SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 216

RJEŠAVANJE PROBLEMA MEĐUDJELOVANJA N TIJELA U FIZIKALNIM SUSTAVIMA

Ivan Jukić

Zagreb, srpanj 2008

Sadržaj

1. UVOD U PROBLEM

1.1 *Znanstvena područja i problem međudjelovanja n tijela*

Postoji mnogo područja znanstvenog djelovanja koja dijele zajedničke probleme, nekad vrlo slične, nekad naizgled različite, ali sa istim temeljnim problemima. Jedan od takvih problema koje možemo povezati sa više različitih područja djelovanja je i problem međudjelovanja *n* tijela. Tako se ovaj problem pojavljuje u astronomiji kao određivanje gibanja planeta i galaksija, u molekularnoj biologiji kao slaganje proteina, u fizici povezan sa elektroničkom strukturom materijala i gibanja tijela, te u mnogim drugim područjima. Međutim, problem međudjelovanja je ponajviše fizikalni problem i kao takav se najčešće pokušava riješiti.

Od vremena Isaaca Newtona pa sve do danas, najveći matematički umovi pokušavali su riješiti ovaj problem. Općenito, početak možemo povezati sa 1687. godinom, kada je Isaac Newton u svojoj knjizi *Philosophiae Naturalis Principia Mathematica* opisao gibanje planeta u solarnom sustavu kao problem *n* tijela. Godine 1820. Laplace je sanjao o mogućnosti predviđanja dugotrajnog gibanja tijela u prirodi. Početkom 20-tog stoljeća, Poincare je otkrio da Newtonova mehanika ne daje analitičko rješenje problema, što ga je kasnije dovelo do definiranja teorije kaosa. Tijekom 20-tog stoljeća mnogi znanstvenici su se također odlučili pozabaviti problemom međudjelovanja u čestičnim sustavima, ali ne na analitički način kao njihovi kolege u prošlosti, već pomoću računalnih algoritama što je ovdje i prikazano [4, 5].

Neki od načina algoritamskog rješavanja problema prikazani su ovdje. U u drugom dijelu prikazan je kratak uvod u metode za rješavanje problema, te matematički opis samog problema. U trećem dijelu prikazani su detaljnije razvijeni algoritmi zajedno sa pseudokodovima i kratkim opisom drugih algoritama. U četvrtome dijelu opisana je platforma koju smo iskoristili za postizanje paralelizma kod razvijenih algoritama. U petom dijelu prikazana je usporedba između razvijenih sljednih i paralelnih algoritama, te u šestom dijelu zaljučak.

2. METODE I MATEMATIČKI OPIS PROBLEMA N TIJELA

2.1. *Uvod u algoritme za rješavanje problema n tijela*

Klasični problem međudjelovanja *n* tijela simulira evoluciju sustava sa n čestica, u kojem sila nametnuta na pojedinu česticu proizlazi iz njenog međudjelovanja sa svim ostalim česticama u sustavu [1]. Algoritmi koji rješavaju taj problem međudjelovanja čestica imaju mnogo primjena u različitim područjima kao što su astrofizika, dinamika proteina i fizika plazme. Takvi sustavi simuliraju se pomoću računalnih algoritama, stoga je nužno za napomenuti da svi takvi algoritmi ovise o vremenskim koracima u kojima se sustav promatra i unutar kojih se vrše proračuni (diskretizacija vremena). Nakon svakog takvog vremenskog koraka ukupna sila na svako tijelo ponovno se računa te se pomoću nje računaju novi položaj, brzina, te možebitni drugi atributi promatranog tijela. Ovdje se susrećemo i sa kompleksnosti tih algoritama, jednim od temeljnih problema zbog kojeg je i pokrenut razvoj različitih metoda za rješavanje problema međudjelovanja *n* tijela. Iako najtočnija i najjednostavnija, *particle-to-particle* metoda, ujedno je i najkompleksnija sa složenosti $O(n^2)$. Da bi se ta složenost smanjila razvijene su mnoge druge meotde uobličene u računalne algoritme, poput *tree-code* algoritama (koriste strukturu stabla za pohranu podataka) iz čije skupine izdvajamo Barnes-Hut algoritam sa složenosti $O(n\log n)$ ili poput *particle-to-mesh* algoritama koji koristeći brzu fourieriovu transformaciju (*fast fourier transformation,* FFT) također smanjuju složenost na $O(n\log n)$, te kompleksniji algoritmi kao *Fast Multipole Method* (FMM) čija je složenost $O(n)$ za uniformnu distribuciju čestica u sustavu [1].

Kao sljedeći razuman korak u poboljšanju efikasnosti algoritama za rješavanje problema međudjelovanja *n* čestica u fizikalnim sustavima bila je njihova paralelizacija. Tako npr. *The Stanford Splash benchmarks* sustav za provjeru računalnih performansi, kao jednu od metoda provjere performansi koristi Barnes-Hut algoritam [3]. Mnogi istraživači upješno su paralelizirali različite *n-body* algoritme

koji su se ekstenzivno koristili i još se koriste u astrofizici i molekularnoj dinamici. Također su se provele eksperimentalne i analitičke provjere vezane uz odstupanja simuliranih sustava od stvarnih, dakle istraživanja vezana uz pogreške koje se javljaju u proračunima algoritama. To je dovelo do zaključka da su analitičke granice pogrešaka vrlo pesimistične, dok se u praksi pokazalo da algoritmi daju puno manje greške [3].

2.2. *Matematički opis problema*

Prije daljnjeg prelaska na same *n-body* algoritme, važno je spomenuti da se mnogi fizikalni fenomeni (kod rješavanja diskretnih verzija kontinuiranih problema) mogu simulirati sustavom čestica, u kojem svaka čestica interaktira sa ostalim česticama prema zakonima fizike. Primjeri uključuju gravitacijsku interakciju između nebeskih tijela, te Coulombovu silu nametnutu od atoma u molekulama. Izazov efikasnog izvršavanja proračuna vezanih uz međudjelovanje čestica u sustavu općenito je poznat kao *n-body problem* [1].
Matematički, problem međudjelovanja *n* tijela u fizikalnim sustavima možemo definirati kao:

$$U(x_0) = \sum_i F(x_0, x_i) \qquad (1)$$

gdje je $U(x_0)$ fizikalna veličina u x_0 koju dobivamo sumirajući međusobne interakcije $F(x_0, x_i)$ u sustavu čestica.

Gledajući gravitacijsku silu i interakcije koje se javljaju kao njene posljedice, promatrajmo neku česticu x u tom sustavu sa masom m. Ukupnu silu koja djeluje na tu česticu možemo izraziti kao:

$$F(x) = \sum_{i=1}^{N} Gmm_i \frac{x - x_i}{\left|x - x_i\right|^3} \qquad (2)$$

gdje je G gravitacijska konstanta i iznosi $6.67 \cdot 10^{-11} Nm^2 / kg^2$. Budući se u stvarnom svijetu sve interakcije odvijaju u trodimenzionalnom prostoru, tako i jednadžbu (2) prikazujemo sa tri komponente, koje je čini pogodnom za algoritamsku implementaciju:

$$F(x) = \sum_{i=1}^{N} Gmm_i \left[\frac{x_1 - x_{i1}}{\left| x - x_i \right|^3} \vec{i} + \frac{x_2 - x_{i2}}{\left| x - x_i \right|^3} \vec{j} + \frac{x_3 - x_{i3}}{\left| x - x_i \right|^3} \vec{k} \right] \qquad (3)$$

Pomoću jednadžbe (3) računamo sile između čestica u sustavu, po komponentama. Ovdje primjećujemo da bi uspješno evaluirali jednadžbu (1) za svih *n* čestica u sustavu trebamo *n-1* operacija nad svakom česticom, čime dolazimo do kompleksnosti $O(n^2)$, što smo ranije spomenuli kao problem koji se uspješno može smanjiti na $O(n \log n)$, pa čak i na $O(n)$.

Ako bi sada promatrali neki fizikalni sustav sa *n* tijela, recimo da u njemu djeluje gravitacijska sila, tada bi sa svakom česticom povezivali dvije osnovne veličine – njenu masu i njen položaj koji ovisi o vremenu. Klasična Newtonova fizika kaže nam da se tada promjena položaja čestice može iskazati kao:

$$\frac{\partial}{\partial x^2} x_j(t) = \sum_{i=1,\, j \neq i}^{N} Gm_i \frac{x_j - x_i}{\left| x_j - x_i \right|^3} \qquad (4)$$

Budući se takvi sustavi promatraju u određenim diskretnim vremenskim koracima, desnu stranu jednadžbe (3) moramo integrirati za svaki vremenski korak od t_k do t_{k+1} kako bi dobili promjenu položaja [1]. Iz toga proizlazi:

$$\Delta x_j = \iint_{[t_k, t_{k+1}]} F(x_j(t)) dt dt \qquad (5)$$

gdje je:

$$F(x_j) = \sum_{i=1}^{N} Gm_i \frac{x_j - x_i}{\left| x_j - x_i \right|^3} \qquad (6)$$

Integracija jednadžbe (5) nije trivijalna, budući je x_j pristuan sa obje strane. Također x_i je ovisan o vremenu t, što znači da u svakom koraku imamo *n* vezanih integracijskih jednadžbi. Uz određene pretpostavke dobivamo diskretnu verziju jednadžbe (5) čiji je opći oblik:

$$\Delta x_j = \sum_{i=1}^{k} c_i F\left(x_j\left(t + h_i\right)\right), \qquad k < \infty \qquad (7)$$

i koja predstavlja linearnu kombinaciju funkcije (6) evaluirane u različitim vremenskim trenutcima.

Kako bi si olakšali računanje, uvodimo pojam brzine čestice. Sada će se u svakom vremenskom koraku mijenjati i brzina, zajedno sa položajem čestice. Provođenjem integracijske sheme (Eulerova integracijska shema) nad (7) i upotrebom brzine čestice, nove položaje čestica možemo definirati kao:

$$x_j' = x_j + T \cdot v_j' \qquad (8)$$

gdje je x_j' novi položaj čestice, x_j stari položaj čestice, T je diskretni vremenski korak, a v_j' je brzina čestice definirana kao:

$$v_j' = v_j + T \cdot F(x_j) \qquad (9)$$

gdje je v_j' nova brzina čestice, v_j stara brzina čestice, T je diskretni vremenski korak, a $F(x_j)$ je ukupna sila na česticu izračunata pomoću jednadžbe (5).

Jednadžbe (5), (8) i (9) vrlo su važne za ovdje prikazana algoritamska rješenja *n-body* problema, te čine oslonac tih algoritama.

3. ALGORITAMSKO RJEŠAVANJE PROBLEMA N TIJELA

U sljedećem dijelu biti će detaljnije prikazani algoritmi za rješavanje *n-body* problema. Posebna će pozornost biti posvećena prema četiri algoritma posebno razvijena za međusobnu usporedbu performansi, te određivanja prednosti paralelnih algoritama nad slijednim.

3.1. *Metoda Particle-to-Particle*

Za razliku od svih ostalih algoritama, *particle-to-particle* [1, 2] (u daljnjem tekstu PP) jedina je metoda koja ne koristi aproksimacije, pa je točnost jedino ovisna o točnosti računala koje vrši proračune. PP je ujedno i najjednostavnija metoda za implementaciju. Princip rada PP metode prilično je jednostavan. Za svaku od *n* čestica u sustavu radimo *n-1* operacija. Prvo izračunamo ukupnu silu na svaku česticu pomoću jednadžbe (6), zatim integriramo jednadžbu gibanja (7) jednom od integracijskih shema (tipično Euler ili Runge-Kutta) da dobijemo promjenu položaja, te sve ponovimo za sljedeći vremenski korak.

