УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за рачунарство и аутоматику
Одсек за рачунарску технику и рачунарске комуникације

ДИПЛОМСКИ (BACHELOR) РАД

Кандидат: **Миле Радовановић**
Број индекса: **11359**

Тема рада: **Једно решење паралелног прорачуна интеграла дате функције**

Ментор рада: **Мирослав Поповић**

Нови Сад, мај 2016.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Дипломски (Bachelor) рад
Аутор, **АУ**:		**Миле Радовановић**
Ментор, **МН**:		**Мирослав др. Поповић**
Наслов рада, **НР**:		**Једно решење паралелног прорачуна интеграла дате функције**
Језик публикације, **ЈП**:		Српски / латиница
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		**2016**
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		**4/28/0/6/10/18/0**
Научна област, **НО**:		Електротехника и рачунарство
Научна дисциплина, **НД**:		Рачунарска техника
Предметна одредница/Кључне речи, **ПО**:		**системска програмска подршка, кластери, паралелизација, интеграл**
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		**У овом пројекту је реализована паралелизација математичког прорачуна интегрaљења на кластеру рачунара**
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	**Драган Кукољ**
	Члан:	**Бранислав Атлагић** — Потпис ментора
	Члан, ментор:	**Мирослав Поповић**

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Bachelor Thesis	
Author, **AU**:		**Mile Radovanović**	
Mentor, **MN**:		**Miroslav Popović, PhD**	
Title, **TI**:		**Parallelization of mathematical calculation of function's integral**	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		**2016**	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		**4/28/0/6/10/18/0**	
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S/KW**:		**Parallelization, cluster, Task Tree Executor**	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		**In this project parallelization of algorithm for integration is realized on computer cluster.**	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	**Dragan Kukolj**	
	Member:	**Branislav Atlagić**	Menthor's sign
	Member, Mentor:	**Miroslav Popović**	

SADRŽAJ

1. Uvod....1
2. Poglavlje....2
2.1 Teorijske osnove....2
2.1.1 Task Tree Executor....2
2.1.2 Podela zadataka i izrada stabla zadataka....3
2.1.3 Protokol prenosa podataka....5
2.2 Koncept rešenja....5
2.2.1 Prevođenje TTE na Linux....5
2.2.2 Analiza problema....6
2.2.3 Dizajniranje modula i algoritama....7
2.3 Opis rešenja....9
2.3.1 Osnovni modul....9
2.3.2 Modul raspoređivača....12
2.3.3 Modul UDP protokola....17
2.3.4 Modul zadatka....18
2.3.5 Modul predefinisanih konstanti....18
2.4 Ispitivanje....19
3. Zaključak....24
4. Literatura....25

SPISAK SLIKA

Slika 2.1 Primer stabla zadataka 2
Slika 2.2 Površina koju zaklapa funkcija sa x-osom 3
Slika 2.3 Različite preciznosti pri integraljenju 4
Slika 2.4 Funkcija u opsegu [-4,4] 4
Slika 2.5 Funkcija u opsegu [0,25] 4
Slika 2.6 Funkcija u opsegu [0,1000] 5
Slika 2.7 Primeri raspodele zadataka 6
Slika 2.8 Blok dijagrami main (levo) i interfejs (desno) niti 7
Slika 2.9 Blok dijagrami prihvatilac (levo) i raspoređivač (desno) 8
Slika 2.10 Izgled UTP paketa 13
Slika 2.11 Grafikon vremena obrade za preciznost $1/2^{10}$ 20
Slika 2.12 Grafikon vremena obrade za preciznost $1/2^{11}$ 20
Slika 2.13 Grafikon vremena obrade za preciznost $1/2^{12}$ 21
Slika 2.14 Grafikon vremena obrade za preciznost $1/2^{13}$ 21
Slika 2.15 Grafikon vremena obrade za preciznost $1/2^{14}$ 22
Slika 2.16 Grafikon vremena obrade za preciznost $1/2^{15}$ 22
Slika 2.17 Grafikon vremena obrade za preciznost $1/2^{16}$ 22
Slika 2.18 Grafikon vremena obrade za preciznost $1/2^{17}$ 22
Slika 2.19 Grafikon vremena obrade za preciznost $1/2^{18}$ 22
Slika 2.20 Grafikon vremena obrade za preciznost $1/2^{19}$ 22
Slika 2.21 Grafikon vremena obrade za preciznost $1/2^{20}$ 23

SPISAK TABELA

Tabela 2.1 Rezultati za dubine 10, 11 i 12 19
Tabela 2.2 Rezultati za dubine 13, 14, 15 i 16 20
Tabela 2.3 Rezultati za dubine 17, 18, 19 i 20 21

SKRAĆENICE

TTE - *Task Tree Executor, izvr*šenje stabla zadataka

CPU - *Central Processor Unit*, Centralna izvršna jedinica

1. Uvod

Izrada veoma velikih distribuiranih sistema poput eketro-mreže veleg grada ili čitave regije, oduvek se smatralo za veliki poduhvat u oblasti inžinjeringa sistema zasnovanih na računaru. Danas, takvi sistemi obrađuju desetine miliona ulaznih vrednost, koristeći uglavnom složene matematičke proračune. Uzevši u obzir činjenicu da su današnji procesori dostigli granicu uvećanja radnog takta, počelo se sa istraživanjem novih tehnoloških mogućnosti. Jedno od mogućnosti rešenja dolazi sa pojavom višejezgarnih procesora i prilagođenja programske podrške na njima. Idući korak dalje, raspodelom zadataka na više računara u mreži, može se postići veliko ubrzanje obrade matematičkih proračuna.

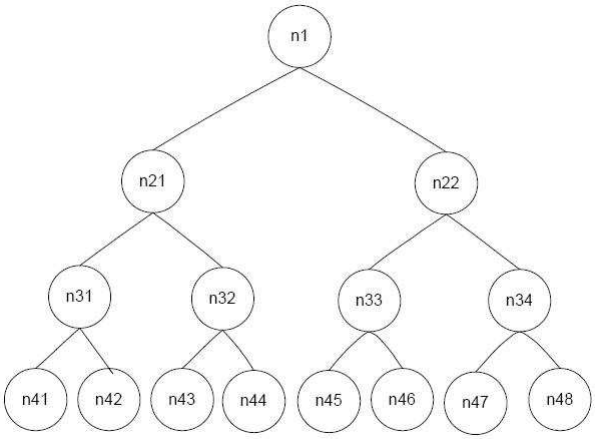
Cilj ovog projekta je paralelizacija i raspodela zadataka na više računara složenog matematičkog proračuna u vidu izračunavanja integrala velike preciznosti nad zadatom funkcijom nad zadatim domenom. Postojeće programsko rešenje profesora M. Popovića, za paralelno izvršenje prvobitno sekvencijalnih programa, na višejezgarnom procesoru, je iskorišteno u rešenju ovog projekta. Analizom problema zadatka je uočeno četiri celine. Prvi korak u rešavanju problema je izmena i prilagođenje izvornog koda TTE biblioteke za izvršenje na linux operativnom sistemu. Drugi korak predstavlja izmenu i prilagođenje TTE koncepta za rad na mreži računara. Zatim je prilagođeno izvršenje matematičkog proračuna na mreži računara. Poslednji korak podrazumeva ispitivanje brzine izvršenja programa i ubrzanja u odnosu na referentno rešenje koje se izvršava sekvencijalno, na jednom računaru.

2. Poglavlje

2.1 Teorijske osnove

2.1.1 Task Tree Executor

Ideja TTE-a je zasnovana na refaktorizaciji originalnog koda, raspodeli obrade na delove koji se nazivaju zadaci, i kreiranju grafa zadataka, tj stabla zadataka (slika 1). Za razliku od sekvencijalnog koncepta gde je jedan zadatak zadužen za celu obradu, dok kod TTE-a svaki zadatak izvršava samo deo obrade za koji je zadužen. TTE može izvršavati graf (stablo) na dva načina: odozgo-nadole (eng. top-down) i odozdo-nadole (bottom-up). Nivoi u stablu su međusobno zavisni. Ukoliko se izvršava npr odozdo-nagore, onda je prvo potrebno izvršiti donji nivo, da bi sledeći mogao nastaviti sa izvršenjem. Stablo je moguće paralelno izvršavati po nivoima, što znači da se mogu svi zadaci u okviru jednog nivoa u isto vreme izvršavati.



Slika 2.1 Primer stabla zadataka

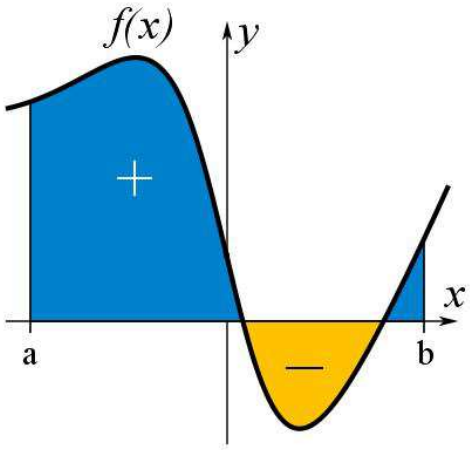
2.1.2 Podela zadataka i izrada stabla zadataka

U osnovi, ovo rešenje je zasnovano na deljenu domena funkcije na jednake delove, i obrada svakog dela predstavlja jedan zadatak. Pošto je izvršenje zadataka međusobno nezavisno, stablo ima samo jedan nivo, i svi zadaci se mogu paralelno izvršavati. Rezultat proračuna jednog zadatka u stablu predstavlja površinu koju funkcija zaklapa na intervalu za koji je zadužen dati zadatak. Zbir rezultata svih zadataka predstavlja konačan rezultat programa.

