Fakultet tehničkih nauka

Univerzitet u Novom Sadu

Računarski sistemi visokih performansi

Implementacija programa u Intel CnC metodologiji

Autor:
Dejan Grubišić

Indeks:
E2 83/2018

17. februar 2019

Sažetak

Ovaj rad bavi se implementiranjem aplikacije za izračunavanje unutrašnjeg proizvoda (eng. *inner-product*) u Intel CnC metodologiji, korišćenjem MADNESS modela. U poglavljima 2, 3, 4 dat je kratak sažetak CnC metodologije i dat je primer njenog korišćenja. Poglavlje 5 se bavi MADNESS metodologijom. Poglavlje 6 se bavi opisivanjem načina rada osnovne implementacije unutrašnjeg proizvoda, dok naredna poglavlja opisuju način optimizacije granularnosti i izlažu dobijene rezultate.

Sadržaj

Lista slika

1 Uvod

Savremena nauka u velikoj meri se zasniva na simulacijama i proračunima velike kompleksnosti. Koristeći običan računar, mnoge naučne simulacije potrajale bi mesecima pa čak i godinama, zbog čega se javila potreba za uvođenjem naučne discipline - naučnog računarstva (eng. *Scientific Computing*) koja ima za cilj uvođenje novih metodologija u načinu izračunavanja i distribuiranju podatakakako bi se smanjilo vremene trajanja naučnih simulacija.

Osnova naučnog računarstva zasniva se na paralelnoj obradi koristeći umrežene računarske resurse. Glavni izazovi u ovome predstavljaju ostvarivanje efikasne komunikacije i prenosa podataka između mašina, kao i pronalaženja pogodnog algoritma koji može određeni problem da podeli na što nezavisnije celine. Takođe je važno da su vremena izračunavanja nezavisnih celina približno jednaka, kako ne bi došli u situaciju da se samo jedan blok izvršava dok ostali blokovi čekaju na dobijanje rezultata.

Da bi se postiglo efikasno iskorišćavanje resursa i omogućila paralelna obrada, osmišljeno je mnogo metodologija i alata. Mnogo napora uloženo je u osmišljavanje metodologije za paralelizaciju sekvencijalnog koda, međutim ovakvi pokušaji idalje ne daju zadovoljavajuće rezultate i najveći fokus dat je na kreiranje programskih modela koji omogućavaju programeru da utiče na tok izvršavanja programa. Postoje nekoliko pristupa u specificiranju paralelizma ovakvim modelima i izdvajaju se:

1. Modeli sa deljenom memorijom
2. Modeli zasnovani na razmeni poruka
3. Modeli sa implicitnim paralelizmom

Model sa deljenom memorijom daje apstrakciju uniformne memorije, kojoj svaki proces može da pristupi. Sa druge strane ovakav model uvodi probleme kao što su problem štetnog preplitanja ili problema trke (eng. *data race*), koji nastaju kao rezultat pristupanja promenjivoj od strane jednog procesa, između čitanja i ažuriranja promenjive od strane drugog procesa. Kako bi se ovakvi problemi izbegli uvodi se pojam međusobnog isključivanja koje se ostvaruje mehanizmima kao što su muteks i semafor.

Postoje takođe mnogi jezici i biblioteke koje u sebi sadrže ove mehanizme i automatski paralelizuju delove koda za koje programer napiše određenu direktivu. Najčešće u svojoj osnovi ovakvih jezika nalazi se fork-džoin model (eng. *Fork-Join model*), kojim se specificira koji delovi koda mogu da se izvršavaju paralelno, a programski prevodilac na odgovarajući način određuje kada se šta izvršava, poštujući zadata ograničenja. U ovakvu grupu spadaju Cilk, OpenMP i Threading Building Blocks.

Modeli zasnovani na razmeni porka daju mnogo veću fleksibilnost i bolje oslikavaju realan sistem nego modeli sa deljenom memorijom. Uvođenjem poruka moguće je jasno definisati komunikaciju između procesa, što sa druge strane dovodi do značajno komplikovanije implementacije. Ovakvi modeli predstavljaju osnovu distribuiranih sistema u kojima je od izuzetnog značaja optimizovati troškove komunikacije, kao i omogućiti da sistem radi pouzdano u slučaju otkaza pojedinih komponenti ili u slučajevima otežane razmene poruka. MPI (eng. *Message Passing Interface*) standard pruža ovakvu funkcionalnost, a pored njega tu su i jezici kao što su GO i Scala.

U oba predhodna modela uključuju definisanje interakcija između procesa od strane korisnika. Za razliku od njih, model sa implicitnim paralelizmom realizuje procese i komunikaciju između njih u vreme kompajliranja ili u vreme rada programa. Na ovaj način se dobijaju najjednostavniji modeli sa stanovišta programera, koji ne sadrže dodatnu kompleksnost uvođenjem sinhronizacionih elemenata i komunikacije. Programer ne bi trebao da brine o podeli zadataka, granularnosti ili troškovima komunikacije, već samo na algoritam koji je potrebno implementirati. Ovakav pristup prepušta optimizaciju programskim prevodiocima i izvršnom sistemu, što može imati za posledicu sub-optimalano izvršavanje. Proces pronalaženja grešaka prilikom razvoja, takođe može da bude problem, jer ne postoji jasan uvid u tok izvršavanja. Ipak, sa razvojem programskih prevodioca, ovakav pristup mogao bi obezbediti bržu i efikasniju implementaciju paralelnih problema nego u predhodnim modelima, jer se uvodi najmanje ograničenja što dovodi do potenicjalno većeg prostora zadovoljavajućih implementacija. IntelCNC (eng. *Intel Concurrent Collections*) je ovakav jedan model biće detaljnije opisan u sledećem poglavlju.