Iako je PP metoda dosta jednostavna, prilikom njene implementacije ponekad je dobro pažljivo razmisliti o numeričkim detaljima same fizike problema, koje ćemo formulirati u digitalnom obliku, tako da dobijemo što vjerodostojnije rezultate. Pretpostavimo li da se čestice približavaju, lako je za zaključiti kako se sile i akceleracije povećavaju. Ako kod integracijske sheme koristimo konstantan vremenski korak kako bi izračunali nove brzine i položaje, tada postoji mogućnost preljeva (*overflow*) rezultata. Preljev se izbjegava tako da se koriste varijabilni vremenski koraci umjesto konstantnih.

Mana PP algoritma je njegova kompleksnost ($O(n^2)$). Zbog velikih vremenskih potreba, PP metoda se ne koristi za proračune nad većim skupom čestica, ali se može efikasno koristiti za manje sustave čestica. Upravo zbog svojih vremenskih potreba, PP algoritam idealan je za paralelizaciju, tako da se uz zadržavanje točnosti, ali

nažalost i kompleksnosti, brže izvedu proračuni, te eventualno omogući njegova šira primjena.

3.2. *Razvijeni PP algoritmi*

Pogledajmo detaljnije strukture algoritama razvijenih u ovom radu. Na slici 1 prikazan je pseudokod slijednog algoritma, a na slici 2 paralelnog algoritma. Po definiciji, algoritam je slijedni ako sve operacije nad podatcima izvršava točno određenim redom, dok paralelni algoritmi mogu u jednom trenutku izvršavati više operacija nad svojim podatcima ili istu operaciju nad više podataka. Glavna ideja oba programa je ista, ostvarenje je drugačije. Oba algoritma slijedno izvršavaju proračune do trenutka kada su podatci potrebni za prelazak na paralelno računanje prikupljeni. Točnije, paralelni algoritam slijedno izračunava ukupne sile na svaku česticu kao i slijedni algoritam, ali zatim paralelno izračunava promjene brzine i promjene položaja, dok sa druge strane slijedni algoritam i dalje iterira po svakoj čestici mijenjajući joj brzinu i položaj.

```
//inicijalizacija sustava
postavi broj cestica;
postavi diskretni vremenski korak;
postavi broj iteracija;
dodijeli cesticama inicijalne polozaje;
dodijeli cesticama masu;
postavi pocetne brzine cestica na nulu;

//pocetak proracuna
dok ( broj iteracija < konacan broj iteracija )
{
    //izracunamo ukupne sile po cesticiama
    za svaku cesticu i
    {
        za svaku drugu cesticu j
        {
            izracunaj silu F(i,j);
            pribroji novoizracunatu silu ukupnoj sili na
                cesticu i;
```

```
        }
    }
    //nakon izracunatih sila po cestici racunamo promjene
    //polozaja i brzina
    za svaku cesticu i
    {
        izračunaj novu brzinu:
        izračunaj promjenu polozaja;
    }
//kraj iteracije
}
```

Slika 1 - pseudokod sljednog PP algoritma

```
//inicijalizacija sustava
postavi broj cestica;
postavi diskretni vremenski korak;
postavi broj iteracija;
dodijeli cesticama inicijalne polozaje;
dodijeli cesticama masu;
postavi pocetne brzine cestica na nulu;

//pocetak proracuna
dok ( broj iteracija < konacan broj iteracija )
{
    //izracunamo ukupne sile po cesticiama
    za svaku cesticu i
    {
        za svaku drugu cesticu j
        {
            izracunaj silu F(i,j);
            pribroji novoizracunatu silu ukupnoj sili na
                cesticu i;
        }
    }
    //racunamo promjene polozaja i brzina
    //VAZNO -> ovo se izvrsava paralelno
    paralelno_izracunaj_promjene();

//kraj iteracije
}
```

Slika 2 - pseudokod paralelnog PP algoritma

Iz pseudokoda ova dva algoritma lako se primjećuje sličnost algoritama, ali teže shvaća razlika. Ona nije samo u ovoj jednoj liniji pseudokoda već se odnosi na cijeli

sustav koji izvodi algoritam i alat kojim je pralalelizam postignut, što je i objašnjeno u poglavljima 4 i 5.

Samo ubrzanje izvođenja može varirati i ponekad može ovisiti i o samoj distribuciji čestica u sustavu u kojem se proračuni obavljaju. Najveće ubrzanje postiže se za veći broj čestica u sustavu, dok su za manje brojeve vremena izvođenja približno jednaka ili ovise o protoku podataka u sustavu.

3.3. *Metoda Particle-to-Mesh*

Dosada se PP metodom jedino diskretiziralo vrijeme, ali sa *particle-to-mesh* metodom (PM u daljnjem tekstu) ide se jedan korak dalje i diskretizira prostor [1, 2].
Ali prije detaljnijeg pojašnjenja, prvo se mora uvesti pojam potencijala.

Pretpostavimo da postoji veličina Φ koja je u odnosu sa fizikalnom veličinom U , koju smo ranije spomenuli, na sljedeći način:

$$U = \nabla\Phi \qquad (10)$$

I dalje da vrijedi:

$$\nabla \bullet U = c\rho \qquad (11)$$

gdje je *ρ* funkcija gustoće (npr. gustoće mase ili naboja) dobivena iz distribucije čestica, a *c* je konstanta. Iz toga dolazimo do Poissonove jednadžbe.

$$\nabla^2\Phi = c\rho \qquad (12)$$

U nam ovdje opet predstavlja silu, a Φ odgovara potencijalnoj energiji gravitacijskog polja koje stvaraju čestice u sustavu, a *ρ* je funkcija gustoće mase.

Ideja PM metode je da uspostavimo mrežu u prostoru u nad kojom izvodimo proračune tako da izračunamo potencijale u određenim točkama na toj mreži, a sile

koje djeluju u tim točkama mreže dobivamo tako da izračunamo gradijent potencijala tih točaka.

3.4. *Određivanje funkcije gustoće*

Funkcija gustoće [1, 2] određuje tip PM metode pa postoji više načina, tj. metoda, koje možemo upotrijebiti za njeno određivanje, stoga razmotrimo osnovna tri:

a) NG (*nearest grid point*)
b) CIC (*cloud-in-cell*)
c) Interpolacije višeg reda (*high order interpolation*)

Pomoću *nearest-grid-point* (NG), masa svake čestice pridjeljuje se točki u prostoru (točnije na mreži koju smo postavili u prostor) koja je najbliža toj čestici. Ovaj način određivanja funkcije gustoće često se referencira kao interpolacija nultog reda (*zero order interpolation*). Ipak ova metoda se rijetko koristi, jer jedan od problema su diskontinuitetne vrijednosti sila kao njena posljedica.

Cloud-in-cell (CIC) je metoda kod koje se masa također pridjeljuje točkama na mreži se nalaze oko promatrane čestice. Masa se ne pridjeljuje samo jednoj točki već se dio mase dodjeljuje i okolnim točkama, a to ovisi o samom položaju čestice. Oko svake točke u prostoru možemo zamisliti jednu ćeliju koja obuhvaća dio prostora. Sada odaberimo proizvoljnu česticu u prostoru i oko nje jedan „oblak“ koji može predstavljati volumen. Taj „oblak“ oko čestice ne nalazi se kompletno u jednoj ćeliji, već se u svakoj od okolnih ćelija nalazi dio oblaka. Po *nearest-gridpoint* metodi masa te čestice dodjelila bi se samo onoj točki kojoj je ta čestica najbliža, ali po *cloud-in-cell* metodi masa se proporcionalno dodjeljuje točkama čije prostorne ćelije presjecaju „oblak“ od čestice. Masa čestice raspodjeljuje se po susjednim ćelijama u prostoru. U trodimenzionalnom prostoru to je osam ćelija, a u dvodimenzionalnom prostoru četiri ćelije, iz čega se vidi da broj ćelija koje dijele masu čestice ovisi o dimenziji prostora (

2^N - N je dimenzija).

Sa interpolacijama višeg reda možemo povezati *cloud-in-cell* metodu. Kada bi recimo „oblak“ čestice (zamišljeni prostor oko čestice) bio dovoljno velik da zauzima cijele ćelije, pa sve skupa i više od četiri ili osam ćelija, tada bi koristili kompleksnije metode za određivanje funkcije gustoće što bi rezultiralo intrepolacijama višeg reda. Jedna takva metoda je i *triangular shaped cloud* (TSC). Naravno, TSC metoda je daleko točnija od NG i CIC metode, koristi prijenosnu interpolacijsku funkciju, a u jednoj dimenziji koristi tri točke u prostoru.

3.5. *Određivanje potencijala*

Sada kada smo utvrdili načine dobivanja funkcije gustoće, spremni smo za rješavanje jednadžbe (10) da dobijemo potencijale u određenim točkama mreže u prostoru [1]. Glavna prednost PM metode je njena brzina. Broj proračuna koji je potrebno izvesti dovodi do kompleksnosti od $O(N + N_g \log N_g)$ gdje N_g predstavlja broj točaka na mreži. Najsporiji korak kod PM metode je rješavanje jednadžbe (10), dakle jednadžbe potencijala. Ta se jednadžba najčešće rješava „Poissonovim rješavateljem“, koji se često oslanja na brzu fourierovu transformaciju (*fast fourrier transformation,* FFT). Jedne od mogućih alternativnih numeričkih metoda za rješavanje jednadžbe potencijala su i metoda konačnih elemenata ili metoda konačnog volumena.

PM metoda se ne koristi za proučavanje bliskih susreta između čestica, jer algoritam u principu čestice tretira kao položajem neodređene. Za najbolje rezultate, udaljenosti između točaka mreže trebale bi biti manje od veličina koje definiraju naš fizikalni sustav.

Još jedan problem PM metode je i problem sa ne-uniformnom raspodjelom čestica u sustavu, za koju se dobiva ograničena rezolucija (razlučljivost). Taj se problem rješava tako da se u područjima od veće važnosti gustoća mreže, koja određuje točke

prostora, poveća. Takve gušće mreže omogućuju točnije modeliranje interakcija u područjima sa većim brojem čestica.

3.6. *Razvijeni PM algoritmi*

Po uzoru na PM metodu u ovom radu napravljena su dva algoritma, jedan slijedni (slika 3) i jedan paralelni (slika 4). Princip rada algoritama sljedi osnovna načela PM metode, ali radi zadržavanja jednostavnosti umjesto proračuna potencijala, računaju se sile između određenih ćelija u prostoru, a za određivanje funkcije gustoće koristi se *nearest grid point* metoda. Bitno je za uočiti da samim time nismo dobili na smanjenju kompleksnosti, ali zbog ograničenja koje smo nametnuli na prostor (prostor je fiksne veličine) koji promatramo, sada i sam položaj čestica ovisi o tome da li ih uzimamo u obzir kod računanja. Ako se čestice ne nalaze u promatranom prostoru ne uzimamo ih u obzir kod računanja, a uzimamo ako se nalaze unutar prostora. Prostor možemo nazvati i *prostorom interesa*, jer nas zanimaju samo čestice unutar njega, a ako je potrebno, prostor se može i povećati. Ovim skupom aproksimacija postižemo znatno ubrzanje u odnosu na PP algoritme.

