Paralelizacija predviđanja pristajanja malih molekula na proteine

Abramović, Maja

Sveučilište u Rijeci – Odjel za informatiku

Jednopredmetni preddiplomski studij informatike

Maja Abramović

Paralelizacija predviđanja pristajanja malih molekula na proteine

Završni rad

Mentor: v. pred. dr. sc. Vedran Miletić

Rijeka, 18. rujna 2020.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 18.02.2020.

Zadatak za završni rad

Pristupnik: Maja Abramović

Naziv završnog rada: Paralelizacija predviđanja pristajanja malih molekula na proteine

Naziv završnog rada na eng. jeziku: Parallelization of protein – small molecule docking

Sadržaj zadatka:

Visokoprotočni virtualni pregled molekula koje su potencijalni lijekovi kroz predviđanje pristajanja tih molekula na proteine je tzv. sramotno paralelna operacija; može se trivijalno paralelizirati tako da se pregled svake pojedine molekule vrši na različitom (logičkom) procesoru. Za tu svrhu se na superračunalima uglavnom koriste upravitelji opterećenjem opće namjene kao što je Slurm ili specijalno razvijeni sustavi za skaliranje virtualnog pregleda molekula kao što je Virtual Flow. Na manjim računalnim sustavima kao što su poslužitelji, radne stanice, stolna i prijenosna računala ovakvi sustavi se ne koriste pa je virtualni pregled molekula često ograničen na korištenje samo jednog logičkog procesora i očekuje od korisnika da ručno razdijeli ulazne datoteke s molekulama na više dijelova i pokrene program više puta kako bi iskoristio sve dostupne logičke procesore. Kako suvremena prijenosna računala imaju od 4 do 16 logičkih procesora, stolna računala i radne stanice od 8 do 32 (u nekim slučajevima i do 128), a poslužitelji od 32 do 256, takvo ručno pokretanje zahtijeva izrazito puno rudimentarnog i repetitivnog rada od strane korisnika. Cilj rada je u alatu za visokoprotočni pregled molekula RxDock, forku alata rDock moderniziranom korištenjem C++-a 11, implementirati podršku za višenitni rad korištenjem mehanizmom dostupnim u C++-u po izboru studenta kako bi se olakšalo korištenje dostupnih resursa računalnih sustava s više logičkih procesora.

Mentor

Ime
v. pred dr. sc. Vedran Miletić

Voditelj za završne radove

doc. dr. sc. Miran Pobar

Zadatak preuzet: 20.02.2020.

(potpis pristupnika)

Sažetak

Pojavom multiprocesora i njihove komercijalne dostupnosti nastalo je povoljno tržiste za razvijanje softvera koji koriste multithreading. U toj domeni razlikuju se pojmovi konkurencije i paralelizma. Razlikujemo dvije klasifikacije paralelizma, a brzinu izvođenja paralelnog koda opisujemo Amdhalovim zakonom.

Paralelizacija na djelu vidi se na primjeru softvera RxDock koji je pisan u jeziku C++. Opisane su mogućnosti paraleliziranja koda u tom jeziku te potencijalni problemi koji se mogu dogoditi kod izvođenja. Objašnjen je tijek razmišljanja kod refaktoriranja koda.

Ključne riječi

otapala, udubljenja, ligandi, dokiranje, modeli, multiprocessing, multithreading, konkurentnost, paralelizam, paralelizam podataka, paralelizam zadataka, Amdhalov zakon, sramotni paralelizam, glavna nit, stanje utrke, međusobno isključivanje, deadlock, sinkronizacija, std::thread, std::async, bazen niti

Sadržaj

1. Uvod

RxDock [1] je softver otvorenog koda namijenjen pređivanju korisnih pozicija za prislanjanje malih molekula na proteine i nukleinske kiseline. Softver je fork softvera rDock [2] čije je razvijanje bilo obustavljeno 2014. godine nakon više od deset godina. Ponovno razvijanje softvera počelo je 2019. godine od strane RxTx-a te se i dalje aktivno razvija.

