvi za paralelizaciju, TBB osigurava skalabilnost. U trenutku kada se pokrene programsko rješenje napisano sa TBB bibliotekom, unutarnji raspoređivač zadataka (*engl. task manager*) analizira sustav na kojem se program izvodi, te izabere optimalan broj dretvi za izvršavanje. Zbog ove činjenice TBB omogućava da će izvršavanje programa biti još brže sa dolaskom procesora sa 4, 8, 16 i više jezgri. Otvorenost TBB biblioteke (koja je dostupna za razne sustave) je također predstavljen činjenicom da se program napisan tom bibliotekom može izvršavati na procesoru bilo kojeg proizvođača i na bilo kojem operacijskom sustavu. Jedini uvjet je da se na sustavu nalazi prevodilac za C++ jezik, te da je u sklopu programskog rješenja uvršten i tbb.dll.

Na početku pisanja novog programa s pomoću TBB paketa bitno je postaviti imenički prostor (*engl. namespace*) na TBB i to sa naredbom *using namespace tbb*. Ovo nije nužno ali sa ovom definicijom kasnije nije potrebno stavljati tbb ispred svake TBB naredbe. U main dijelu programa bitno je inicijalizirati *tbb::task_scheduler_init* objekt koji zapravo predstavlja raspoređivač dretvi. Bez ove inicijalizacije dretve nemaju mogućnost korištenja ugrađenih algoritama u sklopu TBB. Za rad sa Microsoft Visual Studio razvojnom okolinom preporučuje se korištenje dodataka za Visual Studio koji automatski podešava okolinu za TBB,

poput dodavanja putanja do potrebnih biblioteka. Taj se dodatak (*engl. plug-in*) može skinuti sa službenih stranica Intel Threading Building Blocks.

# 3. Komponente TBB biblioteke

Biblioteka Intel Threading Building je sastavljena od šest glavnih komponenti sa kojima se ostvaruje paralelizacija programa.




**Slika 3.1 Prikaz komponenti TBB biblioteke**

## 3.1 Algoritmi

Intel Threading Building Blocks biblioteka je sastavljena od nekoliko ključnih algoritama sa paralelizaciju programskih rješenja. Među jednostavnijima su *parallel_for* i *parallel_reduce*.

### 3.1.1 parallel_for

Algoritam *parallel_for* koristi se za paralelno izvršavanje dijelova kôda za koje se zna da su neovisni od ostalih dijelova. Uz ovaj uvjet potrebno je također znati i koliko puta se treba ponoviti određeni dio kôda. *Parallel_for* je u biti paralelna

verzija for petlje uz uvjet da se sadržaj unutar petlje pretvori u zadatak koji se može samostalno izvršavati. Taj će se zadatak dalje dijeliti pojedinim dretvama. Sadržaj koji se nalazi unutar for petlje stavi se u metodi *operator()*. Za ostvarenje ove funkcionalnosti potrebno je tu metodu ugraditi u jednoj klasi čija je glavna funkcija izvršavanje paralelnog posla. Programski odsječak koji slijedi prikazuje primjer takve klase.

```
class MnozMat
{
      public:
            int rd;
            Matrica a1, b1, c1;
            void operator()( const blocked_range<size_t>& r) const
            {
                  int i = rd;

                  for(size_t j = r.begin(); j < r.end(); j++)
                  {
                        c1.d[i][j] = 0;
                        for(int k = 0; k < b1.n; k++)
                        {
                              c1.d[i][j]+=a1.d[i][k] * b1.d[k][j];
                        }
                  }
            }

      MnozMat(int redak, Matrica a, Matrica b, Matrica c):
            rd(redak), a1(a), b1(b), c1(c)
            {}
};
```

U klasi *MnozMat* nalazi se konstruktor koji prima podatke koji će biti potrebni za posao koji se nalazi unutar funkcije *operator()*. Ovaj konstruktor se mora nalaziti unutar funkcije koja koristi *parallel_for* zbog toga što se u njoj stvaraju posebne kopije podataka za svaki zadatak (*engl. task*). Destruktor u klasi nije potreban ali bez konstruktora izvršavanje nije moguće. Metoda *operator()* kao argument prima *const blocked_range<size_t>& r* koji predstavlja iteracijski prostor od 0 do n-1. Taj iteracijski prostor predstavlja koliko će se puta izvršiti programske naredbe koje se nalaze unutar metode *operator()*. *Blocked_range<T>* je tip podatka koja je ugrađena unutar TBB biblioteke i predstavlja jednodimenzionalni iteracijski prostor. Također postoje i *blocked_range2d* koji predstavlja dvodimenzionalni prostor te *blocked_range3d* koji predstavlja trodimenzionalni prostor.

Tijekom poziva *parallel_for* funkcije šalju se tri argumenta, primjer poziva funkcije slijedi.

```
void FunkMnozMatParal(int redak, Matrica a, Matrica b, Matrica c)
      {
            parallel_for( blocked_range<size_t>(0, b.m),
                  MnozMat(redak, a, b, c), affinity_partitioner());
      }

for(int i = 0; i < mat1.n; i++)
      {
            FunkMnozMatParal(i, mat1, mat2, rez);
      }
```

Navedena *for* petlja koja se nalazi u programskom odsječku zapravo se nalazi unutar *main()* dijela programa te poziva funkciju *FunkMnozMatParal()* n (broj redaka u rezultantnoj matrici) puta i to sa argumentima *i* koji predstavlja redak koji se treba množiti, *mat1* koja je prva matrica za množenje, *mat2* druga matrica, i matrica *rez* u kojoj se sprema rezultantna matrica. Unutar funkcije *FunkMnozMatParal()* nalazi se *parallel_for* koji se poziva sa tri argumenta. Prvi argument je *blocked_range* koji predstavlja iteracijski prostor, dok je drugi argument konstruktor klase u kojoj se nalazi metoda *operator()* te podaci koji se šalju konstruktoru. Zadnji argument predstavlja podatak koji se zove zrnatost (*engl. grain size*). Taj podatak predstavlja količinu posla koji se treba ugraditi u svaki blok koji se šalje procesoru na izvršavanje. Ovaj podatak je bitan jer pomaže u učinkovitijoj raspodjeli posla, jer ako se u određeni blok stavi premalo posla stvoriti će se negativan efekt jer bi se gubilo više vremena na pripremi za izvršavanje nego na sami posao. Preporuka je inače da se podatak postavi na vrijednost od najmanje 10000 do 100000 instrukcija po bloku ako je moguće. Određivanje ove vrijednosti je inače dosta složeno pa su zbog toga ugrađeni posebni automatski operatori. Unosom *auto_partitioner* ili *affinity_partitioner* kao argument omogućava se automatsko određivanje veličine raspodjela posla. Uz tu funkcionalnost *affinity_partitioner* također omogućava automatsko određivanje načina kako će se koristiti priručna memorija procesora. Inače najbolje se ubrzanje postiže korištenjem *affinity_partitioner* kada se u metodi *operator()* radi više računskih operacija na dohvaćene podatke iz memorije, te kada podaci sa kojim se trenutno radi mogu u potpunosti stati u priručnoj memoriji, i kada postoje više od dvije hardverske dretve. Ako se ne postavi ni jedan od ovih dvaju argumenata,

onda se prihvati početna postavka *simple_partitioner* koja postavlja veličinu istu kao i prostor iteracije.

