SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 16

PODRŠKA JEZIKA C# VIŠEPROCESORSKOM PROGRAMIRANJU

Anthony Lipovac

Zagreb, lipanj 2010.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 16

PODRŠKA JEZIKA C# VIŠEPROCESORSKOM PROGRAMIRANJU

Anthony Lipovac

Zagreb, lipanj 2010.

Sadržaj

Uvod 1

1. Višedretvenost 2

   1.1. Osnovno o višedretvenosti 2

   1.2. Prednosti višedretvenosti 3

   1.3. Nedostaci višedretvenosti 3

   1.4. Primjena višedretvenosti 4

2. Jezik C# 6

3. .NET Framework 7

4. Izravno korištenje dretvi (klasa *Thread*) 10

   4.1. Stanja dretvi 10

   4.2. Prioriteti dretvi 12

   4.3. Stvaranje i pokretanje dretvi 13

   4.4. Interakcije između dretvi 14

   4.5. Sinkronizacija dretvi 14

5. Pre-alokacija radnih dretvi (klasa *ThreadPool*) 17

6. Proširenja programskog jezika (klasa *Parallel Extensions*) 19

   6.1. Konstruktori za paralelizaciju podataka 20

      6.1.1. Parallel.For 20

      6.1.2. Parallel.ForEach 21

   6.2. Konstruktori za paralelizaciju zadaća 21

      6.2.1. Parallel.Invoke 21

7. Množenje matrica 23

   7.1. Algoritam množenja matrica 23

   7.2. Implementacija algoritma 24

      7.2.1. Izvedba izravnog korištenja dretvi (klasa *Thread*) 25

7.2.2. Izvedba pre-alokacijom radnih dretvi (klasa *ThreadPool*) ... 25
7.2.3. Izvedba korištenja proširenja programskog jezika ... 26
7.3. Trajanje izvođenja implementacija ... 27
8. Quicksort ... 29
8.1. Algoritam sortiranja Quicksort ... 29
8.2. Implementacija algoritma ... 29
8.2.1. Izvedba izravnog korištenja dretvi (klasa *Thread*) ... 30
8.2.2. Izvedba pre-alokacijom radnih dretvi (klasa *ThreadPool*) ... 30
8.2.3. Izvedba korištenja proširenja programskog jezika ... 31
8.3. Trajanje izvođenja implementacija ... 32
9. Problem trgovačkog putnika sa simuliranim kaljenjem ... 33
9.1. Algoritam rješenja problema trgovačkoga putnika sa simuliranim kaljenjem ... 33
9.2. Implementacija algoritma ... 35
9.2.1. Izvedba izravnog korištenja dretvi (klasa *Thread*) ... 35
9.2.2. Izvedba pre-alokacijom radnih dretvi (klasa *ThreadPool*) ... 36
9.2.3. Izvedba korištenja proširenja programskog jezika ... 36
9.3. Trajanje izvođenja implementacija ... 37
10. Analiza rezultata ... 38
10.1. Analiza ubrzanja paralelnih implementacija ... 38
10.2. Usporedba rezultata paralelnih implementacija i biblioteke TBB ... 41
Zaključak ... 43
Sažetak ... 45
Ključne riječi ... 45
Summary ... 46
Keywords ... 46

Uvod

U prošlim desetljećima, višeprocesorski sustavi mogli su se samo naći u primjeni u industriji i akademskoj zajednici. Dolaskom višejezgrenih procesora namijenjenih za osobna računala, višeprocesorski sustavi danas se nalaze u bezbroj domova i institucija. Iako je pojava višeprocesorskih sustava u svakodnevnome životu sve češća, njihove mogućnosti se u potpunosti ne iskorištavaju. Dosadašnji razvoj programskih rješenja poput poslovnih ili zabavnih aplikacija fokusiran je na radu s jednoprocesorskim ili jednojezgrenim sustavima. Uz nedostatno sklopovlje, dodatni razlog ove orijentacije leži u činjenici da je razvoj programske podrške za ovu vrstu sustava najčešće dosta zahtjevan. Za bolje iskorištavanje mogućnosti višeprocesorskih sustava potrebno je uvesti određenu mjeru paralelizacije u programski kôd što zahtijeva drukčiji način razmišljanja prilikom razvoja programskih rješenja.

U ovom radu obrazlagat će se i analizirati mogućnosti izrade programskih rješenja u jeziku C# koje podržavaju rad višeprocesorskih sustava. Predstavit će se razni mehanizmi dostupni programskom inženjeru za lakšu i efikasniju izradu paraleliziranoga kôda.

U prvom poglavlju pojasnit će se pojam višedretvenosti koji je osnovni pojam višeprocesorskih sustava. U poglavljima koji slijede predstavit će se i objasniti osnove jezika C# i *.NET Framework* kao i njihove mogućnosti za prilagođavanje programskih rješenja za rad s višeprocesorskim sustavima. Koristeći podršku jezika C# za višeprocesorske sustave, u zadnjim poglavljima predstavit će se primjena tih mehanizama u rješavanju nekoliko reprezentativnih problema paralelnoga programiranja. Na kraju analizirat će se složenost i efikasnost primijenjenih mehanizama u usporedbi s ostalim mogućnostima paralelnoga programiranja.

Sva programska rješenja pisana su u jeziku C# te je korištena razvojna okolina *Microsoft Visual Studio 2010* s *.NET Framework 4.0*. Implementacija i izvršavanje programskih rješenja obavljena je na operacijskom sustavu *Windows 7 Professional* na računalu s dvojezgrenim procesorom Intel Core i5 M520 2,40 GHz i 4,0 GB memorije.

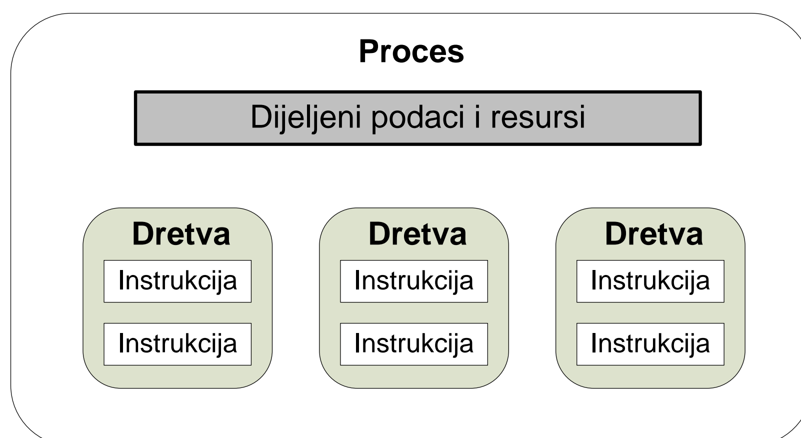
1. Višedretvenost

Pojam višedretvenost je jedan od osnovnih pojmova višeprocesorskih sustava te je ključan za učinkovitije korištenje mogućnosti tih sustava.

1.1. Osnovno o višedretvenosti

Prilikom pokretanja aplikacije, sva memorija, programski kôd i ostali resursi koji su potrebni za izvršavanje aplikacije izdvojeni su u jedinicu koja se zove proces (engl. *process*). Sam računalni program je skup naredbi, a proces predstavlja samo izvršavanje tih naredbi. Pojedina aplikacija može imati više procesa, a memorija i resursi koji su dodijeljeni jednom procesu, izolirani su od ostalih procesa. Svaki proces sadržava jednu ili više dretvi (engl. *thread*). U računalnoj znanosti dretva ima dva značenja. Prvo značenje predstavlja slijed izvršavanja za procesor računala i zove se dretva upravljanja. Drugo značenje predstavlja odnos u višedretvenom sustavu u kojem dretva predstavlja najmanju cjelinu koja se izvodi neovisno.

Na slici 1.1. prikazan je odnos proces-dretva. Procesi inače sadrže veću količinu podataka o stanju izvršavanja dok dretve unutar jednoga procesa dijele podatke o stanju, memoriju i druge resurse. Dretve također dijele adresni prostor dok svaki proces ima vlastiti adresni prostor. Pojam višedretvenost (engl. *multithreading*) predstavlja koncept upravljanja više dretvi nad programom. Program nad kojim je primijenjen pojam višedretvenost posjeduje više nezavisnih tokova izvođenja gdje svaki tok samostalno upotrebljava instrukcije programa.



Slika 1.1. Prikaz odnosa procesa i dretvi

1.2. Prednosti višedretvenosti

Primjena koncepta višedretvenosti u programskom kôdu ima sljedeće prednosti:

- Za računala koje posjeduju višejezgrene procesore ili višeprocesorske sustave, moguće je iskoristiti dodatne resurse sustave da bi se istovremeno izvodilo više dretvi, te se time skraćuje vrijeme ukupnog izvođenja programa.
- U slučaju da više dretvi rade na istom skupu podataka, moguće je taj cijeli skup podataka prebaciti u priručnu memoriju. Ovaj pristup dovodi do učinkovitijega korištenja priručne memorije procesora.
- Može se dogoditi da na jednoprocesorskom ili jednojezgrenom sustavu pojedina dretva u potpunosti ne iskorištava dostupne kapacitete procesora. Koristeći višedretvenost, preostali procesorski resursi se mogu dodijeliti drugoj dretvi što dovodi do ubrzanja rada programa.
- Korištenjem višedretvenosti, moguće je osigurati osjetljivost aplikacije na unos podataka od strane korisnika tijekom izvršavanja dijela programa. Sporednoj dretvi prepušta se izvršavanje dijela koji zahtjeva obradu dok glavna izvođačka dretva može ostati slobodna te čekati unos od strane korisnika. Ponekad je i izvedba jednostavnija, odvajanjem nevezanih radnji u zasebne dretve.

1.3. Nedostaci višedretvenosti

Primjena koncepta višedretvenosti u programskom kôdu ima sljedeće nedostatke:

- Postoje aplikacije i algoritmi koji se ne mogu lako ili uopće paralelizirati, stoga se na njima ne može primijeniti višedretvenost.
- Sama paralelizacija i primjena višedretvenosti na aplikacije je složena i teška. Prilikom programiranja, programer se mora prilagoditi na drukčiji način razmišljanja o tijeku izvođenja programa. Također se mora obratiti pažnja na moguće poteškoće s dijeljenjem podataka te mogućnosti stvaranja zastoja.
- S pretjeranom paralelizacijom programskoga kôda moguće je da umjesto poboljšanja dođe i do lošijih performansi u usporedbi sa serijskom implementacijom. U slučaju da je paralelizirani dio kôda prejednostavan, može se dogoditi da samo stvaranje dretvi traje više vremena od samoga posla koji se treba obaviti.

- Nepravilna sinkronizaciju i zaključavanje može imati za posljedicu da pojedine dretve čekaju predugo da dretva oslobodi korištenje određenoga resursa te se sama paralelizacija odsječka vremenski ne isplati.

1.4. Primjena višedretvenosti

Na sustavu koji posjeduje jedan procesor višedretvenost se primjenjuje tako da procesor slijedno izvodi dretve izmjenjujući pravo na izvršavanje među dretvama. U slučaju da se u sklopu računala nalazi višeprocesorski ili višejezgreni sustav, dopušta se istovremeno izvršavanje dretvi gdje će svaki pojedini procesor ili jezgra izvršavati po jednu dretvu. Višeprocesorski sustavi posjeduju dva ili više procesora i imaju mogućnost raspodjelu zadaća između procesora. Višejezgreni procesori su varijacija višeprocesorskih sustava s razlikom da se umjesto korištenja više odvojenih procesora koristi samo jedan procesor s više jezgri. Inače, dijeljenje priručne memorije između jezgri i ostale karakteristike ovise o izvedbi i proizvođaču procesora.

Jedna od poznatijih tehnologija koja se koristi za primjenu koncepta višedretvenosti zove se *Hyper-Threading*. Sama tehnologija je razvijena od strane vodećega proizvođača procesora Intel te se ujedno koristi na velikom broju njihovih procesora. *Hyper-Threading* omogućava da se na jednoj procesorskoj jezgri istovremeno izvršavaju dvije dretve. Iako fizički postoji samo jedan procesor, s *Hyper-Threading* tehnologijom procesor se u operacijskome sustavu prepoznaje kao dva zasebna procesora. Zbog ove činjenice, procesor s jednom jezgrom može istovremeno raditi s dvije dretve što, ako se pravilno iskoristi, može donekle ubrzati izvršavanje aplikacije, istovremenim korištenjem raznih elemenata procesora od strane dvije dretve.

Da bi se olakšala primjena višedretvenosti na programskom kôdu, postoje mnoga rješenja za paralelno programiranje ovisno o programskom jeziku. U jeziku C++ uz ručno kreiranje dretvi moguće je koristiti biblioteke poput *Intel Threading Building Blocks (TBB)* i *OpenMP-a*.

Umjesto ručnoga stvaranja i sinkroniziranja dretvi koji se koriste tijekom direktnoga rada s njima, u biblioteci *Threading Building Blocks* pristup paralelizaciji je nešto drukčiji. *TBB* pretvara operacije koje se trebaju paralelizirati u zadaće (engl. *tasks*) koje potom raspoređuje po jezgrama procesora. Korištenjem zadaća potpuno se izbjegava

direktni rad korisnika s dretvama, te se korisnik može koncentrirati na paralelizaciju aplikacije.