Predviđanje obavlja na unaprijed zamišljenom kemijskom sustavu koji čine otapala (engl. *solvent*), udubljenja (engl. *cavity*) na koja se molekule vezu i ligandi, molekule za koje se traže pripadajuća udubljenja. Ligandi i udubljenja u prirodi funkcioniraju na principu ključ-brava: jedno udubljenje može popuniti samo jedna molekula specifičnog kemijskog sastava. Molekule se digitalno obrađuju modelima, tj. tekstualnim datotekama čiji zapis identificira molekulu u kemijskom sustavu. Pristajanje (engl. *docking*) je metoda kojom se pretpostavlja najbolja pozicija za povezivanje dvije molekule, primjerice udubljenja i liganda.

Zadatak ovog završnog rada bio je paralelizacija procesa traženja odgovaruće pozicije za ligande zadane unaprijed određenim setom podataka, odnosno modela svih molekula u zadanom sustavu. Rad opisuje paralelan način izvođenja na višejezgrenim čipovima i smjernice kako refaktorirati kod u paralelni način rada.

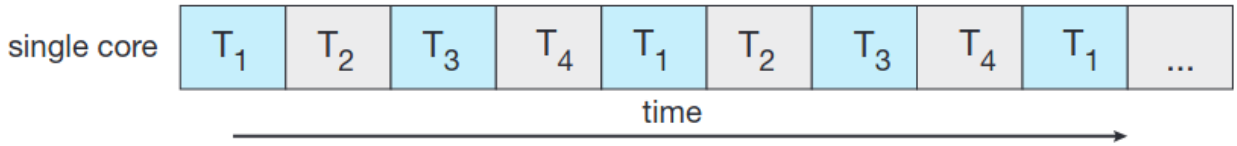
2. Paralelizacija

2.1. Mogućnosti višejezgrenih procesora

Nakon puna dva desetljeća komercijalnog korištenja mikročipova, u IBM-ovim laboratorijima nastao je prvi višejezgreni procesor: IBM100 Power 4 1GHz dvojezgreni mikroprocesor [3]. Nedugo nakon njegovog predstavljanja 1996. i službenog izlaska na tržište 2001. godine, višejezgreni procesori postali su standardni hardver. Danas je većina čipova višejezgrena i postavljaju temelj za heterogeno računarstvo, mogućnost da se koristi više različitih jezgri ili procesora odjednom. Višejezgreni procesori išli su u korak s Mooreovim zakonom[1] koji je 2016. godine dosegao svoje ograničenje, kako je i sam Moore predvidio da će se dogoditi. Također, vrijedi da se s više kompleksnosti procesorskog čipa povećao trošak proizvodnje.

Višejezgreni procesori omogućili su nam nove načine rada – višeprocesnost (engl. *multiprocessing*) i (engl. *multithreading*) [4]. Iako su dva pojma naizgled vrlo slična, koristimo ih u različitom kontekstu. Višeprocesnost je mogućnost da više jezgri obavlja različite poslove simultano, ali ne podrazumijeva nužno da više jezgri sudjeluje u obavljaju jednog posla. Kada govorimo o jednom poslu koji koristi više jezgri, onda koristimo višenitnost gdje je nit (engl. *thread*) dio koda predviđen da ga izvodi samo jedna jezgra. Također, prepoznajemo i pojmove konkurentnosti koja je mogućnost da jedna jezgra izvodi više procesa isprepleteno (konkurentno) u isto vrijeme te paralelizma koji je mogućnost da više jezgri izvodi više procesa simultano.

Možemo zaključiti da konkurentnost može postojati bez paralelizma, ali može biti i paralelizirana [5].