Kod samog rada programa, oba algoritma do određenog djela obavljaju iste operacije, ali u trenutku kada su svi potrebni podatci prikupljeni, paralelni algoritam prelazi na paralelno računanje, a slijedni i dalje izvršava naredbu po naredbu.

```
//inicijalizacija sustava ista je kao i za PP algoritme
//ali ovdje primjećujemo i postavljanje veličine
//promatranog prostora da odredimo prostornu mrežu
postavi broj cestica;
postavi diskretni vremenski korak;
postavi broj iteracija;
dodijeli cesticama inicijalne polozaje;
dodijeli cesticama masu;
postavi pocetne brzine cestica na nulu;
```

```
postavi veličinu promatranog prostora;
//pocetak proracuna

dok ( broj iteracija < konacan broj iteracija )
{
    //izracunamo ukupne sile po cesticiama
    za svaku cesticu i
    {
        pogledaj koja joj je tocka mreze najbliza;
        ako je cestica izvan prostora ne uzimamo je dalje
            u obzir kod racunanja;
        inace dodaj masu cestice promatranoj tocki;
        ako tocka nije u listi, dodaj tocku u listu
            promatranih tocaka;
    }
    dok pointer lista != NULL {
        dok pomocni pointer lista != NULL {
         //računamo silu između svake dvije točke u listi
         //i dodajemo ukupnoj sili na promatranu točku
            izracunaj silu između tocaka;
            dodaj silu ukupnoj sili na promatranu tocku;
        }
    }
    //izracunamo konacne promjene parametara
    za svaku cesticu {
        ako unutar prostora racunanja {
            izracunaj novu brzinu;
            izracunaj novi polozaj;
        }
    }
//kraj iteracije
}
```

Slika 3 - pseudokod sljednog PM algoritma

```
//inicijalizacija sustava ista je kao i za PP algoritme
//ali ovdje primjećujemo i postavljanje veličine
//promatranog prostora da odredimo prostornu mrežu
postavi broj cestica;
postavi diskretni vremenski korak;
postavi broj iteracija;
dodijeli cesticama inicijalne polozaje;
dodijeli cesticama masu;
postavi pocetne brzine cestica na nulu;
postavi veličinu promatranog prostora;
//pocetak proracuna
dok ( broj iteracija < konacan broj iteracija )
{
    //izracunamo ukupne sile po cesticiama
    za svaku cesticu i
    {
        pogledaj koja joj je tocka mreze najbliza;
        ako je cestica izvan prostora ne uzimamo je dalje
            u obzir kod racunanja;
        inace dodaj masu cestice promatranoj tocki;
        ako tocka nije u listi, dodaj tocku u listu
            promatranih tocaka;
    }
    dok pointer lista != NULL {
        dok pomocni pointer lista != NULL {
         //računamo silu između svake dvije točke u listi
         //i dodajemo ukupnoj sili na promatranu točku
            izracunaj silu između tocaka;
            dodaj silu ukupnoj sili na promatranu tocku;
        }
    }
    //spremamo podatke u posebno polje za paralelno
    //računanje
    dok pointer lista != NULL {
        stavi silu na česticu u točki prostora u polje
            sile;
    }

    //PARALELNO IZRAČUNAMO PROMJENE PARAMETARA
    paralelno_izracunaj_promjene();

//kraj iteracije
```

Slika 4 - pseudokod paralelnog PM algoritma

3.7. *Ostale metode rješavanja problema n tijela*

a) Particle-particle/particle-mesh metoda (P^3M)

P^3M [1, 2] je hibridna metoda koja pokušava rješiti probleme PM metode kod modeliranja sila koje djeluju na malim udaljenostima. Stoga se sile u sustavu dijele na brzo mijenjajuće sile kratkog dosega i sporo mijenjajuće sile dugog dosega. Tada se PM metoda koristi za za određivanje sila dugog dosega, a PP metoda za sile kratkog dosega. Obično se pod sile kratkog dosega smatraju sile koje djeluju na udaljenosti manjoj od $3 \cdot x$, gdje je x udaljenost između točaka mreže.

b) Tree codes

Tree codes su algoritmi koji koriste strukturu stabla za prikaz interakcija između čestica [1, 2]. Mnogi fizikalni sustavi imaju svojstvo da izračun danog polja $U(x)$ unutar nekog malog područja može biti iznimno kompleksan, ali jednostavan na nekoj udaljenosti od tog područja. Dakle, skupove čestica na određenoj udaljenosti od promatrane čestice možemo aproksimirati jednostavnijim prikazom (npr. jednom većom i težom česticom).
U dvodimenzionalnom prostoru ta struktura je *quadtree* (svaki čvor stabla ima četiri djeteta), a u trodimenzionalnom *octree* (svaki čvor stabla ima osam djeteta). Ovaj pristup rješavanju problema međudjelovanja *n* tijela je efikasan jer se može primjeniti rekurzivno. Problemi vezani uz *tree code* algoritme su veća potražnja za memorijom, te manja točnost od jednostavnijeg PP algoritma. Poseban tip *tree code* algoritma je ranije spomenuti Barnes-Hut algoritam. Algoritam je predstavljen 1986. godine i intenzivno se koristi u astrofizici, a ujedno je i efikasno paraleliziran.

c) Fast multipole method (FMM)

FMM metodu možemo smatrati jednom vrstom *tree code* algoritma koja koristi dvije reprezentacije polja potencijala: daleko polje (*far field* – multipole) i lokalne ekspanzije (*local expansions*) [1, 2, 3]. Ova metoda koristi brze proračune potencijalnog polja. Izračunski je lakše obraditi potencijal od sile, jer je sila vektor, dok je potencijal skalar, a sila je negativna vrijednost gradijenta potencijala. U principu dijeli paradigmu sa Barnes-Hut algoritmom, ali sa nekim određenim razlikama (jedna od njih je da se koristi upravo potencijal, Barnes-Hut koristi silu).

d) Neopisani algoritmi

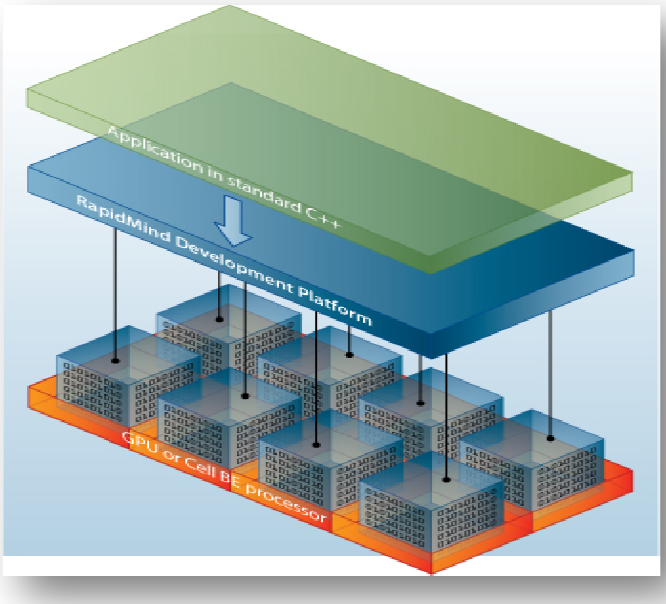
Naravno da ovih pet metoda ovdje opisanih (od kojih dvije i implementirane) nisu jedine metode za rješavanje problema *n* tijela. Većina ostalih metoda koristi kombinacije ovdje navedenih metoda, ali postoje i druge metode koje koriste bitno drugačije i kompleksnije pristupe. Dvije takve su *Self-consistent field* (SCF) – za razvijanje, evoluiranje, sunčevih sustava bez sudara (čestice ne interaktiraju međusobno direktno već kroz doprinos gravitacijskom polju sustava), te *Symplectic* metoda (temelji se na Hamiltonijanskoj mehanici) [2].

4. RAPIDMIND PLATFORMA

4.1. *Uvod u Rapidmind platformu*

Iako prvotno zamišljen kao jezik za pisanje programa eksplicitno za grafičku karticu, *Rapidmind* se razvio u platformu koja daje punu podršku za paralelno programiranje. Programeru je pri tome na raspolaganju jednostavan i standardiziran način programiranja aplikacija. O samom izvođenju aplikacija brine *Rapidmind* platforma koja zadatke raspodjeljuje ovisno o dostupnim resursima u sustavu. Rapidmind se koristi kao skup knjižnica i modula, a može se implementirati u sve ISO standardizirane C++ prevoditelje [9, 10].

Dobra strana kod uporabe *Rapidmind* platforme je ta da programer ne mora znati ništa o procesoru za koji radi aplikaciju. Kada se aplikacija pokrene, *Rapidmind* platforma sama utvrđuje da li će se izvoditi na grafičkom procesoru, Cell procesoru ili nekom trećem, te prevodi kod koji je programer napisao u kod koji određeni procesor podržava.



Slika 5 - odnos Rapidmind platforme i sustava

4.2. *Paraleliziranje aplikacije*

Postoje dva glavna pristupa paraleliziranju aplikacija [7, 8, 11]. Prvi je praleliziranje programskih zadataka. Model kod kojeg se paraleliziraju zadatci zahtjeva da se aplikacija podijeli u određen broj manjih zadataka koji se obično implementiraju korištenjem višedretvenosti. Ako imamo mali broj zadataka, tada je aplikaciju teško paralelizirati na više od malog broja procesora ili procesorskih jezgri, a ako je broj zadataka velik, tada se javljaju problemi sa sinkronizacijom podataka i samo ispravljanje pogrešaka (*debuging*) može predstavljati veliki problem.

Rapidmind koristi drugi model paralelizacije kod kojeg se ne paraleliziraju zadatci, već podatci [9, 10]. Sve dok postoji dovoljan broj podataka koji se trebaju procesirati, model paralelizacije podataka može se skalirati na velik broj procesorskih jezgri, a pritom zadržati jednostavnost. Zbog toga se aplikacija mora modelirati kao skup izračuna koji će se izvršiti nad elementima polja koji sadrže podatke. U samoj *Rapidmind* platformi, podatci se spremaju u posebna polja koja su definirana u platformi, a paralelni izračuni se obavljaju pomoću programskih objekata, također definiranih u platformi. Na programeru je samo dizajnirati polja koja će sadržavati podatke, a onda samo definirati izračune nad tim podatcima u obliku programskih objekata. Ovo je glavni razlog zbog kojeg prethodno opisani paralelni algoritmi djelomično izračune obavljaju na slijedni način, upravo zato da prikupe podatke u prikladna polja i tek tada izvrše paralelne izračune nad njima koristeći odgovarajuće programske objekte.

Prilikom samog izvođenja programskih objekata, *Rapidmind* lomi polja tih podataka u dijelove koji se tada prenose svim dostupnim jezgrama. Što je više jezgri raspoloživo to je više dijelova u koje se podatci lome. Kako bi se osiguralo da se jezgre maksimalno iskoriste, sustav dodjeljuje zadatke prema tome kako se koja jezgra oslobodi (*on the fly*). Na primjer, ako jedna jezgra posebno dugo obavlja neki izračun, ostale jezgre koje izračune obavljaju brže, dobivaju za to vrijeme druge zadatake i tako se cijelo vrijeme iskorištavaju [9].