### 3.1.2 parallel_reduce

Kada je potrebno da se na jednom skupu podataka primjeni jedna operacija poput sume ili logičke operacije I, onda je učinkovito koristi funkciju *parallel_reduce*. Kao primjer nekakve operacija mogla biti suma svih elemenata nekog polja. Primjer serijskog načina zbrajanja slijedi.

```
int suma( int polje[], int n )
{
      int s = 0;
      for( int i = 0; i < n; i++ )
      s += f(polje[i]);
      return s;
}
```

Korištenje funkcije *parallel_reduce* dosta je slično implementaciji funkcije *parallel_for*. Kao argumente prima *blocked_range* koji predstavlja iteracijski prostor, instancu klase u kojoj se nalazi metoda *operator()*, te naziv operatora za određivanje veličine granulacije poput *affinity_partitioner*.

Najveća razlika između *parallel_reduce* i *parallel_for* je u tome što *parallel_reduce* posjeduje metodu koja se zove *join()*. Ova metoda se poziva kada jedan zadatak završi obradu podataka te treba spojiti vlastiti rezultat sa rezultatima ostalih zadataka. U *parallel_reduce* i *parallel_for* inače se kao prostor iteracije definiraju *blocked_range* tipovi koji idu od 0 do n-1. U slučaju da je potrebno da iteracija ide od 0 pa do nekog n, pa od n do m koristi se poseban operator koji se zove split. Ovaj operator uzme broj r koji je jednak n+m te podijeli iteracijski prostor na dva dijela. Nakon što se iteracijski prostor podijeli na dva dijela, može se krenuti u istovremenom izvršavanju tih dijelova.

### 3.1.3 parallel_scan

Funkcija *parallel_scan* se koristi kad se želi izračunati paralelni prefiks $\oplus$. Za proračun ovog podatka algoritam prolazi kroz isti set podataka dva puta. Ova

karakteristika je već ugrađena u funkciji *parallel_scan* i to unutar metode *operator()*. Kao i sa *parallel_for* te *parallel_reduce*, potrebno je definirati klasu u kojoj će se nalaziti metoda *operator()* i u kojoj se obavlja operacija paralelnog prefiksa. Uz *operator()*, u klasi je također definirana metoda *reverse_join()* koja je slična metodi *join()* koja se nalazi u *parallel_reduce*, samo što u slučaju funkcije *parallel_reduce* poziv *reverse_join()* spaja rezultat s podatkom koji se nalazi sa desne strane operatora ⊕. *parallel_reduce* kao parametar također prima bilo koji od *auto_partitioner* objekata poput *affinity_partitioner*.

### 3.1.4 parallel_do

Kada se rješavaju problemi u kojima nije poznata gornja granica iteracije do koje se mora ići onda je prikladno koristiti funkciju *parallel_do*. Sama funkcija kao argumente prima donju granicu i privremenu gornju granicu sa vrijednostima poput *list.begin()* i *list.end()*, te konstruktor klase u kojoj se nalazi metoda *operator()*. Dodatna prednost funkcije *parallel_do* je mogućnost dodavanje dodatnih elemenata za obradu kada je obrada početnih elemenata već krenula. Ovo se postiže sa pozivom *feeder.add(item)*, gdje je *item* podatak koji se dodaje. U programskom odsječku [2] koji slijedi prikazan je jednostavan primjer korištenja parallel_do.

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

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

Unutar funkcije *ParallelApplyFooToList(),* poziva se *parallel_do* i to sa podacima početka liste, kraja liste, te konstruktorom klase *ApplyFoo()* u kojoj se nalazi metoda *operator().* Unutar te metode prima se svaki pojedini element liste te se taj podatak šalje funkciji *Foo()* na daljnu obradu. Za razliku od *parallel_for*, *parallel_do*

nema fiksne granice te je idealna za rad sa dinamičkim strukturama podataka poput listi.

### 3.1.5 pipeline

U TBB biblioteci ugrađena je i struktura koja oponaša cjevovod (*engl. pipeline*). Ta struktura se ponaša tako da podaci prolaze kroz određene faze te se u tim fazama izvršavaju određene promjene nad podacima. U klasičnom radu implementacije cjevovoda sa dretvama gdje se na jednoj dretvi izvršava jedna faza postoje dva problema. Prvi problem je u tome što je ubrzanje rada programa ograničeno sa količinom faza koje postoje. Dok je druga poteškoća ta što kada dretva završi sa poslom, svoje rezultate mora predati drugoj dretvi. Da bi se ovi problemi riješili, u TBB biblioteci za svaku se fazu odredi da li se može izvršiti serijski ili paralelno. Sa ovom karakteristikom postiže se ubrzanje zbog toga što se dopušta fazama koje su obilježene kao paralelne da slobodno rade sa podacima bez da čekaju da se neka druga faza završi. Na kraju ispadne da je količina podatak koja će biti obrađena zapravo ovisna o najsporijoj serijskoj fazi.