Računanje određenog integrala nad funkcijom f(x) u intervalu [a,b] predstavlja implementiranje sledeće funkcije:

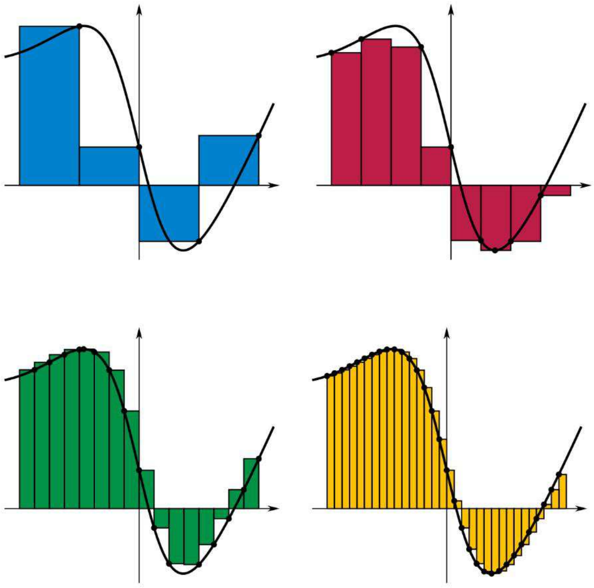
$$\int_a^b f(x)dx = F(b) - F(a)$$

Rezultat ovog proračuna praktično predstavlja površinu koju funkcija zaklapa sa x-osom. Ukoliko se funkcija nalazi iznad x-ose, onda se sabira sa rezultatom a ukoliko je ispod, onda se površina oduzima od ukupne, kao što je naznačeno na slici 2.2.



Slika 2.2 Površina koju zaklapa funkcija sa x-osom

Da bi se programski ova površina mogla izračunati, mora se računati u više iteracija. Potrebno je izdeliti x-osu na jednake delove, i za svaki od tih manjih intervala izračunati integral, dobijajući na taj način niz pravougaonih površina. Zbir tih površina predstavlja celokupan rezultat. Sa većim brojem iteracija,tj sa smanjenjem veličine pomeraja po x-osi dobijamo veći broj površina i veću preciznost. To je slikovito prikazano na slici 2.3. Možemo videti da je površina koju funkcija zaklapa sa x-osom najpribližnija zbiru površina žutih pravougaonika. Teoretski, kada bi veličina pomeraja težila nuli, rezultat bi bio najprecizniji moguće. To bi značilo da bi bilo beskonačno mnogo proračuna, pa je za realizaciju programskog rešenja na računaru potrebno pronaći kompromis između preciznosti proračuna i raspoloživih resursa u obliku vremena.

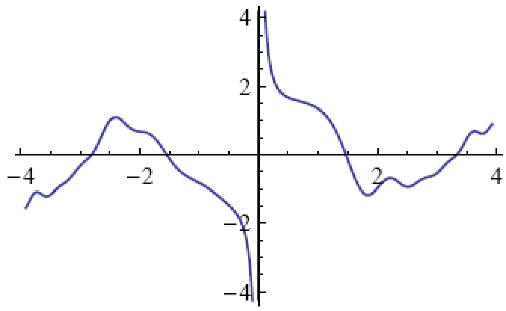


Slika 2.3 Različite preciznosti pri integraljenju

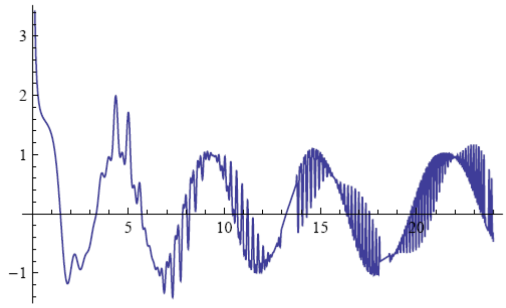
Sledeća funkcija je korištena u programu jer je dovoljno kompleksna i troši puno procesorskog vremena, a to upravo ono što nam treba:

$$\frac{1}{2}\left(\cos\left(a^2\right)\sin^2(a)-\sin(a)\cos\left(a^2\right)\right)+\sin\left(a\sqrt{\log\left(\frac{a}{100}\right)}\right)+\frac{\log\left(e^{\frac{1}{a}}\right)}{\log(10)}$$

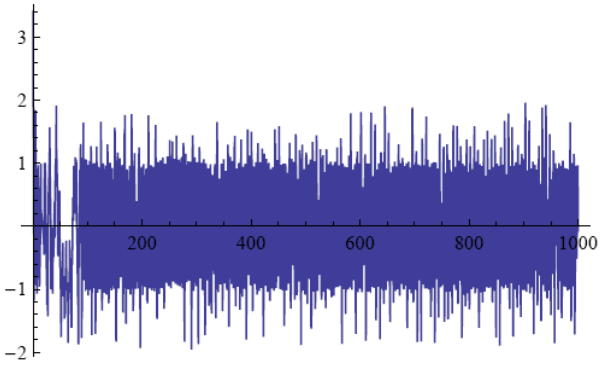
Grafici funkcije nad različitim opsezima:



Slika 2.4 Funkcija u opsegu [-4,4]



Slika 2.5 Funkcija u opsegu [0,25]



Slika 2.6 Funkcija u opsegu [0,1000]

2.1.3 Protokol prenosa podataka

Za komunikaciju između računara korišten je UDP (eng. User Datagram Protocol) protokol jer je najpogodniji za naše rešenje. To je najjednostavniji protokol koji, za razliku od TCP/IP protokola ne zahteva stalnu vezu niti prethodno povezivanje dva računara, već se poruka šalje u proizvoljnom trenutku. Ne poseduje ni kontrolu poslatih poruka u vidu izveštaja računara koji prihvata poruku. Zbog tih osobina UDP protokol je veoma brz, što je najbitnije za naše rešenje, a njegove mane, i uopšteno, sigurnost prenosa podataka, se rešavaju pažljivim planiranjem toka programskog rešenja.

2.2 Koncept rešenja

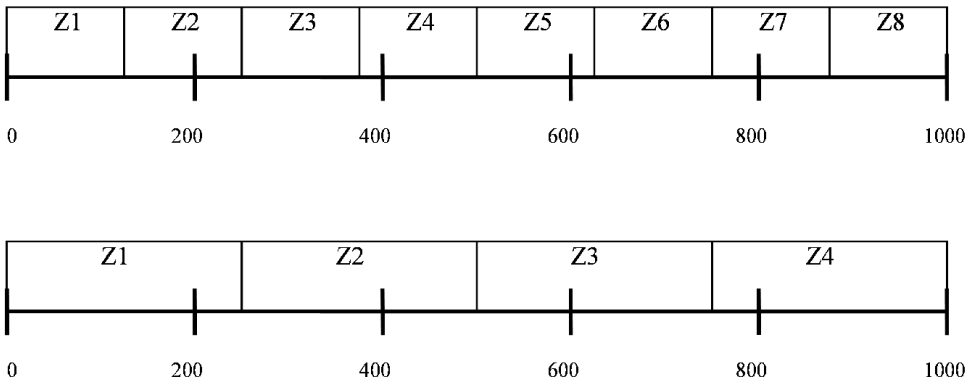
2.2.1 Prevođenje TTE na Linux

Obzirom da je originalna verzija TTE koda, od prof. Miroslava Popovića pisana za operativni sistem Windows, najpre je potrebno prevesti TTE za rad na operativnom sistemu Linux. Prednosti Linux-a u odnosu na Windows, koje se tiču ovog projekta su sledeće:

- Merenje vremena trajanja izvršenja programa. Na Linux platformi je moguće meriti vreme u nanosekundama za razliku od Windows platforme, na kojoj je preciznost merenja vremena 15ms. Za potrebe našeg programa, potrebno je meriti sa rezolucijom od bar 1ms.
- Rad sa nitima u Linux operativnom sistemu je efikasnije. Moguće je instancirati više niti, koje istovremeno troše manje sistemskih resursa.