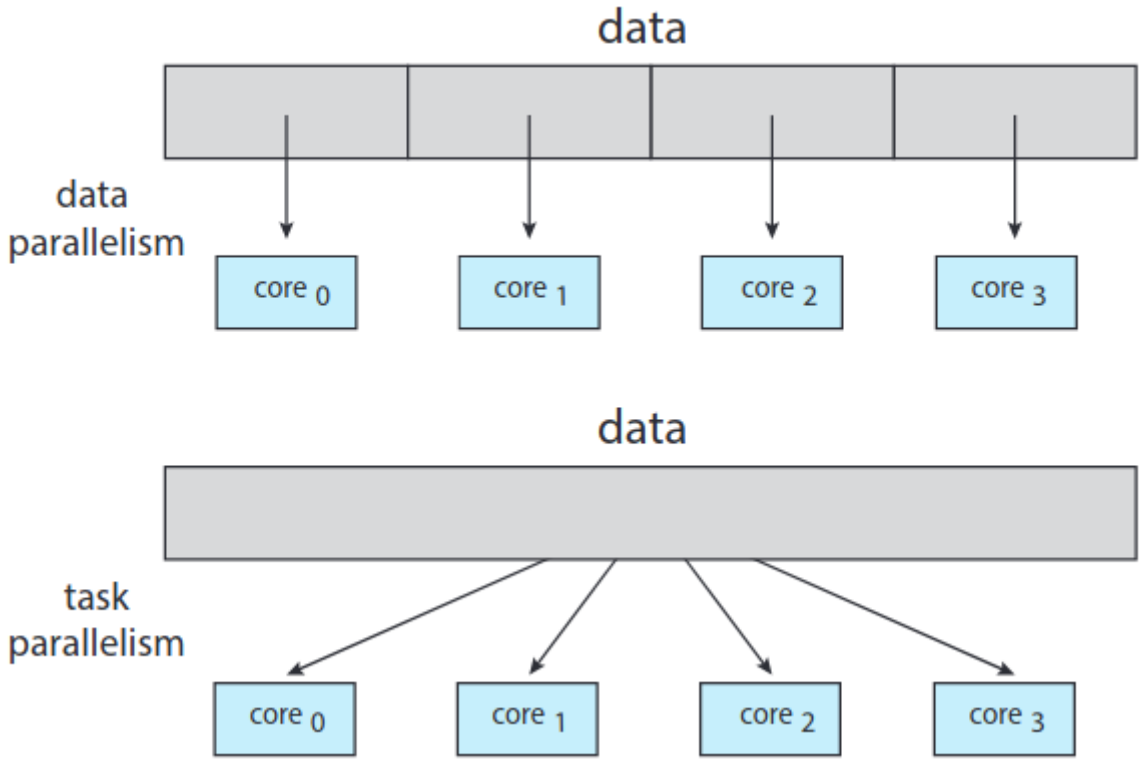


Slika 1: Grafički prikaz konkurentnosti

Paralelizam se dijeli na dvije vrste: paralelizam podataka (engl. *data parallelism*) i paralelizam zadataka (engl. *task parallelism*). Kada paraleliziramo podatke, svaka nit obavlja neku zadaću nad drugim podskupom podataka koji obrađujemo. Paralelizam zadataka svakoj jezgri daje posebnu zadaću, a svaka ta zadaća može i ne mora raditi na istom skupu podataka. Dvije vrste paralelizma se