Cjevovod te svaka faza u TBB biblioteci su opisani klasama *pipeline* i *filter*. Prvi korak u radu sa ovim klasama je instanciranje novog cjevovoda i to sa naredbom *pipeline pipe*. Sami filtri su predstavljeni klasama gdje u svakoj postoji metoda *operator()* unutar koje se nalaze naredbe koje se izvršavaju nad podacima u cjevovodu. Svaka metoda *operator()* mora vratiti neku vrijednost, osim zadnjeg filtra koji vraća NULL. Ta povratna vrijednost je zapravo pokazivač na podatak koji je namijenjen sljedećem filtru u cjevovodu i koju sljedeći filter pretvorbom (*engl. casting*) pretvara u potreban tip. Unutar konstruktora svake klase također se mora definirati da li se filter mora izvršiti serijski *(tbb::filter(serial))* ili se može izvršavati paralelno *(tbb::filter(parallel)).*

U dijelu programa gdje se instancirao cjevovod dodaju se i filtri. Proces dodavanje je takav da se prvo instancira novi objekt klase koji predstavlja filter. Drugi korak dodavanja je pozivanje metode *add_filter()*, koja kao argument prima novu instancu klase koja predstavlja pojedini filter. Bitno je da se filtri dodaju redoslijedom kojem će se i nalaziti u cjevovodu. Primjer dodavanja filtra slijedi.

```
MojFilter novi_filter;
pipe.add_filter( novi_filter );
```

Nakon dodavanja svakog filtra poziva se metoda *pipe.run(limit)*, gdje *limit* predstavlja broja tokena koji se nalaze u cjevovodu. Tokeni u biti predstavljaju podatak u cjevovodu, gdje serijski filter smije raditi na samo jednom tokenu. Dok paralelni filter ima mogućnost rada na više tokena istovremeno. Kada se postigne maksimalni broj tokena u cjevovodu stane se stvaranjem novih na ulaznoj strani te se čeka da se poništi token koje izlazi iz cjevovoda. Na kraju rada sa cjevovodom potrebno ga je i uništiti i to naredbom *pipe.clear()*.

Kapacitet cjevovoda je na kraju određen sa dva podatka: broj tokena te kapacitet najsporijeg serijskog filtra. Bitno je odabrati pravilan broj tokena, jer ako je broj premalen efekt paralelizacije se neće moći osjetiti. Prevelik broj tokena bi prouzrokovao potrebu za prevelikom količinom resursa. Problem najsporijeg serijskog filtra se jedino može ublažiti tako da se osigura da svaki serijski filter radi svoj posao što je brže moguće. Jedan spori serijski filter može umanjiti sav pozitivan efekt ostalih paralelnih filtra.

### 3.1.6 parallel_sort

U sklopu TBB biblioteke ugrađena je funkcija koja omogućava jednostavno sortiranje niza koji ima definiran raspon. Sortiranje je karakterizirano kao nestabilno zbog toga što nekad elemente sa istom vrijednošću ne poreda istim redoslijedom. Funkcija se poziva sa dva parametra gdje je prvi argument početak niza a drugi kraj niza. Također se funkcija može pozvati i sa tri argumenta, gdje treći argument predstavlja poredak sortiranja(padajuće ili rastuće). Izgled poziva *parallel_sort* slijedi:

```
void parallel_sort(RandomAccessIterator begin,
     RandomAccessIterator end,const Compare& comp );
```

## 3.2 Podatkovne strukture

Unutar TBB biblioteke ugrađeni su i spremnici (*engl. containers*) koji dopuštaju da nad jednim spremnikom više dretvi izvršavaju jednu metodu. Vrste tih spremnika su *concurrent_queue*, *concurrent_vector*, te *concurrent_hash_map*.

### 3.2.1 concurrent_queue

Struktura *concurrent_queue* predstavlja red čekanja gdje više dretvi mogu istovremeno dodavati (*engl. push*) i skidati (*engl. pop*) elemente iz reda čekanja. U ovoj verziji reda čekanja postoji problem u tome što se ne zna precizno koji je prvi element koji će se dobiti skidanjem iz reda. Ovaj problem se stvara zbog toga što više dretvi istovremeno skidaju i stavljaju elemente u red. Da bi se održao redoslijed kojima su elementi dodani, kada neka dretva skida određene elemente, redoslijed je jednak redoslijedu unošenja sa strane neke druge dretve. Za umetanje u red koristi se metoda *push(const T& source)*, koja stavlja kopiju podatka na kraj reda, dok se za skidanje elementa koristi metoda *pop(T& destination)* koja kopira prvu vrijednost iz reda u varijablu koja joj se šalje. Inače se ne preporučuje korištenje *concurrent_queue* zbog gubitaka u vremenu, koji se stvara zbog toga što dretve moraju čekati dok se ne skine podatak koji je njim potreban.

### 3.2.2 concurrent_vector

Struktura *concurrent_vector* predstavlja dinamičko jednodimenzionalno polje. U toj strukturi dopušteno je pristupanje elementima polja čak i dok polje raste. Jedino se ne smije pristupiti podatku koji je u procesu stvaranja. Za dodavanje elementa koriste se metode *size_type grow_by( size_type delta )* te *void grow_to_at_least( size_type n )*. Prva metoda dodaje broj elemenata *delta* na kraj polja, dok druga metoda dodaje elemente sve dok polje ne postigne veličinu koja je definirana sa vrijednošću n. Za pristupanje elementima koristi se metoda *reference operator[]( size_type index )* koja vraća referencu na podatak sa zadanim indeksom.

### 3.2.3 concurrent_hash_map

Struktura *concurrent_hash_map* predstavlja strukturu raspršene tablice, sa tim da dopušta više istovremenih pristupa sa strane više dretvi. Ta raspršena tablica je u biti karta koja pretvara ključ u neki određeni tip. Za istovremeno pristupanje više dretvi na jednoj tablici smiju se koristiti samo tri operacije *find*, *insert*, i *erase*. Pozivom metode *bool find( const_accessor& result, const Key& key ) const*, dopušta se čitanje seta podataka sa zadanim ključem, ako postoji u tablici. Metoda

*bool insert( const_accessor& result, const Key& key )* traži za set podataka sa istim ključem koji se dodaje. Ako se ne nađe uneseni ključ, dopušta se dodavanje podataka u tablici. Brisanje seta podataka omogućeno je sa naredbom *bool erase( const Key& key )* samo ako se nađe podatak sa zadanim ključem.

### 3.2.4 Mutex

U Intel Threading Building Blocks biblioteci ugrađeni su spremnici koji se automatski bave omogućavanjem istovremenog pristupa istih podataka sa strane više dretvi. Iako Intel ne preporučava njihovo korištenje također su ugrađeni elementi koji dopuštaju korisniku da sam odredi sinkroniziranje i zaključavanje. U TBB postoje dvije varijante zaključavanja, mutex te atomske operacije (*engl. atomic operations*) koje su nešto brže i jednostavnije od mutex isključivanja.