Za razliku od biblioteke *Intel TBB*, *OpenMP* implementira pojam višedretvenosti tako da zadaće izravno dijeli po dretvama. Sučelje *OpenMP* je u osnovi stvoreno za računalne arhitekture sa zajedničkom dijeljenom memorijom. U ovoj vrsti arhitekture, dretve koje se nalaze na pojedinim procesorima ili jezgrama imaju mogućnost korištenja dijeljenih podataka kada im je potrebno. Zbog ove činjenice, moguće je primijeniti *OpenMP* sučelje nad raznim vrstama računalnih arhitektura poput višeprocesorskih ili višejezgrenih arhitektura.

Za razliku od C++, programski jezik C# posjeduje više ugrađenih biblioteka koje omogućavaju primjenu višedretvenosti. Ove mogućnosti će se detaljnije predstaviti u sljedećim poglavljima.

2. Jezik C#

Programski jezik C# je višeparadigmni programski jezik koji je predstavljen 2001. od strane Microsoftovoga tima pod vodstvom Andersa Hejlsberga. Sam jezik je namijenjen kao novo rješenje za objektno orijentirano programiranje u Microsoftovoj razvojnoj platformi *.NET Framework,* a slijedi iz projekta razvoja komponente *Common Language Runtime* (*CLR*). C# je izvorno izveden iz programskih jezika C i C++ dok mu je sintaksa dosta slična C++ i Javi jer koristi istu blok strukturu s vitičastim zagradama i iste ključne riječi.

U jeziku C# sve metode i njihovi članovi moraju biti definirane unutar klasa iz čega proizlazi da ne postoje globalne varijable i globalne funkcije. C# podržava i operacije *boxing* i *unboxing* koje omogućavaju pretvaranje vrijednosti od nekog definiranoga vrijednosnog tipa u vrijednost odgovarajućega referentnoga tipa i suprotno. C# omogućava potpunu podršku za klase i objektno orijentirano programiranje što uključuje nasljeđivanje sučelja i implementacija, virtualne funkcije i preopterećivanje operatora. Jezik ima i ugrađenu podršku za automatsko generiranje dokumentacije koji je sličan sustavu *Javadocs* koji se koristi u jeziku *Java* s razlikom da je osnovan na jeziku *XML.*

U sklopu jezika C# postoji i mogućnost stvaranja pokazivača i direktan pristup memoriji, ali je sam jezik smišljen tako da je u većini vremena njihovo korištenje potpuno nepotrebno. U slučaju da je njihovo korištenje potrebno, blok kôda u kojem se koriste označi se s oznakom *unsafe.* Za program s takvim blokovima kôda potrebno je posebno dopuštenje za pokretanje, inače nije moguće osloboditi memoriju koja se smatra upravljanom s obzirom da je ugrađeno automatsko oslobađanje memorije koja oslobađa programera od odgovornosti oslobađanja memorije koja se više ne koristi, sprečavajući stvaranje problema s curenjem memorije.

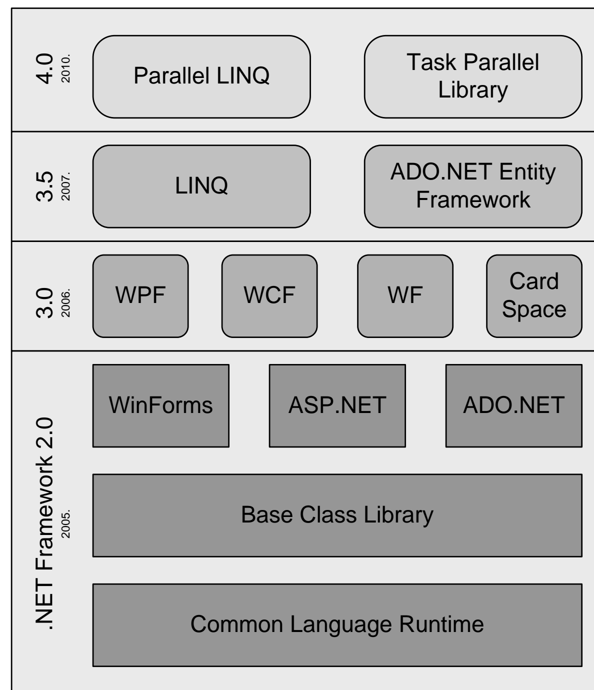
Dodatna značajka C# jezika je mogućnost prevođenja programskoga kôda u izvršnu datoteku ili u biblioteku .NET komponenti koje mogu biti pozvane u sklopu drugoga kôda, slično kao i *ActiveX* kontrole. Pošto je C# dio *.NET Frameworka* ima pristup cijeloj .NET biblioteci osnovnih klasa te i jednostavan pristup *Windows API*-ju. C# je moguće također koristiti za pisanje *ASP.NET* dinamičkih web stranica i XML web servisa.

3. .NET Framework

Microsoftov *.NET Framework* je sastavna *Windows* komponenta koja podržava izgradnju i omogućava rad za sljedeću generaciju aplikacija i *XML* Web servisa. Platforma je predstavljena 2002. i može se ugraditi na računala s ugrađenim operacijskim sustavom *Microsoft Windows*. Zamišljeni ciljevi za *.NET Framework* su sljedeći (Microsoft, 2010):

- Osigurati objektno orijentiranu programsku okolinu neovisno o tome sprema li se objektni kôd i izvršava lokalno, udaljeno ili lokalno, ali je distribuirano po Internetu.
- Osigurati okolinu za izvršavanje kôda koji minimizira konflikte koji nastaju prilikom razvoja softvera i novih verzija.
- Osigurati okolinu za izvršavanje programskoga kôda koji podržava i promiče sigurno izvršavanje programskoga kôda što uključuje i kôd koji je pisan od strane nepoznate ili treće strane.
- Osigurati okolinu za izvršavanje programskoga kôda koja otklanja poteškoće s performansama skriptnih i interpretiranih okolina.
- Osigurati dosljedan doživljaj radne okoline za razvojnoga programera tijekom razvoja različitih oblika aplikacija što uključuje razvoj aplikacija za *Windows* i *Web*.
- Osigurati da je sva komunikacija u skladu s industrijskim standardima što omogućava integraciju *.NET* kôda s drugim programskim kôdom.

Na slici 3.1. prikazani su sastavni dijelovi *.NET Frameworka* i vremenska podjela dolaska novih inačica. Dvije glavne komponente *.NET Frameworka* su *Common Language Runtime (CLR)* i biblioteka klasa *.NET Frameworka*.



Slika 3.1. Prikaz dijelova .NET Frameworka (Saunders, 2008)

CLR je temelj *.NET Frameworka* jer predstavlja agenta koji upravlja programskim kôdom tijekom izvršavanja. *CLR* je u biti virtualno računalo na kojemu se obavlja izvršavanje aplikacije te predstavlja sloj između aplikacije i operacijskoga sustava. Sama komponenta osigurava osnovne usluge poput upravljanja memorijom, upravljanja dretvama i verifikaciju sigurnosti programskoga kôda. Sve ove karakteristike opisuju oblik programskoga kôda koji se može nazvati i upravljani kôd (engl. *managed code*). Upravljane programske komponente posjeduju različite razine povjerenja koje ovise o faktorima poput podrijetla, tj. nalaze li se komponente na Internetu ili na računalu. Ovisno o razini povjerenja određuje se ima li određena komponenta pravo na operacije poput operacija pristupa datotekama, operacije pristupa registru operacijskoga sustava ili druge osjetljive operacije.

U sklopu *CLR-a* postoje dodatne komponente poput infrastrukture *Common Type System* (*CTS*) koja potiče robusnost programskoga kôda. Također postoji komponenta koja obavlja automatsko upravljanje memorijom što rješava dva najčešća problema aplikacija- curenje memorije i neispravne memorijske reference.

Jedna dodatna karakteristika *CLR*-a je i funkcija *just in time compiling* koja omogućava da se upravljani kôd ne mora interpretirati, već se može izvršavati u izvornom strojnom jeziku sustava. Uz to upravitelj memorije uklanja mogućnost stvaranja

fragmentirane memorije, povećavajući lokalnost referenciranja u memoriji. Kombinacija ovih pristupa omogućava poboljšane performanse razvijene aplikacije.

Drugi dio *.NET Framework* je biblioteka klasa koja je opsežan skup tipova za višekratnu upotrebu (engl. *reusable types*) koje su usko integrirane s *CLR-om*. Sama biblioteka je objektno orijentirana što omogućava implementaciju funkcionalnosti i u vlastitome programskom kôdu. U sklopu biblioteke nalazi se raznovrstan skup tipova koji omogućavaju obavljanje programskih zadataka poput rada sa podatkovnim tipom *string*, podatkovnim skupovima, povezivanja s bazama podataka i rad s datotekama. Uz klase za osnovne zadatke, postoje i biblioteke koje omogućavaju razvoj aplikacija za posebne namjene. Primjeri aplikacija koje se mogu razviti s pomoću *.NET Frameworka* su konzolne aplikacije, *Windows GUI* aplikacije, *Windows Presentation Foundation*(*WPF*) aplikacije, *ASP.NET* aplikacije, Web servisi, servisno orijentirane aplikacije koje koriste *Windows Communication Foundation (WCF)* i aplikacije koje prate tijek rada koristeći *Windows Workflow Foundation (WF)* (Microsoft, 2010) .

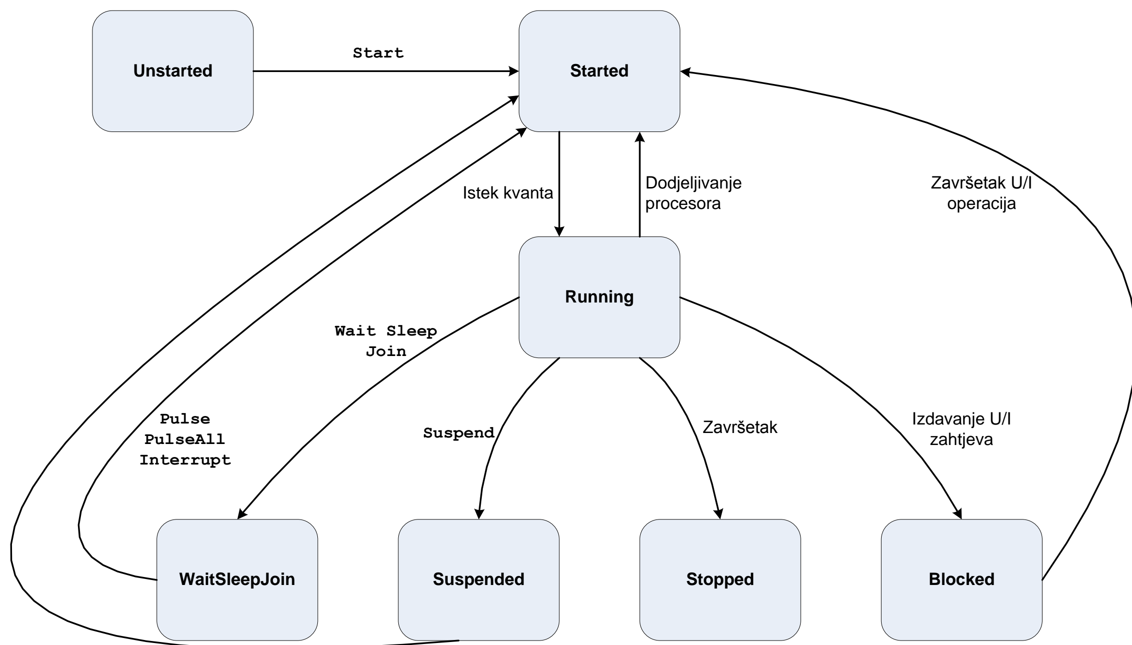
4. Izravno korištenje dretvi (klasa *Thread*)

U sklopu *.NET Frameworka* kao dio imeničkoga prostora System.*Threading* nalazi se klasa *Thread* koja se koristi za izravno korištenje dretvi. Klasa se sastoji od više metoda s kojima se može upravljati s dretvama što uključuje stvaranje dretvi, pokretanje dretvi, sinkronizacija dretvi te interakcija između dretvi.

4.1. Stanja dretvi

Svaka se dretva u bilo kojem trenutku nalazi u jednom od nekoliko stanja (engl. *thread states*). Na slici 4.1. prikazan je dijagram stanja dretve. Nova dretva počinje svoj životni ciklus u stanju *Unstarted* te u njemu ostaje sve dok program ne pozove metodu *Start* iz klase *Thread* te dretva potom prelazi u stanje *Started*. Nakon što uđe u stanje *Started,* kontrola izvođenja se ponovno vraća pozivajućoj dretvi te je dopušteno istovremeno izvođenje dretve koja je pozvala *Start*, nove stvorene dretve i svih ostalih postojećih dretvi.

Dretva koja ima najviši prioritet prelazi u stanje *Running* u trenutku kada joj operacijski sustav dodijeli pristup procesoru ili jezgri. Kada dretva prvi put prelazi iz stanja *Started* u *Running*, dretva izvršava vlastiti delegat *ThreadStart* koji određuje koji će se poslovi obavljati prilikom života dretve. Dretva koja se nalazi u stanju *Running* ulazi u stanje *Stopped* tek kad mu vlastiti delegat *ThreadStart* završi.