2.2.2 Analiza problema

Osnovna problematika u primenjivanju TTE algoritma na konkretan zadatak, je način raspodele proračuna na više nezavisnih zadataka. Računanje integrala se može jednostavno raspodeliti jer se svaki poddomen može računati posebno, da bi se na kraju svi rezultati jednostavno sabrali i dobili tačan rezultat. Jednostavnom podelom čitavog domena na jednake delove, dobijamo poddomene koji će predstavljati zadatke. Da bi proračun bio što tačniji, potrebno je domen izdeliti na veliki broj tačaka. Pri testiranju, najmanja preciznost je iznosila 1024 tačke, ali to ne znači da će postojati 1024 zadatka, jer bi to bilo previše, i svaki pojedinačni proračun bi bio previše jednostavan, te bi sam proračun kraće trajao od prenosa zahteva i rezultata preko UDP-a. Zbog toga je potrebno praviti zadatke od niza uzastopnih tačaka. Domen nad kojim će se vršiti obrada je predefinisan u telu programa. Ulazni parametri programa su preciznost obrade, ali kao stepen broja 2 (npr za 10, preciznost od 1024 tačke, za 15, preciznost od 32768) i broj zadataka. Poznavajući te parametre, program će razdeliti opsege automatski, na jednake delove i na osnovu njih praviti zadatke. Zadaci su identifikuju svojim rednim brojem i dovoljno je u zahtevu navesti identifikacioni broj zadatka da bi računar klijent znao koji je domen potrebno obraditi.



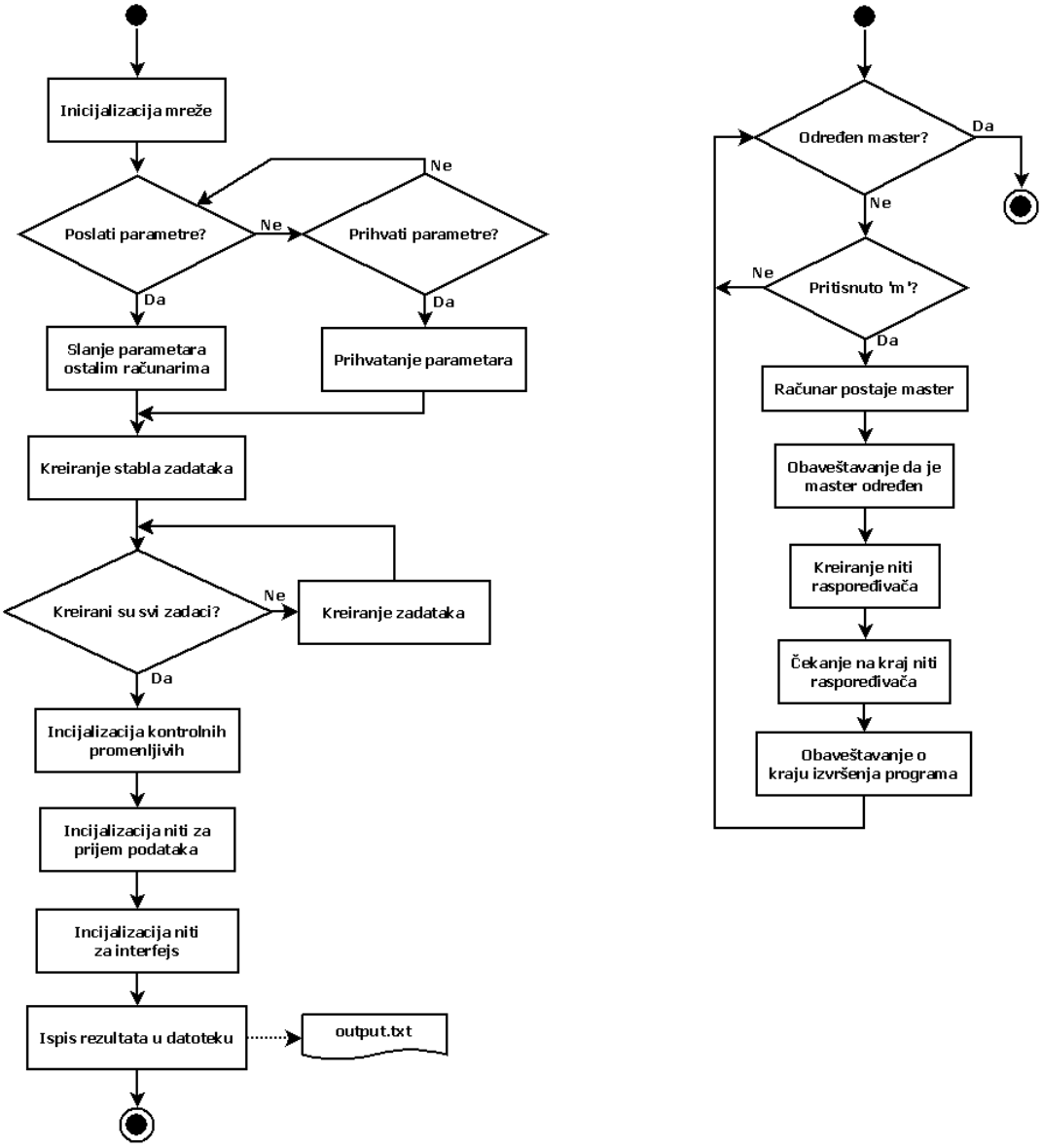
Slika 2.7 Primeri raspodele zadataka

Na slici 2.7 su prikazana dva primera raspodele zadataka, gde je preciznost računanja, tj broj tačaka 1000. Na gornjem delu slike, domen je podeljen u osam zadataka, što znači da jedan zadatak sadrži 1000/8 = 125 tačaka. Prvi zadatak obrađuje tačke od 0-125, drugi 125-250 itd. Na donjem delu slike, ulazni parametar za broj zadataka je četiri, pa jedan zadatak sadrži 250 tačaka za obradu.

2.2.3 Dizajniranje modula i algoritama

Moduli u našem rešenju su nasleđeni iz originalnog TTE koda.

Osnovni (eng. main) modul je zadužen za inicijalizaciju mreže u učitavanje IP adresa računara koji učestvuju u proračunu, iz tekstuelne datoteke. Kada su na svim računarima pokrenuti programi, potrebno proslediti parametre proračuna svim računarima, sa bilo kog računara. Parametri predstavljaju dubinu proračuna, tj preciznost, i ukupan broj zadataka, broj na koji će čitav proračun biti podeljen. Parametri se unosi kao parametri programa. Ukoliko se ne navedu, koriste se podrazumevane vrednosti. Kada jedan računar prosledi ostalima svoje parametre, svi računari kreiraju stablo zadataka i zadatke u njima. Glavni modul još pokreće nit za prihvatanje podataka od ostalih računara u mreži, i nit za početni interfejs. Po završetku celokupnog proračuna, ovaj modul će rezultate ispisati u izlaznu datoteku. Algoritam osnovnog modula je prikazan na slici 2.8, levo.

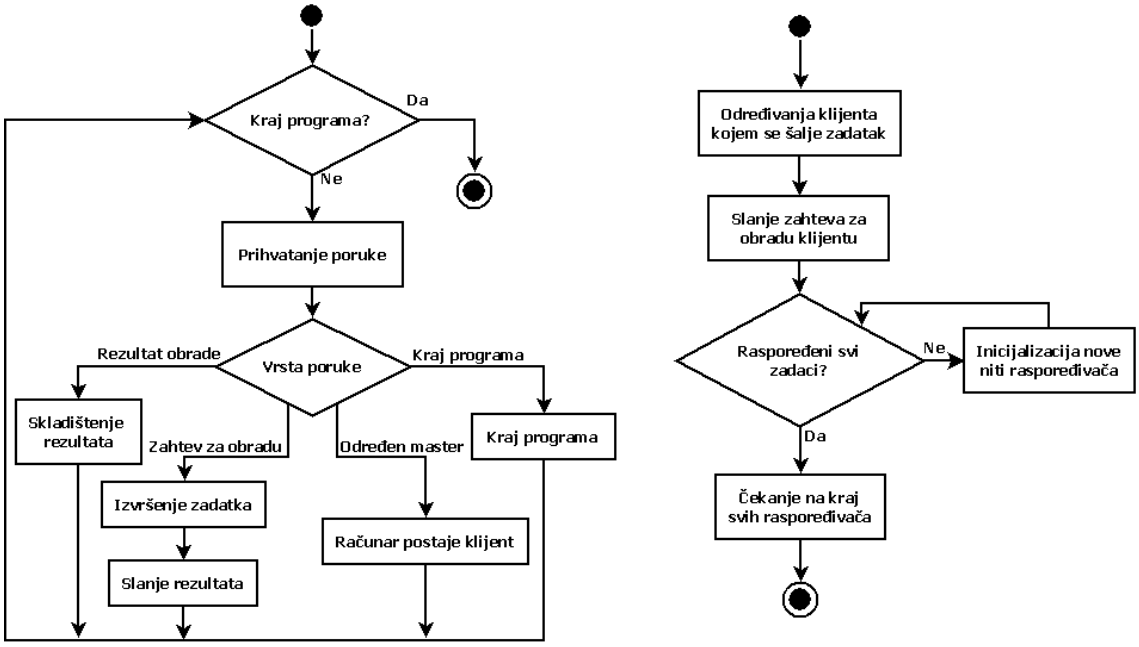


Slika 2.8 Blok dijagrami main (levo) i interfejs (desno) niti

Modul zadužen za raspoređivanje niti, TaskScheduler sadrži tela niti za interfejs, za prihvat podataka i nit za koordiniranje i kontrolu raspoređivanja zadataka. Interfejs funkcija (eng. Interface) se pokreće kada su svi računari spremni za početak rada i potrebno je odrediti računar koji će biti glavni (eng. Master), koji će raspoređivati zadatke i vršiti kontrolu. Nakon što se glavni računar odredi, pritiskom na taster ‘m’ na tastaturi, on obaveštava ostale računare o svom statusu, i ostali računari postaju klijenti, i postaju izvršioci zadataka. Nakon što je računar određen kao glavni, pokreće nit za raspoređivanje zadataka, i čeka da se ta ista nit završi. Završetak te niti označava kraj proračuna i nakon nje nit interfejs obaveštava sve računare u mreži da je proračun završen i da se mogu isključiti. Algoritam je prikazan na slici 2.8, desno.