Mutex je globalna varijabla koja osigurava da ni jedan drugi zadatak ne piše ili čita određeni podatak dok neki drugi zadatak baš u tom trenutku piše taj podatak. U sklopu TBB biblioteke postoji nekoliko verzija mutex zaključavanje gdje je najjednostavniji *spin_mutex* koji se koristi kada je potrebno zaključati odsječak koji sadržava manji broj instrukcija. Karakteristike toga mutex-a su te da se ne prati redoslijed sa kojim su zadaci došli već dopušta one zadatke koji su aktivni da uđu prvi. *Spin_mutex* također ne dopušta zadacima koji su zaključali programski odsječak da ga opet zaključaju. Sa ovim karakteristikama ova vrsta mutex-a je najbrža. Postoje i druge vrste mutex-a poput *queuing_mutex* koji je sličan *spin_mutex* samo što se drži redoslijeda prispijeća zadataka za dodjelu pristupa programskom odsječku.

# 4. Implementacija TBB

Svi primjeri su pokrenuti na računalu sa Intel Core 2 Duo P8600 (2.26 GHz) procesorom te 4GB memorije. Operacijski sustav je Windows Vista 64bit, te je kao razvojna okolina korišten Microsoft Visual Studio 2005.

## 4.1 Množenje matrica

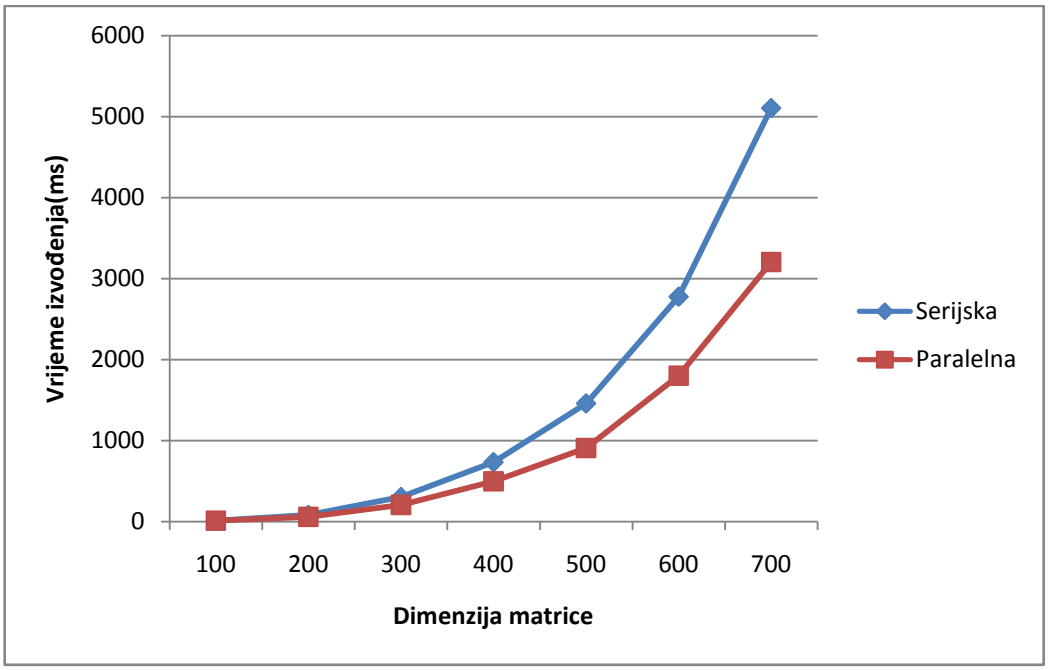
Množenje matrica je vrlo dobar primjer operacije gdje je paralelizacija efikasna zbog toga što se cjelokupni posao može lako podijeliti u nezavisne dijelove. Uvjet za množenje je da su matrice oblika *NxM* te *MxP*, te je rezultanta matrica izgleda *NxP*. Osnovni algoritam za množenje matrica je definiran tako da se svaki redak iz prve matrice množi sa svakim stupcem u drugoj matrici. Ovaj osnovni algoritam je korišten za stvaranje TBB implementacije množenja matrica, uz promjenu da se množenje svakog retka dijelilo na više dijelova.

Iz TBB biblioteke korišten je algoritam *parallel_for* koji je bio najprimjereniji za ovaj zadatak zbog toga što se unaprijed zna količina podataka sa kojem se radi. U *main()* dijelu programa poziva se funkcija koja kao argument šalje redni broj retka koji se treba množiti. U toj funkciji koja se zove *FunkMnozMatParal* nalazi se poziv funkciji *parallel_for*. U pozivu te funkcije kao argumenti šalju se *blocked_range<size_t>* koji je definiran od nule do P, te poziv konstruktoru *MnozMat* u kojem je definirana metoda *operator()*. Uz ta dva argumenta šalje se i podatak *affinity_partitioner* koji omogućava automatsko određivanje veličine granulacije te automatski rad sa priručnom memorijom procesora. Unutar metode *operator()* koja pripada klasi *MnozMat* obavlja se množenje elemenata dvaju ulaznih matrica koja rezultira kao dio jednog retka u rezultatnoj matrici.

Isti algoritam u kojem se odvojeno računa svaki redak rezultatne matrice, također je preslikan u serijskoj izvedbi, te u izvedbi sa ručnom stvaranjem dretvi. Umjesto *parallel_for*, u serijskoj izvedbi poziva se funkcija *MnoziRedak()* N puta te se tako dobiva rezultanta matrica. Za implementaciju primjera množenja matrica sa ručnom stvaranjem dretvi, korišten je jezik Java koji ima ugrađenu klasu *Thread* te se nova dretva stvara za svaki redak u rezultantnoj matrici. Kao što TBB ima