4.3. *Rapidmind sučelje*

Sučelje *Rapidmind* platforme predstavljaju tri glavna tipa za pohranu podataka i izračuna koje dizajner aplikacije može koristiti: ***Value*** tipovi, ***Array*** tipovi, te ***Program*** tipovi [9]. Ovi tipovi temelje se na dobro poznatim C++ tipovima podatka i formiraju sučelje *engine*-a za paralelno izvršavanje aplikacija, baziranog na generiranju dinamičkog koda pomoću *program* tipa koji je ključna inovacija u *Rapidmind* platformi. Na prvi pogled, možda nam se ne čine posebno zanimljivi ni *value* tipovi, ni *array*, ali semantika ovih tipova namjerno je izabrana kako bi se olakšalo njihovo učenje.

Value tipovi podataka predstavljaju sekvencu brojeva fiksne duljine koji mogu biti npr. tipa *float*, *integer* ili *boolean*. *Value* tipovi su veoma korisni za prikaz podataka sa više komponenti (u našem primjeru sila je predstavljena pomoću tri komponente). *Value* tipove sintaksno definiramo kao **Value<*N*,*T*>**, gdje je N predstavlja broj komponenti (N-torku) koje ta varijabla sadrži, a T je tip podatka. Broj komponenti može biti bilo koji cijeli broj veći od nule. *Rapidmind* prevoditelj (*compiler*) interno, ako tako sklopovlje zahtjeva, varijable razbija u manje dijelove (npr. varijable sa četiri komponente), ali o tome programer ne mora brinuti. Umjesto dugog oblika **Value<*N*,*T*>** možemo koristiti i skraćeni oblik **Value*Nt***, gdje je N opet broj komponenti, a t je skraćeni naziv tipa podatka, npr. **Value<3, float>** možemo također napisati kao **Value3f**. Skraćeni oblik podržava *Value* tipove koji imaju 1, 2, 3 ili 4 komponente.

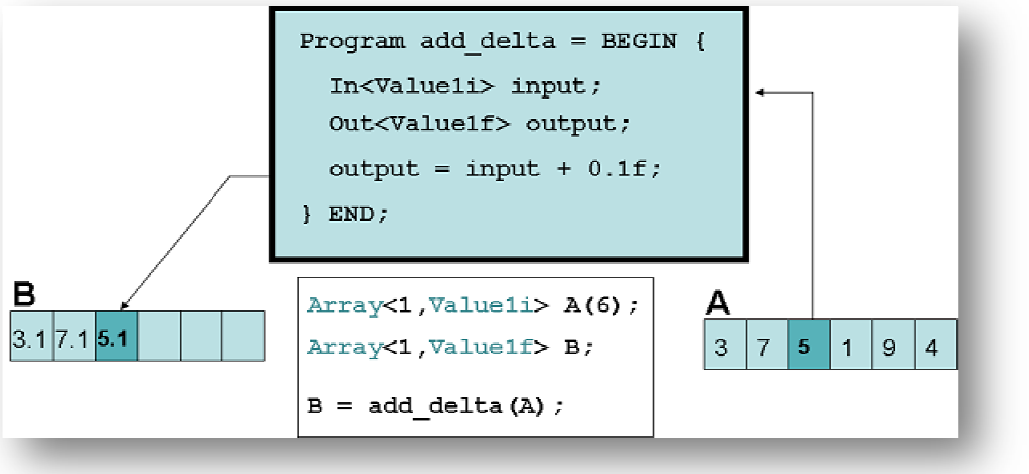
Array tipovi podataka su osnovni tip u *Rapidmind* platformi koji predstavlja proizvoljno velik skup podataka. Oni pokušavaju imitirati C++ polja, ali su puno općenitiji. *Array* tipovi su napravljeni nad *Value* tipovima, pa stoga možemo koristiti *Value* tipove da specificiramo sadržaj polja. *Array* tipovi imaju dimenziju koja može biti 1, 2 ili 3. Sintaktički *Array* tip definiramo kao **Array<*D*,*T*>**, gdje je D dimenzija polja, a T je *Value* tip elementa, npr. **Array<3, Value3f>** predstavlja trodimenzionalno polje čiji su elementi *Value* tipovi koji sadrže tri *float* komponente.

Elementima polja možemo pristupiti na više načina. Jedan dobro poznati je pomoću uglatih zagrada, ali možemo im pristupiti i kao standardnim C++ poljima pomoću **Array::read_data()** i **Array::write_data()** funkcija, ovisno o tome koje operacije želimo izvršiti nad poljem (čitanje ili pisanje). Ove dvije funkcije vraćaju standardne C++ pokazivače koje možemo iskoristiti za pristup elementima polja, ali rade i puno više od toga. Budući se podatci mogu nalaziti u memoriji različitoj od one kojoj procesor koji obrađuje podatke ima pristup, mogu se dogoditi situacije u kojima je potrebna sinkronizacija ili prijenos podataka što može vremenski biti zahtjevno, stoga na ovaj način eksplicitno definiramo operaciju koju želimo izvršiti.
Array tipovi mogu se pomicati (*shift*), može im se mijenjati veličina i još puno drugih operacija, a ne sadržavaju uvijek nužno podatke (*Virtual Arrays*).

Program objekti su skupovi izračuna koji se obavljaju sa *Rapidmind* tipovima i operacijama. Ti izračuni se obavljaju na ciljanom računalnom skolopovlju (centralni procesor, grafički procesor, Cell procesor). Programski objekti slični su funkcijama u standardnom C++, predstavljaju skup izračuna koji se pozivaju u programu, ali za razliku od standardnih C++ funkcija objekte ne prevodi C++ prevodioc. Umjesto toga, oni se formiraju prilikom pokretanja programa (*runtime*). Poput C++ funkcija mogu imati više ulaznih parametara, ali za razliku od standardnih funkcija koje imaju samo jednu povratnu vrijednost, programski objekti mogu imati više izlaznih vrijednosti.
Rapidmind programski objekti slijede *Single Program Multiple Data* (SPMD) model , što znači da ne sadrže samo jednostavne nizove naredbi za računanje sa podatcima, već mogu sadržavati i naredbe za kontrolu toka poput *if* naredbi i *for* petlji.
Naredbe koje čine *Program* objekt sadržane su unutar dvije makro naredbe, RM_BEGIN i RM_END. Same naredbe programskog objekta izvršavaju se u tzv. *retained* modu. Prilikom pokretanja aplikacije, naredbe unutar objekta se snimaju i spremaju za izvršavanje kada to bude potrebno, dakle njihovo se izvršavanje zadržava (eng. *retain*). Naredbe izvan definicije programskog objekta izvršavaju se u tzv. *immediate* modu, što znači da se izvršavaju odmah (eng. *immediately*). Ovo vrijedi samo za *Rapidmind* tipove podataka, dok za standardne C++ tipove i naredbe smatramo da se uvijek izvršavaju u *immediate* modu.
Sučelje *Program* objekta čine njegovi ulazni i izlazni argumenti, koji su uvijek *Value*

tipa. Unutar samog objekta ulazne i izlazne parametre specificiramo pomoću odgovarajućih omotača, ako je argument ulazni definiramo ga kao **In<T>**, ako je izlazni definiramo ga kao **Out<T>**, a ako je i jedno i drugo tada koristimo **InOut<T>** omotač. T ovdje predstavlja *Value* tip podatka. Redosljed kojim definiramo ulazne i izlazne argumente vrlo je bitan. Kada se programski objekt pozove iz programa, redosljed kojim se argumenti predaju objektu i redosljed izlaznih argumenata moraju odgovarati prethodnoj definiciji ulaznih i izlaznih argumenata u objektu. Po konvenciji, ulazni i izlazni argumenti definiraju se na početku programskog objekta, iako to nje nužno.

Na slici 6 možemo konceptualno prikazati kako točno programski objekt vrši operacije nad elementima polja. U ovom primjeru, podatci se uzimaju iz polja A i spremaju u polje B. Koristeći model paralelizacije podataka, programski objekt iterira po podatcima u polju A, nad svakim elementom polja A obavi iste izračune i rezultate sprema u polje B. Pritom primjećujemo da se novi podatci spremaju na isti indeks mjesta u polju B, kao što su ga imali podatci u polju A koje smo iskoristili za dobivanje novih podataka. U plavom kvadratu možemo vidjeti samu sintaksu programskog objekta **`add_delta`** kojim smo definirali proračune , dok se u bijelom kvadratu (ispod plavog) vidi sintaksa kojom smo definirali dva polja, prvo polje A sa šest elemenata, polje B za koje možemo primjetiti da nema definiranu veličinu, te sintaksa poziva programskog objekta. Za polje B *Rapidmind* će sam odredit veličinu gledajući veličinu polja A. Polja koja predajemo u programski objekt i polja u koja spremamo izlaz objekta (u ovom slučaju A i B) moraju uvijek biti iste dimenzije i veličine, ali ne moraju sadržavati iste *Value* tipove. Izračuni programskog objekta izvršavaju se na ciljanom sklopovlju i iako se sa slike 6 može shvatiti kao da se elementi polja A obrađuju jedan po jedan, u stvarnosti se elementi obrađuju paralelno tako da se izvršavanje programskog objekta podijeli na više dostupnih procesorskiih jezgri.



Slika 6 - princip rada programskih objekata

4.4. *Koraci korištenja Rapidmind sučelja*

Promotrimo korake kod oblikovanja *Rapimind* aplikacije [9].

a) Zamjena tipova

Programer mijenja klasične C++ tipove podataka sa ekvivalentnim tipovima *Rapidmind* platforme.

b) „Hvatanje proračuna“ (*capture computations*)

Ovo „hvatanje proračuna“ se zapravo odnosi na spremanje operacija koje će se izvesti unutar definiranih programskih objekata.

c) *Stream* izvršavanje aplikacije

Odnosi se na izvršavanje programskih objekata nad poljima podataka.

Recimo da promatramo aplikaciju sa slike 7, koja svaki element trodimenzionalnog polja b pomnoži skalarom, zbroji sa elementom polja a na istom indeksu, te rezultat spremi u polje a.

```
#include <iostream>

int main() {

  float f = 0.1;
  const unsigned int w=512,h=512;
  float a[w*h*3], b[w*h*3];  // a[h][w][3], b[h][w][3]

  for (int y = 0; y < h; y++)
  for (int x = 0; x < w; x++)
  for (int e = 0; e < 3; e++)
  {
    a[(y * w + x) * 3 + e] += b[(y * w + x) * 3 + e] * f;
  }

  return 1;
}
```

Slika 7 - program pisan u standardnom C++

Pa krenimo sada redom i primjenimo korake oblikovanja *Rapidmind* aplikacije na program sa slike 7. Prvo mijenjamo standardne C++ tipove podataka sa ekvivalentnim tipovima *Rapidmind* platforme. U ovom primjeru varijablu **float f** mijenjamo sa **Value1f f**, te polja **a** i **b** sa poljima tipa **Array<2, Value3f>**. Primjetimo pritom da su polja a i b u standardnom obliku zapravo definirana kao trodimenzionalna polja veličine $512 \times 512 \times 3$ elementa, dok su *Rapidmind* polja definirana kao dvodimenzionalna veličine 512×512 elemenata. Ovu treću dimenziju nadopunjujemo korištenjem *Value* tipa koji ima tri komponente tipa float, dakle imamo 512×512 elemenata, a svaki element ima još tri komponente.