Slika 4.1. Prikaz dijagrama stanja dretve (Deitel et al., 2002)

U slučaju potrebe da se dretva zaustavi, može se pozvati metoda *Abort* iz klase *Thread*. Sama metoda će aktivirati iznimku *ThreadAbortException* tijekom rada dretve što će prouzročiti zaustavljanje dretve. Kada se dretva nalazi u stanju *Stopped* te kada više ne postoje reference pripadajućem *Thread* objektu, upravitelj memorije briše dretvu iz memorije. Dretva može ući u stanje *Blocked* u slučaju da izda ulazno/izlazni zahtjev. Tada operacijski sustav blokira izvođenje dretve sve dok ne završi ulaznu/izlaznu operaciju na koju dretva čeka. Nakon što se taj uvjet ispuni, dretva se vraća u stanje *Started* te se nastavlja izvođenje. Dretva koja je u stanju *Blocked* ne može koristiti procesor čak i ako postoji neki koji je slobodan.

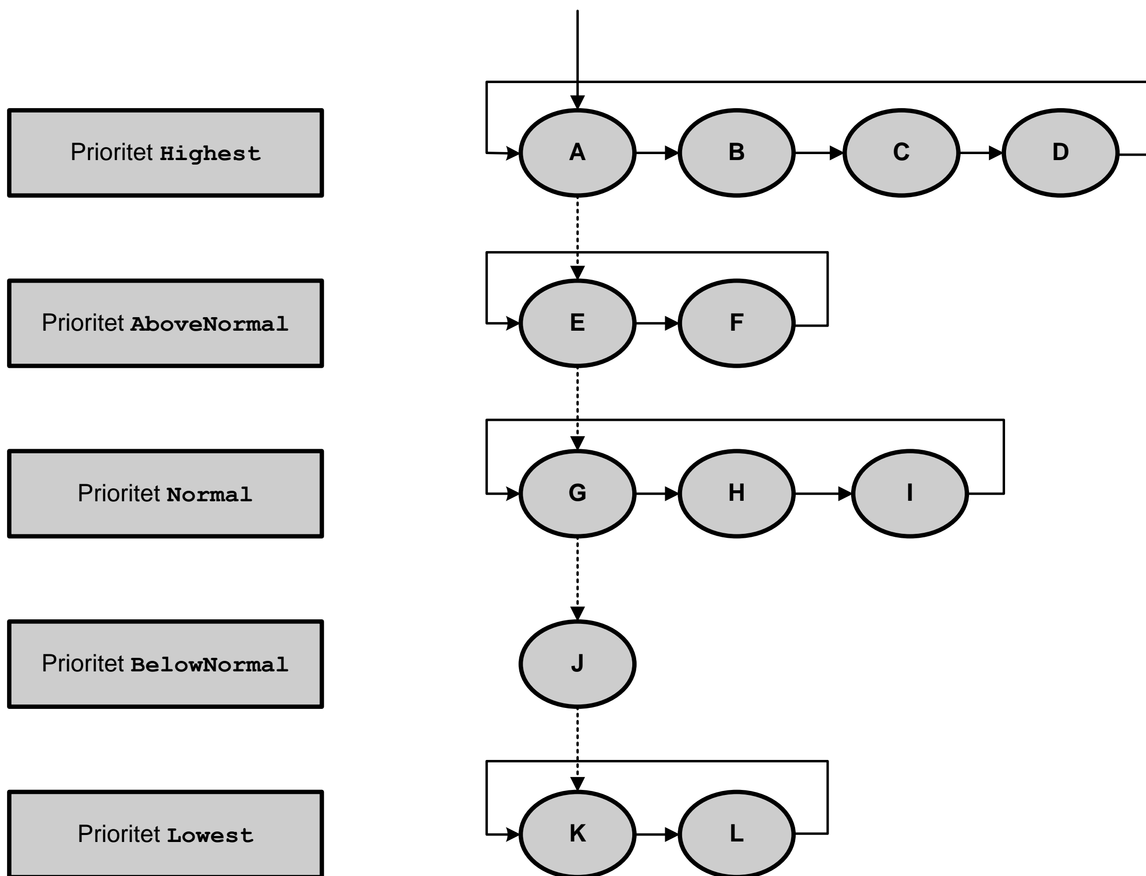
Za prijelaz iz stanja *Running* u *WaitSleepJoin* postoje tri moguća načina. U slučaju da dretva dođe do programskoga kôda koji se još ne može izvršiti jer svi uvjeti nisu ispunjeni, dretva može pozvati metodu *Wait* iz klase *Monitor* te prijeći u stanje *WaitSleepJoin*. Dretva se iz ovoga stanja može vratiti u stanje *Started* tako da neka druga dretva pozove metode *Pulse* ili *PulseAll,* obje iz klase *Monitor*. Metoda *Pulse* vraća prvu sljedeću dretvu koja čeka u stanje *Started,* dok metoda *PulseAll* vraća sve dretve koje čekaju u stanje *Started*.

Dretva koja je u stanju *Running* pozivom metode *Sleep* može prijeći u stanje *WaitSleepJoin* i to za vrijeme koje je specificirano kao ulazni argument metode. Nakon što prođe specificirano vrijeme dretva se vraća u stanje *Started*. U slučaju da postoji dretva

koja ne može nastaviti s izvršavanjem sve dok neka druga dretva ne završi, prva dretva može pozvati *Join* metodu druge dretve. Pozivom ove metode, prva dretva prelazi u stanje *WaitSleepJoin* i tamo ostaje sve dok druga dretva ne završi s izvršavanjem. Pozivom metode *Suspend*, dretva koja se nalazi u stanju *Running* prelazi u stanje *Suspended* i ostaje tamo sve dok neka druga dretva ne pozove *Resume* metodu zaustavljene dretve.

4.2. Prioriteti dretvi

Prioritet svake dretve nalazi se u rasponu između vrijednosti *ThreadPriority.Lowest* do *ThreadPriority.Highest* koje su definirane u enumeraciji *ThreadPriority*. Sam skup se sastoji od pet vrijednosti (*Lowest*, *BelowNormal*, *Normal*, *AboveNormal*, *Highest*) te je vrijednost *Normal* zadana vrijednost svake dretve. U sklopu operacijskoga sustava podržan je koncept *timeslicing* koji omogućava dijeljenje procesora od strane više dretvi istoga prioriteta. Koristeći koncept *timeslicinga,* svaka dretva dobije kvant (engl. *quantum*) vremena što je u biti kratak trenutak procesorskoga vremena tijekom kojega se dretva može izvršiti. Nakon što se kvant vremena završi, provjerava se stanje dretve. Ukoliko dretva nije završila s poslom, oduzima joj se pravo korištenja procesora i dodjeluje se drugoj s istim prioritetom. Komponenta raspoređivač dretvi (engl. *thread scheduler*) ima zadatak osigurati izvršavanje dretve s najvišim prioritetom. Raspoređivač osigurava da ako postoji više dretvi s najvišim prioritetom, izvršenje u kvantu vremena i to koristeći pristup *round-robin*. Na slici 4.2. prikazan je primjer izvršavanja prema pristupu *round-robin*. U tom pristupu izvršavanje dretvi počinje dretvom s najvišim prioritetom. Kada dretva završi ili ako nije gotova s poslom u kvantu vremena, dretva s istim prioritetom dobiva pravo na korištenje procesora. Ovaj pristup se ponavlja za sve dretve istoga prioriteta. Tek kada sve dretve s istim prioritetom završe s izvođenjem, prelazi se na dretve s nižim prioritetom te se postupak ponavlja.



Slika 2.2. Prikaz pristupa *round-robin* (Deitel et al., 2002)

4.3. Stvaranje i pokretanje dretvi

Za stvaranje i pokretanje nove dretve u C# potrebno je nekoliko koraka. Sam proces zahtjeva stvaranje novoga objekta klase *Thread*, instanciranje novoga delegata *ThreadStart* s metodom delegata te prenošenje delegata *ThreadStart* u novi objekt. Nakon što je dretva stvorena, ona se nalazi u stanju čekanja te ju je potrebno pokrenuti. Pozivom metode *Start* od novoga objekta *Thread,* pokreće se izvođenje dretve. U slučaju da je nužno poslati podatke dretvi koja će obaviti određeni posao, potrebno je prilagoditi naredbe tako da se stvori nova klasa čiji će članovi biti vrijednosti koje su potrebne da se izračuna određeni posao koji se obavlja na dretvi. U donjem isječku programskoga kôda prikazano je stvaranje i pokretanje nove dretve. Klasa *Parametar* služi za prijenos parametara te sadržava metodu koja će se paralelno obavljati. Poziv na tu metodu se prenosi preko delegata *ThreadStart* s kojim se stvara nova dretva. Na samome kraju pozivom metode *Start* pokreće se dretva.

```
Parametri prvi = new Parametri(N, Q, M, 0, a, ref A, ref B, ref C);
Thread pThread = new Thread(new ThreadStart(prvi.Mnozenje));

pThread.Start();
```

4.4. Interakcije između dretvi

Nakon stvaranja i pokretanja dretve potrebno je pričekati da se rad dretve završi prije nego što glavna pozivajuća dretva nastavi s izvođenjem. Za ostvarenje ove funkcionalnosti postoji metoda *Join,* prikazana u donjem isječku, koja blokira pozivajuću dretvu sve dok dretva ne završi ili dok ne prođe vrijeme koje se specificiralo prilikom poziva metode.

```
pThread.Join();
```

Ako je potrebno blokirati dretvu koja se izvršava postoji metoda *Sleep()* koja šalje zahtjev da se izvršavanje dretve ne nastavi dok ne protekne vrijeme uneseno kao argument prilikom poziva. Pozivom ove metode procesor se oslobađa na određeno vrijeme, izvršavajući poslove ostalih dretvi. U slučaju da je potrebno da pozivajuća dretva prepusti izvršavanje drugoj dretvi koja je spremna za izvršavanje na procesoru, postoji metoda *Yield()* koja omogućava tu funkcionalnost.

```
Thread.Sleep(100);
Thread.Yield();
```

Samo selektiranje dretve obavlja operacijski sustav. Kao dodatna mogućnost interakcije s dretvama postoji i mogućnost da se postavi prioritet raspoređivanja dretve koja će se izvršiti, kao i mogućnost dohvata identifikacijske oznake dretve. Inače se samo zaustavljanje i oslobađanje dretvi obavlja od strane *CLR-a*.

4.5. Sinkronizacija dretvi

Koristeći metode za stvaranje i pokretanje dretvi, moguće je stvoriti veći broj dretvi koje će se istovremeno izvršavati. U slučaju da se svaka dretva izvršava neovisno bez korištenja zajedničkih resursa ne bi trebalo doći do poteškoća u izvršavanju. U slučaju da

više dretvi imaju potrebu dijeliti određeni resurs, potrebna je kontrola pristupa. U jeziku C# postoje metode kojima se mogu koordinirati i sinkronizirati aktivnosti između dretvi. Koristeći sinkronizaciju, programima se omogućava iskorištavanje prednosti koje proizlaze iz višedretvenoga rada dok se istovremeno održava integritet stanja objekata i podataka.

U jeziku C# naredba *lock* dopušta samo jednoj dretvi izvršavanje dijela programskoga kôda dok ostale dretve moraju čekati da ta dretva završi. Sama naredba je dio C# jezika dok sama izvedba naredbe koristi klasu *Monitor* koja je dio *.NET Frameworka*. U donjem isječku prikazana je jednostavna klasa *LockTest* koja posjeduje jednu metodu *Test*. U sklopu te metode obavlja se zaključavanje liste gdje je omogućeno da samo jedna dretva može istovremeno dodati novi element.

```
class LockTest
{
    List<string> list = new List<string>();
    void Test()
    {
        lock (list)
        {
            list.Add("Item");
        }
    }
}
```

Iako je naredba *lock* brža i jednostavnija za korištenje, klasa *Monitor* posjeduje ugrađenu mogućnost zaključavanja s vremenskim ograničenjem. Ova funkcionalnost može biti vrlo korisna u slučaju da se unutar zaključanoga dijela nalazi programski kôd koji traje predugo ili nema kraja poput beskonačne petlje. C# posjeduje i izvedbu *Mutex* zaključavanja koja pruža istu funkcionalnost kao i *lock,* ali ima prednost rada preko više procesa. Ta funkcionalnost omogućava zaključavanje ne samo nad aplikacijom, nego i nad cijelim računalom.

Ako je za izvedbu programa potrebna struktura koja oponaša semafor, postoji klasa *Semaphore* koja dopušta da određeni broj dretvi izvršava dio kôda. Inače objekt klase *Semaphore* koji dopušta najviše jednu dretvu je dosta sličan naredbama *lock* ili *Mutex* s tim da *Semaphore* nema pozivajuću dretvu ili vlasnika. U donjem programskom isječku prikazan je primjer korištenja klase *Semaphore* gdje je inicijaliziran novi objekt klase *Semaphore* i to s ograničenjem od pet dretvi. Unutar metode *Posao* dopušteno je izvršavanje do pet dretvi istovremeno, te se pozivom metode *Release()* oslobađa mjesto u semaforu za još jednu dretvu.

```
class SemaphoreTester
{
    static Semaphore sf = new Semaphore(5, 5);

    static void Main()
    {
        for (int i = 0; i < 15; i++)
            new Thread(Posao).Start();
    }

    static void Posao()
    {
        while (true)
        {
            sf.WaitOne();
            Thread.Sleep(200);
            sf.Release();
        }
    }
}
```

5. Pre-alokacija radnih dretvi (klasa *ThreadPool*)

Klasa *ThreadPool* je dio Microsoftove biblioteke *.NET Framework* te se nalazi u imeničkom prostoru *System.Threading* i koristi se za pre-alokaciju radnih dretvi. Osnovna zadaća klase je stvaranje skupa dretvi koji se može koristiti za obavljanje radnih zadataka, obrađivanje asinkronih ulazno/izlaznih operacija i čekanje na ostale dretve. *ThreadPool* stvara skup dretvi koji se mogu koristiti i za obavljanje više zadaća u pozadini što primarnu dretvu ostavlja slobodnom asinkrono obavljati ostale zadaće. Kada dretva koja pripada skupu *ThreadPool* obavi svoj posao, ona se vraća na red dretvi koji se mogu ponovno koristiti. Za taj postupak jedna dretva prati status više operacija čekanja dretvi koji se nalazi u redu skupa dretvi. Kada se operacija čekanja završi, jedna dretva iz skupa dretvi izvršava odgovarajuću funkciju povratnoga poziva (engl. *callback function*). Dretve koje se nalaze u upravljanom skupu dretvi su pozadinske dretve (engl. *background threads*) što znači da one neće ostaviti aplikaciju u stanju izvršavanja nakon što su sve dretve u prvom planu (engl. *foreground threads*) gotove s izvršavanjem.