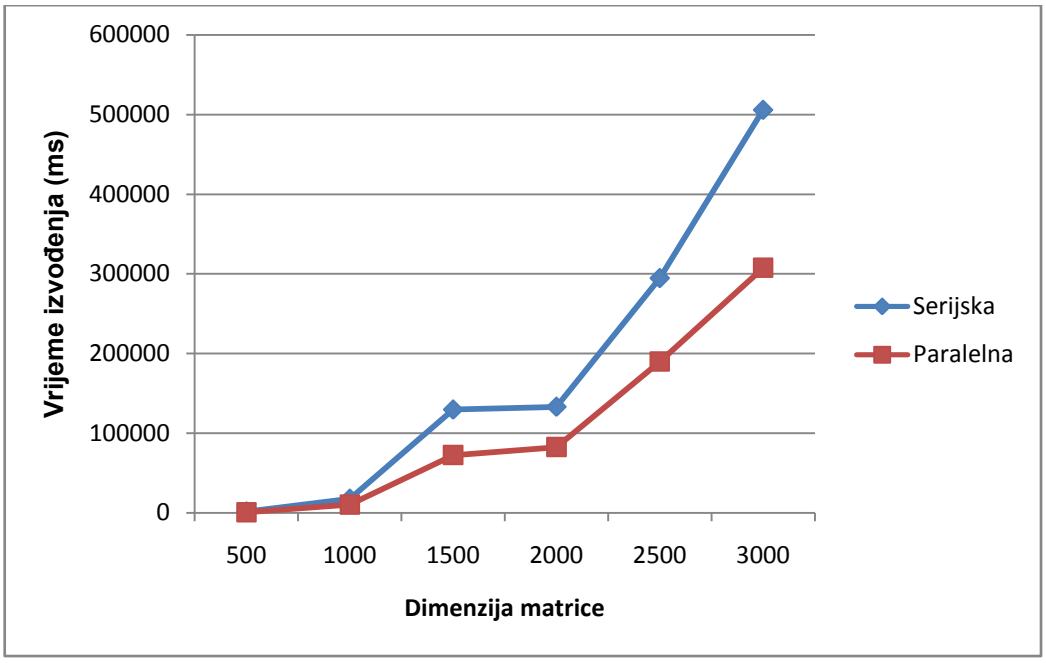
metodu *operator()*, u klasi *Thread* koristi se metoda *run()*. U toj se metodi nalazi sadržaj koji će se izvršavati na pojedinoj dretvi.

Na slici 4.1.1 prikazana je usporedba izvršavanja paralelne TBB verzije programa za množenje matrica, te izvršavanje verzije programa koje koristi slijedno izvršavanje. Iz prikaza se može vidjeti da kada su dimenzije matrica *N=M=P=100*, *N=M=P=200* ili *N=M=P=300* da je razlika u vremenu izvođenja između dva tipa rada dosta malena. Efekt paralelizacije s pomoću TBB biblioteke tek se vidi nakon što su dimenzije matrica veće od 600 te je prednost u vremenu izvođenja jednaka 2000 ms. Objašnjenje ovakvog rezultata je činjenica da matrice sa dimenzijama manje od nekoliko stotina elemenata ne predstavljaju veliki problem procesoru za izračun te se paralelizacijom ne dobiva na vremenu.



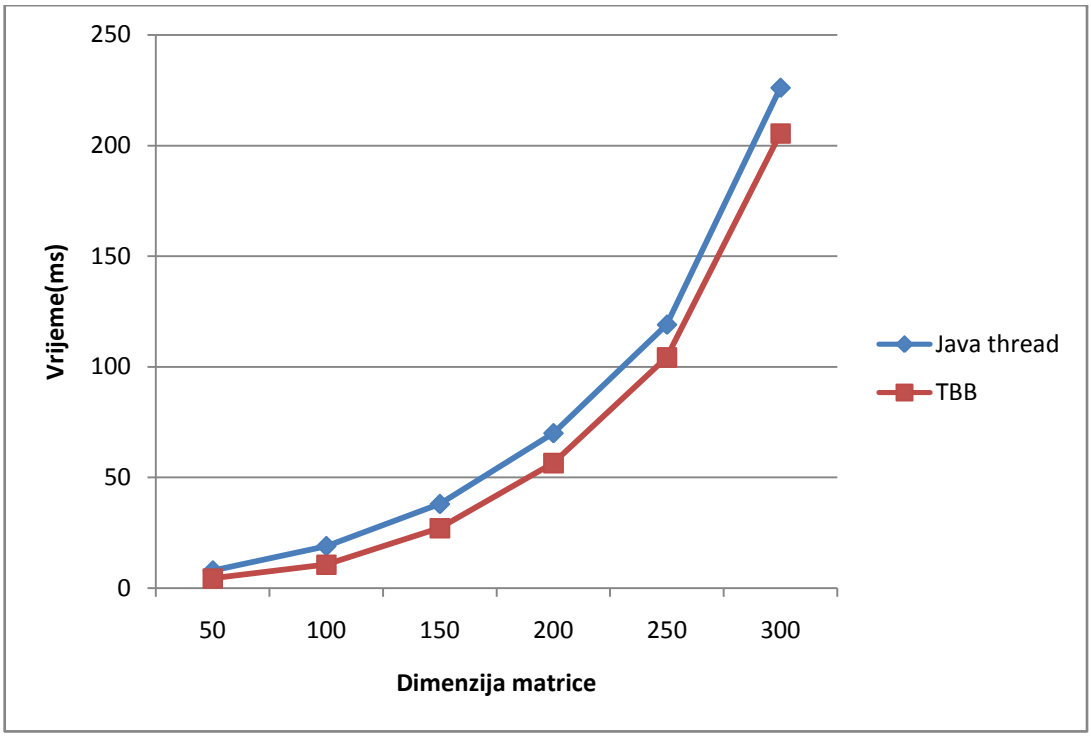

**Slika 4.1.1 usporedba izvođenja TBB i serijske izvedbe**

Prava razlika o vremenima tek se ostvaruje kada dimenzije premašuju nekoliko tisuća te se na slici 4.1.2 može vidjeti da je vremenska razlika čak 200000 milisekundi ili 200 sekundi. Kada se taj podatak usporedi sa vremenima izvršavanja ispadne da se s pomoću TBB biblioteke ostvaruje ubrzanje od 40%.