2 Intel Concurrent Collections model

IntelCnC je programski model sa ugrađenim implicitnim paralelizmom. Zasniva se na specificiranju zadataka koji treba da budu izvršeni i definisanju samo neophodnih međuzavisnosti. Uvedene zavisnosti jasno definišu graf izvršavanja zadataka, koji se sastoji od podataka koji treba da se obrade i zadataka nad njima. Kada će se određeni zadatak izvršiti isključivo zavisi od dva faktora:

- Da li su podaci koje taj zadatak koristi dostupni
- Da li je zadatak dobio signal da treba da se izvrši

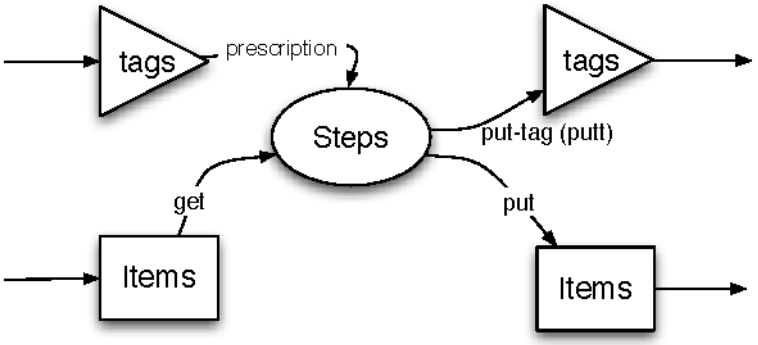
Na ovaj način specificira se početni graf izvršavanja, koji dalje programski prevodioci i izvršni sistemi optimizuju, podešavajući granularnost zadataka pogodnu za efikasno izvršavanje na određenoj mašini. Još jedna prednost ovakvog pristupa je velika skalabilnost i laka prenosivost programa, čiji je aplikativni deo uvek isti, dok se jedino menja programski prevodilac. Sa druge strane razdvajanje početnog problema na aplikativni deo i optimizaciju znači da osoba koja piše aplikaciju ne mora skoro ništa da zna o paralelnom izvršavanju, kao što ni osoba ili programski prevodilac koji se bave optimizacijom ne moraju mnogo da znaju o problemu koji optimizuju. [5]

Sam tok izvršavanja prilično je intuitivan i slikovit primer koji opisuje kako IntelCnC radi može se naći u literaturi pod [1].

3 Arhitektura Intel CnC modela

Osnovni elementi IntelCnC modela predstavljaju kolekcije koraka(eng. *step collections*), kolekcije podataka (eng. *data collections*) i kolekcije kontrolnih oznaka (eng. *tag collections*). Unutar kolekcije, svaka instanca podataka, koraka i kontrolih oznaka određena je jedinstvenim identifikatorom, koji može biti bilo kog tipa, za koji su određene operacije jednakosti i *hash* funkcija. Obično ovi identifikatori imaju specifično značenje unutar aplikacije. Oni mogu označavati iteratore kroz strukturu petlje ili bilo kog drugog prostora kroz koji se prolazi kao što su grafovi i stabla.

Kolekcije koraka se izvršavaju sekvencijalno i u sebi mogu da sadrže naredbe čitanja i upisa nad kolekcijama podataka, kao i kontrolnih oznaka, čime se može inicirati nastajanje novih podataka, kao i zadataka koji trebaju da budu izvršeni. Osobina da jedan korak može da ima kontrolu nad kreiranjem novih zadataka daje CnC modelu veću fleksibilnost od tipičnog protočnog modela. Dijagram osnovnih pojmova u CnC metodologiji dat je na slici 1.



Slika 1: CnC kolekcije

Operacije upisa i čitanja dobijaju se *put* i *get* metodama, pri čemu podaci imaju osobinu nepromenjivosti i kada se jednom upišu, ne mogu više da se menjaju. Ova osobina je od posebnog značaja u paralelnom programiranju, jer isključuje probleme štetnog preplitanja.

Kolekcije kontrolnih oznaka ponašaju se kao generatori zadataka. Svaki put kada se izvrši *put* operacija nad jednom kontrolnom oznakom, kreira se novi zadatak, odnosno korak, koji će biti konačno izvršen kada na svom ulazu bude imao dostupne podatke koje konzumira.

Izvršavanje CnC programa započinje inicijalnim kreiranjem zadataka i traje dokle god postoje zadaci koji su generisani od strane kontrolnih oznaka i na svom ulazu imaju dostupne podatke koje konzumiraju.

Primer korišćenja CnC modela dat je u sekciji Prvi dodatak.

4 Mapiranje na određenu platformu

Mapiranje na određenu platformu predstavlja najvažniji korak u CnC modelu u pogledu performansi. Postoje mnogi faktori koji utiču na brzinu izvršavanja kao što su veličina bloka, lokalnost podataka, broj *put* i *get* operacija, kao i strategija raspoređivanja zadataka koji se izvršavaju. U CnC modelu podešavanje optimizacionih parametara su mogući u vreme prevođenja programa, ali takođe i u vreme izvršavanja. Jedan od ciljeva razvoja CnC modela je da se proces pronalaženja optimalne veličine i strukture blokova automatizuje i da se omogući dinamičko menjanje ovih blokova u toku izvršavanja u zavisnosti od količine podataka koje dobijamo na ulazu.

Implementacije CnC modela obično generišu kod, zasnovan na specificiranom CnC grafu izvršavanja, koji se može koristiti preko aplikativnog interfejsa izvršnog sistema. Poznati jeziti u kojima je implementiran CnC model su [5]:

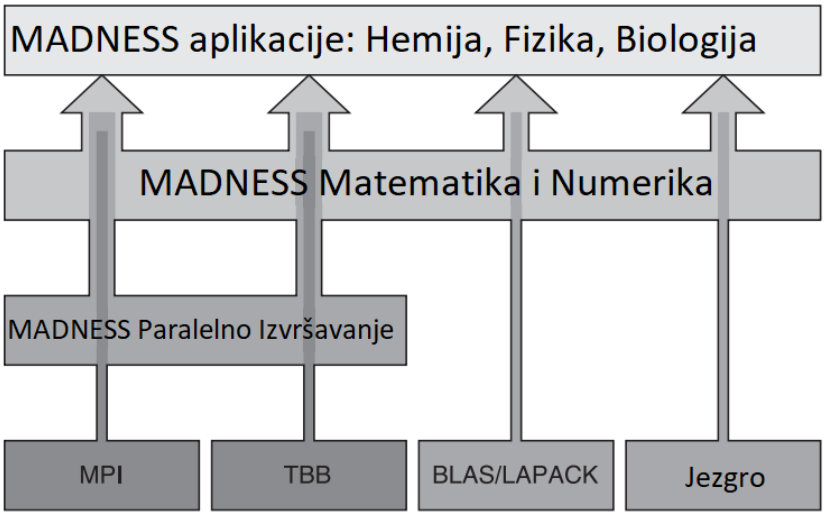
- C++ - zasnovan na tehnologiji (eng. *Intel Treading Building Blocks*)
- Java - zasnovana na tehnologiji (eng. *Java Concurrency) Utilities*
- .NET - zasnovan na tehnologiji (eng. *.NET Task Parallel) Library*
- Haskell