1 Prvi Mooreov zakon (1965.): broj tranzistora na silikonskom čipu dvostruko se uvećava svake godine.

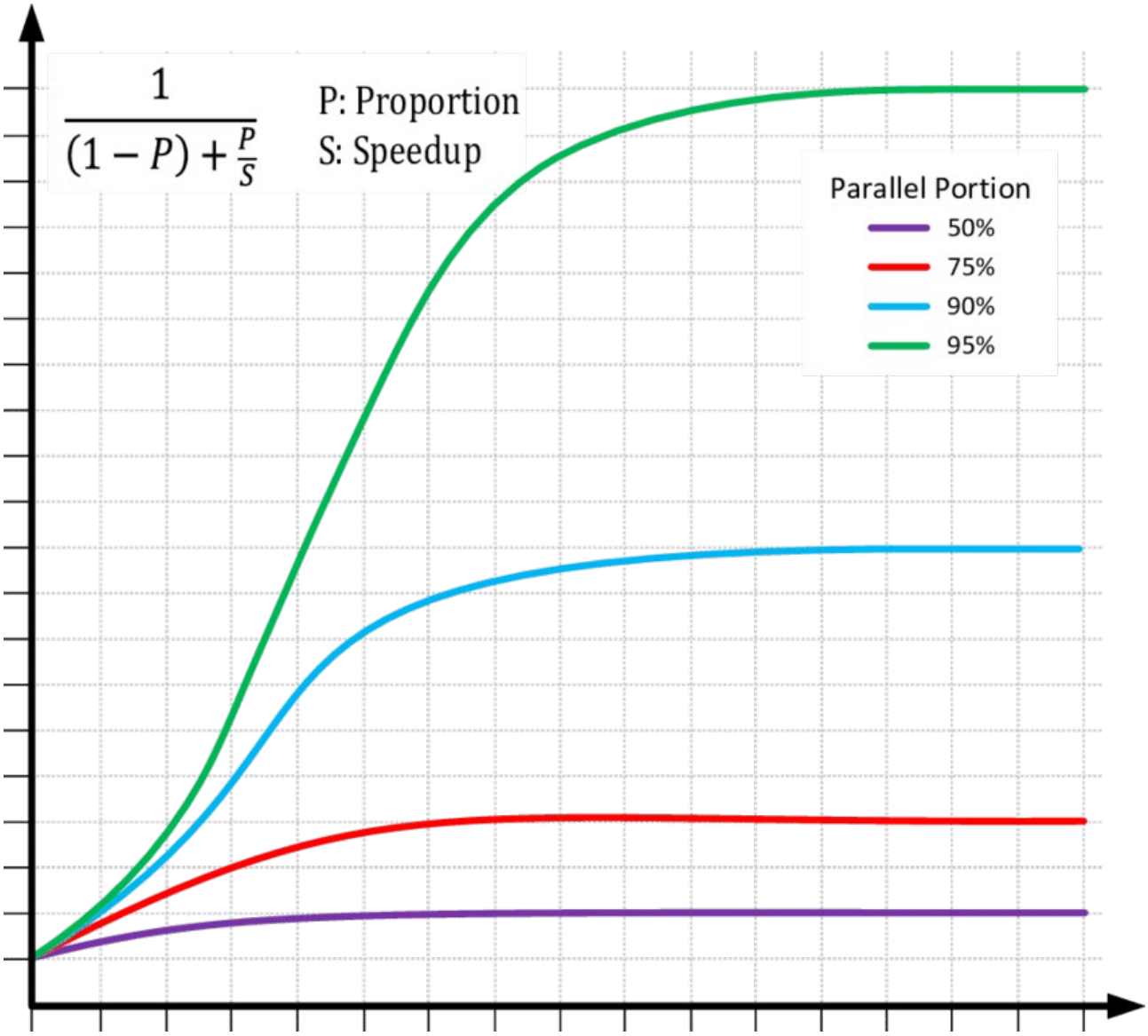
ne isključuju međusobno i moguće je razviti rješenje hibridnog oblika.



Slika 2: Grafička usporedba paralelizma podataka i paralelizma zadataka

Najvažnija prednost kod paralelizacije koda je očigledno ubrzanje obavljanja posla, odnosno skraćivanje vremena njegovog izvođenja. U idealnoj situaciji u kojoj pretpostavljamo da su sve jezgre identične bi dvojezgreni procesor bio duplo brži od onoga s jednom jezgrom, ali u praksi su idealne situacije vrlo rijetke. Dodavanje jezgri kod procesora je jedna od mogućih hardverskih promjena za koju se smanjenje latencije[2] može opisati Amdhalovim zakonom; tada je oznaka P broj niti potreban da se posao obavi, a S je broj dostupnih procesora. Vidimo izraz (1 – P) jer računamo da je jedna nit ona glavna od koje dolaze pozivi svima drugima. Sljedeći graf prikazuje ubrzanje latencije za postotak posla koji je moguće paralelizirati (os y) kroz vrijeme (os x). Dakle, što više posla možemo paralelizirati, to je latencija manja.