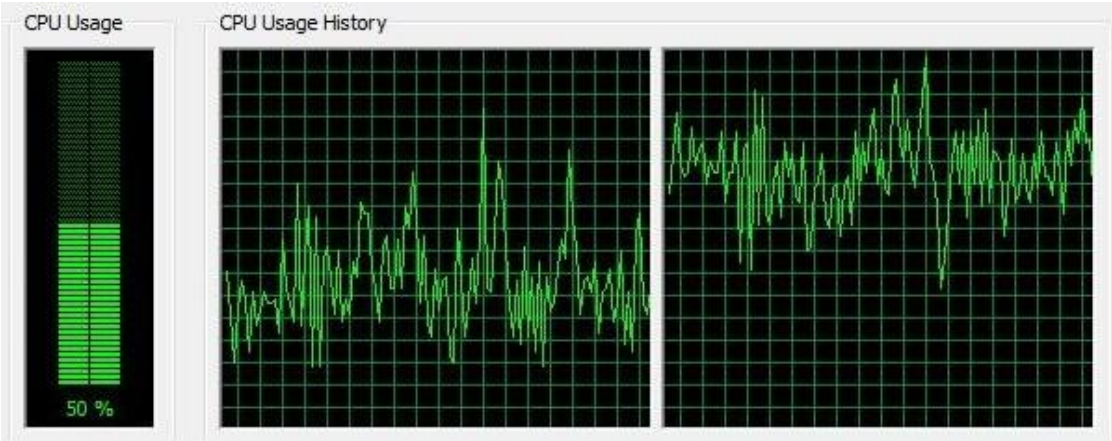

**Slika 4.1.2 usporedba izvođenja TBB i serijske izvedbe za veće matrice**

Kada se usporede vremenski rezultati izvođenja TBB rješenja i rješenje sa ručnim stvaranjem dretvi, vide se nešto drugačiji rezultat. Na slici 4.1.3 prikazano je kako dimenzije matrice utječu na vremenu izvođenja između dvije metode paralelizacije množenja matrica. Može se primijetiti da je razlika u vremenima manja nego sa usporedbom TBB verzije i slijedne verzije rješenja. Ubrzanje korištenjem TBB biblioteke iznosi od 10-20%, što nije loše sa obzirom da obe metode rješavaju problem paralelno. Ta prednost u vremenu je stvorena zbog činjenice da korištenjem postavki poput *affinity_partitioner*, TBB bolje prilagođava izvršavanje programskog rješenja sa višejezgrenim procesorom.



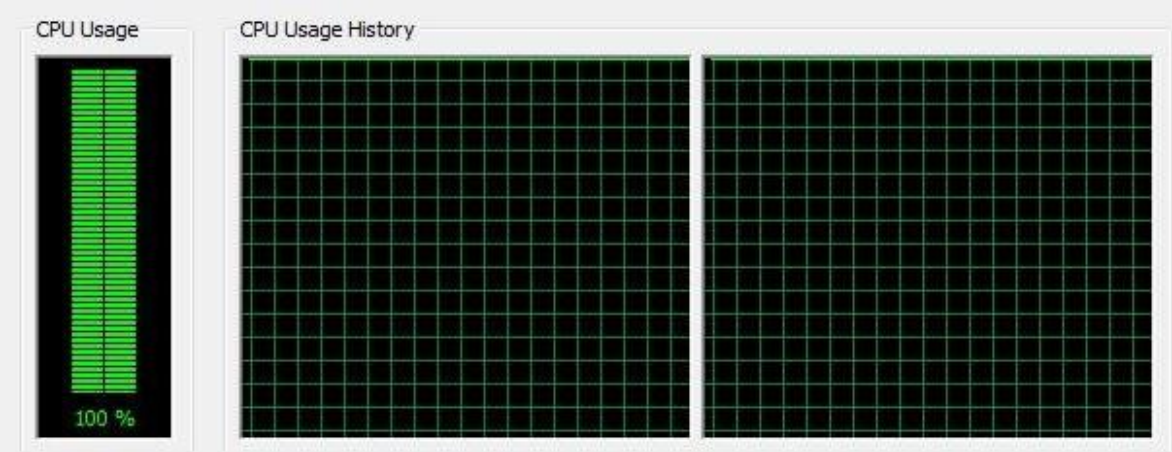

**Slika 4.1.3 usporedba izvođenja TBB i paralelne izvedbe**

Razlika u izvođenju programa sa TBB vidljivo je i u aplikaciji *Task Manager* koja je ugrađena u sklopu Windows operacijskog sustava. Na slici 4.1.4 prikazan je ispis toga programa tijekom izvođenja programa za množenje matrica koji ne koristi paralelizaciju. Može se primijetiti da je iskorištenost procesora oko 50%. Također se može primijetiti da jezgre nisu podjednako iskorištene.




**Slika 4.1.4 prikaz rada procesora tijekom izvršavanja programa bez TBB**

Na slici 4.1.5 prikazan je ispis programa Task Manager tijekom izvršavanja programa za množenje matrica koji koristi TBB biblioteku. Iz slike se vidi da je iskorištenost procesora 100%, što znači da su obe jezgre potpune iskorištene.



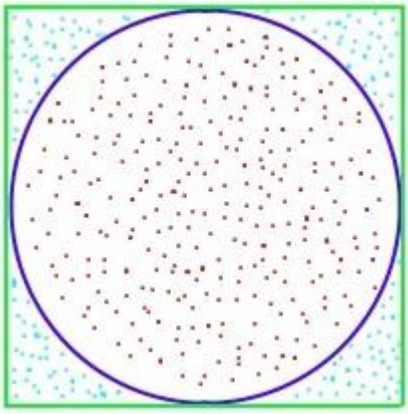

**Slika 4.1.5 prikaz rada procesora tijekom izvršavanja programa sa TBB**

## 4.2 Generiranje vrijednosti π

Za proračun procjene matematičke konstante π može se koristiti Monte Carlo metoda. Ta metoda se osniva na statistici gdje se pretpostavlja da uz poznatu vjerojatnost može se očekivati da će se ostvariti neki događaj uz određene uvjete. Ovi uvjeti se u biti simuliraju na računalu kao slučajno generirane vrijednosti koje se generiraju više puta. Osnovna ideja u korištenju ove metode za aproksimaciju vrijednosti π je da se generiraju vrijednosti između -1 i 1. Potom se gleda da li su te vrijednosti unutar jedinične kružnice. Na slici 4.2.1, prikazana je grafička reprezentacija jedinične kružnice sa slučajno generiranim točkama.

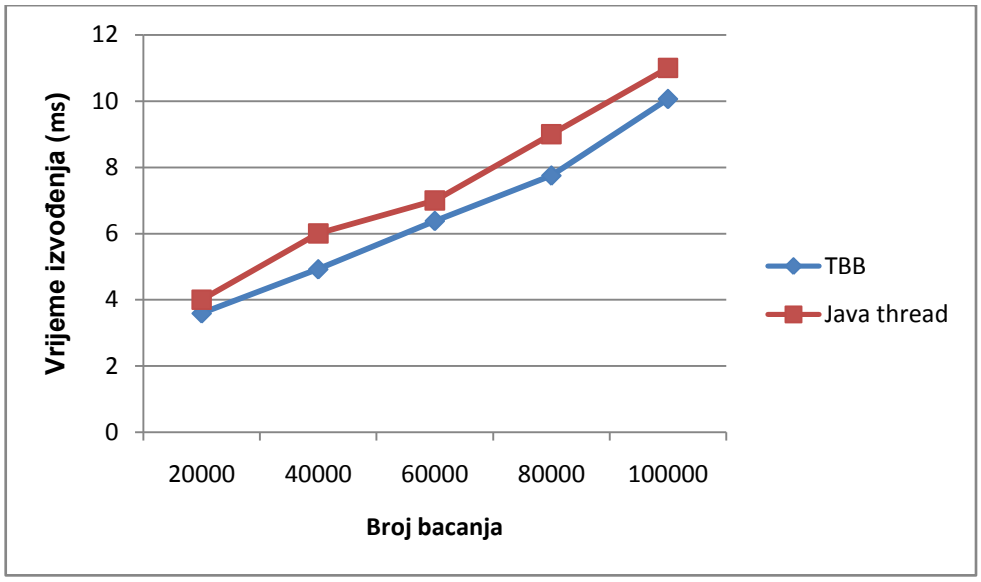
Vrijednost koja je približno jednaka π dobije se tako da se podijeli broj točaka u jediničnoj kružnici sa ukupnim brojem točaka unutar kružnice i kvadrata, te se taj broj pomnoži sa 4. Implementacija sa TBB bibliotekom koristi algoritam *parallel_for*, gdje je gornja granica iteracije definirana sa strane korisnika. Unutar metode *operator(),* nalazi se generiranje slučajne točke te ispitivanje da li je točka unutar jedinične kružnice. Rješenje koje ručno stvara dretve slijedi isti algoritam, ali posjeduje razliku u tome što se pojedinoj dretvi predaje posao generiranja 100