Implementacija *put* i *get* metoda može da se razlikuje u ovim jezicima. Operacija *put* nad kontrolnom oznakom, odnosno kreiranje zadatka, može odmah da pokrene izvršavanje koraka, pri čemu postoji povratni mehanizam u koliko ne postoje sve kolekcije podataka koje se konzumiraju (eng. *Roll-back mechanism*). Ovo je slučaj u Intelovoj implementaciji, koja je implementirana u C++ programskom jeziku. Druga mogućnost je da se pri kreiranju zadatka proverava da li postoje podaci na ulazu. Operacija *get* može da bude blokirajuća, u slučaju da se ne garantuje dostupnost podataka koji se prihvataju, ili može da bude ne blokirajuća u slučaju da sistem garantuje dostupnost podataka.

CnC garantuje determinizam i svaka od ovih implementacija ima mehanizme provere uslova jedinstvene dodele podataka (eng. *Single Assignment*). Pored ovoga, postoje mehanizmi brisanja kolekcija podataka koje se više ne koriste od strane izvršnog sistema. U slučaju C++ implemetacije, specificira broj korišćenja neke kolekcije koja nakon toga obriše [5].

5 Više dimenziono, adaptivno numeričko okruženje za naučne simulacije (eng. MADNESS - Multiresolution, Adaptive Numerical Environment for Scientific Simulation)

MADNESS predstavlja softversko okruženje visokog nivoa apstrakcije, koje omogućava rešavanje integrala i diferencijalnih jednačina u više dimenzija. Zasniva se na korišćenju adaptivnih metoda i metoda brze harmonijske analize sa garantovanom preciznošću [4]. Struktura MADNESS okruženja data je na slici 2.



Slika 2: Struktura MADNESS okruženja

Cilj ovog pristupa je da se omogući jednostavano rukovanje kompleksnih naučnih simulacija, bez potrebe da se simulacioni model prilagođava strukturama podataka simulacionog alata. MADNESS pristup omogućava definisanje računskih operacija između funkcija (u analognom smislu) bez potrebe da se funkcije predstavljaju kao vektori ili matrice.

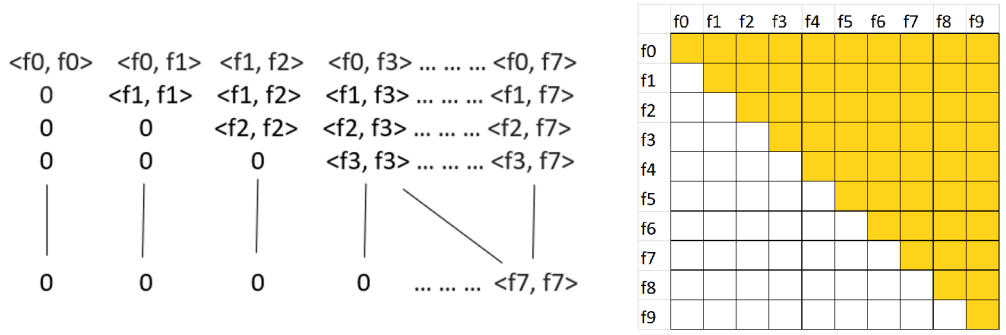
Kada se funkcija kreira, reprezentuje se strukturom stabla, pri čemu se svaka dimenzija sukcesivno deli na pola i aproksimira sa Ležandrovim polinomima, za koje je moguće podesiti stepen polinoma, kao i broj tačaka aproksimacije. Stablo se dalje profinjuje sve dok greška aproksimacije ne postane manja od zadate. Za svaki operator između funkcija definiše se greška pri čemu se funkcije eventualno dodatno profinjavaju, kako bi rezultat u svim delovima zadovoljavao uslov.

Takođe, funkcije mogu biti opisane sa dve reprezentacije. Prva reprezentacija odnosi se na najfiniji nivo granularnosti - listove stabla (eng. *real space basis*). Druga reprezentacija predstavlja nivo veće granularnosti i odstupanje od najfinije granularnosti (eng. *multiwavelet basis*). Definisana je još i brza transformacija između ove dve reprezentacije (eng. *Fast wavelet transform*), koja omogućava da možemo izabrati pogodnu reprezentaciju i nad njom izvršiti neku operaciju. Može se povući paralela sa Furijeovom transformacijom, za koju je konvolucija dve funkcije u vremenskom domenu, zapravo njihov proizvod u spektralnom domenu, zbog čega se kao takva može efikasnije izračunati.

MADNESS koristi matematičke operacije iz *BLAS* i *LAPACK* biblioteka kao što su operacije iz linearne algebre, numeriče analize, a podržava i visoko dimenzione funkcije(do šest dimenzija). Pored ovoga omogućava paralelno izvršavanje na velikim umreženim sistemima korišćenjem MPI 3.0 standard i Intel TBB (eng. *Threading Building Blocks*) tehnologiju. Ovo omogućava rešavanje kompjutaciono zahtevnih problema, dok MADNESS izvršno okruženje pruža visok nivo apstrakcije.

6 Osnovna implementacija aplikacije unutrašnjeg proizvoda u Intel CnC metodologiji

Ovo poglavlje posvećeno je optimizacionom delu aplikacije koja se bavi izračunavanjem unutrašnjeg proizvoda (eng. *inner-product*) između svaka dva člana iz dva vektora funkcija. U euklidskom prostoru unutrašnji proizvod naziva se skalarni proizvod i svaka funkcija predstavljena je jednodimenzionim vektorom.