2 Latencija je vrijeme potrebno da se zadatak započne, izvrši i završi.



Slika 3: Graf ubrzanja izračunatog Amdahlovim zakonom [15]

U paralelnom računarstvu situaciju u kojoj kod zahtjeva minimalne ili nikakve promjene da bi kod bio paraleliziran prepoznajemo kao sramotni paralelizam (engl. *embarrassingly parallel*) [6]. Česti primjeri ove pojave su Monte Carlo simulacije, numeričke metode među kojima je i numerička integracija, render računalne grafike, brute-force pretraživanja u kriptografiji i slični algoritmi.

2.2. Način razmišljanja i programska podrška

Važno pitanje kod refaktoriranja serijskog koda je koje dijelove koda je moguće paralelizirati. U praksi uvijek postoji jedna glavna nit koja je pokretač svih zadataka koji se obavljaju u domeni jednog posla. Glavna nit ili nit roditelj (engl. *main thread*, *parent thread*) općenito možemo opisati:

- poziva ostale skripte, biblioteke i zaglavlja potrebna da se kod kompajlira
- definira varijable podatkovnih struktura koje su potrebne da se kod izvede
- obavlja input seta podataka potrebnog za izvođenje i output konačnog rezultata
- poziva funkcije koje koje manipuliraju ulaznim setom podataka

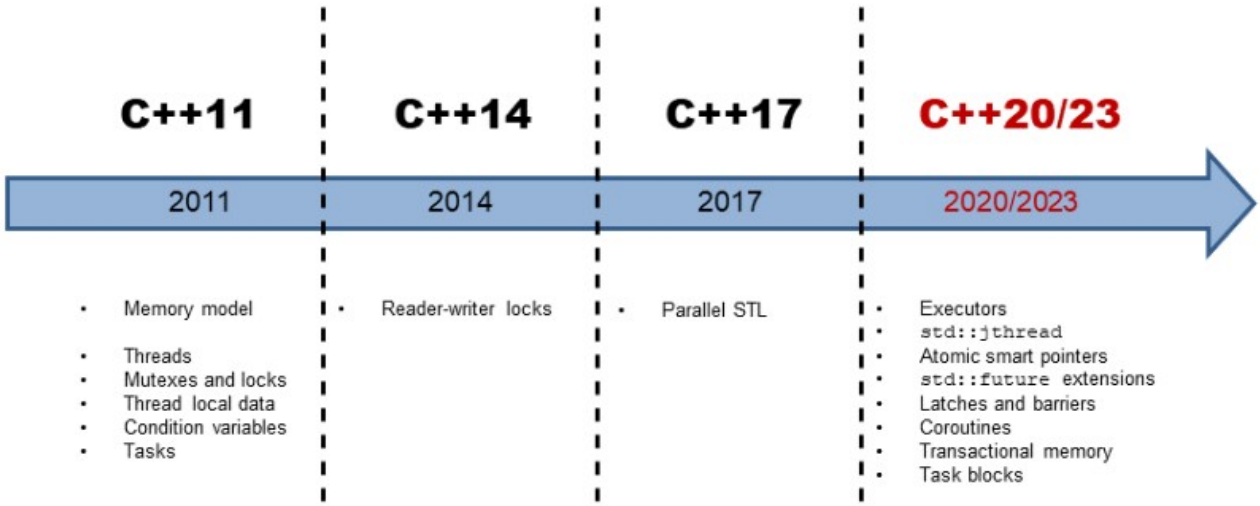
Osim smanjenja latencije, prednosti paralelnog izvođenja su značajna ušteda na resursima i mogućnost rješavanja većih, kompleksnijih problema. Međutim, paralelizirani kod zna biti nezgodan ukoliko je pogrešno dizajniran. Primjerice, susrećemo situaciju koja se u praksi zove stanje utrke (engl. *race condition*) koja opisuje problem koji nastaje zbog istovremenog izvođenja niti. Primjer takve situacije je kada jedna nit zapisuje novo stanje na memorijskoj lokaciji na kojoj druga nit čita – takva situacija može rezultirati pogrešnom vrijednosti na memorijskoj lokaciji i sveukupnim pogrešnim rezultatom izvođenja.