Istovremeno sa pokretanjem niti interfejs, inicijalizuje se i nit za prihvatanje podataka (slika 2.9, levo). Ona je zadužena za osluškivanje mrežnog interfejsa i interpretaciju UDP paketa koji su pristigli. Nakon što se prepozna postojanje UDP poruke, određuje se tip poruke. Tip poruke može biti:

- Zahtev za obradu – računar dobija redni broj zadatka koji je potrebno obraditi i nadalje se pokreće obrada određenog zadatka, i po završetku se rezultat prosleđuje glavnom računaru
- Rezultat obrade – po prijemu rezultata, potrebno ga je smestiti na određeno mesto u memoriji glavnog računara
- Određen master – ova poruka označava da je odabran glavni računar i da se trenutni računar postavlja kao klijent, zadužen samo za obradu zadataka. Prepoznaje se adresa pošaljioca poruke i pamti se u memoriji kao adresa glavnog računara
- Kraj programa – obaveštenje o završetku programa



Slika 2.9 Blok dijagrami prihvatilac (levo) i raspoređivač (desno)

Nit raspoređivača je najvažnija u ovom modulu. Ona je zadužena za raspoređivanje zadataka po računarima u mreži. Kreira se paket sa zahtevom za obradu i identifikacionim brojem zadatka, i prvom klijentu u listi klijenata se prosleđuje UDP paket. Kreiraju se niti za svaki zadatak koji je poslat klijentu i svaka nit ponaosob čeka da odgovarajući klijent vrati rezultat svoje obrade. Kada svaka nit dočeka odgovor od svog klijenta, završava se osnovna nit raspoređivača. Blok dijagram raspoređivača je prikazan na slici 2.9, desno.

2.3 Opis rešenja

U ovom poglavlju su pojašnjeni moduli, pojedinačno, način funkcionisanja svakog od njih kao i celokupnog rešenja. Moduli koje sadrži projekat su:

- *main* – osnovni modul
- *TaskScheduler* – raspoređivač zadataka
- *UDPfunc* – funkcionalnost UDP protokola
- *Task* – zadatak
- *Defines* – predefinisane konstante

Pre početka opisa ovih modula, prikazaćemo *makefile* koji je korišten za prevođenje kompletnog rešenja:

```
main.bin: main.o TaskScheduler.o
  g++  UDPfunc.cpp Task.cpp main.cpp TaskScheduler.cpp -pthread -lrt -o test

clean:
  rm -f *.o udp_time.bin
```

Argumenti `-pthread` i `-lrt` služe da bi se omogućilo prevođenje koda koje sadrži rad sa nitima i za rad sa merenjem vremena izvršenja dela koda, respektivno.

2.3.1 Osnovni modul

Osnovni modul se nalazi u *main.cpp* datoteci i odatle kreće izvršenje programskog rešenja. Tu je implementirano prenošenje parametara sa komandne linije, kreiranje stabla zadataka, inicijalizacija TTE koda i na kraju, ispis rezultata u izlaznu datoteku. Takođe u ovom modulu se nalate globalne promenljive koje služe za povezivanje između modula, zatim programska funkcija u kojoj je implementirana matematička funkcija nad kojom će se vršiti računanje integrala, kao i programska funkcija koja realizuje računanje integrala nad proizvoljnom funkcijom. I još je vredna pomena zasebna pomoćna nit koja realizuje prihvatanje parametara od nekog računara iz mreže.

Pokretanje programa je moguće bez parametara, gde se za paremetre uzimaju vrednosti koje su predefinisane u *defines.h*. Nakon toga se inicijalizuje mreža pomoću funkcije *InitializeNetwork()*, koja je implementirana u drugom modulu i o tome će biti reči kasnije.

Nakon toga dolazi do razmene parametara između računara na sledeći način:

```
pthread_create(&receiveParamsHandle, NULL, &ReceiveParams, NULL);
    while (!paramsDone)
    {
        if (_kbhit() != 0)
            {
            pressed = getchar();
            if ((pressed == 's') || (pressed == 'S'))
            {
                paramsDone = true;
                memcpy( &tmp[0], (char *)(&numberOfTasks), NODE_ID_LENGTH );
                memcpy( &tmp[NODE_ID_LENGTH], (char *)(&depth), NODE_ID_LENGTH );
                UDPBroadcast(udp.sockfd, udp, &tmp[0], NODE_ID_LENGTH*2);
                UDPSend(udp.sockfd, udp.local, &pressed, 1);
            }
        }
        usleep(25000);
    }
void* ReceiveParams(void*)
{
    sockaddr_in rmt;
    int received;
    received = UDPReceive(udp.sockfd, rmt, inputParams, NODE_ID_LENGTH*2);
    if( received > 0 )
    {
         if(!paramsDone)
         {
             printf("\nParameters acquired!");
             printf("\n\tDepth %d (%d Calculations)\t\tNumber of tasks:
%d\t\tNumber of calculations per task: %d\n",depth, 1<<depth, numberOfTasks,
(1<<depth)/numberOfTasks );
             memcpy( &numberOfTasks, &inputParams[0], NODE_ID_LENGTH );
             memcpy( &depth, &inputParams[NODE_ID_LENGTH], NODE_ID_LENGTH );
             paramsDone = true;
         }
    }
}
```

Najpre se inicijalizuje nit *ReceiveParams* koja osluškuje UDP poruke koje pristignu za eventualno prihvatanje parametara od drugih računara u mreži. Istovremeno, se čeka unos sa tastature od korisnika da trenutni računar pošalje ostalima svoje parametre. Ukoliko se pritisne taster ‘s’, računar će poslati ostalima UDP paket koji sadrži parametre *nubmerOfTasks* i *depth*, kao i sebi, da bi se nit *ReceiveParams* ugasila. U oba slučaja, kada se prenos parametara dogodi, logička promenljiva *paramsDone* postaje istinita i nastavlja se sa izvršenjem programa.

Dalje, kreiramo graf stabla i uvezujemo zadatke u stablo:

```
TS_CreateTaskGraph(0,BU_CallBackF,TD_CallBackF,300);
for ( int i = 1; i < numberOfTasks; i++)
      TS_AddTask(0,i);
```

Funkciji *TS_CreateTaskGraph* prosleđujemo dva puta istu funkciju jer u ovom rešenju nema potrebe za obrađivanje stabla u dva smera, a poslednji argument predstavlja maksimalan broj kreiranja niti.

Konačno, dolazimo do iniciranja modula *TaskScheduler* funkcijom *TS_Init* i po završetku pozivamo *TS_Close*, kako bi se oslobodili resursi koji su zauzeti u toku rada modula *TaskScheduler*.

```
TS_Init();
TS_Close();
```

Poslednje zaduženje osnovnog modula je ispis rezultata u izlaznu datoteku, gde se rezultat svakog zadatka ponaosob upisuje u datoteku *output.txt* i konačni rezultat se ispisuje na ekran korisnika.

```
FILE* file;
file = fopen("output.txt", "w");
if (!file)
{
    fprintf(stderr, "\n\n Could not open file output.txt for reading!\n");
    return(-1);
}
finalResult=0;
for(int i=0; i<(numberOfTasks); i++)
{
    fprintf( file, "%f ", result[i]);
    finalResult += result[i];
}
printf("\nFinal result = %f\n",finalResult);
pthread_join(receiveParamsHandle, NULL);
fclose(file);
```

2.3.2 Modul raspoređivača

Od značajnijih globalnih promenljivih u modulu *TaskScheduler.cpp* pomenućemo:

```
static Task *tgraph;                        - Početak grafa stabla
static sockaddr_in masterAddr;              - Adresa glavnog računara
static char inputBuffer[BUFFER_SIZE];       - Prihvatni bafer
static char refreshBuffer[BUFFER_SIZE];     - Bafer za osvežavanje rezultata
static char localIP[16];                    - Adresa lokalnog računara
static bool end = false;                    - Označavanje kraja rada programa
static int status;                          - Označava da li je računar master ili klijent
static list<sockaddr_in> listOfServers;     - Lista klijenata
```

Funkcija *_kbhit()* predstavlja istoimenu funkciju koja postoji u biblioteci prevodioca programskog jezika C na Windows platformi ali ne i na Linux platformi, pa je bilo potrebno ručno je implementirati.

Funkcije *TS_CreateTaskGraph, TS_AddTask, TS_DeleteTask, TS_SetBottomUpFun, TS_SetTopDownFun, TS_DestroyTaskGraph, TS_ExecuteTopDown, TS_ExecuteBottomUp, TS_ExecuteTopDownSequentially* i *TS_ExecuteBottomUpSequentially* su nasleđene iz originalnog TTE koda i neće biti komentarisane.