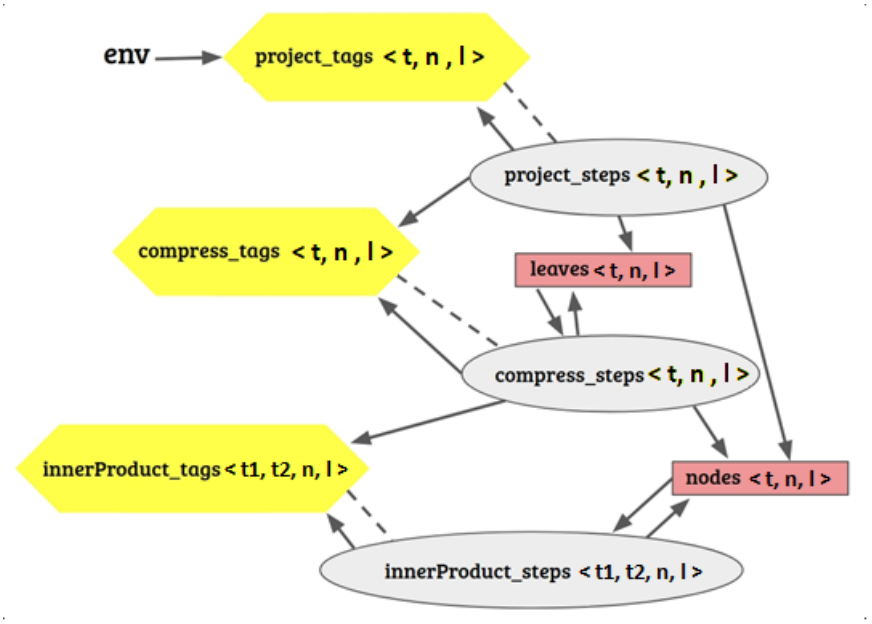
Slika 3: Izračunavanje unutrašnjeg proizvoda između dva vektora funkcija

Aplikacija je implementirana u Intelovoj CnC tehnologiji i koristi MADNESS pristup, kao model za reprezentaciju funkcija i operatora nad njima. CnC graf sastoji se iz tri koraka:

1. Korak: *Project Step* - Reprezentacija funkcija u formi stabla, zasnovanu na Ležandrovim koeficientima
2. Korak: *Compress Step* - Transformacija reprezentacije u *multiwavelet* formu
3. Korak: *Inner-Product Step* - Izračunavanje unutrašnjeg proizvoda između svakog čvora dva stabla na istoj poziciji

Dubina stabla ograničena je na 30 nivoa, kako ne bi došlo do preopterećivanja memorijskih resursa, što omogućava preciznost aproksimacije reda veličine 10^{-18}.

Svaka operacija izvršava se na nivou čvora u stablu, i u odnosu na to identifikator koraka za korake *Project* i *Compress* predstavlja torka $< t, n, l >$, gde je t - redni broj funkcije, n - dubina čvora u stablu i l - udaljenost od najlevlje pozicije na datom nivou. Za *Inner-Product* korak identifikator je $< t1, t2, n, l >$ pri čemu t1 i t2 označavaju funkcije nad kojima se vrši operacija za poziciju čvora definisanom sa n i l. Prikaz CnC grafa izvršavanja dat je na slici 4.



Slika 4: CnC graf izvršavanja

Izvršavanje počinje izvršavanjem operacije *put* nad kontrolnim oznakama $< Project\_tags : t, n = 0, l = 0 >$ za svako t u vektoru funkcija. Nakon *Project* koraka kreira se korenski čvor u stablu, od kog započinje kreiranje celog stabla. U ovom koraku se dati interval deli na pola i izračunava se aproksimacija za svaku polovinu. U koliko je greška veća od predvićene inicira se stvaranje *Project* koraka za decu datog čvora, a u koliko je greška aproksimacije zadovoljavajuća, izračunate aproksimacije se upisuju na mesto $< t, n + 1, 2 * l >$ i $< t, n + 1, 2 * l + 1 >$ u kolekciji listova (eng. *leaves*), koje označava poziciju dece u odnosu na dati čvor. Zatim se inicira kreiranje *Compress* koraka za dati čvor.

Compress korak konzumira podatke sa kolekcije listova za decu u odnosu na dati čvor, vrši njihovu konkatanaciju i vrši transvormaciju u *multiwavelet* formu. Ovi podaci se upisuju u kolekcije čvorova (eng. *nodes*) i listova (eng. *leaves*) za dati čvor. Pored ovoga vrši se i *put* operacija nad korakom *Inner-Product* između datog čvora i čvorova svih funkcija čiji je identifikator veći od identifikatora date funkcije. Ovo znači da će se generisati koraci za *Inner-Product*, za polja $< fn, fn >$ sa slike 3 i sva polja desno od tog polja, za datu poziciju čvora u stablu.

Korak *Inner-Product* će se izvršiti jedino ako su dostupna oba podatka iz kolekcije čvorova za date čvorove. Ovaj korak izračunava operaciju unutrašnjeg proizvoda između vektora aproksimacije koji opisuju čvorove i rezultat će akumulirati u pomoćnoj promenjivoj *results* koja je tipa *concurrent_vector* iz biblioteke

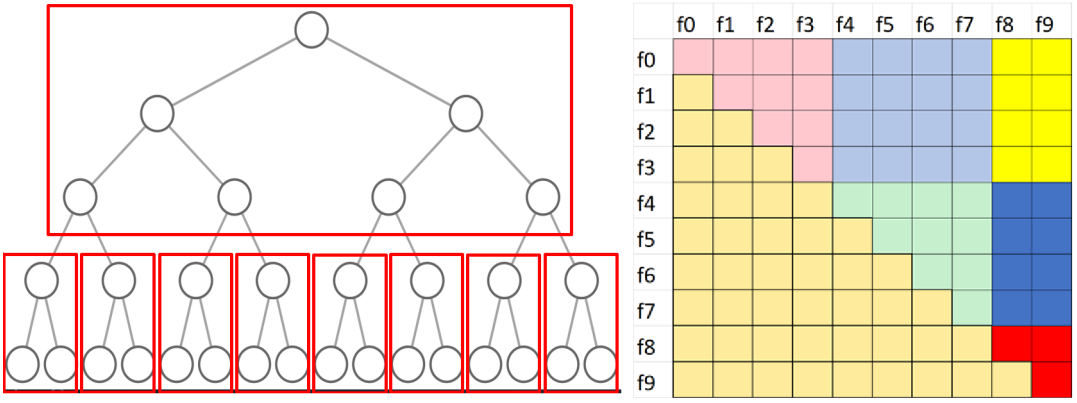
IntelTBB. Po završetku izvršavanja u ovoj promenjivoj smešteni su rezultati za proizvode između svake dve funkcije i mogu se čitati iz glavnog programa.