Da bismo spriječili situacije u kojima se javlja race condition, koristimo zaključavanja - međusobno isključenje (engl. *mutual exclusion*) poznato kraticom *mutex*. Tada nit zaključava resurs koji joj je u tom trenutku potreban i druge niti mu ne mogu pristupiti. Međutim, situaciju u kojoj jedna nit drži resurs zaključanim predugo i time obustavlja ili usporuje cijelo obavljanje zadatka zovemo potpuni zastoj (engl. *deadlock*).

Ono što smo postigli koristeći mutex je da su se niti međusobno vremenski uskladile – dogodila se sinkronizacija. Razlikujemo više vrsta sinkronizacija poput semafora ili monitora, ali preferiramo mutex zato sto želimo spriječiti međusobno blokiranje niti.

RxDock pisan je u programskom jeziku C++ i zato nam je interesantno kako je paralelizacija implementirana u tom jeziku i potencijalno njegovim dodatnim bibliotekama i proširenjima. C++11 iz 2011. godine je bio prvi koji je predstavio paradigmu u standardnoj knjižnici. Tada je paralelizacija bila moguća na dva načina – *std::thread* i *std::async*. C++17 omogućio je paralelnu izvedbu često korištenih algoritama poput algoritama za sortiranje vektora ili matrica, traženja elementa vektora ili matrica, prebrojavanje elemenata, kopiranje objekta, zamjene elementa i slično [7]. Potpuni popis podrzanih algoritama možemo pronaći na web sjedištu CppReference [8].

Općenito u paralelnom računarstvu razlikujemo dva načina pozivanja niti, sinkroni i asinkroni način. Nit roditelj započinje niti djecu, kod sinkronog načina roditelj čeka kraj djece da bi nastavio; kod asinkronog načina roditelj i djeca niti se izvršavaju samostalno, konkurentno. Upravo su to razlike dva načina paralelizacije u standardnoj knjižnici jezika C++ – *std::thread* obavlja se sinkrono, a *std::async* asinkrono. Da bismo omogućili rad više jezgri odjednom, u kodu implementiramo ono što nazivamo bazen niti (engl. *thread pool*), a to je struktura koja sadrži maksimalno niti koliko je dostupnih jezgri procesora.



Slika 4: Noviteti vezani za paralelno programiranje u standardima C++11, C++14, C++17, C++20 [14]

Vrijedi spomenuti da postoji mogućnost korištenja i drugih task programming frameworka kao što je Taskflow [9] ili korištenje sustava za redanje poslova na superračunalu kao što je Slurm [10].

3. Paralelizacija na djelu - RxDock

RxDock je od 2019. godine u aktivnom razvijanju, a to je omogućilo brzu i efikasnu prilagodbu u varijatnu koja podržava paralelizaciju pristajanja. Modernizacija koda je glavna misija današnjeg razvijanja - developeri teže k tome da se kod prilagodi suvremenim standardima jezika. S obzirom da su prve linije izvornog koda rDocka napisane sada već davne 1998. godine, razumno je da promatramo zastarjeli kod koji prati stariji standard C++-a.

Kako je opisano u prethodnom poglavlju, paralelizacija počinje od jedne niti roditelja, u ovom slučaju je to datoteka *rbdock.cxx* koju je moguće vidjeti u bilo kojem od repozitorija GitHub [11], GitLab [12] i ostalim poveznicama dostupnim na službenom webu RxDocka [1]. Ukoliko gledamo verzije koda bez paralelizacije, primijetiti ćemo da nova, paralelizirana verzija opisana ovdje u radu i one koje se mogu naći u repozitoriju rade u potpunosti iste operacije, samo sto novija verzija to radi paralelno.

Ukratko, možemo opisati da ova nit radi sljedeće:

- parsira parametre koji su dani kod pozivanja RxDocka i ispisuje ih korisniku
- prema datim parametrima postavlja mnogo varijabli koje nam nisu relevantne za paralelizaciju
- poziva *workspace* objekt koji se bavi informacijama vezanim za receptor, ligand i otapalo
- čita modele molekula i preusmjerava ih objektima pripadajućih klasa
- mjeri vrijeme izvođenja
- obavlja pristajanje liganda na receptor i upisuje rezultate u datoteke
- obrađuje greške

Za paralelizaciju ovog koda korišteni su *std::thread* za razdvajanje izvedbe u više niti i *std::mutex* za sinkronizaciju niti. S obzirom da je kod trebao malene varijacije da bi se mogao obraditi paralelno, možemo reći da se obavlja sramotna paralelizacija i da se radi o paralelizaciji podataka.