Svaka instanca klase *ThreadPool* ima zadani broj od 250 radnih dretvi (engl. *worker threads*) po svakom procesoru ili jezgri te 1000 dretvi za izvršavanje ulazno/izlaznih operacija (engl. *I/O completion threads*). Broj dretvi u njihovom skupu može se promijeniti koristeći metodu *SetMaxThreads*. Svakoj stvorenoj dretvi dodjeljuje se stog zadane veličine, ali i zadani prioritet. Pozivom metode *QueueUserWorkItem*, moguće je narediti da određeni zadatak obavlja jedna dretva u *ThreadPoolu*. Ova metoda kao parametar prima referencu na metodu ili na delegata koji će biti pozvan od strane jedne dretve odabrane iz skupa dretvi. Nakon što se radni zadatak proslijedi pojedinoj dretvi, ne postoji način za poništenje izvršavanja.

U programskom isječku koji slijedi prikazan je primjer korištenja klase *ThreadPool*. Klasa je korištena za paralelizaciju operacije množenja matrica. Da bi se ostvarila što bolja iskorištenost procesora te i bolji vremenski rezultati, sam postupak množenja matrica je podijeljen na četiri dijela te su zato korištene i četiri instance klase *ThreadPool*. Svaka instanca *ThreadPool* klase prima radni zadatak preko delegata *WaitCallback* te i novi objekt klase *Parametar* koja sadržava sve potrebne parametre za izvršavanje operacije množenja matrica. Za samu sinkronizaciju svih četiriju objekta

korištena je klasa *CountdownEvent* koja predstavlja brojilo koje se smanjuje kada svaki *ThreadPool* objekt završi izvršavanje dodijeljenoga posla.

```
private static CountdownEvent counter;

counter = new CountdownEvent(4);

ThreadPool.QueueUserWorkItem(new WaitCallback(MnozenjeTP), new Parametri(N, Q, M, 0, a,
ref A, ref B, ref C));

ThreadPool.QueueUserWorkItem(new WaitCallback(MnozenjeTP), new Parametri(N, Q, M, a, b,
ref A, ref B, ref C));

ThreadPool.QueueUserWorkItem(new WaitCallback(MnozenjeTP), new Parametri(N, Q, M, b, c,
ref A, ref B, ref C));

ThreadPool.QueueUserWorkItem(new WaitCallback(MnozenjeTP), new Parametri(N, Q, M, c, N,
ref A, ref B, ref C));

counter.Wait();
```

6. Proširenja programskog jezika (klasa *Parallel Extensions*)

Razvijen od strane jedne od Microsoftovih razvojnih skupina, biblioteka *Parallel Extensions* je rješenje za upravljani istovremeni rad i paralelizaciju kôda unutar aplikacija. Biblioteka se koristi od 2007., a danas je dio platforme *.NET Framework 4.0*. Osnovni koncept u biblioteci je zadaća (engl. *task*) koja je mali dio programskoga kôda koji se može samostalno izvršiti te je predstavljen lambda funkcijom. U sklopu biblioteke postoje metode koje omogućavaju podjelu programskoga kôda u dijelove koji se mogu izvršiti kao zadaće.

Za upravljanje svim zadaćama postoji objekt *Task Manager* koji sadržava globalni red zadaća koje se potom izvršavaju na određenom skupu dretvi. Po zadanim postavkama, broj stvorenih dretvi ovisi o broju procesora ili procesorskih jezgri, iako postoji i mogućnost ručnoga postavljanja broja dretvi. Svaka dretva posjeduje vlastiti red zadaća te kada je u besposlenom stanju (engl. *idle state*), preuzima novi skup zadaća, stavlja ih na vlastiti lokalni red te izvršava zadaću po zadaću. Ako je globalni red prazan, dretva će tražiti zadaće u redovima drugih dretvi te preuzeti one koje su najdulje u redu. Kada se pokrene izvršavanje, svaka se zadaća izvršava neovisno od ostalih zadaća što u slučaju korištenja dijeljenoga resursa zahtijeva uvođenje sinkronizacije koristeći funkciju zaključavanja (engl. *locks*).

Sama biblioteka *Parallel Extensions* sastavljena je od dva dijela: *Parallel LINQ* i *Task Parallel Library*. *Parallel LINQ* (*PLINQ*) je stroj za istovremeno izvršavanje LINQ upita nad bazom podataka. Koristeći sučelje *IParallelEnumerable* definirano u *PLINQ,* moguće je paralelizirati izvršavanje upita nad objektima (*LINQ to Objects*) i XML podacima (*LINQ to XML*).

Druga komponenta biblioteke *Parallel Extensions* je *Task Parallel Library (TPL)* koja omogućava pisanje upravljanoga kôda koji automatski iskorištava mogućnosti višeprocesorskoga sustava. Pomoću *TPL-a* moguće je paralelizirati postojeći serijski programski kôd koji omogućava bolju iskoristivost postojeće računalne opreme. Sami poslovi stvaranja dretvi, što uključuje i određivanje broja dretvi te uništavanja dretvi, obavljaju se od strane biblioteke. Biblioteka sadržava skup konstruktora s kojima se dijeli posao u zadaće. Sam skup konstruktora se može podijeliti na skup konstruktora za paralelizaciju podataka i na skup konstruktora za paralelizaciju zadaća.

6.1. Konstruktori za paralelizaciju podataka

U slučaju da se radi sa skupom podataka u kojemu se nad svakim elementom izvršava određena operacija, moguće je taj dio programskoga kôda paralelizirati. S naredbama *Parallel.For* i *Parallel.ForEach* može se ostvariti paralelno izvođenje određenih operacija nad više elemenata koji se nalaze u polju ili nekakvom drugom skupu podataka. Kada se pokrene paralelna izvedba petlje, biblioteka *TPL* dijeli skup podataka tako da se petlja može izvršiti na više dijelova istovremeno. Komponenta *Task Scheduler* dijeli zadaće na osnovi sistemskih resursa i opterećenja procesora. Kada je potrebno, *Task Scheduler* ima i mogućnost redistribucije posla između više dretvi i procesora ukoliko opterećenja postanu neuravnotežena.

6.1.1. Parallel.For

Koristeći metodu *Parallel.For,* moguće je paralelizirati izvođenje osnovne *for* petlje. Sama metoda prima tri argumenta za izvođenje. Prvi argument je numerička vrijednost predstavljena *integer* varijablom *fromInclusive* koja predstavlja inicijalnu vrijednost iteratora petlje, dok je drugi argument varijabla *toExclusive* koja predstavlja zadnju vrijednost iteratora. Treći argument je delegat koji se poziva jedanput po iteraciji. Taj delegat je programski kôd koji se inače nalazi u sklopu *for* petlje. U programskom odsječku koji slijedi prikazan je primjer korištenja konstruktora *Parallel.For* za operaciju množenja matrica. Umjesto *for* petlje korišten je konstruktor *Parallel.For* te su argumenti poziva metode zapravo donja i gornja granica inicijalne *for* petlje. Sadržaj petlje se potom prenosi metodom korištenja delegata koji se onda istovremeno izvršava.

```
Parallel.For(0, N, delegate(int i)
            {
                for (int j = 0; j < Q; j++)
                {
                    for (int k = 0; k < M; k++)
                    {
                        C[i][j] += A[i][k] * B[k][j];
                    }
                }
            });
```

6.1.2. Parallel.ForEach

Slično kao i *Parallel.For*, metoda *Parallel.ForEach* koristi se za paralelizaciju petlji, ali u ovom slučaju nad skupom podataka *IEnumerable <TSource>*. Metoda inače prima samo dva argumenta i to varijablu *source* koja predstavlja izvor podataka i varijablu *body* koja predstavlja delegata koji prenosi programski kôd koji će se istovremeno izvršavati. U donjem isječku prikazan je jednostavan primjer korištenja metode *Parallel.ForEach* u kojemu je izvršena obrada polja objekata. Pozivom metode *Parallel.ForEach* istovremeno će se pozvati metoda *obradaSenzora* za svaki objekt koji se nalazi u polju *senzori.*

```
Sensor[] senzori = {
	new Sensor{id = 1, rez = 56},
	new Sensor{id = 2, rez = 129},
	new Sensor{id = 3, rez = 19}
};
Parallel.ForEach(senzori, obradaSenzora);
```

6.2. Konstruktori za paralelizaciju zadaća

Kako je biblioteka *Task Parallel Library* osnovana na radu sa zadaćama, postoji i mogućnost direktnoga stvaranja zadaća. Zadaća inače predstavlja asinkronu operaciju i njeno stvaranje dosta podsjeća na stvaranje nove dretve, ali na višoj razini apstrakcije. Jedna prednost korištenja zadaća je efikasnije i više skalabilno korištenje sistemskih resursa. U pozadini izvršavanja, zadaće su stavljene na red koji pripada objektu klase *ThreadPool* koji je poboljšan dodatnim algoritmima koji određuju i namještaju broj potrebnih dretvi koji će dati maksimalnu propusnost. Druga prednost korištenja zadaća je veća programska kontrola nego što je moguća s dretvama. U sklopu *TPL-a* nalazi se veći skup API-ja koji podržavaju čekanje, nastavak nakon otkazivanja, robusno rukovanje s iznimkama, dohvat detaljnih statusa i samostalno planiranje.

6.2.1. Parallel.Invoke

Metoda *Parallel.Invoke* omogućava jednostavan način za pokretanje paralelnoga izvršavanja više neovisnih izraza. Kao argumente metoda prima *System.Action* objekte koji predstavljaju akcije ili događaje koji se moraju obaviti. Metoda paralelno pokreće sve događaje paralelno dijeleći ih po jezgrama. Tip *System.Action* je u biti imenovani delegat koji predstavlja podrutinu koja ne prima parametre i ne vraća ništa, prenoseći samo adrese

podrutina koje se trebaju pokrenuti. Metoda *Parallel.Invoke* kao argument prima parametar *array* što omogućava da se odjednom može proslijediti cijelo polje akcija umjesto pojedinačnih akcija. Iako se sve akcije istovremeno pozivaju te se izvršavaju paralelno, pojavi li se situacija gdje dvije dretve istovremeno pristupaju istim varijablama može doći do poteškoća. Koristi li se zaključavanje, postoji mogućnost da dvije dretve probaju zaključati iste resurse istovremeno što može dovesti do potpunoga zastoja (engl. *deadlock*). U tom slučaju kada se ovo dogodi obje dretve su u stanju čekanja jer svaka čeka za resurs koji je druga dretva zaključala.

U isječku kôda ispod teksta nalazi se primjer korištenja metode *Parallel.Invoke* na primjeru algoritma *Quicksort* gdje su poslovi sortiranja lijeve i desne strane polja podijeljeni u dvije akcije. Nakon što je definirano koji će se posao obaviti u sklopu svake akcije, poziva se metoda *Parallel.Invoke* s akcijama kao argumentima metode.

```
QuickSort lijevi = new QuickSort(L, pivot, ref polje);
Action leftFX = () => { lijevi.SortiranjeFX(); };

QuickSort desni = new QuickSort((pivot + 1), R, ref polje);
Action rightFX = () => { desni.SortiranjeFX(); };

Parallel.Invoke(leftFX, rightFX);
```

7. Množenje matrica

Problem množenja matrica je odličan primjer problema koji se može efikasno paralelizirati jer se cijeli postupak može lako podijeliti u neovisne dijelove. Sam postupak ima $N \times M \times P$ operacija množenja gdje su N i M dimenzije prve matrice, a M i P dimenzije druge matrice.

7.1. Algoritam množenja matrica

Osnovni algoritam množenja matrica gdje se množe matrica A i B dimenzija $N \times M$ i $M \times P$ izgleda ovako:

```
procedura množenje;
     za i:=1 do N radi
          za j:=1 do P radi
               za k:=1 do M radi
                    C[i, j]:= C[i, j] + A[i, k] x B[k, j];
kraj množenje;
```

Algoritam je sastavljen od triju *for* petlji, gdje prva petlja s iteratorom *i* kreće od 1 te ide sve do broja *N* koji predstavlja broj redaka matrice *A* i rezultante matrice *C*. Druga petlja s iteratorom *j* kreće od prvoga elementa i ide sve do *P* koji predstavlja broj stupaca matrice *B* i rezultantne matrice *C*. Treća petlja s iteratorom *k* ide od 1 do *M* koji predstavlja broj stupaca matrice *A* i broj redaka matrice *B*. U sklopu zadnje petlje obavlja se množenje pojedinačnoga elementa matrica *A* i *B* koji se potom zbraja s odgovarajućim elementom u matrici *C*.