Prema sljedećem koraku definiramo programski objekt koji će izvesti operaciju množenja ulaznog argumenta skalarom, zbrajanja, te dodjeljivanja dobivene vrijednosti izlaznom argumentu. Programski objekt imat će dva ulazna argumenta i jedan izlazni. Važno je primjetiti da varijablu **f**, koja predstavlja skalar, nije potrebno eksplicitno dodati kao argument kod poziva programskog objekta, već se ona kao takva može slobodno koristiti unutar tog objekta. Ipak sve eventualne promjene te varijable unutar programskog objekta neće utjecati na varijablu globalno, te će ona zadržati staru vrijednost iako je možda promjenjena unutar objekta. Programski

objekt započinjemo sa imenom tipa, dakle *Program*, te dodajemo ime varijable, u ovom slučaju *mul_add*. Navođenjem makro naredbe RM_BEGIN signaliziramo *Rapidmind* platformi da prijeđe u *retained* mod, tako da se operacije koje slijede poslije te makro naredbe spreme, a ne izvrše. Ovo također omogućuje programskom objektu da se prevede (kompajlira) samo jednom prilikom pokretanja aplikacije (*at runtime*), a zatim izvede više puta bez potrebe za prevođenjem. Iz *retained* moda izlazimo pomoću makro naredbe RM_END i vraćamo se u *immediate* mod. Nakon što smo definirali naš programski objekt jedino što moramo napraviti je pozvati ga. Umjesto da smo koristili *for* petlje, program se zapravo sastoji od samo jedne izračunske linije i izvršava se paralelno nad oba polja. Nakon što smo uspješno slijedili korake oblikovanja dobili smo program sa slike 8. Važno je napomenuti da se program neće prevesti sve dok se prvi puta ne pokrene. Sva ostala pokretanja programa ne zahtjevaju ponovno prevođenje.

```
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>

using namespace rapidmind;

int main() {

  Value1f f = Value1f(0.1);

  const unsigned int w=512,h=512;
  Array<2,Value3f> a(w, h), b(w, h);

  Program mul_add = RM_BEGIN {
    In<Value3f> i1, i2;
    Out<Value3f> o;

    o = i1 + i2 * f;
  } RM_END;

  a = mul_add(a, b);

  return 1;
}
```

Slika 8 - paralelizirani program, sa slike 7, korištenjem *Rapidmind* platforme

4.5. *Paralelizacija razvijenih aplikacija*

Također, sljedeći gornje korake, paralelizirali smo i razvijene slijedne algoritme. Budući da je temeljna ideja ista, pronaći sile pa odrediti promjene položaja i brzina, prikažimo dio koda PP algoritma koji smo paralelizirali.

```
for ( i=0; i < brojCestica; i++ )
{     // prvo postavljamo nove polozaje, idemo česticu po cesticu
      //x koordinata
      cestica[i].polozaj[0] = cestica[i].polozaj[0] +
          cestica[i].brzina[0]*deltat;

      //y koordinata
      cestica[i].polozaj[1] = cestica[i].polozaj[1] +
          cestica[i].brzina[1]*deltat;

      //z koordinata
      cestica[i].polozaj[2] = cestica[i].polozaj[2] +
          cestica[i].brzina[2]*deltat;

      //postavljamo nove brzine cestice
      //x komponenta brzine
      cestica[i].brzina[0] = cestica[i].brzina[0] +
          cestica[i].sila[0]*deltat;

      //y komponenta brzine
      cestica[i].brzina[1] = cestica[i].brzina[1] +
          cestica[i].sila[1]*deltat;

      //z komponenta sile
      cestica[i].brzina[2] = cestica[i].brzina[2] +
          cestica[i].sila[2]*deltat;
}
```

Slika 9 - kod slijednog PP algoritma, izračunavanje novih brzina i položaja

Svaka čestica definirana je klasom i nalazi se u polju čestica u kojem indeks mjesta polja određuje broj čestice. Nakon izračunate ukupne sile po čestici, iteriramo po polju mijenjajući pritom stare vrijednosti položaja i brzina čestica novim vrijednostima. Prikažimo sada paralelno rješenje.

```
//unutar konstruktora klase cestica definiramo program objekt
//koji radi istu funkciju kao i kod sa slike 9

...
```

```
m_update = RM_BEGIN
{
      //racunanje novih brzina i polozaja cestica
      InOut<Value3f> position;
      InOut<Value3f> velocity;
      In<Value3f> force;

      position += velocity*m_time_delta;
      velocity += force*m_time_delta;

} RM_END;

...

//sljedece je linija koda koja poziva gornji programski objekt nakon
//sto su sile na cestice izracunate

Bundle(p_position, p_velocities)
                      = m_update(p_position, p_velocities, p_force);

...
```

Slika 10 - kod paralelnog PP algoritma, izračunavanje novih brzina i položaja

Kod sa slike 9 i sa slike 10 obavlja istu funkciju, jedina je razlika što je kod sa slike 9 slijedni, a kod sa slike 10 paralelni. Nakon izračuna ukupnih sila, programskom objektu predaju se polja koja sadrže stare položaje i brzine čestica, te novoizračunate sile. Unutar samog programskog objekta, na temelju predanih vrijednosti izračunavaju se nove, koje se zatim spremaju natrag u polja koja sadrže položaje i brzine čestica. Ovdje je zapravo primjetna jednostavnost ovakvog načina izračuna, ali i velika efikasnost paralelnog načina obrade podataka.

4.6. *Ostali alati za razvoj paralelnih aplikacija*

a) Compositional C++

Compositional C++ je opće – namjenski programski jezik za programiranje paralelnih aplikacija [11]. Sve ključne riječi koje se nalaze u C++ nalaze se i u Compositional C++, ali uz dodatak još šest drugih ključnih riječi. Dakle, svaki standardni C++ program, iako ne koristi Compositional C++ ključne riječi, ujedno je važeći Compositional C++ (CC++) program. Ovaj pristup modeliranju paralelnih

programa temelji se na modelu praleliziranja zadataka, za razliku od *Rapidmind* platforme gdje se paraleliziraju podatci. Na CC++ možemo gledati kao na skup proširenja za C++. Ta proširenja CC++ možemo podijeliti na šest različitih apstrakcija:

i) PROCESOR OBJEKT

Možemo ga nazvati jedinicom lokalnosti. Predstavlja adresni prostor unutar kojeg se svi pristupi podatcima smatraju kao lokalni pristupi, dakle pristupi lokalnoj memoriji definiranoj za taj objekt. Dretva koja se izvodi unutar objekta *processor* može pristupiti strukturama podataka alociranim ili definiranim unutar tog objekta *processor* koristeći obične pokazivače. Ključna riječ ***global*** identificira objekt klase *processor.*

ii) GLOBALNI POKAZIVAČ

Zapravo se koristi za povezivanje processor objekata, a identificira se po tipu global. Globalni pokazivač mora se koristiti kako bi se pristupilo strukturama podataka ili kako bi se napravili proračuni unutar drugih processor objekata.

iii) DRETVE

Stvaraju se neovisno o processor objektu i unutar jednog takvog objekta može ih se izvoditi više. Ključne riječi ***par***, ***parfor*** i ***spawn*** koriste se za stvaranje dretvi. Izvođenje dretve se može premjestiti iz jednog unutar drugog processor objekta pomoću remote procedure poziva. Ovdje također vrijedi i upotreba globalnih pokazivača koji se koriste kako bi dretva mogla pozvati neku javnu funkciju koja se nalazi unutar nekog drugog processor objekta.

iv) SYNC VARIJABLA

Označava se pomoću ključne riječi ***sync***. Koristi se za sinkroniziranje izvođenja dretvi. Koristi se u slučajevima kada jedna dretva treba pričekati

na izvođenje neke druge kako bi mogla dovršiti svoje izvođenje. Kada neku varijablu deklariramo kao *sync*, tada ona dobiva svoju početnu vrijednost koja je *'undefined'*. Vrijednost se takvoj varijabli može dodijeliti samo jednom i tada se smatra kao konstanta. Kada dretva pokuša pročitati *sync* varijablu čija je vrijednost *'undefined'*, tada će ta dretva ostati čitati taj blok podataka sve dok se vrijednost te varijable ne promijeni. Bilo koji tip varijable može se proglasiti *sync* varijablom, pa tako i globalni pokazivači.

v) ATOMIC FUNKCIJE

Specificira se klučnom riječi ***atomic***. Koristi se kako bi se kontroliralo ispreplitanje dretvi koje se izvode unutar istoga processor objekta. Ako deklariramo neku funkciju kao *atomic*, to znači da se izvođenje te funkcije neće kositi sa izvođenjem bilo koje druge *atomic* funkcije deklarirane unutar istoga processor objekta. Drugim riječima, dok se jedna *atomic* funkcija izvodi, druga mora čekati sve dok se prva funkcija ne izvede u cijelosti.

vi) TRANSFER FUNKCIJE

Imaju preddefiniran tip *CCVoid*. Dozvoljavaju proizvoljnim strukturama podataka da se prenose između procesora kao argumenti *remote procedure* poziva.

b) MPI (Message Passing Interface)

MPI je jezično neovisan API (*application programming interface*) za komunikaciju preko poruka na paralelnim sustavima [6]. Općenito, MPI se smatra *low-level* alatom, izražavajući paralelizam eksplicitno, a ne implicitno. Smatra se veoma uspješnim u postizanju visokih performansi i velikom prenosivošću između jezicika i platformi, ali mu se zamjera na nekim lošim *low-level* osobinama. Namijenjen je da pruži osnovnu virtualnu topologiju, te mogućnosti sinkronizacije i komunikacije između procesa. MPI uvijek radi na komunikaciji

između procesa, iako se često krivo govori o da radi sa procesorima. Danas se za postizanje maksimalnih mogućnosti sa MPI-jem odabire jedan proces po procesoru (jezgri).

MPI-jem možemo ostvariti *point-to-point send/recieve* operacije dogovorenog tipa, razmjenu podataka između procesa, kombiniranje djelomičnih rezultata različitih procesa, sinkroniziranje procesa, a omogućuje nam i dobavljanje informacija kao što su broj procesa u izvedbi, indentitet procesora kojem je pridjeljen neki proces, susjedni procesi dostupni u logičkoj topologiji, itd.

Danas se koriste MPI-1 i MPI-2. MPI-2 je noviji i sadrži veći raspon mogućnosti. Ipak i dalje je glavna mana niska razina apstrakcije. MPI i *Rapidmind* su potpuno različiti tipovi alata za modeliranje paralelnih aplikacija. Kao glavna prednost *Rapidmind* platforme je razina apstrakcije koja nam u svega par linija koda omogućuje postizanje visoke razine paralelizacije podataka na različitom sklopovlju. MPI sa druge strane omogućuje međuprocesnu komunikaciju i postizanje paralelizma korištenjem većeg broja računala sa različitim sustavima, dok je *Rapidmind* orjentiran na sklopovlje unutar jednog sustava.