Važno je razaznati što ćemo promijeniti u serijskom kodu, a što će se odvijati paralelno. Ako pogledamo CppReference stranicu za *std::thread* [13], vidimo da svaka nit kao argument uzima neku funkciju ili klasu. U ovome slučaju, to je značilo da se kod koji će se odvijati paralelno mora prepisati u posebnu funkciju koju zovemo *dock_ligand()*. Ova funkcija će obavljati samo

pristajanje, zapisivanje rezultata i dohvaćanje greški za svaku pojedinu nit. Nit roditelj će nastaviti obavljati pripremu za pristajanje: podešavanje parametara, čitanje modela molekula, pozivanje niti, prikupljanje greški od svake pojedine niti i ispis.

Praksa kod korištenja biblioteke *thread* je da se niti inicijaliziraju unutar *thread friendly* strukture podataka, najčešće unutar vektora ili posebne klase. U ovom slučaju, korišten je *std::vector*. Na isti način su pozvani *workspace* i ligand objekti. Kod upisivanja u datoteke je mutex koji zaključava pristup datoteci za vrijeme upisivanja. Ideja izvedbe je da se broj dostupnih niti dobije korištenjem *std::thread::hardware_concurrency()*, a onda se učita isti broj molekula za koje želimo da se izvrši pristajanje. Iz tog razloga nam je potrebno jednako toliko *workspace* objekata; jedan objekt se bavi jednom molekulom.

Redom, napravljene su sljedeće promjene u kodu:

1. Inicijaliziran je vektor *workspace* objekata veličine broja dostupnih jezgri koji u nastavku ove for petlje podešava argumente za svaki objekt koji sadrži. Kada objektu podesi parametre, napravi *push_back()* objekta u vektoru.

```
std::size_t nCPUs = std::thread::hardware_concurrency();

// Create bimolecular workspaces
std::vector<BiMolWorkSpacePtr> vecWS;
for (std::size_t i = 0; i < nCPUs; i++) {
  BiMolWorkSpacePtr spWS(new BiMolWorkSpace());
```

Slika 5: Workspace objekti

2. Stvorena je while petlja iz koje se izlazi onda kada više nema modela za čitanje. Unutar nje se se inicijalizira vektor od maksimalno *nCPUs* (broj jezgri) liganada. Manje je liganada onda kada čita posljednje modele dostupne, a maksimalni broj liganada nije višekratnik broja jezgri.
3. Poziva se onoliko niti koliko je liganada učitano. Potom niti završavaju operacijom *join()*.

```
while (goOn) {

  std::vector<BiMolWorkSpacePtr> vecLigand;
  for (nRec = 0; nRec < nCPUs; spMdlFileSource->NextRecord(), nRec++) { ...}

  std::vector<std::thread> threads;
  for (std::size_t i = 0; i < vecLigand.size(); i++) {
    std::thread t(dock_ligand, bOutputHistory, nSeed, nFailedLigands, nRec,
                  loopBegin, totalDuration, nDockingRuns,
                  strOutputHistoryFilePrefix, bSeed, vecWS[i], vecLigand[i],
                  nUnnamedLigands);
    threads.push_back(std::move(t));
  }
  for (auto &t : threads) {
    t.join();
  }
}
```

Slika 6: Petlja u kojoj se čitaju ligandi i povezuju niti

4. Unutar funkcije *dock_ligand()* dodan je mutex kako bi se pravilno izvršilo pisanje u datoteku. Mutex varijabla je morala biti globalna.