Telo funkcije *InitializeNetwork* se nalazi u ovom modulu iako se poziva iz osnovnog modula. Ona je zadužena za otvaranje komunikacije preko UDP prolaza

```
UDPOpen(udp.sockfd, udp.local, PORT, localIP);
```

i kreiranja liste klijenata, očitavajući ulaznu datoteku *IP.txt*.

```
while (fscanf(file, "%s", IP) != EOF)
    {
        if (UDPGetRemoted(addr, PORT, IP) == 0)
        {
            if (strcmp(IP, localIP))
            {
                udp.neighbours.push_front(addr);
            }
            listOfServers.push_front(addr);
        }
```

Inicijalizacija ovog modula se vrši pokretanjem funkcije *TS_Init*, i tu postavljaju promenljive na početne vrednostim kreiraju se kontrolne promenljive u vidu semafora i kritičnih sekcija:

```
for( int i=0; i < numberOfTasks+1; i++ )
    sem_init(&taskSemBU[i], 0, 0);
sem_init(&refreshSem, 0, 1);
sem_init(&terminatorSem, 0, 0);

pthread_mutexattr_t mutexattr;
pthread_mutexattr_settype(&mutexattr, PTHREAD_MUTEX_RECURSIVE_NP);
pthread_mutex_init(&listOfServersCriticalSec, &mutexattr);
pthread_mutexattr_destroy(&mutexattr);
```

I na kraju se kreiraju niti za prihvatanje podataka od ostalih računara u mreži *Receive* i nit za interfejs *Interface* o kojima je bilo reči ranije. Još se kreira nit *Terminator* koja služi za uništavanje niti koje kreira osnovna nit *ExecuteCluster*, kako bi se dinamički, u toku trajanja programa, oslobađali zauzeti resursi tim nitima.

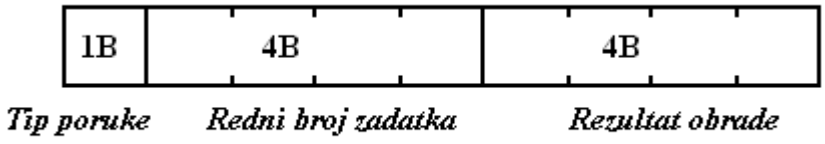
Analogno, funkcija *TS_Close* omogućava oslobađanje resursa kontrolnih promenljivih koje su kreirane ranije, čekanje na završetak rada kreiranih niti u *TS_Init* funkciji i zatvaranje UDP prolaza.

```
pthread_join(interfaceHandle, NULL);
pthread_join(receiveHandle, NULL);
pthread_join(terminatorHandle, NULL);

    for(int i=0; i < numberOfTasks+1; i++ )
        sem_destroy(&taskSemBU[i]);

pthread_mutex_destroy(&listOfServersCriticalSec);
sem_destroy(&refreshSem);
sem_destroy(&terminatorSem);
UDPClose(udp.sockfd);
```

Nit zadužena za prihvatanje podatak preko UDP prolaza je realizovana u ovom modulu i nazvana je *Receive*. Dok god logička promenljiva *end* ne postane istinita, ova nit osluškuje UDP prolaz koristeći funkciju *UDPReceive* preko prolaza *udp.sockfd* koji je inicijalizovan u funkciji *InitializeNetwork*, i čim pristigne poruke, počinje njeno parsiranje. Na osnovu prvog bajta pristigle poruke, propoznaje se tip poruke. Format UDP poruke možemo videti na slici 2.10



Slika 2.10 Izgled UTP paketa

Tip poruke je smešten u prvi bajt, za redni broj zadatka je rezervisano četiri bajta, kao i za rezultat obrade, koji dolazi na kraju. Međutim, struktura je različita za različite tipove. Objava da je pronađen glavni računar i dozvola za završetak programa su poruke koje sadrže samo prvo polje, tip poruke. Zahtev za obradu pored tipa poseduje i redni broj zadatka. Jedino kada je tip poruke – rezultat obrade, sva polja se koriste. Sledeći deo koda prikazuje preuzimanje i prepoznavanje poruke.

```
while(!end)
{
    received = UDPReceive(udp.sockfd, remote, inputBuffer, BUFFER_SIZE);
    if( received > 0 )
    {
        msgID = inputBuffer[0];
        switch (msgID)
```

Nadalje u kodu vidimo da postoje četiri vrste poruka, i za svaku se vežu različite vrste aktivnosti:

```
case REQUEST:
   memcpy( &taskID, &inputBuffer[1], NODE_ID_LENGTH );
   pthread_create(&calculateHandle[taskID], NULL, &Calculate, (void*)(taskID));
   tasksToTerminate[terminatorCounter++] = taskID;
   sem_post(&terminatorSem);
```

Ukoliko je računar dobio zahtev za obradu, preuzima iz sledećeg polja redni broj zadatka i smešta ga u *taskID* promenljivu, kreira nit za proračun – *Calculate* i evidentira se početak rada te niti, smeštanjem identifikacionog broja u niz *tasksToTerminate*, da bi se mogao dočekati kraj te niti i regularno ugasila. Na kraju signaliziramo semaforom *terminatorSem* niti *Terminator* da može nastaviti sa svojim radom, koji ćemo opisati kasnije.

```
case DATA:
   memcpy( &taskID, &inputBuffer[1], NODE_ID_LENGTH );
   sem_wait(&refreshSem);
   memcpy(refreshBuffer, inputBuffer + 1, received - 1);
   pthread_create(&calculateHandle[taskID], NULL, &Refresh, (void*)(received-1));
   tasksToTerminate[terminatorCounter++] = taskID;
   sem_post(&terminatorSem);
```

Kada prihvatimo tip poruke koji sadrži rezultate obrade, kao i u prvom slučaju, najpre preuzimamo redni broj zadatka i smeštamo u *taskID*. Zatim se čeka na propuštanje semafora *refreshSem*, da ne bi u prihvatni bafer smestili ove podatke i prepisali podatke koji se možda u datom momentu upisuju. Kada dobijemo dozvolu semafora, prenosimo podatke iz prihvatnog bafera u bafer *refreshBuffer*, odakle će nit koja se sledeća kreira *Refresh*, preneti podatke na potrebnu lokaciju u memoriji. Ponovo se na isti način kao i u prethodnom slučaju signalizira niti *Terminator* za nastavak rada.

```
case MASTER:
   masterFound = true;
   status = SLAVE;
   memcpy (&masterAddr, &remote, sizeof(remote));
   break;
```

Kada prihvatimo ovaj tip poruke, potrebno je odraditi nekoliko aktivnosti. Logička promenljiva koja označava prisutnost glavnog računara u mreži se postavlja na istinito, da bi se nit za interfejs mogla isključiti. Postavlja se status računara na *slave* iliti klijent, i memoriše se adresa pošaljioca poruke kao glavnog računara, kojem će se kasnije slati rezultati obrade.

```
case END_OF_PROGRAM:
   end = true;
   tasksToTerminate[terminatorCounter++] = taskID;
   sem_post(&terminatorSem);
```

Pri završetku obrade svih zadataka, glavni računar šalje zahtev za završavanje programa. Logička promenljiva *end* se postavlja na istinito i sve niti koje zavise od te promenljive završavaju svoju aktivnost. Još se šalje i obaveštenje niti *Terminator* da nastavi sa izvršenjem i da završi i svoju aktivnost.

Sledeća nit u ovom modulu je *Interface.* Ona je aktivna dok god se ne odabere glavni računar u mreži, kao što se vidi iz koda:

```
while (!masterFound)
{
   if (_kbhit() != 0)
      {
         pressed = getchar();
         if ((pressed == 'm') || (pressed == 'M'))
```

Ukoliko je neki drugi računar odabran kao master, to će nit *Receive* evidentirati i postaviti logičku promenljivu *masterFound* na istinito, pa će ova nit završiti svoje izvršenje. Do tada, nit *Interface* čeka da korisnik pritisne taster ’m’. Kada se pritisne taster, opet se postavlja *masterFound* na istinito i postavlja se status na *MASTER*. Šalje se obaveštenje ostalim računarima, kreira se nit *executeCluster* i čeka se na kraj njenog izvršenja. Napominjemo da se ova nit izvršava samo na glavnom računaru.

```
masterFound = true;
pressed = MASTER;
status = MASTER;
UDPBroadcast(udp.sockfd, udp, &pressed, 1);
clock_gettime(CLOCK_REALTIME, &starttime);
pthread_create(&(tgraph->hThread), NULL, &executeCluster, (void*)tgraph);
pthread_join(tgraph->hThread, NULL);
clock_gettime(CLOCK_REALTIME, &endtime);
```

Vidimo da se pre kreiranja niti *executeCluster* poziva funkcija *clock_gettime*. Ta funkcija memoriše trenutno vreme na računaru i upisuje u promenljivu koja je navedena kao drugi argument funkcije, a to je *starttime*. Nakon obrade, tj završetka niti, opet se poziva ista funkcija sa argumentom *endtime*. Sa upisanim vremenima u te dve promenljive, možemo, izračunati precizno koliko je bilo vreme izvršenja. Promenljive *starttime* i *endtime* su tipa *timespec*, struktura koja sadrži vreme u dva segmenta: protekle sekunde i protekle mikrosekunde.