ili 1000 točaka (broj ovisan o odabiru korisnika). Ovo se moralo napraviti zbog toga što je generiranje nove točke prejednostavan posao, te bi se stvarali veliki gubici u performansama programa ako bi se jednoj dretvi dodijelio posao generiranje samo jedne točke.



**Slika 4.2.1 prikaz kružnice sa slučajno generiranim točkama**

Na slici 4.2.2 prikazana je usporedba izvođenja Monte Carlo simulacije aproksimacije vrijednosti π realizirano sa TBB bibliotekom te sa realizacijom ručnog stvaranje dretvi. Iz prikaza se vidi da ne postoje neke velike razlike, ali da je TBB izvedba ipak brža i to za otprilike 9-14%. Krivac ovakvom rezultat je to što su obje implementacije paralelne te da je sami posao koji se obavlja unutar svakog programa dosta jednostavan te prednosti koji su ugrađeni u TBB biblioteci ne dolaze do izražaja.




**Slika 4.2.2 usporedba izvođenja TBB i paralelne izvedbe**

## 4.3 Dijkstra

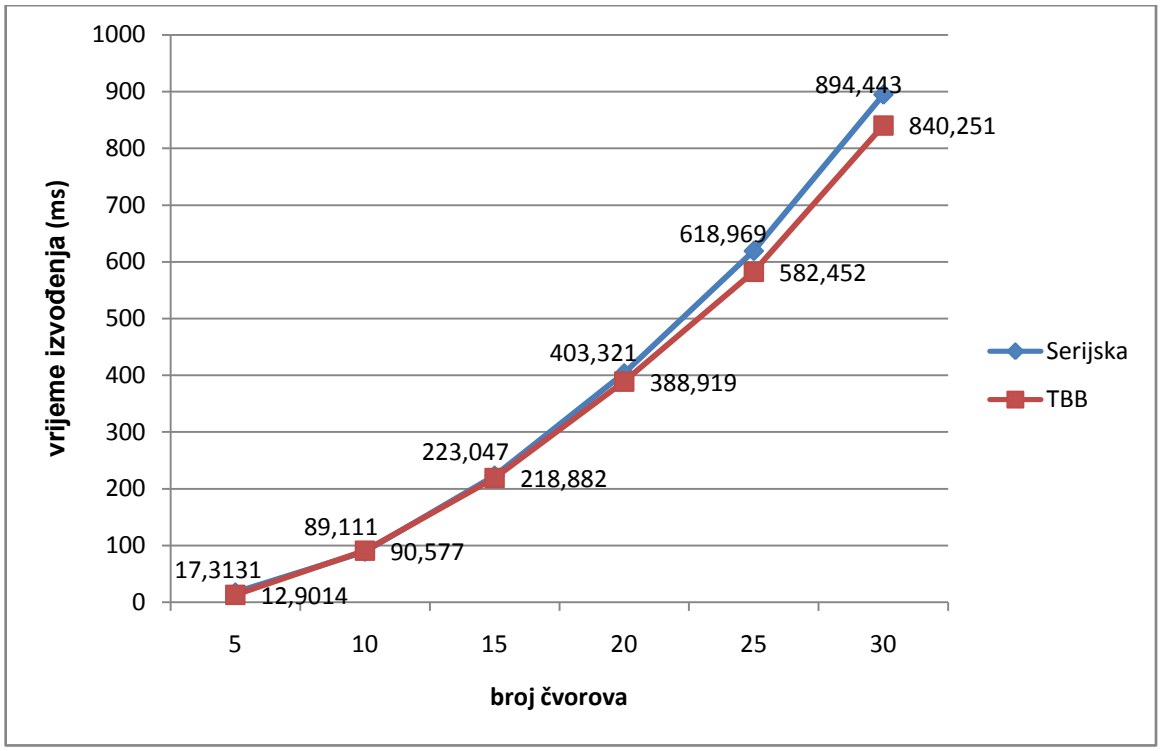
Pronalazak najkraćeg puta između dviju točaka je problem koji ima veliki broj primjena. Za pronalazak najkraćeg puta koristi se Dijkstrin algoritam. Osnovni pojam Dijkstrinog algoritma je da svaki vrh koji se nalazi u najkraćem putu može saznati svojeg neposrednog prethodnika te tako rekonstruirati cijeli put od polazišta pa do sebe. Pseudokod algoritma izgleda ovako:

```
Dijkstra (polazni čvor, odredišni čvor)
     za sve čvorove v→ udaljenost(polazni čvor
          ,v)=beskonačno;
     udaljenost(polazni čvor , polazni čvor)=0;
     unos svih čvorova u polje zaProvjeru;
     dok postoji čvor u polju zaProvjeru
          x =čvor iz zaProvjeru sa najmanjom
               udaljenost(polazni  čvor,x);
          ako x == odredišni čvor
               kraj→pronađen je najkraći put;
          inače izvadi x iz polja zaProvjeru;
          za sve čvorove y u zaProvjeru i koji su susjedni sa
               x
               novaudaljenost(polazni čvor ,y)=
               udaljenost(polazni čvor,x)+težina(x,y);
               ako je novaudaljenost(polazni
```

```
čvor,y) < udaljenost(polazni čvor,y)
udaljenost(polazni čvor,y)
     = novaudaljenost(polazni čvor,y);
prethodnik (y) = x;
```

U rješenju sa TBB bibliotekom umjesto da se pronalazi samo najkraći put od jednog čvora do drugog implementirano je da se pronalaze najkraći putevi između svih čvorova. To rješenje je izabrano da bi se moglo primijeniti paralelizacija na samom zadatku. Iz TBB biblioteke korišten je *parallel_for* algoritam kojemu je iteracijski prostor postavljen od 0 do broj čvorova unutar grafa. Svakom se bloku dodjeljuje posao pronalaska najkraćeg puta između jednog čvora i svih ostalih čvorova unutar grafa. Cijeli graf je prikazan matricom susjedstva koja je se nalazi unutar jedne tekstualne datoteke.