7 Grupisanje izvršnih koraka u blokove

Aplikacija koju piše korisnik u CnC modelu predstavlja najfiniju implementaciju u pogledu granularnosti. Rezultat ovoga je kreiranje velikog broja koraka koji unose dodatno kašnjenje. Optimizacioni deo aplikacije zadužen za pronađenje optimalne veličine blokova koji se izvršavaju u svakom koraku. U ovoj aplikaciji urađeno je skaliranje po tri dimenzije:

- Dimenzija stabla
- Dimenzija vektora
- Dimenzija operatora

Grupisanje u dimenziji stabla odnosi se na podešavanje dubine podstabala konačne dubine, nad kojim se vrše izvršni koraci umesto na nivou čvora. Grupisanje u nivou vektora označava veličinu bloka funkcija čija se podsabla na istoj poziciji zajedno posmatraju kao jedan blok. Ovo se može videti na slici 5.



Slika 5: Izgled grupisanih blokova u optimizacionom delu

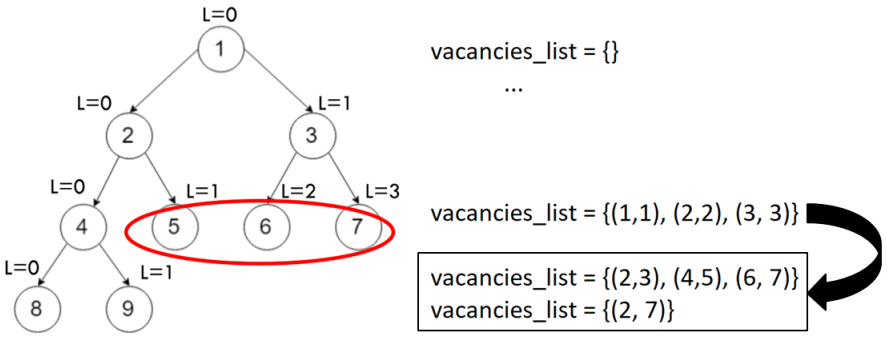
Grupisanje u dimenziji operatora se odnosi na spajanje susednih izvršnih koraka koji mogu da se izvrše jedan za drugim nad istim identifikatorom. Na ovaj način prevazilazi se potreba za stvaranjem dodatnih kolekcija podataka koje će posredovati između dva izvršna koraka. Da bi se odredilo koje kolekcije koraka

mogu da se spoje, potrebno je odrediti međuzavisnosti između podataka koji se konzumiraju i koji se kreiraju.

U *Inner-Product* aplikaciji kreiranje stabla vrši se od korena prema listovima i korak *Project* se kreira tek kada se ovaj korak izvrši nad roditeljima datog čvora. *Compress* korak sa druge strane konzumira podatke nastale od ovog koraka nad decom datog čvora. *Inner-Product* korak na jednoj poziciji u stablu, je nezavistan od izvršavanja na drugim pozicijama i može da se izvrši čim su podaci za dati čvor definisani. Ovo znači da je moguće korake *Compress* i *Inner-Product* izvršiti sekvencijalno za svaku poziciju u stablu.

8 Implementacija optimizovane aplikacije

Grenerisanje čvorova u blokovima podstabla implementirano je iterativno u koraku *Project*. Da bi se izbeglo ponovno izračunavanje aproksimacije za čvorove čiji je predak već imao dovoljnu preciznost uvedena je promenjiva *vacancies_list*, koja označava intervale koje treba preskočiti (Slika 6).



Slika 6: Lista praznina u optimizacionom delu

Na početku ova lista je prazna i svaki put kada neki čvor ima dovoljnu preciznost, njegova udaljenost od najlevljeg čvora na tom nivou se upisuje u listu kao par (l, l). Svakim prelaskom u sledeći nivo vrednosti liste postaju $(2 * l, 2 * l + 1)$ i vrši se grupisanje susednih intervala. Tokom iteriranja kroz nivo podstabla intervali iz liste se preskaču.

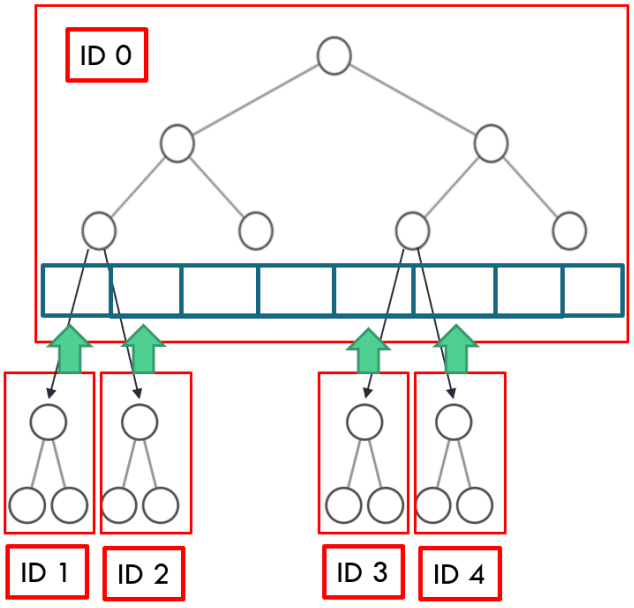
Druga optimizacija u prolasku kroz stablo je brojač kompletnosti (eng. *completeness counter*). Ovaj brojač nam omogućava da odredimo trenutak, kada podstablo ima sve listove odgovarajuće preciznosti. Svaki put kada se generiše čvor

odgovarajuće preciznosti na brojač se doda broj jednak broju čvorova koje bi trenutni čvor imao na poslednjem nivou. Kada brojač postane jednak $2^{dubina\_podstabla}$ svi čvorovi su generisani.