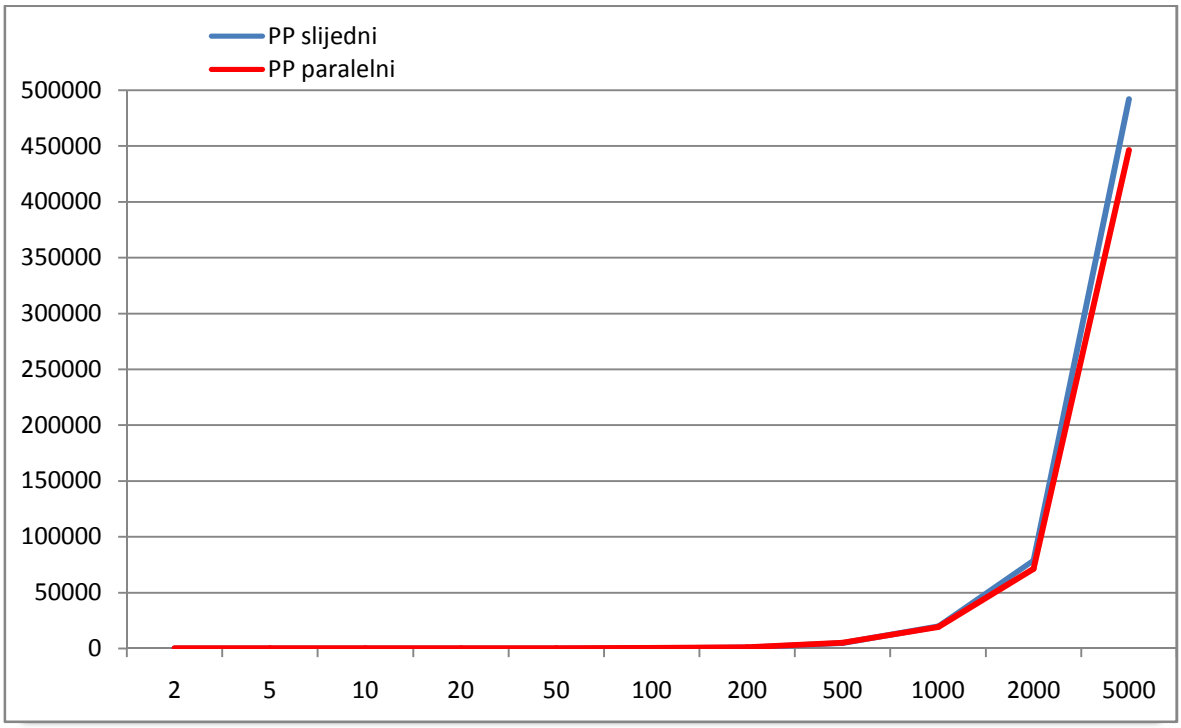
5. USPOREĐIVANJE PERFORMANSI RAZVIJENIH ALGORITAMA

U ovom dijelu uspodređuju se razvijeni algoritmi i određuje da li se paralelizacija podataka uvijek isplati, i ako da koliki su dobitci na performansama. Prvo ćemo usporediti algoritme koji rade PP metodom, a zatim algoritme koji rade PM metodom. Važno je spomenuti sklopovlje na kojem su se simulacije pokretale. Svi sljedni programi pokretani su na Intel Core Duo 32-bitnom centralnom procesoru koji radi na taktu od 1.83 GHz, a paralelni programi pokretani su na NVIDIA grafičkoj kartici serije GeForce*Go* 7600 sa 256 MB memorije.

Mjerenje se izvodi na sljedeći način: svaki algoritam se promatra kroz 100 iteracija sa diskretnim vremenskim korakom postavljenim na 0.1 sekundu. Parametar koji se mijenja su čestice u sustavu. Promatraju se sustavi sa 2, 5, 10, 20, 50 pa sve do 5000 čestica. Već za sustav od 5000 čestica simulacija je trajala oko osam minuta, a za sustave sa 10 000 čestica oko 32 minute. Kod sustava sa 10 000 čestica i PM algoritma mjerenje navedenim računalom nije se moglo izvesti zbog prevelikih memorijskih zahtjeva, pa tih rezultata nema. Svaki algoritam bio je pokrenut deset puta za svaki broj čestica. Dobiveni rezultati su se zapisali, a kao konačan rezultat uzeta je srednja vrijednost dobivenih vremena.

5.1. *Usporedba slijednog i paralelnog PP algoritma*

PP metoda vremenski je najzahtjevnija od svih opisanih metoda rješavanja problema *n* tijela. Imajući to na umu, logično je za pretpostaviti da je ta metoda ujedno i najbolji kandidat za paralelizaciju. Nakon provedenog mjerenja na slici 11, grafički je prikazan odnos između vremena izvođenja paralelnog i sljednog algoritma.



Slika 11 - odnos paralelnog i slijednog PP algoritma

Broj čestica	2	5	10	20	50	100
PP sljedni	0 ms	0 ms	0 ms	4 ms	37 ms	199 ms
PP paralelni	325 ms	326 ms	320 ms	340 ms	374 ms	527 ms
Broj čestica	200	500	1000	2000	5000	10 000
PP sljedni	802 ms	4950 ms	19 706 ms	78 786 ms	492 072 ms	> 32 min
PP paralelni	1134 ms	5272 ms	19 358 ms	71 484 ms	446 447 ms	> 30 min

Slika 12 - prikaz prosječnih vremena PP algoritama u odnosu na brojeve čestica

Na slici 11 plavom linijom označen je sljedni algoritam, a crvena linija predstavlja paralelni algoritam. Na osi apscisa nalazi se broj čestica u sustavu (od 2 do 5000), a na

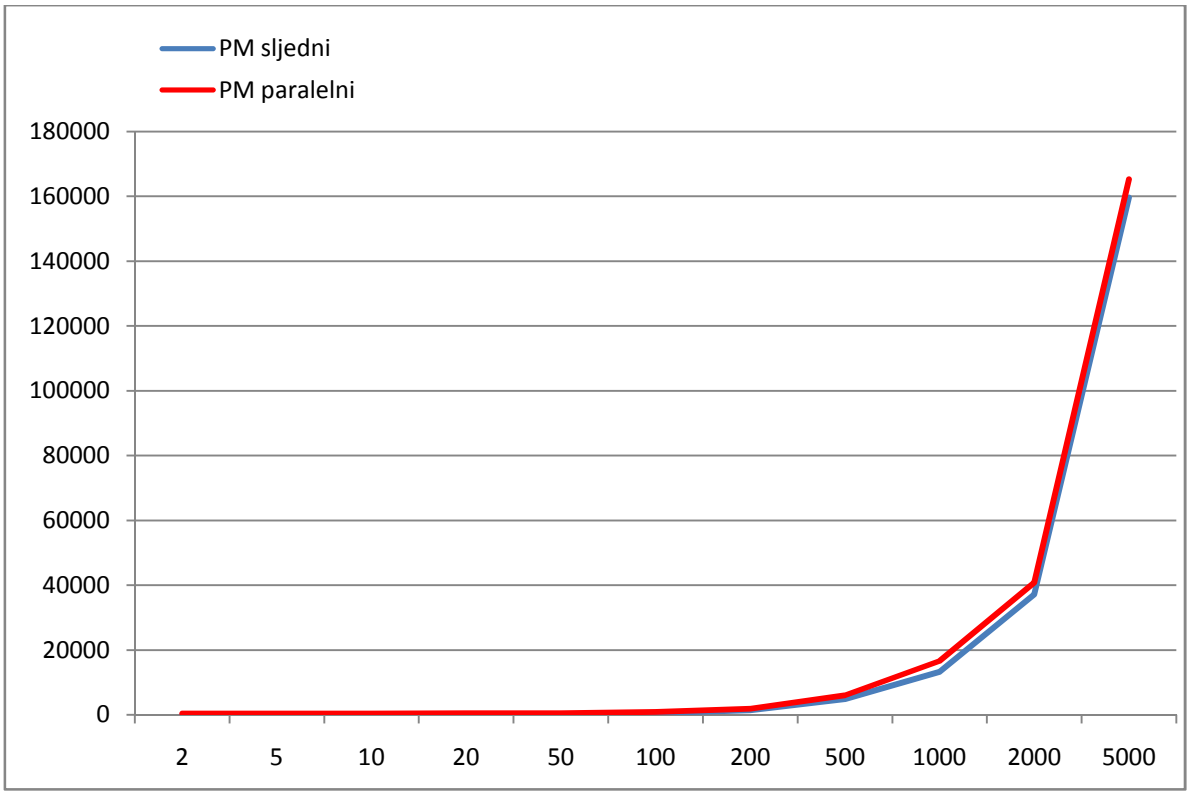
osi ordinata se nalazi vrijeme izvođenja algoritma u milisekundama. Primjećujemo da se linija sljednog algoritma (plava) ne vidi sve dok se ne dođe do broja od 1000 čestica u sustavu. Za manje brojeve čestica (od 2 do 500), paralelni algoritam pokazao je nešto lošije rezultate izvodeći se na grafičkoj kartici, pa je zbog toga na grafu prikazana njegova cijela linija. Ipak, početni rezultati izvođenja paralelnog algoritma na grafičkoj kartici (od 2 do 50 čestica u sustavu) pokazivali su jednu vremensku konstantu, bez bitnijeg rasta vremena izračuna (oko 340 milisekundi), dok se za iste brojeve čestica u sustavu, izvođen na centralnom procesoru, sljedni algoritam kretao između nula i 37 milisekundi. Razumno je za pretpostaviti da se razlika vremena utroši na prijenos podataka iz glavne memorije u memoriju grafičke kartice kada su ti podatci potrebni, pa bi prema tome ukupni prijenos podatak trebao iznositi oko 320 – 330 milisekundi.

Na oko 1000 čestica u sustavu, paralelni algoritam preuzima prednost nad sljednim. Razlika u prosječnom vremenu izvođenja simulacije nad sustavom sa 1000 čestica iznosila je 348 milisekundi u korist paralelnog algoritma. Na 2000 čestica ta razlika iznosila je 7.3 sekundi, a na 5000 čestica 45.6 sekundi u korist paralelnog algoritma. Iz ovoga zaključujemo da je model paralelizacije podataka pogodan za paralelizaciju PP metode i nesumnjivo je da bi za veći broj čestica u sustavu, i još dodatno uz uporabu sklopovlja visokih performansi (u ovom slučaju grafičkih kartica), vremenska razlika između vremena izvođenja ova dva algoritma bila daleko u korist paralelnog algoritma.

5.2. *Usporedba slijednog i paralelnog PM algoritma*

Kod ova dva algoritma situacija je drugačija. Sam algoritam se razlikuje od uobičajene implementacije *particle-mesh* metode, pa prisjetimo se najprije njegovog principa rada. Prostor se podijeli na mrežu i svakoj točki mreže se dodijeli ukupna masa čestica koje se nalaze u neposrednoj blizini te točke. Zatim se između svake dvije točke prostora koje su „dobile“ masu izračuna gravitacijska sila (kod uobičajenih

implementacija računa se gravitacijski potecijal pomoću brze fourierove transformacije). Ta sila se tretira kao ukupna sila na čestice oko te točke i pomoću nje se izračunaju nove brzine i položaji čestica. Također, sam koordinatni prostor se tretira kao *prostor interesa*, što znači da se sve čestice koje izađu iz tog prostora više ne uzimaju u obzir kod računanja funkcija gustoće i sila. Budući da smo i taj algoritam pokušali paralelizirati metodom paralelizacije podataka, pogledajmo na slici 13 kako se odnose razvijeni paralelni i sljedni algoritam.