U slučaju množenja kvadratnih matrica s istim dimenzijama, može se koristiti sljedeći poboljšani algoritam:

```
procedura množenjekvad;
     za i:=1 do N radi
          za j:=1 do P radi
               za k:=1 do M radi
                    C[i, k]:= C[i, k] + A[i, j] x B[j, k];
kraj množenjekvad;
```

Za razliku od prvoga algoritma, ovaj algoritam množenja u sklopu zadnje petlje dohvaća cijeli redak matrice samo jednom. Ovaj postupak je bolji jer prilikom množenja cijeli se redak, ako može stati, stavlja u priručnu memoriju. Zbog te činjenica ne gubi se vrijeme koje bi se inače koristilo za dohvaćanje elemenata različitih redaka prilikom izračuna jednoga elementa rezultante matrica.

7.2. Implementacija algoritma

Da bi se u potpunosti ispitale mogućnosti paralelizacije kôda u jeziku C#, izrađene su četiri implementacije algoritma množenja matrica. Za svaku varijantu korištene su iste matrice koje su popunjene s proizvoljnim numeričkim vrijednostima dvostruke preciznosti, dok je rezultanta matrica *C* popunjena s nulama. Uz različite varijante ugrađena je mogućnost korištenja poboljšanoga algoritma za kvadratne matrice za sve četiri varijante. Algoritam množenja matrica implementiran je u sljedeće implementacije: serijska izvedba, izvedba koja koristi klasu *Thread*, izvedba koja koristi klasu *ThreadPool* i izvedba koja koristi biblioteku *Parallel Extensions*.

Implementacija množenja izvedena je prema osnovnom algoritmu. Odsječak programskoga kôda prikazuje osnovnu serijsku izvedbu množenja:

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < Q; j++)
        for (int k = 0; k < M; k++)
        {
            C[i][j] += A[i][k] * B[k][j];
        }
```

7.2.1. Izvedba izravnog korištenja dretvi (klasa *Thread*)

Za korištenje klase *Thread* potrebne su određene izmjene osnovnoga algoritma. S obzirom da se samo izvršavanje i mjerenje izvodilo na dvojezgrenom procesoru s dodatkom Intelove *Hyper-Threading* tehnologije, bilo je potrebno podijeliti posao množenja na četiri dijela gdje svaki dio predstavlja dio rezultante matrice. U odsječku koji slijedi prikazano je stvaranje dvije dretve kojima su dodijeljeni poslovi množenja prve dvije četvrtine rezultante matrice *C*.

```
Parametri prvi = new Parametri(N, Q, M, 0, a, ref A, ref B, ref C);
Thread pThread = new Thread(new ThreadStart(prvi.Mnozenje));

Parametri drugi = new Parametri(N, Q, M, a, b, ref A, ref B, ref C);
Thread dThread = new Thread(new ThreadStart(drugi.Mnozenje));

pThread.Start();
dThread.Start();

pThread.Join();
dThread.Join();
```

Nakon što se kreiraju objekti klase *Parametri* preko kojih će se prenijeti parametri potrebni za množenje, stvaraju se i nove instance klase *Thread* koje predstavljaju pojedinu dretvu. Svakoj se dretvi dodjeljuje posao za obavljanje preko delegata koji referencira metodu *Množenje* klase *Parametri*. Samo pokretanje dretvi je inicirano pozivom metode *Start* dok se sinkronizacija dretvi ostvaraje s pozivom metode *Join* koja čeka da dretva završi s poslom.

7.2.2. Izvedba pre-alokacijom radnih dretvi (klasa *ThreadPool*)

Kao i izvedba koja koristi klasu *Thread,* izvedba množenja matrica pomoću klase *ThreadPool* ostvaruje se tako da se posao množenja dijeli na četiri dijela. U programskom odsječku koji slijedi prikazana je implementacija:

```
counter = new CountdownEvent(4);

ThreadPool.QueueUserWorkItem(new WaitCallback(MnozenjeTP), new Parametri(N, Q, M, 0, a,
ref A, ref B, ref C));

ThreadPool.QueueUserWorkItem(new WaitCallback(MnozenjeTP), new Parametri(N, Q, M, a, b,
ref A, ref B, ref C));

ThreadPool.QueueUserWorkItem(new WaitCallback(MnozenjeTP), new Parametri(N, Q, M, b, c,
ref A, ref B, ref C));

ThreadPool.QueueUserWorkItem(new WaitCallback(MnozenjeTP), new Parametri(N, Q, M, c, N,
ref A, ref B, ref C));

counter.Wait();
```

Za svaki objekt klase *ThreadPool* preko delegata stavlja se metoda *MnozenjeTP* u red čekanja i prenosi se novi objekt klase *Parametri,* koji sadržava podatke koji će biti potrebni za operaciju množenja. Za sinkronizaciju pozvanih *ThreadPool* objekata, što osigurava da je svaki dio množenja gotov, koristi se klasa *CountdownEvent* i metoda *Wait* klase *Monitor*. Sama sinkronizacija se postiže tako da se na kraju svakoga obavljenog posla pozove metoda *Signal* klase *CountdownEvent* koja smanjuje postavljeno brojilo. Metoda *Wait* zaustavlja glavnu pozivajuću dretvu sve dok se brojilo ne smanji s četiri na nulu. Nakon ispunjenja toga uvjeta, izvođenje glavne dretve se može nastaviti.

7.2.3. Izvedba korištenja proširenja programskog jezika

Izvedba množenja matrica koja koristi biblioteku *Parallel Extensions* je dosta slična osnovnom algoritmu množenja matrica. U kôdu koji slijedi prikazan je poziv metode *Parallel.For* klase *Parallel*:

```
Parallel.For(0, N, delegate(int i)
{
    for (int j = 0; j < Q; j++)
    {
        for (int k = 0; k < M; k++)
        {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
});
```

Sam poziv metode ustvari zamjenjuje vanjsku petlju osnovnoga algoritma te kao argumente prima donju i gornju granicu izvođenja izvorne *for* petlje. Zadnji argument koji se prenosi preko delegata ustvari je posao ili zadatak koji će se višestruko izvršavati. Može

se primijetiti da unutarnje *for* petlje nisu zamijenjene metodom *Parallel.For* petlje. Korištenje dodatnih poziva je izostavljeno zbog neefikasnosti izvođenja jer dovode do pretjerane paralelizacije (engl. *overexposed parallelism*). Za količinu posla koji se obavlja te broju dostupnih jezgri, dovoljnom se pokazala paralelizacija samo vanjske petlje.

7.3. Trajanje izvođenja implementacija

Vremenski rezultati izvođenja operacije množenja prikazani su u tablici 7.3.1. Za dobivanje rezultata korištena je osnovna verzija algoritma množenja. U tablici su prikazani rezultati množenja matrica s dimenzijama 100×100, 200×200, 500×500, 1000×1000 i 2000×2000. Za svaku pojedinu veličinu matrice prikazana su vremena izvođenja za sve četiri implementacije algoritma (*Serijska, Thread, ThreadPool, Parallel Extensions*).

Za matrice veličine 100×100 može se primijetiti da je jedino implementacija koja koristi *Parallel Extensions* brža od serijske implementacije i to za 37%. Razlog tomu je činjenica da zbog manjih matrica sama veličina posla nije dovoljno velika da bi se nadoknadilo potrebno vrijeme za podjelu posla po dretvama te njihovo istovremeno izvršavanje. Pokazalo se da za množenje matrica čije su dimenzije 200×200 i više, paralelizacija ima znatan efekt na vrijeme trajanja izvođenja i to od 34% (*Thread*) do 115% (*Parallel Extensions*). Kada se usporede izvođenja svih paralelnih implementacija, može se zaključiti da je implementacija koja koristi biblioteku *Parallel Extensions* imala najbolje rezultate gdje za matrice s dimenzijama 2000×2000 stvara ubrzanje od čak 151% u odnosu na serijsku implementaciju.

Implementacija	**t (s)**				
	Dimenzije matrica				
	100	**200**	**500**	**1000**	**2000**
Serijska	0,0148904	0,1270713	2,3280329	33,3508906	324,1811715
Thread	0,0189312	0,0944348	1,1992622	12,4561162	131,4825301
ThreadPool	0,0167840	0,0726772	1,1358487	12,3942857	131,5523570
Parallel Extensions	0,0108398	0,0590652	1,1168606	12,6925817	128,7440313

Tablica 7.3.1. Vremenski rezultati izvođenja osnovnoga algoritma množenja matrica

Vremenski rezultati izvođenja množenja matrica s poboljšanim algoritmom su prikazani u tablici 7.3.2. Za matrice veličine 100×100 može se primijetiti da su rezultati

izvođenja sporiji od serijske implementacije algoritma, osim verzije u kojoj je korištena biblioteka *Parallel Extensions* koja je brža za 66%. Prilikom množenja matrica s dimenzijama 200×200 sve paralelne implementacije izvršavaju se brže od serijske i to od 37% (*ThreadPool*) do 110% (*Parallel Extensions*). Kada se usporede rezultati množenja za matrice s dimenzijama 500×500, mogu se primijetiti veće razlike i to do 116% (*Parallel Extensions*). Najveća ubrzanje od 132% (*Parallel Extensions*) ostvaruje se tek prilikom množenja matrica s dimenzijama 1000×1000 i 2000×2000.

Usporedbom rezultata može se ustvrditi da su implementacije koje koriste poboljšanu verziju algoritma brže od implementacija koji koriste osnovni algoritam. Razlog ovog boljeg rezultata je činjenica da prilikom množenja matrica s manjim dimenzijama u oba algoritma veličina operanada je manja od kapaciteta priručne memorije. Tek kad se množe veće matrice, uočava se da poboljšani algoritam ima mogućnost smještanja svih potrebnih operandi za operaciju u priručnoj memoriji. Za matrice većih dimenzija osnovnom algoritmu je potrebno višestruko dohvaćanje podataka iz sekundarne memorije koji se potom prebacuju u priručnu memoriju. To prebacivanje podataka iz jedne razine memorije u drugu stvara trošak vremena koji dovodi do sporijega izvođenja.

Kao i osnovnoj izvedbi algoritma, pokazalo se da je od tri paralelne varijante najbolje rezultate imala verzija u kojoj je korištena biblioteka *Parallel Extensions*. Iako su vremena sve tri implementacije dosta blizu, implementacija s *Parallel Extensions* daje bolje rezultate. Razlog ovog boljeg rezultata je to što je korištenjem konstruktora poput *Parallel.For* moguće ostvariti dosta jednostavniju paralelizaciju te činjenica da se sav posao upravljanja s dretvama i memorijom obavlja od strane platforme *.NET Framework*.

Implementacija	**t (s)**				
	Dimenzije matrica				
	100	**200**	**500**	**1000**	**2000**
Serijska	0,0137847	0,1103625	1,7280741	14,1879061	117,07264334
Thread	0,0194979	0,0740211	0,8003305	6,3826344	53,8392165
ThreadPool	0,0159152	0,0802437	0,819443	6,3683667	53,5881714
Parallel Extensions	0,0082974	0,052391	0,79962	6,2840698	50,3314149

Tablica 7.3.2. Vremenski rezultati izvođenja poboljšanoga algoritma množenja matrica

8. Quicksort

Jedan od popularnijih algoritama za sortiranje više vrsta elemenata je algoritam *Quicksort*. Sam algoritam razvio je C. A. R. Hoarse, te je potrebno u prosjeku $O(n \log n)$ usporedbi za sortiranje n elemenata. U najgorem slučaju potrebno je $O(n^2)$ usporedbi što je vrlo rijetko ako je ispravna izvedba algoritma. Sortiranje velikih polja je inače računalno i vremenski zahtjevna operacija te je zato odličan primjer za paralelizaciju. Uz to algoritam *Quicksort* je prikladan za paralelizaciju zbog mogućnosti dijeljenja polja u neovisne dijelove čije se sortiranje može obavljati istovremeno.

8.1. Algoritam sortiranja Quicksort

Za sortiranje polja prema *Quicksort* algoritmu potrebno je slijediti sljedeće korake:

1. Potrebno je odabrati element koji će predstavljati stožer polja.
2. Nakon odabira stožera, potrebno je prebaciti sve elemente koji su manji od stožera na poziciju ispred stožera.
3. Potrebno je prebaciti sve elemente koji su veći od stožera poslije stožera. Kada se polje u potpunosti uredi, stožerni element se nalazi na mjestu na kojemu će ostati do kraja sortiranja.
4. Rekurzivno sortirati podpolje elemenata koji su manji od stožera sve dok je broj elemenata u podpolju veći od jedan.
5. Rekurzivno sortirati podpolje elemenata koji su veći od stožera sve dok je broj elemenata u podpolju veći od jedan.

8.2. Implementacija algoritma

Implementacija algoritma u potpunosti slijedi osnovni algoritam sortiranja. U prvom koraku traži se stožerni element polja. Radi što kraćega vremena trajanja sortiranja polja, potrebno je kvalitetno odabrati stožerni element. Zbog tog razloga stožerni element se odabire tako da se određeni broj elemenata uzme iz polja kao uzorak. Broj elemenata koji će se uspoređivati iznosi $((broj\ elemenata + 1) \div 10) + 3$. Od svih elemenata odabire se element koji je najbliži srednjoj vrijednosti cijeloga podskupa elemenata. Nakon odabira stožera polje elemenata se uređuje tako da se svi elementi manji od stožera prebace prije stožera, a svi elementi veći od stožera poslije stožera. Poslije ovoga koraka polje je

spremno za sortiranje. Ova dva inicijalna koraka se obavljaju za sve serijske i paralelne izvedbe algoritma. Sam postupak sortiranja podpolja obavlja se tako da se uspoređuju elementi s početka i kraja podskupa te se obavlja izmjena u slučaju da se element nalazi na pogrešnoj strani u odnosu na stožer. Ovaj zadnji korak se obavlja rekurzivno i to rekurzivnim pozivom funkcije za sortiranje.

8.2.1. Izvedba izravnog korištenja dretvi (klasa *Thread*)

Koristeći klasu *Thread* iz imeničkog prostora *System.Threading,* ostvarena je paralelna verzija sortiranja polja. Sama izvedba koristi isti algoritam za sortiranja, ali za razliku od serijske izvedbe omogućava istovremeno pokretanje sortiranja oba podpolja elemenata koji se nalaze s lijeva i desna stožernoga elementa. U programskom isječku koji slijedi nalazi se dio programskoga kôda koji stvara pojedine dretve:

```
QuickSort sortLeft = new QuickSort(0, (pivot - 1), ref poljeT);
Thread leftThread = new Thread(new ThreadStart(sortLeft.Sortiranje));

QuickSort sortRight = new QuickSort((pivot + 1), (brelem - 1), ref poljeT);
Thread rightThread = new Thread(new ThreadStart(sortRight.Sortiranje));

leftThread.Start();
rightThread.Start();
leftThread.Join();
rightThread.Join();
```

Sortiranje je predstavljeno klasom *QuickSort* čiji konstruktor prima osnovne parametre za sortiranje polja. Svaka stvorena instanca klase *Thread* prima po jedno podpolje za sortiranje i to preko *ThreadStart* delegata koji prenosi referencu na metodu *Sortiranje* klase *QuickSort* koja obavlja sortiranje. Samo izvođenje dretve počinje s pozivom metode *Start* dok se sinkronizacija dretvi obavlja metodom *Join*.

8.2.2. Izvedba pre-alokacijom radnih dretvi (klasa *ThreadPool*)

Kao druga varijanta za paralelizaciju sortiranja s algoritmom *Quicksort,* korištena je je klasa *ThreadPool* koja za razliku od klase *Thread* upravlja sa skupom dretvi. Osnovna podjela posla se obavlja tako da se stvara novi *Threadpool* objekt za dva podpolja koja se nalaze slijeva i desna stožernoga elementa. U donjem programskom isječku prikazana je izvedba implementacije koja koristi klasu *ThreadPool*. Za sinkronizaciju te i osiguravanje sortiranosti oba podpolja, korištena je klasa *CountdownEvent* koja je instancirana s

vrijednošću dva što predstavlja za koliko *ThreadPool* objekata naredba *Wait* mora čekati na njihov završetak. Kao i u implementaciji koja koristi klasu *Thread*, samo sortiranje se obavlja u sklopu klase *QuickSort.* Svaki ThreadPool objekt preko metode *QueueUserWorkItem* prima referencu na metodu *SortiranjeTP* u sklopu koje se obavlja sortiranje zadanoga podpolja.

```
counter = new CountdownEvent(2);

ThreadPool.QueueUserWorkItem(new WaitCallback(SortiranjeTP), new QuickSort(0, (pivot - 1), ref poljeT));

ThreadPool.QueueUserWorkItem(new WaitCallback(SortiranjeTP), new QuickSort((pivot + 1), (brelem - 1), ref poljeT));

counter.Wait();
```

8.2.3. Izvedba korištenja proširenja programskog jezika

Uz klase *Thread* i *ThreadPool* implementirana je i varijanta koja koristi konstruktore biblioteke *Parallel Extensions*. Pošto u samoj implementaciji algoritma *Quicksort* ne postoji odsječak programskoga kôda koji je prikladan za paralelizaciju s konstruktorom *Parallel.For,* korišten je konstruktor *Parallel.Invoke* kao što je prikazano u donjem programskom isječku. Sortiranje je kao i u prijašnjim implementacijama predstavljeno klasom *QuickSort* te je podijeljeno na dva dijela gdje je svaki dio predstavljen kao instanca klase *Action*. Pokretanje sortiranja počinje pozivom metode *Parallel.Invoke* koja kao argumente prima poslove koji bi se trebali obaviti paralelno.

```
QuickSort sortLeftFX = new QuickSort(0, (pivot - 1), ref poljeT);
Action leftFX = () => { sortLeftFX.Sortiranje(); };

QuickSort sortRightFX = new QuickSort((pivot + 1), (brelem - 1), ref poljeT);
Action rightFX = () => { sortRightFX.Sortiranje(); };

Parallel.Invoke(leftFX, rightFX);
```

Uz navedenu varijantu izrađena je implementacija algoritma koja nakon inicijalne paralelizacije paralelizira i daljnje korake sortiranja. U sklopu funkcije za sortiranje prilikom rekurzivnog pozivanja, umjesto običnoga poziva izmijenjen je poziv da se ponovna podjela posla obavlja s naredbom *Parallel.Invoke* i tako sve do kraja rekurzije.

8.3. Trajanje izvođenja implementacija

U tablici 8.3.1. nalaze se rezultati izvođenja svih pet implementacija algoritma sortiranja *Quicksort*. Pošto su se sortirali jednostavni elementi, sam efekt paralelizacije se može primijetiti tek kod nešto većega broja elemenata. Za polja s 100, 1000 i 10000 elemenata pokazalo se da je serijska implementacija bila brža. Tek od polja s 100000 elemenata dolazi do boljih rezultata izvođenja za paralelne implementacije. Samo ubrzanje iznosi oko 77% (*ThreadPool*) u odnosu na serijsku implementaciju, dok je sortiranje polja od 1000000 elemenata, koristeći paralelne implementacije (*Parallel Extensions*), brže za 84% od serijske implementacije.

Kao i s množenjem matrica, implementacija koja koristi biblioteku *Parallel Extensions* ima najbolje rezultate. Može se primijetiti da je druga varijanta implementacije *Parallel Extensions* imala lošije rezultate od ostalih paralelnih implementacija. Razlog ovoga lošeg rezultata je prekomjerna paralelizacija jer su se za svaku podjelu u podpolja stvarala dva nova događaja (*Actions*). Zbog jednostavnosti sortiranja, više se vremena gubilo na stvaranje dretvi i sinkronizaciju nego na sam postupak sortiranja.

Implementacija	**t (s)**				
	Broj elemenata				
	100	**1000**	**10000**	**100000**	**1000000**
Serijska	0,000235	0,0003336	0,0063046	0,0590682	0,7223109
Thread	0,0059603	0,0064766	0,0292522	0,0395125	0,4465997
ThreadPool	0,0026955	0,0033307	0,0091264	0,0333853	0,3928434
Parallel Extensions	0,0037028	0,0038764	0,0035197	0,0341570	0,3918258
Parallel Extensions 2.var.	0,0010145	0,0013618	0,0257218	0,0600541	0,6677068

Tablica 8.3.1. Vremenski rezultati izvođenja algoritma *Quicksort*

9. Problem trgovačkog putnika sa simuliranim kaljenjem

Problem trgovačkog putnika (engl. *travelling salesman problem*) je problem iz teorije grafova u kojemu se traži najefikasniji Hamiltonski ciklus koji trgovac može uzeti za obilazak svih *n* gradova. Uz određeni skup gradova te poznate udaljenosti ili trošak puta između pojedinih gradova, potrebno je pronaći najkraći ili najjeftiniji put koji će jednom obići sve gradove te se vratiti u početni grad. Problem je NP-težak te trenutno ne postoji metoda koja bi efikasno i u potpunosti riješila ovaj problem. Zbog ovog razloga koriste se heuristički algoritmi za optimizaciju koji dovode do dovoljno dobrih rezultata, iako ne optimalnih.

Algoritam simuliranoga kaljenja je primjer heurističkoga algoritma koji se koristi za optimizaciju rješavanja problema trgovačkoga putnika. Sam algoritam zapravo predstavlja tehniku kaljenja u metalurgiji gdje se određeni materijal grije pa potom hladi da bi se povećala veličina kristala u samoj strukturi materijala te smanjili defekti tih kristala. Koristeći analogiju samog fizičkoga procesa, algoritam zamjenjuje trenutno rješenje s slučajno susjednim rješenjem koje se odabire na osnovi razlike između duljina puteva.

9.1. Algoritam rješenja problema trgovačkoga putnika sa simuliranim kaljenjem

Osnovni algoritam koji je korišten za rješavanje problema trgovačkoga putnika slijedi (Jakobović, 2004):

```
procedura TSP_SA (N, S, p0, α, KTL);
     generiranje slučajnih koordinata za N gradova;
     izračunaj matricu D(N x N) //matrica udaljenosti svih
kombinacija gradova
     put := početno rješenje;
     dput := duljina(put);
     //određivanje prosječnog povećanja puta
     prosjek_povecanja := 0;
     brojac := 0;
     za i := 1 do N radi {
```

```
        put2 := slucajno_susjedno_rjesenje(put);
        dput2 := duljina(put2);
        ako je dput2 > dput tada
             prosjek_povecanja += (dput2 - dput);
             brojac++;
        }
   c := (prosjek_povecanja / brojac) / ln(1 / P0);
   za i := 1 do S radi {
        za j := 1 do KTL * N2 radi {
             put2 := slucajno_susjedno_rjesenje(put);
             dput2 := duljina(put2);
             promjena := dput2 – dput;
             prihvati := FALSE;
             ako je promjena < 0 tada
                  prihvati := TRUE;
             inače
                  ako je exp(-promjena / c) > random[0,1]
                       tada prihvati := TRUE;
                  ako je prihvati := TRUE tada
                       put := put2;
                       dput := dput2;
             }
             c: = c * α
        }
kraj.
```

9.2. Implementacija algoritma

U implementaciji rješenja korišten je gore navedeni algoritam te je korisniku omogućeno unošenje sljedećih parametara:

- **N**: broj gradova u grafu
- **S**: broj koraka hlađenja (10 – 100)
- $\mathbf{P_0}$: vjerojatnost prihvaćanja lošijeg rješenja (0,7 – 0,8)
- **α**: faktor smanjenja temperature (0,5 – 0,99)
- **KTL**: koeficijent termalne ravnoteže (0,1 – 0,5)

Na temelju navedenih parametara moguće je utjecati na sam proces simuliranoga kaljenja. Serijska implementacija rješenja izrađena je na osnovi algoritma te su svi gradovi predstavljeni s klasom *Grad* koja kao članove sadržava koordinate pojedinoga grada.

9.2.1. Izvedba izravnog korištenja dretvi (klasa *Thread*)

Za paralelizaciju rješenja *TSP-a* koristeći klasu *Thread,* obavljene su promjene u osnovnom algoritmu rješenja. Radi što boljih performansi, koristilo se rješenje u kojem nije bila potrebna sinkronizacija dretvi ili bilo kakvo zaključavanje kritičnih dijelova kôda. Da bi se ovo ostvarilo, vanjska petlja izvršavanja je podijeljena u četiri dijela te se pojedini dijelovi dodjeljuju dretvama za izvršavanje. U donjem isječku programskoga kôda prikazano je samo stvaranje i pokretanje prve dretve:

```
Cooling prvi = new Cooling(N,S1,alpha, KTL, c, dput, ref D, ref put,ref put1);

Thread prviThread = new Thread(new ThreadStart(prvi.pokreniHlad));

prviThread.Start();

prviThread.Join();
```

Metoda *pokreniHlad* klase *Cooling* predstavlja sam proces hlađenja koji je dio simulacije kaljenja. Sav posao koji se obavljao unutar vanjske petlje originalnoga algoritma nalazi se u sklopu te metode. Nakon završetka izvođenja svih četvero dretvi, obavlja se usporedba rješenja do kojih je svaka dretva došla. Krajnje rješenje se odabire tako da se uzme ono rješenje čiji je put najkraći.

9.2.2. Izvedba pre-alokacijom radnih dretvi (klasa *ThreadPool*)

Izvedba algoritma koja koristi klasu *ThreadPool* je dosta slična varijanti koja je izvedena za klasu *Thread*. Kao i u prijašnjoj implementaciji, posao koji je potrebno izvršiti dijeli se na četiri dijela gdje se svaki dio dodjeljuje pojedinoj instanci klase *ThreadPool*. U donjem isječku prikazana je implementacija za samo jednu instancu klase *ThreadPool*:

```
counter = new CountdownEvent(4);

Cooling prvi = new Cooling(N, S1, alpha, KTL, c, dput, ref D, ref put, ref put1);

ThreadPool.QueueUserWorkItem(new WaitCallback(pokreniHladTP), new Cooling(N, S1, alpha,
KTL, c, dput, ref D, ref put, ref put1));

counter.Wait();
```

Za razliku od *Thread* izvedbe, ova izvedba ima prednost što je kraća te je i sama sinkronizacija dretvi bolje riješena. Umjesto da se koristi metoda *Join*, koristi se klasa *CountdownEvent* koja u biti predstavlja brojilo koje odbrojava sve do postavljene granice te se sinkronizacija potom završava i može se nastaviti s izvođenjem programa.

9.2.3. Izvedba korištenja proširenja programskog jezika

U odnosu na ostale dvije paralelne implementacije, izvedba algoritma koja koristi biblioteku *Parallel Extensions* zahtjeva najmanje izmjena osnovnoga algoritma. Umjesto vanjske *for* petlje izvođenja, korišten je konstruktor *Parallel.For* i to prilikom određivanja prosječnoga povećanja puta i prilikom postupka hlađenja, kao što je prikazano u programskom isječku koji slijedi:

```
Parallel.For(0, S, delegate(int i)
{
    //vanjska petlja
    for (int j = 0; j < (KTL * Math.Pow(N, 2)); j++)
    {
        //sadržaj unutarnje petlje
    }
    c = c * alpha;
});
```

Same granice petlje su postavljene od *0* do *S,* kao i u izvornoj *for* petlji, dok je treći argument delegat koji prenosi posao koji će pokušati obaviti paralelno. Može se primijetiti

da se za unutarnju petlju i dalje koristila osnovna *for* petlja umjesto konstruktora *Parallel.For*. Razlog korištenja *for* petlje je taj što se događa da dodatna razina paralelizacije dovodi do lošijih vremenskih performansi zbog premale količine posla za pojedinu paralelnu iteraciju. Zbog korištenja dodatne paralelizacije, samo vrijeme potrebno za stvaranje i sinkronizaciju dretvi trajalo bi više od posla koji bi pojedina dretva trebala obaviti.

9.3. Trajanje izvođenja implementacija

U tablici 9.3.1. prikazana su rješenja izvođenja svih četiriju implementacija rješenja problema trgovačkoga putnika. Testiranje vremenskoga izvođenja se izvodilo tako da se mijenjao broj gradova u grafu od 50 pa sve do 1000 gradova. Ostali parametri su postavljeni na sljedeće vrijednosti: S = 100, P_0 = 0,75, α = 0,75, KTL = 0,3.

Može se uočiti da za manji broja gradova dolazi da boljih rezultata za izvođenje paraleliziranih implementacija. Za broj od 50 gradova najbrža paralelna implementacija (*Thread*) je 202% brža od serijske implementacije. Ovaj odnos u trajanju nastavlja se i za izvođenja programa za 100, 200 i 500 gradova gdje je potom za 500 gradova najbrža paralelna implementacija (*ThreadPool*) 134% brža od osnovne serijske implementacije.

Iako je u primjerima množenja matrica i sortiranja najbolje rezultate imala implementacija u kojoj se koristila biblioteka *Parallel Extensions*, najbrža paralelna implementacija *TSP*-a bila je ona u kojoj je korištena klasa *Thread*. Kada se usporede rezultati rješenja samog problema trgovačkoga putnika, najkraće putove je najčešće imala varijanta u kojoj nije uključena paralelizacija dok su rezultati ostalih implementacija bili od 5% do 50% lošiji od rezultata serijske implementacije.

Implementacija	**t (s)**			
	Broj gradova			
	50	**100**	**200**	**500**
Serijska	1,3633927	6,3066264	30,1062735	299,8186740
Thread	0,4504694	2,22708589	11,9944660	129,3015750
ThreadPool	0,61412538	2,265589	11,9311346	128,0601521
Parallel Extensions	0,5994899	2,2247087	13,3312096	137,3398561

Tablica 9.3.1. Vremenski rezultati izvođenja rješenja problema trgovačkoga putnika

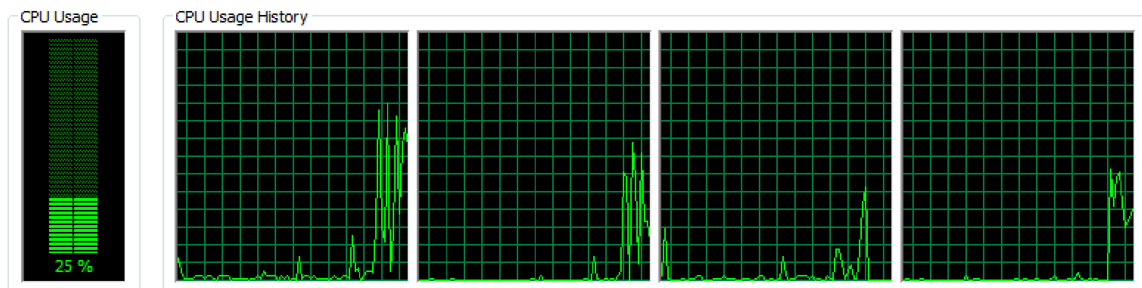
10. Analiza rezultata

Efekt paralelizacije programskih rješenja može se vidjeti direktno iz vremena izvršavanja ali i tijekom samog izvođenja.

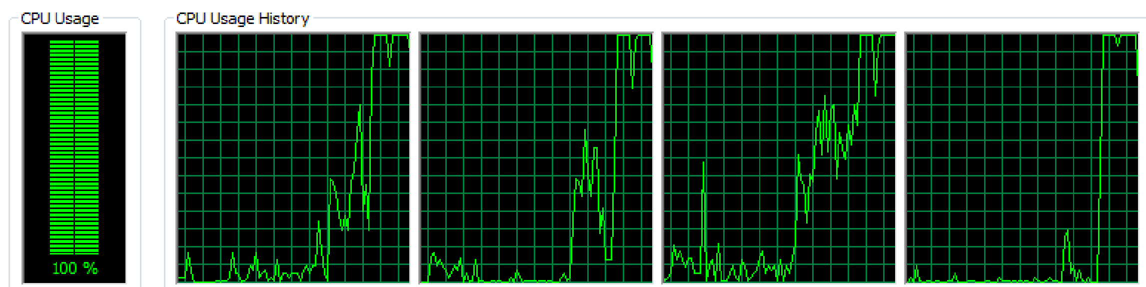
10.1. Analiza ubrzanja paralelnih implementacija

Na slikama 10.1. i 10.2. prikazano je opterećenje procesora tijekom izvođenja implementacija množenja matrica. Može se primijetiti da operacijski sustav prikazuje opterećenja za četiri procesorskih jezgri iako postoje samo dvije fizičke jezgre. Druge dvije su u biti virtualne jer procesor na kojemu su se implementacije izvršavale posjeduje tehnologiju *Hyper-Threading*.

Na slici 10.1. prikazano je opterećenje procesora prilikom serijskog množenja matrica. Tijekom izvođenja programa, iskorištava se najviše 25% mogućnosti procesora. Za isti problem množenja matrica, na slici 10.2. prikazano je stanje procesora tijekom izvođenja paralelnih implementacija (*Parallel Extensions*). Odmah nakon pokretanje implementacije, iskorištenost procesora dostiže razinu od 100% što prikazuje uspješno iskorištavanje dostupnih resursa te opravdava znatno bolje vremenske rezultate.

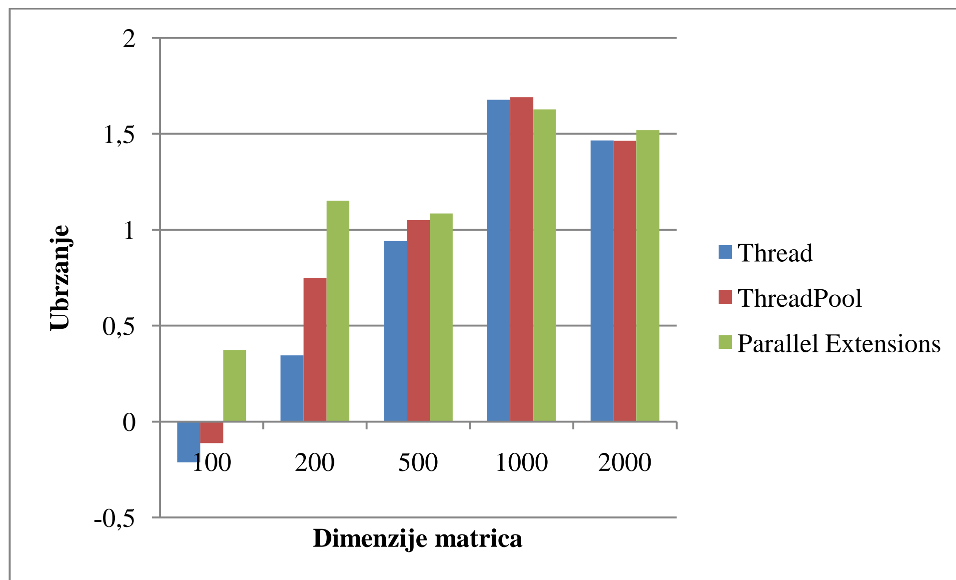


Slika 10.1.1. Prikaz opterećenja procesora prilikom serijskog množenja matrica



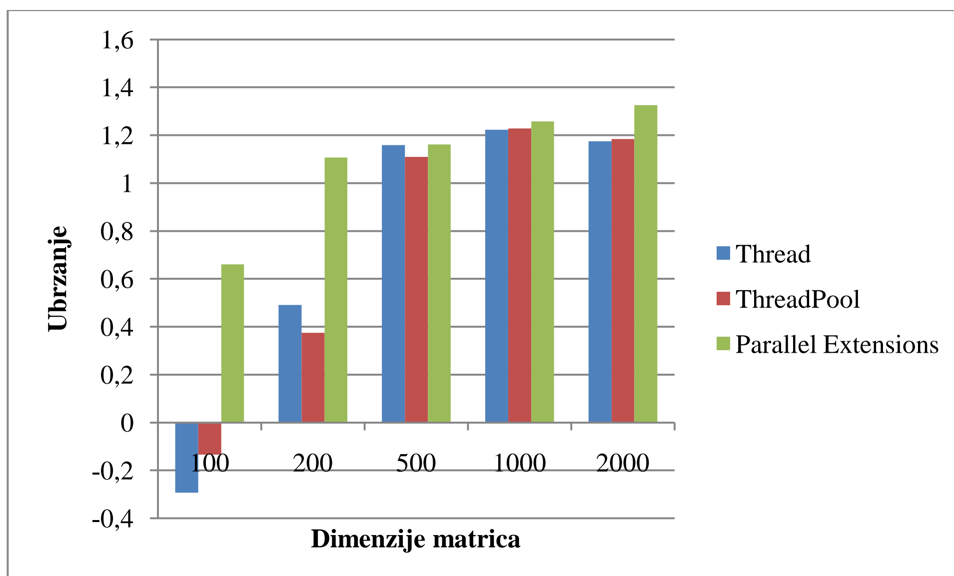
Slika 10.1.2. Prikaz opterećenja procesora prilikom paralelnog množenja matrica

Na slici 10.3 prikazana je usporedba ubrzanja paralelnih implementacija u odnosu na serijske implementacije osnovnoga algoritma množenja matrica. Može se vidjeti da se za matrice dimenzija 100×100 jedino implementacija koja koristi *Parallel Extensions* izvršava brže od serijske implementacije. U ostalim usporedbama sve su paralelne implementacije brže od serijskih i može se primijetiti da nakon dostignuća ubrzanja od 1,69 za matrice 1000×1000, stupanj ubrzanja pada ispod 1,5 za matrice 2000×2000. Iz slike se može vidjeti da za sve slučajeve korištenja osim za dimenzije 1000×1000, najbolje rezultate je imala implementacija u kojoj je korištena biblioteka *Parallel Extensions.*



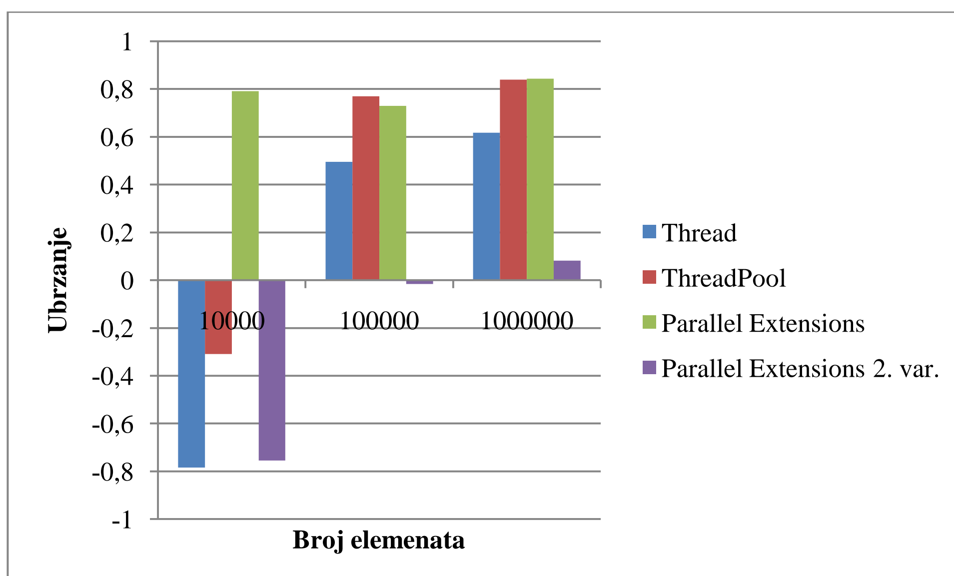
Slika 10.1.3. Rezultati izvođenja osnovnoga algoritma množenja matrica

Usporedba ubrzanja izvođenja poboljšanog algoritma množenja matrica prikazana je na slici 10.4. Kao i s osnovnim algoritmom ubrzanje svih implementacija se ostvaruje tek s množenjem matrica dimenzija 200×200 i veće. Može se primijetiti da je postignuto ubrzanje niže od ubrzanja postignutog tijekom izvođenja implementacija osnovnog algoritma. U većini slučajeva najbolje rezultati dobili su se od implementacije u kojoj je korištena biblioteka *Parallel Extensions*, iako su u zadnja tri slučaja po rezultatima sve tri paralelne implementacije dosta izjednačene.



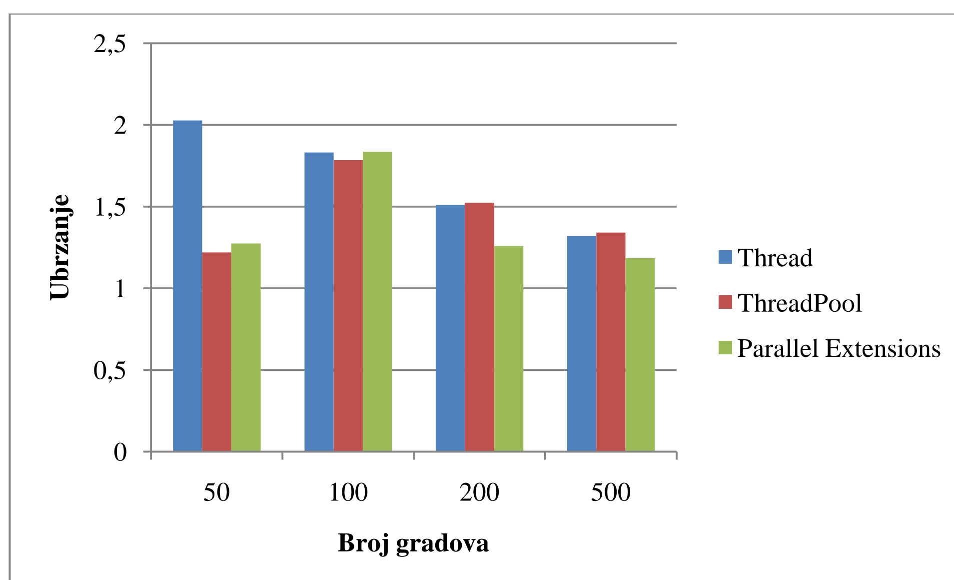
Slika 10.1.4. Rezultati izvođenja poboljšanoga algoritma množenja matrica

Na slici 10.5. prikazano je ubrzanje postignuto od strane paralelnih implementacija u odnosu na serijsku implementaciju izvođenja algoritma sortiranja *Quicksort.* Za prvi testni slučaj od 10000 elemenata jedino je implementacija u kojoj je korištena biblioteka *Parallel Extensions* imala pozitivni rezultat. Taj odnos se mijenja za ostale implementacije u zadnja dva testna slučaja. Može se primijetiti da je druga varijanta implementacije *Parallel Extensions* imala znatno lošije rezultate u odnosu na ostale paralelne implementacije. Iz tog se rezultata može vidjeti da dodatna paralelizacija može imati i negativan efekt na vremenske rezultate izvođenja.



Slika 10.1.5. Rezultati izvođenja algoritma Quicksort

Rezultati izvođenja i ostvareno ubrzanje paralelnih implementacija rješenja problema trgovačkog putnika sa simuliranim kaljenjem prikazano je na slici 10.6. Za razliku od prijašnja dva primjera ubrzanje se ostvaruje u svim testnim slučajevima. Dodatna razlika u odnosu na prijašnje primjere je ta da najbrža implementacija nije ona u kojoj je korištena biblioteka *Parallel Extensions,* nego implementacije u kojima su korištene klase *Thread* i *ThreadPool*. Također se primijetiti i postepeno usporavanje implementacija sa 100 pa na 200 pa na 500 gradova što se može objasniti da za sve veći broj gradova problem postaje složeniji te je potrebno više ulazno/izlaznih operacija kao i više sinkronizacija dretvi što dodatno troši vrijeme.



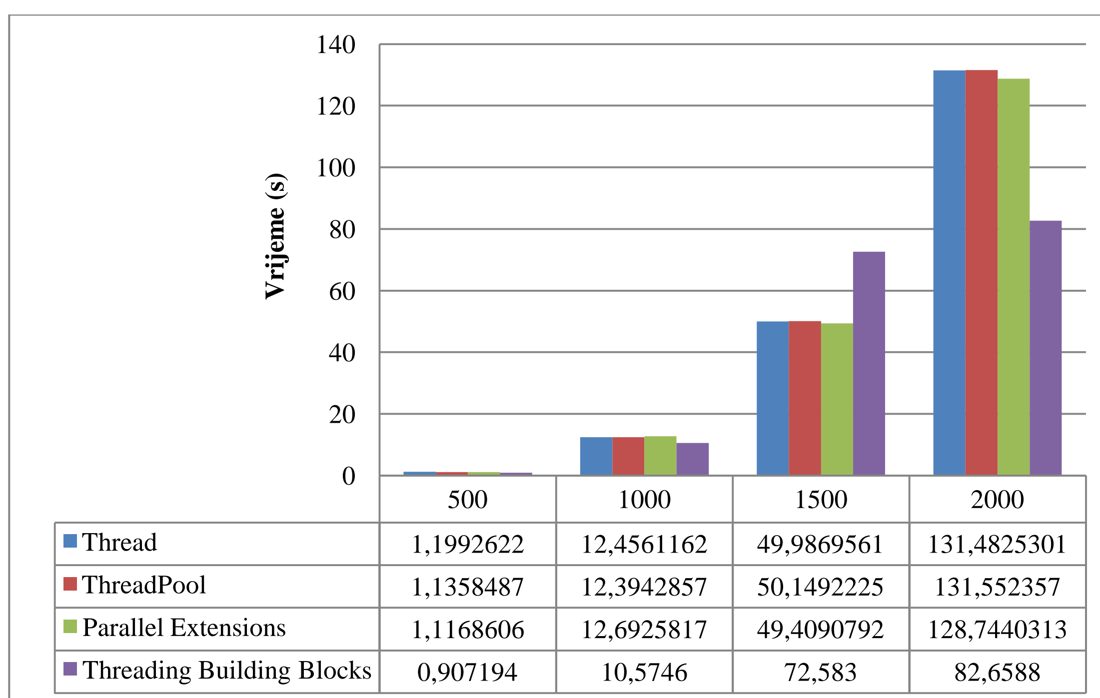