Uvođenje blokova podstabala uvodi problem komunicikacije između dece i roditelja pri iniciranju koraka *Compress*. U početnoj implementaciji, korak *Compress* je konzumirao podatke o levom i desnom detetu koji su uvek bili dostupni, međutim svako podstablo može imati maksimalno $2^{dubina\_podstabla+1}$ dece, pa je potrebno obezbediti da su svi potrebni podaci dostupni kada se korak pokreće.

Da bi se ostvarila uspešna komunikacija između dece i roditelja podstabala uvedena je struktura *concurrent_vector* iz biblioteke Intel TBB, pri čemu svaki element predstavlja podstablo koje ima informacije o poziciji roditelja u vektoru i alocira se postor za upisivanje rezultata svakog deteta podstabla.

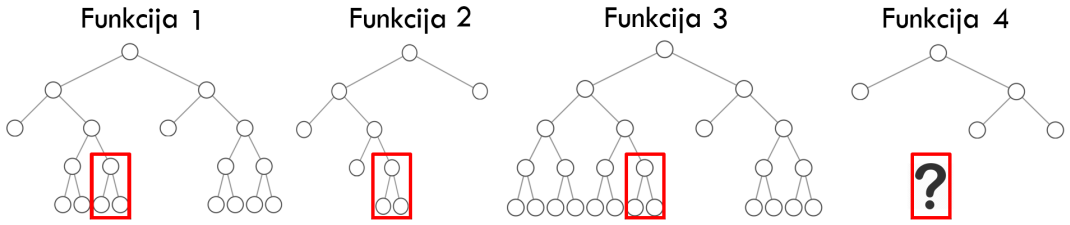
Svaki put kada se kreira novo podstablo u koraku *Project* na vektor se dodaje novi element i uzima se trenutna dužina vektora kao identifikator podstabla. Kako *concurrent_vector* atomično dodaje elemente garantuje se jedinstvenost identifikatora. Potom se dobijeni identifikator šalje svoj deci podstablima, kako bi u koraku *Compress* znali na koje mesto u vektoru treba da upišu svoje rezultate. Struktura podataka podstabala i primer upisa u roditeljsko podstablo dat je na slici 7.



Slika 7: Ilustracija strukture podataka podstabala

Problem koji se javio tokom projekta, bilo je grupisanje po vektorskoj dimenziji. U prvoj verziji projekta grupisanje u vektorskoj dimenziji bilo je urađeno za korak *Inner-Product*, dok su *Project* i *Compress* bili izvršavani za svaku funkciju posebno. Kako se ne može odrediti koje će se stablo kada kreirati, moglo se desiti da za blok od četiri funkcije, tri funkcije završe korak *Compress* za određeno podstablo, a četvrta funkcija izvršava korak *Project* za roditeljsko podstablo (Slika 8).

U ovoj situaciji nije izvesno da li će funkcija četiri kreirati traženo podstablo ili ne. U slučaju da se podstablo ne kreira, bilo bi potrebno obezbediti mehanizam da predhodne tri funkcije započnu korak *Inner-Product* za dato podstablo bez funkcije četiri. Ovako nešto bi izuzetno povećalo komunikaciju, jer izgled stabala nije unapred poznat i ukoliko bilo koja funkcija iz bloka sadrži čvor koji ne postoji u funkciji četiri, bilo bi potrebno da se pošalje poruka za svaki takav čvor. Isto važi i za sve druge funkcije iz bloka.

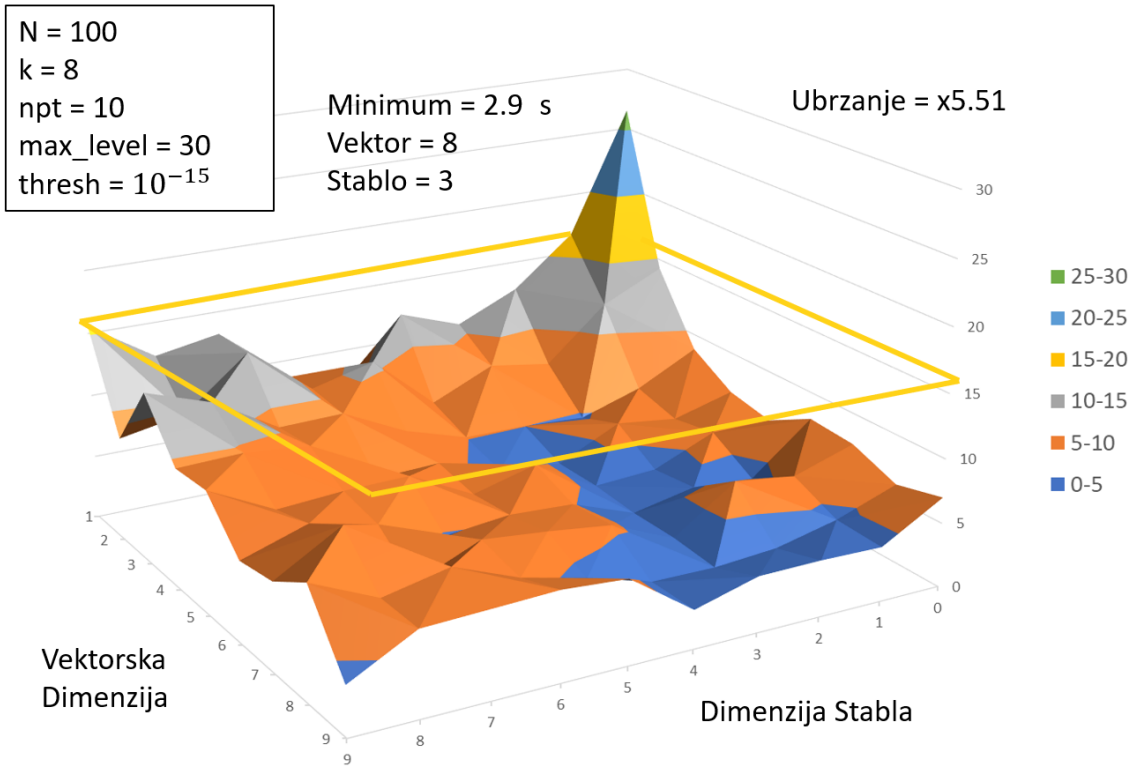


Slika 8: Inner-Product korak za grupu funkcija f1, f2, f2, f3

Rešenje problema bilo je da se koraci *Project* i *Compress* takođe grupišu u vektorskoj dimenziji, da bi funkcije iz bloka zajedno generisale podstabla na istim lokacijama. Na ovaj način nakon svakog *Compress* koraka imali bi sve informacije potrebne za korak *Inner-Product*, zbog čega se ovi koraci mogu integrisati.