Ovo je vrlo jednostavna i funkcionalna implementacija paralelizacije u RxDocku, ali uvijek postoji mogućnost boljeg rješenja. Ideja za poboljšanje ovog koda je učitati novi model po završetku neke niti, a ne svaki put učitati fiksnih nekoliko modela.

4. Zaključak

Paralelni način rada omogućio je brži rad softvera. Takav način rada nije svojstven samo operacijskim sustavima ili skupom softveru koji se koristi profesionalno, već bilo kojem softveru koji pokazuje predispozicije za paralelizaciju, kao što je, primjerice, znanstveni softver. To su nam omogućile dostupne knjižnice za višenitnost i sustavi ta redanje poslova na superračunalu koji se i dalje vrlo intenzivno razvijaju.

U budućnosti, će RxDock dobiti mnogo novog koda napisanog u duhu novijih standarada, a već sad se koristi isključivo C++11 bez nasljednih funkcija. U trenutku pisanja ovog rada, rujnu 2020. godine, C++20 prolazi svoje konačne tehničke provjere te se očekuje da će formalno postati standard krajem 2020. godine. Moguće je očekivati da se ovaj paralelni kod prepiše već za nekoliko mjeseci – C++20 najavljuje novo razvijanje višenitnosti u standardnoj knjižnici novom klasom *std::jthread* koja je poboljšana verzija sadašnjeg *std::thread*. Osim toga, moguća su i poboljšanja u smjeru korištenja asinkronog izvođenja.

Cilj ovog rada je postignut time što je paralelizacija uspješno implementirana unutar RxDocka. To je bitan doprinos jer će daljnje razvijanje sada gledati u još jednom pravcu i to tako da paralelizacija ima što bolje performanse. Otvoren kod dopušta proizvoljne promjene na bolje, a to je ono najbolje što budućnost može nositi za RxDock.

5. Literatura

[1] „RxDock“, *rxdock.org*. https://www.rxdock.org/ (pristupljeno ruj. 15, 2020).
[2] „rDock“, *rdock.org*. https://www.rdock.org/ (pristupljeno ruj. 15, 2020).
[3] „IBM100 - Power 4 : The First Multi-Core, 1GHz Processor“, ožu. 07, 2012. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/power4/ (pristupljeno ruj. 15, 2020).
[4] A. Silberschatz, G. Gagne, i P. B. Galvin, *Operating System Concepts, 10th Edition*. Wiley, 2018.
[5] A. Williams, *C++ Concurrency in Action: Practical Multithreading*. Manning, 2012.
[6] D. Vrajitoru, „Embarrassingly Parallel Programs“. https://www.cs.iusb.edu/~danav/teach/b424/b424_23_embpar.html (pristupljeno ruj. 15, 2020).
[7] B. Filipek, *C++17 in Detail*. Leanpub, 2018.
[8] „cppreference.com“. https://en.cppreference.com/w/ (pristupljeno ruj. 15, 2020).
[9] „Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System“. https://taskflow.github.io/ (pristupljeno ruj. 15, 2020).
[10] „Slurm Workload Manager - Documentation“. https://slurm.schedmd.com/ (pristupljeno ruj. 15, 2020).
[11] *rxdock/rxdock*. RxDock, 2020.
[12] „Projects · RxDock / RxDock“, *GitLab*. https://gitlab.com/rxdock/rxdock (pristupljeno ruj. 15, 2020).
[13] „std::thread - cppreference.com“. https://en.cppreference.com/w/cpp/thread/thread (pristupljeno ruj. 15, 2020).
[14] „A new Thread with C++20: std::jthread - ModernesCpp.com“. https://www.modernescpp.com/index.php/a-new-thread-with-c-20-std-jthread (pristupljeno ruj. 15, 2020).
[15] N. De Witte, R. Vincke, S. Van Landschoot, E. Steegmans, i J. Boydens, „(PDF) Evaluation of a Dual-Core SMP and AMP Architecture based on an Embedded Case Study“, *ResearchGate*. https://www.researchgate.net/publication/309829008_Evaluation_of_a_Dual-Core_SMP_and_AMP_Architecture_based_on_an_Embedded_Case_Study (pristupljeno ruj. 15, 2020).

Tablica slika