Na kraju, posle ispisa utrošenog vremena za računanje, šaljemo svim računarima u mreži paket tipa *END_OF_PROGRAM* i ova nit završava svoju aktivnost.

```
char c = END_OF_PROGRAM;
UDPBroadcast(udp.sockfd, udp, &c, 1);
UDPSend(udp.sockfd, masterAddr, &c, 1);
```

Funkcija *Calculate* se pokreće kao nit, i to iz niti *Receive*. Zadužena je za izvršenje dobijenog zadatka od glavnog računara. Identifikacioni broj zadatka je prosleđen kao parametar, pa se prvo taj parametar prihvata i na osnovu njega se kreira lokalna instanca objekta zadatak. Pokrećemo funkciju, preko pokazivača *TS_bufPtr*, koja se nalazi u osnovnom modulu i kao argument opet prosleđujemo identifikacioni broj zadatka. Posle obrade, ukoliko trenutni računar nije glavni, potrebno je rezultate poslati glavnom računaru. To se radi koristeći funkciju *ShareResult*, koju ćemo kasnije opisati. Naposletku, oslobađamo odgovarajući semafor kako bi *executeCluster* znao da je taj zadatak završen.

Funkcija *ShareResult* prihvata kao argument identifikacion broj zadatka, i njena uloga je da rezultate koji su smešteni u memoriju lokalnog rezultata za dati zadatak pošalje preko UDP protokola na glavni računar. Kreiramo lokalni bafer *tmp* i u njega smeštamo, na prvo mesto tip podatka, tj *DATA*, zatim identifikacioni broj, i na kraju rezultate. Na kraju se sve to šalje funkcijom *UDPSend.*

```
char tmp[BUFFER_SIZE];
tmp[0] = typeOfData;
memcpy( &tmp[1], (char *)(&taskID), NODE_ID_LENGTH );
memcpy( &tmp[NODE_ID_LENGTH+1], (char*)&result[taskID], NODE_ID_LENGTH );
UDPSend( udp.sockfd, masterAddr, &tmp[0], NODE_ID_LENGTH*2 + 1 );
```

Refresh funkcija se pokreće kao nit, i hronološki dolazi nakon slanja podataka pomoću gore opisane funkcije *ShareResult.* Naime, kada glavni računar prihvati tip podatka *DATA*, kreira se *Refresh* nit. Iz prihvatnog bafera preuzimamo identifikacion broj zadatka i smeštamo ga u lokalnu promenljivu *taskID*, a zatim rezultat pohranjujemo na odgovarajuću memorijsku lokaciju, tj u niz *result.* Oslobađaju se semafori za nadzor izvršenja zadataka u niti *executeCluster – taskSemBU[taskID]*, kao i za oslobađanje prihvatnog bafera za unos nekih sledećih podataka – *refreshSem.*

```
int taskID;
memcpy((char *)(&taskID), &refreshBuffer[0], NODE_ID_LENGTH );
memcpy( (char*)&result[taskID], &refreshBuffer[NODE_ID_LENGTH], NODE_ID_LENGTH );
sem_post(&taskSemBU[taskID]);
sem_post(&refreshSem);
```

Nit *executeCluster* raspodeljuje zadatke računarima u mreži, koristeći njihove adrese koje se nalaze u globalnoj promenljivoj *listOfServers.* Zadaci se formiraju sa zahtevom za obradu i identifikacionim brojem i redom se šalju računarima u mreži, u krug. Zbog toga je preporučljivo da broj zadataka bude srazmeran broju računara, kako bi svaki računar dobio jednak broj zadataka. Prvi zadatak se šalje glavnom računaru, tj samom sebi. Nit na kraju čeka da se svaki zadatak završi, koristeći signalizaciju semafora, nakon čega može da završi svoju aktivnost.

Nit *Terminator* je jednostavna i njena uloga je dinamičko uklanjanje niti koje su završile svoju aktivnost, umesto da se sve one uklone po završetku rada aplikacije. Na ovaj način, u toku rada se oslobađaju nepotrebni resursi, te je moguće kreirati više niti, tj više zadataka je moguće kreirati. Ideja je da svaka *Calculate* i *Refresh* nit nakon svog izvršenja upišu svoje identifikacione brojeve u niz *tasksToTerminate,* povećaju brojač tog niza i signaliziraju nit *Terminator* semaforom *terminatorSem,* nakon čega će ona pozvati funkciju *pthread_join* za nit čiji je identifikacioni broj na sledećoj poziciji u nizu.

```
int i = 0;
while(!end)
{
   sem_wait(&terminatorSem);
   pthread_join(calculateHandle[ tasksToTerminate[i++]], NULL);
}
```

2.3.3 Modul UDP protokola

Ovaj modul je realizovan u datotekama *UDPfunc.cpp* i *UDPfunc.h.* Sadrže sve potrebne funkcije za komunikaciju između više računara. Opisaćemo sadržaj *UDPfunc.h* datoteke:

```
#define PORT 6788
```

Konstanta, broj porta kroz kojeg će teći komunikacija.

```
typedef struct UDP
{
    int sockfd;
    sockaddr_in local;
    Connected_t neighbours;
} UDP;
```

Struktura, koja sadrži redom: UDP prolaz, kroz koji teče komunikacija, adresa lokalnog računara i lista adresa svih računara u mreži.

```
int UDPOpen(int &sockfd, sockaddr_in &local, unsigned int port, char* localIP);
int UDPClose(int &sockfd);
int UDPSend(int sockfd, sockaddr_in remoted, char* msg, int size);
int UDPBroadcast(int sockfd, UDP udp, char* msg, int size);
int UDPReceive(int sockfd, sockaddr_in &remote, char* msg, int size);
int UDPGetRemoted(sockaddr_in &remoted, unsigned int port, char* ipAddress);
```

UDPOpen se pokreće pri inicijalizovanju mreže. Kreira se UDP prolaz, u promenljivoj koja je definisana kao prvi argument poziva funkcije. Ostali argumenti su lokalna adresa kao i broj prolaza.

UDPClose, analogno gornjoj funkciji, služi za oslobađanje resursa i zatvaranje UDP prolaza.

UDPSend je funkcija koja šalje potrebne podatke kroz mrežu određenom računaru. Argumenti su redom: UDP prolaz, adresa odredišnog računara, adresa bafera sa kojeg šaljemo podatke i broj bajtova koje treba preneti. Povratna vrednost je broj uspešno prenetih bajtova.

UDPBroadcast služi za slanje podataka svim računarima u mreži. Razlika u argumentima u odnosu na prethodnu funkciju je da umesto odredišnog računara, šaljemo *UDP* strukturu koja sadrži adrese svih računara. Povratna vrednost je broj uspešno prenetih bajtova.

UDPReceive je zadužena za prihvatanje podataka sa mreže. Argumenti su isti kao i kod funkcije *UDPSend*, s tim, da ovog puta umesto adrese odredišnog računara, funkcija upisuje adresu računara koji je poslao poruku. Povratna vrednost je broj uspešno primljenih bajtova.

UDPGetRemoted je funkcija koja na osnovu IP adrese u tekstuelnom formatu i broja prolaza, kreira promenljivu tipa *sockaddr_in* koja će nadalje koristiti kao adresa računara.

2.3.4 Modul zadatka

Modul zadatka je opisan zaglavljem *Task.h* i implementacijom *Task.cpp*. Sadrži klasu zadatak – *Task*, njene metode, kao i destruktore i konstruktore.

2.3.5 Modul predefinisanih konstanti

Ovaj modul se nalazi u datoteci *Defines.h* i sadrži sve konstante koje se koriste u programu. Smeštene su na jedno mesto radi preglednosti i jednostavne izmene po potrebi. Opisaćemo sadržaj, tj značenja pojedinih konstanti.

`#define SLAVE 0`	Oznaka statusa računara
`#define MASTER 1`	Oznaka statusa računara
`#define REQUEST 2`	Oznaka tipa poruke
`#define DATA 3`	Oznaka tipa poruke
`#define END_OF_PROGRAM 4`	Oznaka tipa poruke
`#define MAX_NUMBER_OF_TASKS 256`	Maksimalan broj zadataka za kreiranje
`#define DEPTH 20`	Dubina računanja, ukoliko se drugačije ne podesi
`#define NUMBER_OF_TASKS 128`	Broj zadataka, ukoliko se drugačije ne podesi
`#define BOUND_A 1`	Leva granica x-ose računanja integrala
`#define BOUND_B 1000`	Desna granica x-ose računanja integrala

2.4 Ispitivanje

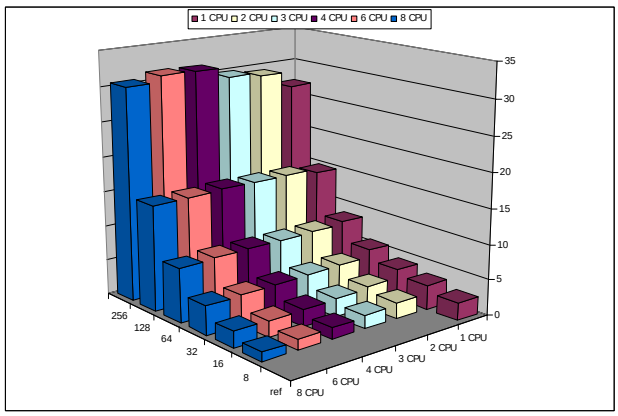
Validacija izlaznih podataka je vršena poređenjem rezultata sa rezultatima koje smo dobili izvršavanjem referentne aplikacije. Referentna aplikacija je jednostavno rešenje problema, koja se izvršava sekvencijalno i bez ikakvih ubrzanja, ali rezultati dobijeni na taj način se mogu usvojiti kao tačni. Dalje, ispitivanja naše aplikacije je vršena na različitom broju računara (1-8 računara), zatim sa različitom dubinom računanja, tj preciznošću ($2^{10} – 2^{20}$) i sa različitim brojem zadataka, na koji je račun podeljen (8, 16, 32, 64, 128 i 256). Sledeće tabele prikazuju merenja za zadate parametre. Data vremena su u milisekundama.