9 Rezultati optimizovane aplikacije

Na slici 9 se može videti dobijeno vreme izvršavanja u zavisnosti na broj funkcija u bloku i dubinu podstabla. N označava broj funkcija, k i npt označavaju stepen polinoma i broj tačaka aproksimacije, *max_level* je maksimalna dubina stabala i *thresh* predstavlja zadatu preciznost. Žutim pravougaonikom označeno je vreme izvršavanja početne implementacije. Najbrže vreme izvršavanja dobija se za veličinu bloka od devet funkcija i dubinu podstabla od šest nivoa.



Slika 9: Zavisnost vremena izvršavanja od veličine blokova

10 Zaključak

U ovom radu dat je prikaz jedne aplikacije i njena optimizacija kroz povećavanje granularnosti. Aplikacija je pisana koristeći Intel CnC metodologiju, koja omogućava kreiranje specifikacije programa intuitivno kroz CnC graf. Ovaj pristup posebno je pogodan kod iregularnih struktura podataka, jer omogućava definisanje identifikatora kroz bilo koju strukturu podataka koja implementira operaciju jednakosti i *hash* funkciju. Još jedna prednost je što se algoritam specificira za osnovnu jedinicu granularnosti, što je bio čvor stabla u našem slučaju.

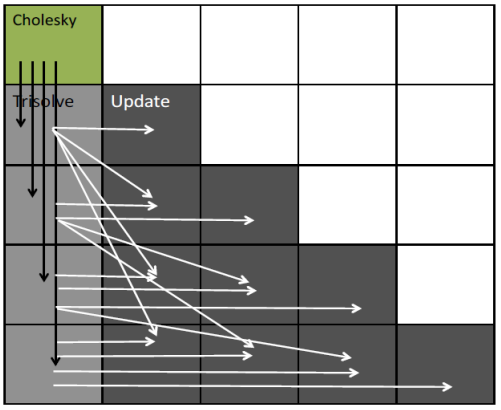
Optimizacioni deo vršio je grupisanje u tri dimenzije: dimenzija stabla, dimenzija vektora funkcija i dimenzija operatora. Uvođenje vektorske dimenzije i dimenzije stabla iniciralo je upotrebu kompleksnih struktura podataka, koje su pogodne za paralelno isvršavanje, kao što su to konkurentni vektori i atomični brojači. Takođe razmatrane su mogućnosti različitog grupisanja koraka izvršavanja i došli smo do zaključka da je potrebno sve korake grupisati u istim dimenzijama, kako bi dobijeni podaci nakon koraka bili na istom nivou hijerarhije.

Dobijeni rezultati govore da grupisanje koraka izvršavanja povećava performanse i da nije bitna samo veličina blokova, već i oblik. Ubrzanje optimizovane aplikacije bi bilo veće kada bi postojala mogućnost da se kolekcijama podataka pristupa sa više nivoa hijerarhije. U tom slučaju bi bilo moguće upisivati podatke na niskom nivou granularnosti, a čitati na visokoj, što bi smanjilo broj čitanja. Istraživački rad na hijerarhijskim kolekcijama podataka je trenutno u toku.

A Prvi dodatak

Primer jednog CnC programa je Čoleski faktorizacija koja ima za cilj dekompoziciju jedne matrice na proizvod dve matrice $L \cdot L^T$, pri čemu je L trougaona matrica. Kao početni algoritam uzeta je paralelno asinhrona verzija podesive granularnosti i sastoji se iz tri koraka: sekvencijalni čoleski algoritam, izračunavanje trougaone matrice i ažuriranje (eng. *cholesky, trisolve, update*) [3]. Veličina ulazne matrice B je nxn, pri čemu je $n = p * b$, gde b predstavlja veličinu bloka, a p broj blokova. Algoritam i ilustracija toka izvršavanja data je na slici 10.

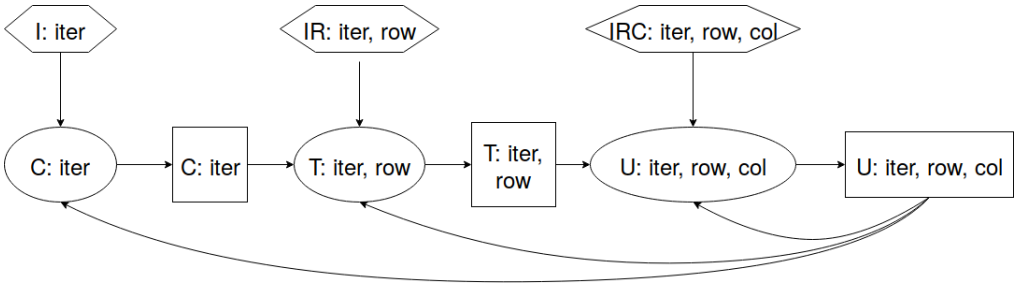
```
1 for k = 1 to p do
2 |   ConventionalCholesky(B_kk, L_kk);
3 |   for j = k + 1 to p do
4 |   |   TriangularSolve(L_kk, B_jk, L_jk);
5 |   |   for i = k + 1 to j do
6 |   |   |   SymmetricRank-
  |   |   |   kUpdate(L_jk, L_ik, B_ij);
```



Slika 10: Algoritam i ilustracija Čoleski faktorizacije

Prvi blok B_{11} dobija se korišćenjem sekvencijalnog čoleski algoritma, nakon čega blokovi koji se nalaze ispod njega u paraleli mogu da izračunavaju korak trisolve, što se može videti iz linija 3 i 4 iz algoritma. Posle svakog od koraka trisolve moguće je izvršiti korak ažuriranja na poljima označenim belim strelicama sa slike, nezavisno od izvršavanja ostalih koraka.