Slika 13 - odnos sljednog i paralelnog PM algoritma

Broj čestica	2	5	10	20	50	100
PM sljedni	0 ms	9 ms	10 ms	46 ms	210 ms	494 ms
PM paralelni	407 ms	409 ms	404 ms	454 ms	509 ms	891 ms
Broj čestica	200	500	1000	2000	5000	10 000
PM sljedni	1356 ms	4832 ms	13 274 ms	37 179 ms	159 604 ms	-
PM paralelni	1893 ms	6045 ms	16 552 ms	40 814 ms	165 236 ms	-

Slika 14 - prikaz prosječnih vremena PM algoritama u odnosu na brojeve čestica

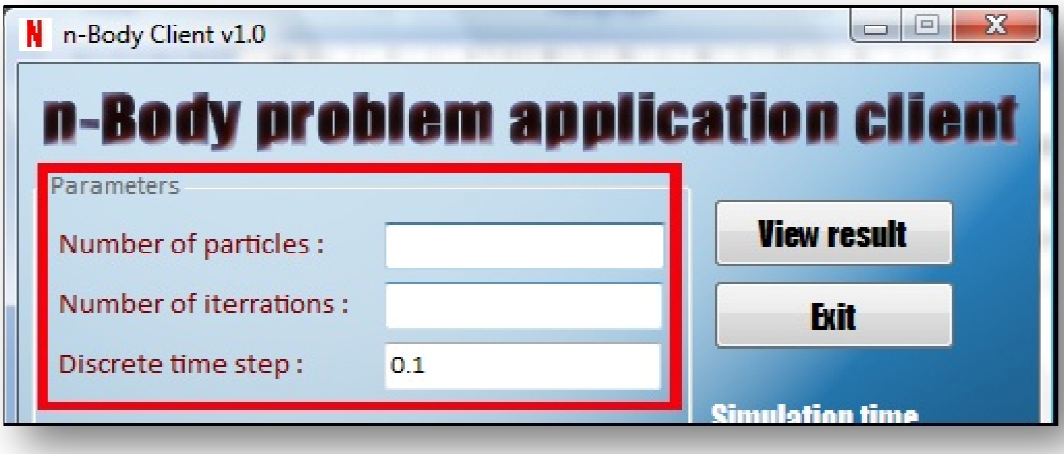
Sa slike 13 nije teško primjetiti da su performanse oba algoritma podjednake, ali sa slike 14 koja nam daje tablični prikaz prosječnih vremena izvođenja algoritama za određen broj čestica, vidimo da sljedni algoritam ipak ima određenu prednost. To možemo pripisati samoj sintaktčkoj konstrukciji algoritma gdje se koriste konstrukti standardnog C++ poput lista, klasičnih pokazivača i polja koji povećavaju količinu podataka u komunikaciju na relaciji grafičkog sklopovlja i glavne memorije, te većeg broja pokazivača potrebnih pri upisu podataka u *Rapidmind* polja podataka koji se stalno pozivaju. Unatoč tome, u odnosu na PP algoritme, PM algoritmi daju vidna ubrzanja, upravo zbog drugačije koncepcije programa gdje je prostor nepromjenjive veličine i čestice koje izađu iz tog prostora se više ne uzimaju u obzir kod izvršavanja proračuna, dok se kod PP prate sve čestice kroz sve iteracije. Isto tako treba uzeti u obzir da se čestice u prostoru raspodjeljuju generiranjem slučajnih koordinata, te da trajanje simulacije ovisi i o prostornoj raspodjeli, jer čestice mogu izaći iz prostora u svega nekoliko iteracija, a isto tako uopće ne moraju izići iz prostora. Ukoliko trebamo veći prostor u kojem promatramo čestice, tada se on može povećati, ali samim time povećavamo i trajanje simulacija, te možebitne probleme ako sustav nema dovoljno memorije. Ipak ove rezultate treba uzeti sa određenom zadrškom, jer je moguće dobiti potpuno drugačije rezultate na drugačijim sustavima, gdje bi paralelni programi možda imali potpunu prednost, ili bi imali vidno manje mogućnosti.

5.3. *Opis razvijenog klijentskog programa*

Kao program koji povezuje četiri algoritma, tj. programe temeljene na tim algoritmima, opisanim kroz prijašnji tekst i pseudokodove, napravljen je poseban klijent sa jednostavnim grafičkim sučeljem preko kojeg pozivamo određeni algoritam. U tom klijentu možemo postaviti parametre simulacije, te pregledati rezultate, odlučimo li se za njihovu pohranu.

Nakon pokretanja klijenta, prije pokretanja simulacije, potrebno je postaviti parametre simulacije. Kako bi se održala jednostavnost, samo su dva parametra

neophodna za ispravan rad algoritma: broj čestica u sustavu, te broj iteracija kroz sustav. Postavljanje tih parametara vrši se njihovim upisom u za to predviđena tekstualna područja (Slika 15 - postavljanje parametara kod klijentskog programa), koja se nalaze na vrhu klijenta.

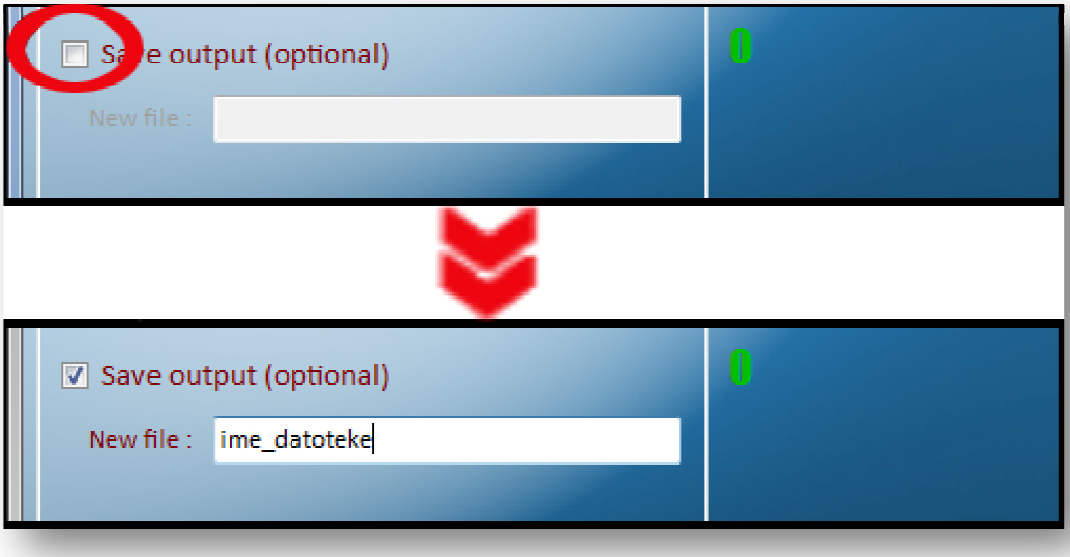


Slika 15 - postavljanje parametara kod klijentskog programa

Na slici 15 vidljivo je da se osim broja čestica (prvo tekst područje) i broja iteracija (drugo tekst područje) može postaviti i diskretni vremenski korak, izražen u sekundama, unutar kojeg promatramo sustav (treće tekst područje). Postavljanje ovog parametra je opcionalno i početna vrijednost koja se vidi u tekst području je unaprijed zadana vrijednost parametra koja će se koristiti u slučaju da se parametar ne promijeni. Velikim dijelom točnost rezultata ovisi o ovom parametru i ne preporuča se postavljanje iznad vrijedosti od jedne sekunde.

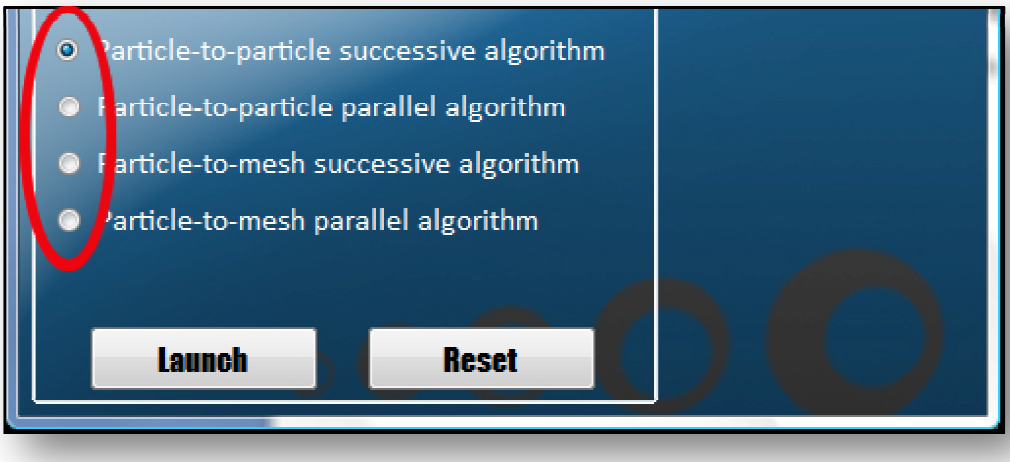
Sljedeći parametar koji možemo postaviti je spremanje rezultata. Ako poželimo spremiti rezultat simulacije potrebno je samo jednostavno označiti za to predviđen *check box* (slika 16). Nakon što je *check box* kućica označena omogućiti će se unos teksta u tekst područje neposredno ispod njega (slika 16). To tekst područje predviđeno je za unos imena datoteke u koju želimo sačuvati rezultate. Postavljanje imena datoteke je također potpuno opcionalno. U slučaju da se ime datoteke ne unese, a označeno je spremanje rezultata, tada će se oni spremiti u za to prije predviđenu (defaultnu) datoteku. Ekstenziju datoteke, dakle tip datoteke, nije

potrebno unositi. Sve datoteke se spremaju u html formatu zbog jednostavnosti i preglednosti tabličnog prikaza rezultata koji omogućuje html.



Slika 16 - spremanje rezultata i unos imena datoteke

U sljedećem koraku odabiremo algoritam koji želimo izvesti. Kako su razvijena četiri algoritma tako postoje četiri opcije između kojih možemo odabrati (na početku je odabran PP slijedni algoritam). Algoritmi su poredani po metodi kojom su izvedeni. Prva dva su PP algoritmi, a druga dva su PM algoritmi, pri čemu je kod obje metode prvi slijedni, a drugi paralelni algoritam. Algoritam se odabire označavanjem tzv. *radio* gumba ispred imena željenog algoritma (slika 17).