Depth	NoOfT	1 CPU	2 CPU	3 CPU	4 CPU	6 CPU	8 CPU	referentni
10	8	3.4	2.3	1.9	1.6	1.5	1.3	2.5
	16	4.2	3	2.7	2.5	2.3	2.3	
	32	5.7	4.7	4.5	4.4	4.2	4.1	
	64	8.6	8.2	8.1	8.1	8	7.7	
	128	14.8	15.4	15.4	15.4	15.2	15.1	
	256	27	29.3	29.8	31.3	31.4	30.5	
11	8	5.8	3.4	2.9	2.3	2.1	1.6	4.9
	16	6.7	4.3	3.7	3.1	2.7	2.6	
	32	8	5.9	5.2	5	4.7	4.4	
	64	11.5	9.5	8.9	8.6	8.6	8.2	
	128	17.5	16.5	15.9	15.9	15.8	15.6	
	256	30.2	31	31.5	31.4	31.4	31.2	
12	8	11	5.9	5.1	3.5	3.3	2.2	10.1
	16	12.1	6.8	5.6	4.4	3.7	3.2	
	32	13.2	8.7	6.8	6.3	5.3	4.9	
	64	16.1	12.3	10.3	9.9	9.2	8.8	
	128	22.4	19.5	17.8	17.4	16.7	16.1	
	256	34.9	33.4	33	33.2	32.7	32.2	

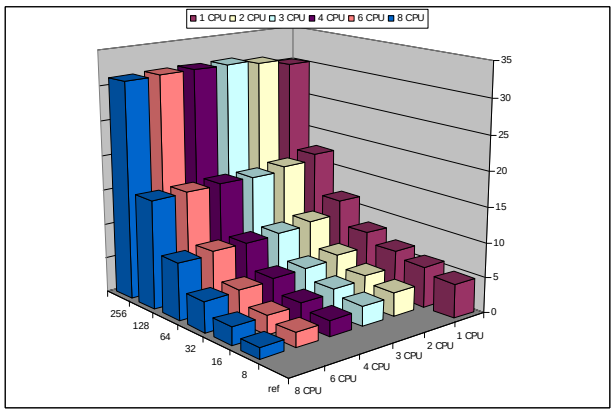
Tabela 2.1 Rezultati za dubine 10, 11 i 12

Depth	NoOfT	1 CPU	2 CPU	3 CPU	4 CPU	6 CPU	8 CPU	referentni
13	8	22.3	11.1	9.7	5.9	5.7	3.6	28.3
	16	21.9	12	9.9	6.8	5.8	4.7	
	32	23.5	13.5	10.3	8.6	7	6.2	
	64	26.2	17.8	13.7	12.1	10.5	10.1	
	128	32.5	24.1	20.9	19.9	18.5	17.8	
	256	45.2	40.5	35.7	35.2	34.4	32.8	
14	8	40.9	21.8	16.5	10.9	10.7	6.7	48.4
	16	42.8	22.7	16.6	11.8	9.9	7	
	32	44.8	24.5	17	13.7	10.7	8.7	
	64	46	28.3	20.4	17.5	13.6	12.5	
	128	52.7	38.7	27.4	24.9	21.7	20.1	
	256	64.5	48.5	42.7	40.2	37.4	36.4	
15	8	82.5	42.2	31.7	21.6	21.1	12.8	88.1
	16	83.1	42.5	31.2	22.2	19	13.3	
	32	84.2	44.4	30.9	23.8	18.2	13.8	
	64	86.5	47.8	33.8	27.2	23.5	17.5	
	128	92.9	55.6	41.5	35	34.6	25	
	256	105.2	69.5	55.9	49.5	44.2	41.2	
16	8	161.3	83.4	62.4	41.9	41	25.1	167.5
	16	162.3	83.4	61.1	42.6	37.5	25.2	
	32	164.4	84.2	59.3	44.3	32.6	25.5	
	64	166.9	87.6	61.5	47.8	35.7	27.6	
	128	173.6	95.2	67.8	55.5	42.4	35.1	
	256	186.6	112.1	82.5	70.5	57.1	50.4	

Tabela 2.2 Rezultati za dubine 13, 14, 15 i 16

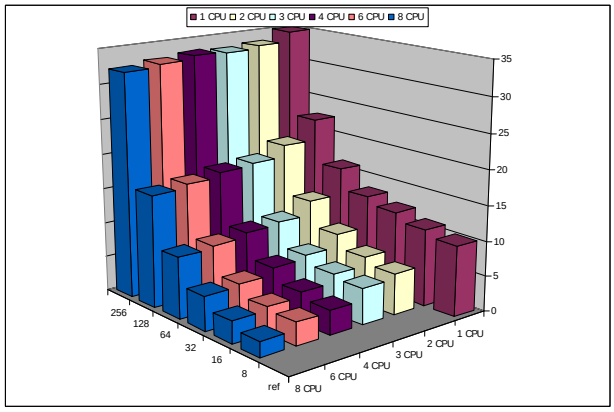


Slika 2.11 Grafikon vremena obrade za preciznost $1/2^{10}$

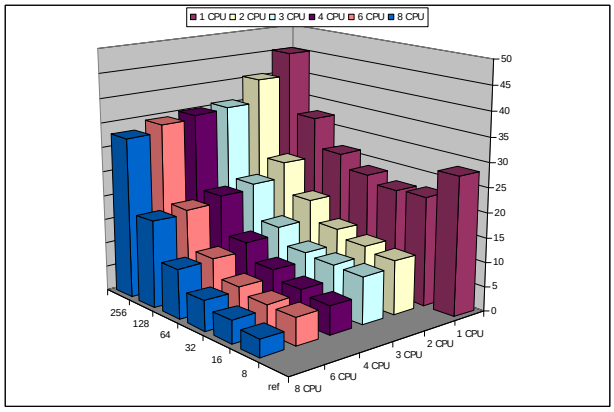


Slika 2.12 Grafikon vremena obrade za preciznost $1/2^{11}$

Očigledno je da za relativno male preciznosti (do $1/2^{12}$, slike 2.11, 2.12 i 2.13) vreme proračuna se višestruko povećava sa većim brojem zadataka. Razlog je mali broj ukupnih tačaka za obradu. Ukoliko odredimo 256 zadataka, znači da će, za najmanju merenu preciznost, svaki zadatak obrađivati tek 4 tačke, što je neisplativo po pitanju vremena izvršenja. U našim merenjima, to je očigledno do preciznosti $1/2^{16}$-$1/2^{17}$ (slike 2.17 i 2.18), nadalje, sa povećanjem preciznosti, postoji dovoljno veliki broj obrade čak i kada postoji veliki broj zadataka.



Slika 2.13 Grafikon vremena obrade za preciznost $1/2^{12}$

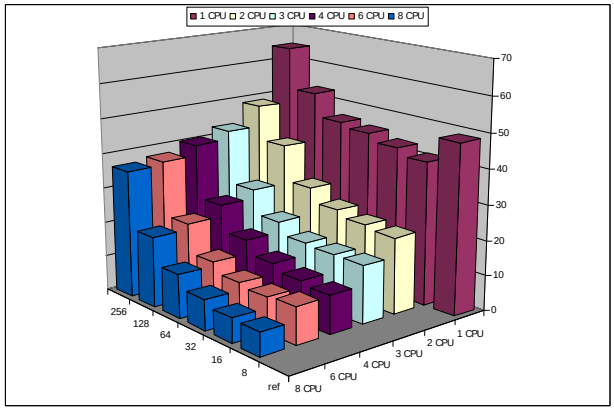


Slika 2.14 Grafikon vremena obrade za preciznost $1/2^{13}$

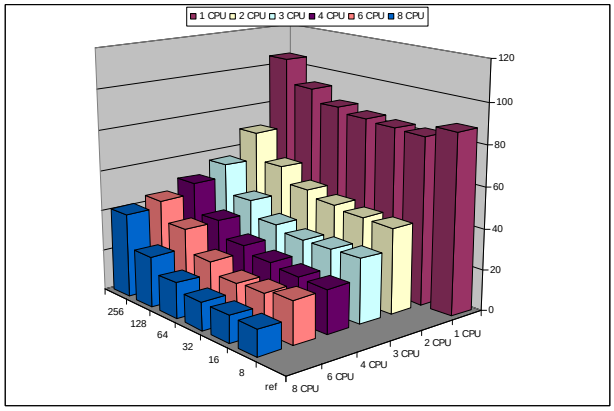
Možemo videti da za male preciznosti, broj računara ne igra značajniju ulogu. Već na slici 2.14, preciznost $1/2^{13}$, primećuje se sniženje utrošenog vremena sa povećanjem broja računara, posebno kada se radi sa manjim brojem zadataka. Razlog tome je veći broj proračuna po jednom zadatku, i računari imaju dovoljno obrade, da bi se gubitak vremena u transferu podataka između računara i kontroli toka podataka, mogao zanemariti. Od slike 2.15 pa nadalje, sa preciznošću $1/2^{14}$ i većom, primećujemo da je sve manja razlika u vremenu za različite brojeve zadataka.