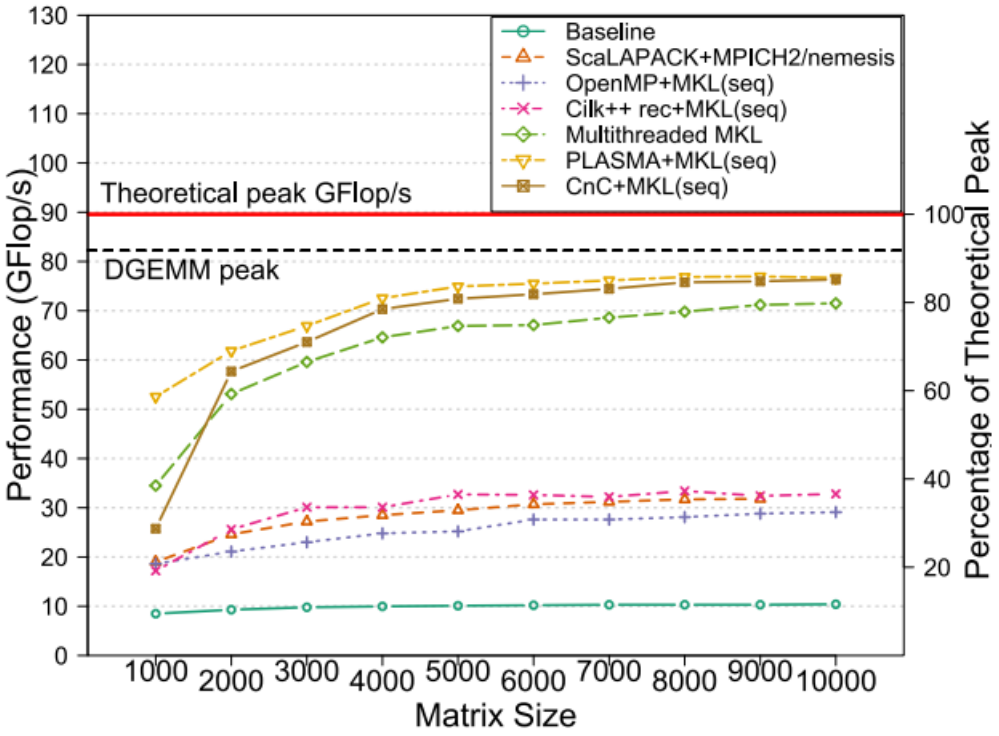
Čoleski algoritam pruža visok nivo paralelizma koji se jednostavno modeluje CnC grafom (Slika 11). Na CnC grafu elipse označavaju korake izračunavanja i obeleženi su velikim početnim slovima operacija (C), (T) i (U), dok oznake *iter, row, col* označavaju identifikator svake instance izračunavanja (eng. *tag*). Pravougaonik označava kolekcije podataka ([C], [T], [U]), dok su kontrolne oznake definisane sa <I:iter>, <IR: iter, row> i <IRC:iter,row, col>.



Slika 11: CnC model Čoleski faktorizacije

Izvršavanje počinje tako što se izvrši *put* operacija nad kontrolnom oznakom $< I : iter >$, pri čemu je iter=1. Ovo dovodi do stvaranja instance koraka izračunavanja, koji izvršava čoleski algoritam i vrši *put* operacije nad oznakama $< IR : iter, row >$, čime se kreiraju novi koraci izračunavanja za operaciju *trisolve* sa parametrima $iter = 1$ i $row = k + 1, k + 2, ..., p$. Takođe generiše se i kontrolna oznaka $< I : 2 >$, nakon čega se kreira korak izračunavanja $(C : 2)$, koji će biti izvršen tek kada budu dostupni podaci koje on konzumira. Na sličan način korak *Trisolve* kreira nove instance *Update* operacije. Kao rezultat *Update* koraka kreiraju se kolekcije podataka potrebne za izvršavanje svih operacija, većeg stepena identifikatora *iter*. Izvršavanje programa prestaje kada nema koraka koji su spremni za izvršavanje i kada se nijedan korak trenutno ne izvršava.

Poređenje CnC modela sa ostalim implementacijama čoleski algoritma je data u radu [2]. U ovom radu za svaki od koraka izračunavanja uzete su optimizovane sekvencijalne implementacije BLAS biblioteke. Kao osnova poređenja uzeta je sekvencijalna implementacija iz *Math Kernel Library* biblioteke, koja radi na oko $10 GFlops/s$. Prikaz dobijenih rezultata dat je na slici 12.



Slika 12: Poređenje performansi Čoleski faktorisacije

Poređenje je vršeno na dvokanalnom četvorojezgarnom procesoru Intel Xeon X5560 (Nehalem) na frekvenciji od $2,8GHz$

Literatura

[1] Cnc in a nutshell. `https://icnc.github.io/api/tutorial.html`. Accessed: 2018-02-11.

[2] Richard Vuduc Aparna Chandramowlishwaran, Kathleen Knobe. Applying the Concurrent Collections Programming Model to Asynchronous Parallel Dense Linear Algebra . (1):2, 2010.

[3] Richard Vuduc Aparna Chandramowlishwaran, Kathleen Knobe. Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems . (1):12, 2010.

[4] ROBERT J. HARRISON et Al. MADNESS: A MULTIRESOLUTION, ADAPTIVE NUMERICAL ENVIRONMENT FOR SCIENTIFIC SIMULATION. *SIAM Journal on Scientific Computing*, (5):123–142, 2016.

[5] Kathleen Knobe Intel Corporation Hudson Massachusetts Ryan Newton Intel Corporation Hudson Massachusetts Vivek Sarkar Rice University Houston Texas Michael G. Burke Rice University Houston, Texas. The Concurrent Collections Programming Model . *Technical Report*, (1):1, dec 2010.