Unutar metode *operator()* koja pripada klasi *Dijkstra* nalazi se cijeli algoritam za određivanje najkraćeg puta te se rezultanti put sprema u polje. Na slici 4.3.1 prikazana je usporedba izvođenja programa koji koristi TBB biblioteku te onog bez TBB biblioteke. Sa slike se vidi da za grafove sa 5 do 20 čvorova nema razlike u vremena izvođenja. Tek se sa 30 i više čvorova osjeti razlika te sa daljnjim povećanjem broja čvorova stvara se veća prednost.




**Slika 4.3.1 usporedba izvođenja TBB i serijske izvedbe Dijsktrinog algoritma**

# 5. Zaključak

Sa dolaskom višejezgrenih procesora u standardna računala, postojeća programska rješenja ne uspijevaju iskoristiti sva poboljšanja u performansa koje nude ti procesori. Da bi pomogli u rješavanju ovog problema, Intel je stvorio rješenje koje se zove Intel Threading Building Blocks (TBB). To rješenje je zapravo C++ biblioteka koja omogućava programskim inženjerima korištenje paralelizacije unutar svojih programa bez potrebe za ručno stvaranje i sinkronizaciju dretvi. U ovom radu glavni TBB algoritmi i podatkovne strukture su objašnjeni i diskutirani. Neki od tih TBB algoritama su korišteni u stvaranje paralelnih programskih rješenja za zadatke kao što su množenje matrica i pronalazak najkraćeg puta. Na kraju da bi se ispitala prednost korištenja TBB biblioteke, uspoređeni su rezultati izvođenja rješenja koje koriste TBB i onih rješenja koje koriste ručno upravljanje sa dretvama. Ovisno o problemu koji se rješava te načinu izvedbe rješenja, ubrzanje rada programskih rješenja je različita. To ubrzanje koje se postiže može biti od 0-40% kao što je pokazano u primjerima, pa i više. Glavni krivac za ovakav raspon ubrzanja je činjenica da određeni problemi nisu dovoljno procesorski zahtjevni da bi se prednosti TBB biblioteke u potpunosti iskoristili. Zbog velikog broja algoritama i struktura podataka koje posjeduje, Threading Building Blocks biblioteka daje korisniku priliku da sa lakoćom svoja rješenja pretvori u paralelizirana programska rješenja.

# 6. Literatura


[1] Reinders, James. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. Sebastopol: O'Reilly Media, Inc., 2007.

[2] Intel Threading Building Blocks Tutorial, 21.08.2008, *Intel Threading Building Blocks 2.1 for Open Source*, http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Tutorial%20(Open%20Source).pdf, 17.09.2008.

[3] Intel Threading Building Blocks Reference Manual, 31.03.2008, *Intel Threading Building Blocks 2.1 for Open Source,* http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference%20Manual%20(Open%20Source).pdf, 17.09.2008.

[4] Robison, Arch. Generic Parallel Algorithms in Intel Threading Building Blocks, 1.08.2008, *Intel Threading Building Blocks 2.1 for Open Source Blocks*, http://www.threadingbuildingblocks.org/uploads/82/127/1.0/GenericParallelAlgorithms.pdf, 20.09.2008.

[5] Voss, Michael. What's New in Threading Building Blocks, 1.08.2008, *Intel Threading Building Blocks 2.1 for Open Source,* http://www.threadingbuildingblocks.org/uploads/82/126/1.0/OSCON_Voss_2008_final.ppt, 29.09.2008.

[6] Robison, Arch., Voss, Michael., Kukanov, Alexey. Optimization via Reflection on Work Stealing in TBB, 21.04.2008, *Intel Threading Building Blocks 2.1 for Open Source*, http://www.threadingbuildingblocks.org/uploads/82/106/1.0/HIPS%20TBB%20Reflection%20Web.ppt, 20.09.2008.

# 7. Dodatak

**Naslov završnog rada**: Primjena i ispitivanje učinkovitosti biblioteke “Intel Threading Building Blocks”


## Sažetak

Sa dolaskom višejezgrenih procesora u standardna računala, postojeća programska rješenja ne uspijevaju iskoristiti sva poboljšanja u performansa koje nude ti procesori. Da bi pomogli u rješavanju ovog problema, Intel je stvorio rješenje koje se zove Intel Threading Building Blocks (TBB). To rješenje je zapravo C++ biblioteka koja omogućava programskim inženjerima korištenje paralelizacije unutar svojih programa bez potrebe za ručno stvaranje i sinkronizaciju dretvi. U ovom radu glavni TBB algoritmi i podatkovne strukture su objašnjeni i diskutirani. Neki od tih TBB algoritama su korišteni u stvaranje paralelnih programskih rješenja za zadatke kao što su množenje matrica i pronalazak najkraćeg puta. Na kraju da bi se ispitala prednost korištenja TBB biblioteke, uspoređeni su rezultati izvođenja rješenja koje koriste TBB i onih rješenja koje koriste ručno upravljanje sa dretvama.



**Ključne riječi**: TBB, Threading Building Blocks, višedretvenost, dretve, zadatak, višejezgreni procesori, C++


**Title**: Application and analysis of the effectiveness of the “Intel Threading Building Blocks” library


## Summary

With the advent of multi-core processors in mainstream computers, existing software hasn’t been able to fully take advantage of the performance enhancements that such processors provide. To help with this problem Intel has created a solution called Intel Threading Building Blocks (TBB), which is a C++ library that enables software developers to use parallelization in their programs without the need for manual creation and synchronization of threads. In this report

the main TBB algorithms and data structures are explained and discussed. Some of those TBB algorithms are used to create parallel software solutions to problems such as matrix multiplication and the shortest path problem. At the end to get a better picture of the enhancements of TBB, performance results of TBB solutions are compared with the results of solutions that use manual management of threads.

**Keywords**: TBB, Threading Building Blocks, multithreading, threads, task, multi-core processors, C++