Depth	NoOfT	1 CPU	2 CPU	3 CPU	4 CPU	6 CPU	8 CPU	referentni
17	8	321	166	124.3	83	80.3	49.6	326.6
	16	322	166.8	121	83.8	74	49.6	
	32	323	166.5	114.6	84.2	63.2	49.9	
	64	325	167.5	112.8	87.4	68.6	50.3	
	128	333	175.2	119.7	95.5	69	55.7	
	256	345	189	136.2	109.7	83.6	70.5	
18	8	640	331	248	166	160	98.3	641.6
	16	643	331	239	165	147.3	98.3	
	32	645	332	227	166	124.3	98.6	
	64	645	334	223	168	135.9	99.3	
	128	651	336	228	175	130.7	99.9	
	256	664	350	242	189	136.4	111.2	
19	8	1279	661	496	330	319	196	1272
	16	1283	661	476	330	293	196	
	32	1283	660	456	332	246	196	
	64	1286	663	440	332	270	197	
	128	1295	666	446	335	259	198	
	256	1306	684	454	350	267	199	
20	8	2556	1335	995	667	638	391	2536
	16	2558	1324	954	664	586	391	
	32	2561	1321	912	661	492	391	
	64	2566	1325	877	663	539	393	
	128	2572	1329	894	662	516	395	
	256	2583	1333	884	675	530	395	

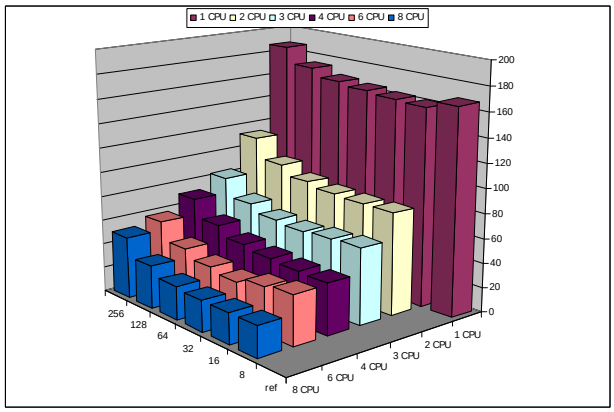
Tabela 2.3 Rezultati za dubine 17, 18, 19 i 20



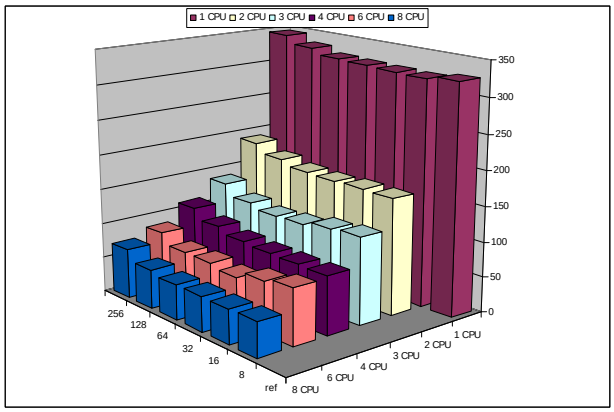
Slika 2.15 Grafikon vremena obrade za preciznost $1/2^{14}$



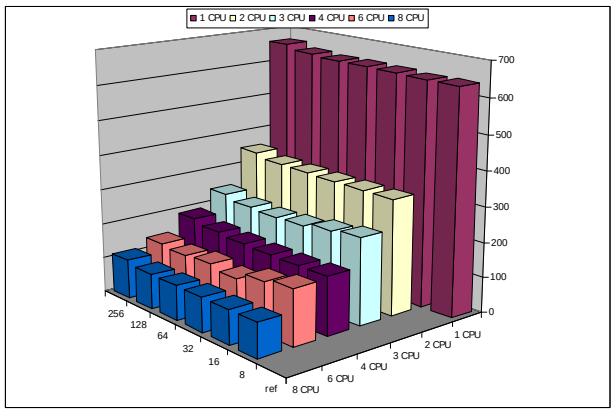
Slika 2.16 Grafikon vremena obrade za preciznost $1/2^{15}$



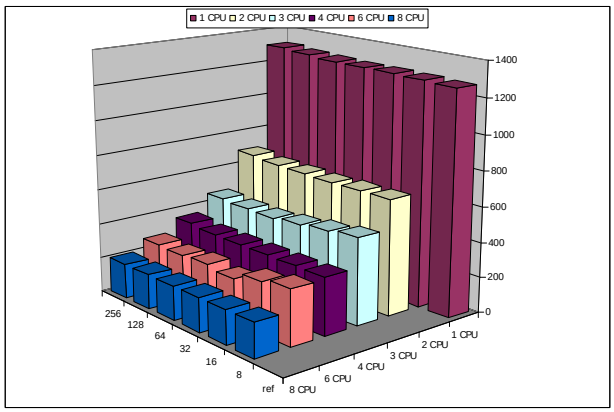
Slika 2.17 Grafikon vremena obrade za preciznost $1/2^{16}$



Slika 2.18 Grafikon vremena obrade za preciznost $1/2^{17}$

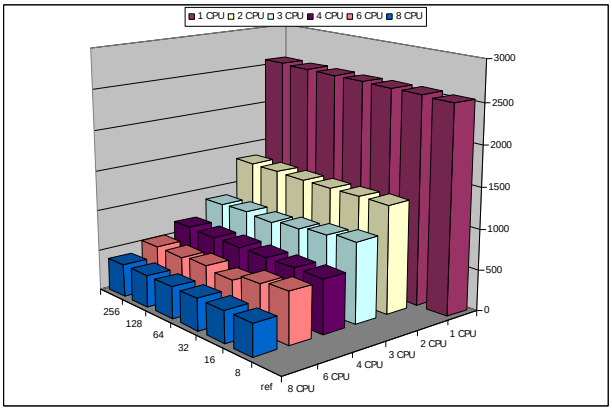


Slika 2.19 Grafikon vremena obrade za preciznost $1/2^{18}$



Slika 2.20 Grafikon vremena obrade za preciznost $1/2^{19}$

Slike 2.18 – 2.21 prikazuju da je došlo do zasićenja i da broj zadataka ne igra nikakvu ulogu. Jasno se vide višestruka ubrzanja sa većim brojem računara. Još se može primetiti, da se jasno vidi duplo bolje vreme izvršenja gledajući rezultate na dva računara u poređenju sa jednim. Kao i poređenjem četiri računara na prema dva. Testiranje na šest računara je postavljeno eksperimentalno. Vidi se da u nekim slučajevima nema velike razlike u poređenju sa rezultatima na četiri računara, ali ponekad su veoma slični rezultatima na osam računara. Uzrok te anomalije ćemo povezati sa brojem zadataka. Ukoliko je broj zadataka deljiv brojem računara (u ovom slučaju šest) onda se dobijaju optimalni rezultati. Ali kada nije, npr osam i šesnaest u našem slučaju, vidimo da je brzina jednaka izvršenju na četiri računara. Razlog je što se ne može ravnomerno raspodeliti broj zadataka i nekoliko računara u mreži radi više (u ovom slučaju duplo više) zadataka u odnosu na ostale.



Slika 2.21 Grafikon vremena obrade za preciznost $1/2^{20}$

3. Zaključak

Postignuti rezultati pri merenju izrađenog programskog rešenja su zadovoljavajući, dobijena ubrzanja su kao što smo i predviđali. Vreme izvršenja se skoro smanjuje srazmerno broju računara koji učestvuju u procesu. Cilj je bio uposliti računare i međusobno ih povezati u proračunu bez značajnih vremenskih gubitaka na međusobnu komunikaciju i samu organizaciju obrade, i to smo uspešno ispunili. Programski kod je dizajniran tako, da je jednostavno izmeniti funkciju nad kojom vršimo proračun, kao i granice proračuna, dok se parametri broja zadataka i preciznosti obrade, unose kao parametri aplikacije. Takođe, aplikacija je izrađena na taj način, da se najbolji rezultati dobijaju kada je broj zadataka proporcionalan broju računara u mreži.

4. Literatura

[1] M. Popovic, I. Basicevic, V. Vrtunski: A Task Tree Executor: New Runtime for Parallelized Legacy Software

[2] http://www.wolframalpha.com/

[3] http://en.wikipedia.org/wiki/Integral

[4] http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

[5] http://www.ibm.com

[6] http://linux.die.net