Slika 10.1.6. Rezultati izvođenja rješenja problema trgovačkoga putnika

Može se zaključiti da se od svih implementacija, najbolji rezultati dobiveni od implementacija koje su koristile biblioteku *Parallel Extensions*. Dodatne prednosti poput lakšeg korištenja te automatskoga prilagođavanja broju dostupni jezgri čine izbor biblioteke boljim izborom za uvođenje osnovne paralelizacije u programska rješenja. Ako su potrebni dodatni mehanizmi poput zaključavanja i sinkronizacije te direktan rad s dretvama onda su, iako nešto sporiji, bolji izbor klasa *Thread* ili klasa *ThreadPool.*

10.2. Usporedba rezultata paralelnih implementacija i biblioteke TBB

Za dodatnu analizu rezultata usporedila su se vremena izvođenja paralelnih implementacija množenja matrica s implementacijom koja je izrađena u jeziku C++. Na

slici 10.2.1. prikazani su vremenski rezultati tri mehanizma iz jezika C# te vremenski rezultati implementacije koja koristi biblioteku *Intel Threading Building Blocks*. U svim implementacijama korišten je isti osnovni algoritam množenja matrica te su prikazani rezultati izvođenja za matrice dimenzija 500×500, 1000×1000, 1500×1500 i 2000×2000. Može se primijetiti da *TBB* implementacija ima bolje rezultate za sve slučajeve osim u slučaju množenja matrica s dimenzijama 1500×1500. Ovi bolji rezultati vezani su uz dva moguća razloga. Prvi razlog je činjenica da su implementacije izrađene različitim programskim jezicima. Kada se izvršava visoko optimizirani programski kôd, jezik C++ ima prednost u trajanju izvođenja. Drugi razlog boljih rezultata je korištenje biblioteke *TBB* koja je posebno izrađena za primjenu na višejezgrenim procesorima te posjeduje veći skup mehanizama za raspodjelu posla u zadaće i uravnoteživanje opterećenja.



	500	1000	1500	2000
Thread	1,1992622	12,4561162	49,9869561	131,4825301
ThreadPool	1,1358487	12,3942857	50,1492225	131,552357
Parallel Extensions	1,1168606	12,6925817	49,4090792	128,7440313
Threading Building Blocks	0,907194	10,5746	72,583	82,6588

Slika 10.2.1. Rezultati izvođenja množenja matrica za C# i C++ implementacije

Zaključak

Višeprocesorski sustavi danas su sastavni dio sve većeg broja računala. Da bi se iskoristile prednosti korištenja tih sustava, potrebno je prilagoditi programska rješenja za paralelni rad. Pokazalo se da jezik C# posjeduje više mehanizama za primjenu pojma višedretvenosti u programskom kôdu. Na primjerima množenja matrica, sortiranja polja s algoritmom *Quicksort* i rješavanja problema trgovačkoga putnika prikazana je primjena tih mehanizama te njihovi vremenski rezultati izvođenja. Korištenje ugrađenih mogućnosti jezika C# za višedretveni rad može se doći do boljega iskorištavanja procesora što dovodi do znatno boljih vremenskih rezultata.

Od svih dostupnih mehanizama najbolji rezultati dobiveni su od implementacija koje su koristile biblioteku *Parallel Extensions*. Uz vrlo dobre rezultate, pokazalo se da je korištenje same biblioteke bilo vrlo jednostavno te se zbog te jednostavnosti korištenja može lako primijeniti u nova, ali i u postojeća programska rješenja. Dolaskom procesora sa sve više jezgri, doći će do izražaja i ugrađena funkcionalnost automatskoga prilagođavanja i iskorištavanja svih dostupnih jezgri za izvođenje aplikacija.

Iako je biblioteka *Parallel Extensions* davala najbolje rezultate, ostali mehanizmi poput klasa *Thread* i *ThreadPool* daju dobre rezultate te posjeduju dodatne prednosti za višedretveni rad. S pomoću dostupnih mehanizama u jeziku C# moguće je jednostavno i efikasno iskoristiti prednosti rada s višeprocesorskim sustavima.

Literatura

[1] TITUS, T., FERRACCHIATI, F., REDKAR, T., SIVAKUMAR, S. *C# Threading Handbook*. New York: Apress, 2004.

[2] TITUS, T. et al. *Pro.NET 1.1: Remoting, Reflection, and Threading*. New York: Apress, 2005.

[3] MAYO, J. C# 3.0: *Unleashed with the .NET Framework 3.5*. 2. izdanje. Indianapolis: SAMS, 2008.

[4] WATSON, B. *C# 4.0: How-To*. 1. izdanje. Indianapolis: SAMS, 2010.

[5] KECKLER, S., OLUKOTUN, K., HOFSTEE, H. P. *Multicore Processors and Systems*. New York: Springer, 2009.

[6] JAKOBOVIĆ, D. *Analiza i projektiranje računalom: Simulirano kaljenje*, 13.4.2004, http://www.fer.hr/_download/repository/Simulirano_kaljenje.pdf. 15.4.2010.

[7] NAGEL, C., EVJEN, B., GLYNN, J., SKINNER, M., WATSON, K. *Wrox Professional C# 2008*. Indianapolis: Wiley Publishing, 2008.

[8] DEITEL, H., DEITEL, P., LISTFIELD, J., NIETP, T., YAEGER, C., ZLATKINA, M., *C# A Programmer's Introduction.* Upper Saddle River: Prentice Hall, 2002.

[9] MICROSOFT, *.NET Framework 4*. http://msdn.microsoft.com/en-us/library/w0x726c2%28v=VS.100%29.aspx, 12.5.2010.

[10] SAUNDERS, J. *An Introduction to the Microsoft .NET Platform.* 29.1.2008, http://geekswithblogs.net/jsaunders/articles/an-introduction-to-the-microsoft-.net-platform.aspx. 18.4.2010.

Naslov

Podrška jezika C# višeprocesorskom programiranju

Sažetak

Uz uvođenje višejezgrenih procesora, u osobnim računalima nikada nije bilo većih potreba za razvoj paralelnih i višedretvenih aplikacija. Da bi se potpuno iskoristile mogućnosti višeprocesorskih sustava, programeri imaju potrebu za alatima koji će im omogućiti stvaranje višedretvenih aplikacija jednostavnije i s više lakoće. Objektno orijentirani programski jezik C# ima ugrađene mehanizme koji omogućavaju razvoj aplikacija pogodnih za višeprocesorske arhitekture. Ti će mehanizmi u ovome radu biti predstavljeni i objašnjeni. Nadalje, primjena ovih mehanizama na nekoliko reprezentativnih problema paralelnoga programiranja bit će detaljno predstavljena i raspravljana. Rezultati izvršenja tih programa i serijskih implementacija usporedit će se i analizirati na kraju ovoga rada.

Ključne riječi

C#, dretva, višedretvenost, .NET Framework, zadaća, višejezgreni procesori, paralelno programiranje

Title

Support for Multiprocesor Programming in C#

Summary

With the introduction of multi-core processors in personal computers never has there been a greater opportunity and need for the development of parallel and multithreaded applications. In order to fully take advantage of multiprocessor systems, developers have a need for tools which will enable them to create multithreaded applications with a greater degree of ease and simplicity. The object oriented C# programming language has built in mechanisms which aid the development of applications suited for multiprocessor architectures. In this paper these mechanisms will be presented and explained. Furthermore the application of these mechanisms on several representative problems of parallel programming will be presented and discussed in detail. At the end of this paper the execution results of these applications and the non-parallel versions will be compared and analyzed.

Keywords

C#, thread, multithreading, .NET Framework, task, multicore procesors, parallel programming