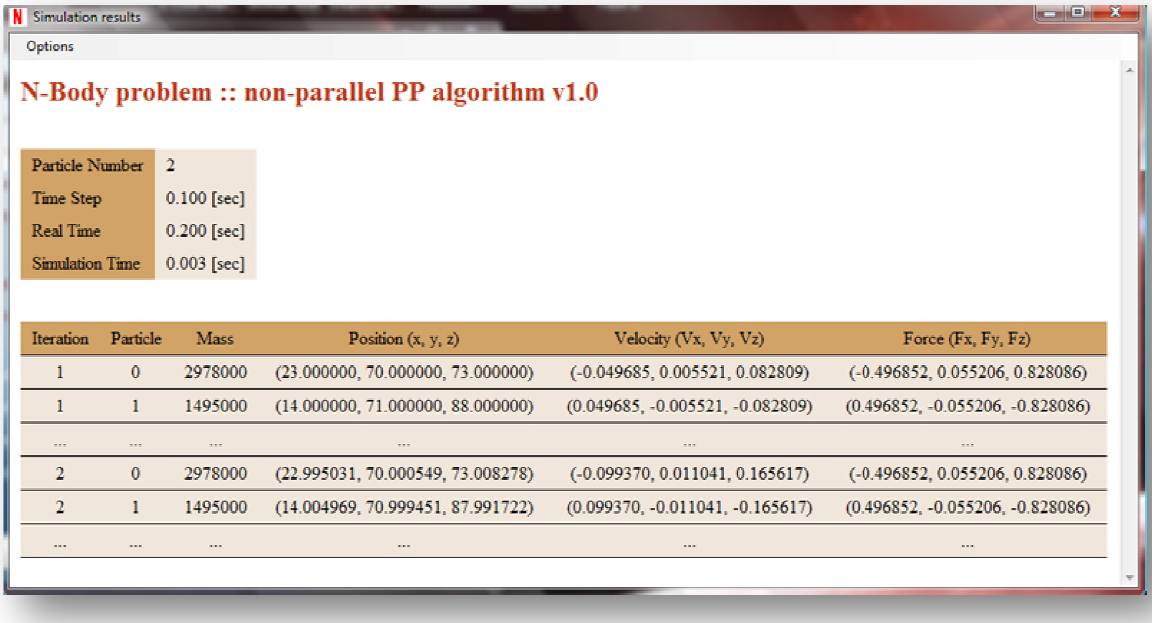


Slika 17 - odabir algoritma i pokretanje simulacije

Na slici 17 također su vidljiva još dva gumba, *Launch* i *Reset*. *Launch* gumb nam služi za pokretanje simulacije nakon što smo postavili sve parametre, a *Reset* gumb nam služi za postavljanje parametara simulacije na izvorne vrijednosti. Nakon pokretanja simulacije otvara se komandni prozor u kojem se prikazuje postotak završetka simulacije. Taj postotak ovisi o trenutnoj iteraciji u kojoj se simulacija nalazi i ukupnom broju iteracija. Nakon što je simulacija završila, u klijentski program se vraća broj milisekundi koji je bio potreban za njeno izvršenje i tada klijentski program prikaže taj broj. Ako je broj milisekundi manji od tisuću rezultat se prikazuje u milisekundama (*ms*), a ako je broj veći od tisuću rezultat se prikazuje u sekundama (*s*). U slučaju da se nužni parametri (broj čestica i broj iteracija) ne unesu, a pokuša se pokrenuti simulacija, klijent će javiti pogrešku i neće pokrenuti odabrani algoritam.

Pregled rezultata je vrlo jednostavan i obavlja se odabirom gumba *View result*. Ovaj gumb može se odabrati iako prethodno nismo obavili simulaciju u kojoj smo spremali rezultate, a ukoliko smo to učinili, tada prikazuje rezultate te posljednje simulacije. Prilikom odabira pregleda rezultata otvara se novi prozor u kojem će se rezultati prikazati (slika 18). U padajućem izborniku na vrhu prozora možemo odabrati otvaranje druge datoteke sa rezultatima neke od prijašnih simulacija te međusobno uspoređivati rezultate (za uspoređivanje moramo imati više otvorenih prozora za pregled rezultata). Također, mogu se otvarati i druge datoteke sa **.html* ekstenzijom.

Rezultati su vrlo pregledni i daju nam informacije o tipu algoritma koji smo koristili (na vrhu stranice), zatim broju čestica u sustavu, diskretnom vremenskom koraku koji smo koristili, vremenu u kojem bi se promatrao takav realan čestični sustav, vremenu potrebnom za simulaciju, te tablični prikaz parametara vezanih uz pojedinu česticu u svakoj iteraciji simulacije.



Slika 18 - primjer rezultata za dvije čestice i dvije iteracije

Zatvaranje prozora za pregled rezultata možemo napraviti odabirom opcije *exit* iz padajućeg *Options* izbornika na vrhu prozora ili standardno klikom na *'X'* u gornjem desnom uglu prozora. Izlaz iz klijentskog programa obavljamo u glavnom prozoru programa (prozor iz kojeg pokrećemo simulaciju) odabirom gumba *Exit* ili također standardnim klikom na *'X'* u gornjem desnom uglu prozora.

Što se tiče tehničkih zahtjeva, za pokretanje klijentskog programa i algoritama potreban je MS Windows XP SP2 ili Windows Vista operacijski sustav, za koje se može dobiti instalacijska datoteka. Također na računalu mora biti instaliran MS *.NET framework*, verzija 2.0 minimalno. Sa strane *hardware*-skih zahtjeva, za izvođenje paralelnih inačica algoritama potrebne su grafičke kartice proizvođača ATI ili NVIDIA. U obzir dolaze serije x1000 i novije od ATI-ja, NVIDIA-ine grafičke kartice serija GeForce 6000, 7000, 8000, te također NVIDIA-ina serija Quadro sa podrškom za

shader model 3.0 (npr. Quadro FX5500). Također, važno je napomenuti da zbog verzije platfome kojom smo postigli paralelnost algoritama, pokretanje paralelnih algoritama na sustavima sa MS Windows Vistom i ATI x1000 serijom grafičkih kartica nije podržano.

6. ZAKLJUČAK

Minimalnim promjenama između paralelnih i sljednih algoritama, ovdje smo pokušali dobiti na performansama (što je primjetno i iz samih pseudokodova algoritama). Promjenom temeljne koncepcije ovdje prikazanih paralelnih algoritama, moguće je povećati njihove performanse, te isto tako uporabom naprednijeg grafičkog sklopovlja. Korištenjem različitih alata (od kojih su neki spomenuti ovdje) dobivamo različite mogućnosti modeliranja paralelnih aplikacija. Kako se slijedni algoritmi mogu izvoditi samo na jednom procesoru ili procesorskoj jezgri, njihove performanse mogu se popraviti samo korištenjem procesora sa bržim radnim taktom, ali ako uzmemo u obzir da smo danas došli do stupnja razvoja sklopovlja temeljenih na siliciju na kojem povećanja taktova rada procesora, zbog disipacije topline, praktički više nisu moguća, slijednim programima ne ostaje puno mjesta za poboljšanje performansi. Iz svega toga lako zaključujemo da će se broj procesorskih jezgri na jednoj pločici, te broj grafičkih procesora na grafičkim karticama konstantno povećavati, a samim time prednost i iskoristivost potencijala računalnih sustava davat će se u ruke paralelnih programa koji zasigurno predstavljaju novu glavnu granu razvoja računarskih disciplina.

7. LITERATURA

[1] Lindholm T., *N Body algorithms*, seminar, 1999.

[2] Nbody/Particle simulation methods, 20.03.2000., *Nbody/Particle simulation methods*, http://www.amara.com/papers/nbody.html, 05.04.2008.

[3] Beloch G., Narlikar G., Parallel algorithms- Series in Discrete Matemathics and Theoretical Computer Science – A Pratical Comparison of N-Body Algorithms, vol.30, 1997.

[4] Izaguirre J.A., Matthey T., Parallel multi-grid summation for the N-Body problem, magistarski rad, magistarski rad, University of Notre Dame, 02.10.2003.

[5] Aguilar L.A., The N-body problem, prezentacija, Instituto de Astronomia/UNAM Mexico

[6] MPI, 04.05.2003., *Message Passing Interface*, http://en.wikipedia.org/wiki/Message_Passing_Interface, 06.04.2008.

[7] Parallel programming Model, 15.07.2005., *Parallel Programming Interface*, http://en.wikipedia.org/wiki/Parallel_programming_model, 06.04.2008.

[8] Parallel Algorithms, 19.11.2002., *Parallel Algorithms,* http://en.wikipedia.org/wiki/Parallel_algorithms, 06.04.2008.

[9] RapidMind development platform overwiev, 20.04.2007., *RapidMind development platform overwiev*, http://www.rapidmind.net/pdfs/RapidmindDatasheet.pdf

[10] Michael D. McCool, WPdrpm, 20.04.2007., *Data-Parallel programming on the Cell BE and the GPU using the RapidMind Devolepment platform*, http://www.rapidmind.net/pdfs/WPdprm.pdf, 07.04.2008.

[11] Foster, I. , Designing and Building Parallel Programs, 20.04.2007*., Designing and Building Parallel Programs: Compositional C++,* http://www-unix.mcs.anl.gov/dbpp/, 1995.

SAŽETAK

Jedan od problema koji se danas pojavljuje u mnogim područjima znanstvenog djelovanja zasigurno je i problem međudjelovanja n tijela. Problem međudjelovanja n tijela možda je i jedan od najčešće nerješivih problema u povijesti znanosti.

U matematičkom modelu rješenja, potrebno je odrediti ukupnu silu na pojedinu česticu, te u skladu s njom odrediti promjenu položaja čestice.

Sa strane algoritamskih metoda postoji mnogo rješenja, koja uglavnom dijele zajedničke temeljne ideje, ali i nekih sa radikalno novim smjerovima.
Najjednostavnija algoritamska metoda je *particle-to-particle* metoda koja direktno računa sile između svake dvije čestice te ne koristi aproksimacije, što joj osigurava veliku točnost, ali zato je vremenski veoma zahtjevna.
Particle-to-mesh metode koriste aproksimacije sa prostorom. Dijele prostor u mrežu, te u točkama u prostoru računaju potencijale. Zatim se iz gradijenta potencijala dobije ukupna sila u toj točki prostora koja djeluje na okolne čestice.
Particle-to-particle/particle-to-mesh metoda objedinjuje najbolje od gornje dvije metode.
Tree code metode koriste strukutru stabla za aproksimacije čestica na određenim udaljenostima. Primjer je poznati Barnes-Hut algoritam.

U ovom radu prikazane su i implementirane *particle-to-particle* i *particle-to-mesh* metode. Za svaku metodu razvijena su dva algoritma, jedan paralelni, a jedan slijedni. Paralelnost smo ostvarili pomoću *Rapidmind* platforme. U testovima, paralelni PP algoritam prikazao je veću efikasnost i moć izračuna nad slijednim PP algoritmom, dok je sa druge strane slijedni PM algoritam imao malu prednost nad paralelnim PM algoritmom.

Rapidmind platforma razvijena je nad standardnim C++ elementima i zamotana u C++ razrede. Omogućuje paralelnost aplikacije na razini paralelizacije podataka, te njihove obrade na višejezgrenim procesorima, grafičkim karticama, te *Cell* procesoru. Koristi tzv.

programske objekte kojima se spremaju operacije, te se kasnije izvode na dostupnom sklopovlju. Proračuni se vrše nad poljima podataka u paraleli. Podatci u poljima se lome, te se raspodjeljuju ovisno i dostupnom sklopovlju za računanje i njegovim mogućnostima.

Performanse slijednih algoritama ovise o radnim taktovima današnjih procesora. Budući da su taktovi rada procesora dostigli granicu kod koje je povećanje praktički nemoguće, budućnost se okreće prema paralelnim algoritmima koji će moći iskoristiti puni potencijal budućeg sklopovlja.

KLJUČNE RIJEČI

Hrvatski	Engleski
paralelni	parallel
slijedni	successive
algoritam	algorithm
problem n tijela	n-body problem
čestica	particle
sustav čestica	particle system
čestica-čestica	particle-to-particle
čestica-mreža	particle-to-mesh
funkcija gustoće	density function
potencijal	potential
Rapidmind platforma	Rapidmind platform
prevoditelj	compiler
višedretvenost	multithreading
programski objekti	program objects
određivanje performansi	benchmarking