Rešavanje sistema linearnih jednačina velikih dimenzija na distribuiranim računarskim sistemima

(preliminarna verzija)

Student: Igor Jeremić < igor@jwork.net >
Profesor: Dr. Desanka Radunović
Asistent: Filip Marić

Sadržaj

1 Abstrakt **3**

2 Spisak fajlova **4**

3 Uvod **5**
3.1 Direktne i iterativne metode za rešavanje sistema linearnih jednačina 5

4 Realizacija i rezultati **5**
4.1 Način realizacije 7
4.1.1 Apstraktna klasa matrica 8
4.1.2 Izvedene klase gustaMatrica, gustaKvadratnaMatrica i gustaSimetrična matrica 9
4.1.3 Izvedena klasa retkaMatrica 10
4.2 Paralelni i distribuirani računarski sistemi 11
4.3 Posix threads - paralelni programi na čvrsto vezanim procesorima 12
4.3.1 Implementacija Jacobi-eve metode 12
4.3.2 Analiza rezultata 13
4.3.3 Implementacija paralelnog množenja matrica 14
4.4 Programiranje slanjem poruka - MPI 14
4.4.1 Tipovi podataka 15
4.4.2 Interprocesorska komunikacija na Računaru sa distribuiranom memorijom . 16
4.4.3 Metode za kolektivnu komunikaciju 16
4.4.4 Dekompozicija računa na više procesa i redukovanje rezultata 16
4.5 MPI Implementacija Kramerovog pravila 17
4.5.1 Implementacija 17
4.5.2 Testiranje programa 18
4.6 MPI Implementacija Jacobi-eve metode 20
4.6.1 Implementacija 20
4.6.2 Analiza i zaključi 21

5 Analiza i zaključci **22**
5.1 Primene 23

6 Literatura i sofware **25**
6.1 Literatura 25
6.2 Internet lokacije 25
6.3 Software 25

1 Abstrakt

Tema seminarskog rada je implementacija Jacobi-evog iterativnog metoda za rešavanje sistema linearnih jednačina. Rad je radjen u jeziku C++ čime je ostavljena mogućnost za veoma jednostavnu dodatnu implementaciju drugih metoda i nosača podataka. Radi poredjenja rezultata kao predstavnik direktnih metoda izabran je Gauss-ov sistem eliminacije sa izborom najvećeg elementa, čija je kompleksnost $O(n^3)$. Takodje, implementirana je i Gauss-Saidel-ova modifikacija Jacobi-evog iterativnog metoda ali samo u slučaju sistema sa jednim procesorom.

Višeprocesorska implementacija Jacobi-evog metoda radjena je u dva interfejsa: **pthreads** za čvrsto vezane višeprocesorske sisteme i **MPI** za labavo vezane višeprocesorske siteme. Takodje, dat je pregled nekih osnovnih metoda za rad sa interfejsom za slanje poruka sa primerima uradjenim u C++

Zbog interfejsa za podršku višeprocesorskih sistema rad je radjen na UNIX platformi ali se klase napisane na C++-u, koje definišu matrice i omogućavaju osnovne manipulacije sa njima, mogu prevesti na bilo kom C++ prevodiocu i bilo kojoj platformi. Kao interfejsi za unos podataka i prikaz rezultata korišćeni su standardni ulaz i standardni izlaz, koji se mogu preusmeriti iz fajla ili u fajl. Ovakav način unosa i prikaza podataka pruža veliku portabilnost na bilo koji drugi interfejs.

2 Spisak fajlova

naziv fajla	veličina	opis
defs.h		definicije makroa zajedničke za sve fajlove
matrice.h		fajl koji include-uje ostale fajlove
matrica.hh		deklaracija apstraktne klase matrica
matrica.cc		implementacija klase matrica
gustaMatrica.hh		deklaracija klase gusta matrica izvedene iz klase matrica
gustaMatrica.cc		implementacija guste matrice
gustaKvadratnaMatrica.hh		kvadratna matrica izvedena iz klase gusta matrica
gustaKvadratnaMatrica.cc		implementacija kvadratne matrice
gustaSimetricnaMatrica.hh		simetrična matrica izvedena iz klase matrica
gustaSimetricnaMatrica.cc		implementacija simetrične matrice
retkaMatrica.hh		deklaracija klase retka matrica izvedene iz matrice
retkaMatrica.hh		implementacija retke matrice pomoću map-e iz STL
funkcije.hh		deklaracija raznih funkcija za rad sa matricama
funkcije.hh		implementacija funkcija
latexExport.hh		funkcije za izvoz matrice ili sistema u LaTeX format
latexExport.cc		implementacija
slj.hh		šabloni - funkcije za rešavanje sistema jednačina
slj.cc		implementacija
jacobiPthreads.hh		funkcije za Jacobi-evu iterativnu metodu pomoću posix niti
jacobiPthreads.cc		implementacija
matmulPthreads.hh		množenje matrica na čvsto vezanom višeprocesorskom sistemu
matmulPthreads.cc		implementacija
timetest.hh		funkcije za merenje vremena UNIX/DOS
timetest.cc		implementacija
primer01.c		primer sabiranja dve pravougaone matrice
primer02.c		primer nalaženja inverzne matrice i množenja matrica
primer03.c		primer rešavanja sistema Gausovom metodom eliminacije
primer04.c		primer Jacobi-eve i Gauss-Saidel-ove metode
kramer-MPI.c		računanje sistema metodom determinanti (MPI)
jacobi-MPI.c		rešavanje sistema Jacobi-evom metodom (MPI)
makefile		gnu make fajl za prevodjenje projekta
readme.tex		izvorni kod ovog dokumenta
readme.pdf		ovaj dokument (PDF format)

***** *Veličine fajlova će biti upisane u finalnoj verziji rada*

3 Uvod

Rešavanje sistema linearnih jednačina spada u jedan od osnovnih zadataka linearne algebre. Za rešavanje sistema postoje egzaktni algoritmi, medjutim kada se govori o sistemima veće dimenzije (npr preko 10.000 jednačina), većina tih algoritama u praksi nije primenljiva. Jedan od algoritama je Kramerovo pravilo ili metoda determinante, ali ovaj algoritam je interesantan samo sa teorijskog stanovišta. Dovoljno je na primer dokazati da je determinanta sistema različita od nule da bi znali da sistem ima tačno jedno rešenje, odnosno kod homogenih sistema da je determinanta jednaka nuli pokazuje da sistem ima i netrivijalnih rešenja. U praksi, za sistem dimenzije n potrebno je izvršiti $O(n!n)$ aritmetičkih operacija, što dovoljno govori o neprihvatljivosti ove metode za rešavanje sistema velikih dimenzija. Nešto prihvatljivija metode za rešavanje sistema jednačina je Gausova metoda eliminacije (koja će u ovom radu biti implementirana radi poredjena), složenost ove metode je $O(n^3)$.

3.1 Direktne i iterativne metode za rešavanje sistema linearnih jednačina

Metode za rešavanje sistema linarnih jednačina dele se na direktne i iterativne. Kod direktnih metoda se u konačnom broju koraka dolazi do tačnog rešenja, dok je u iterativnim metodama rešenje granična vrednost niza uzastopnih aproksimacija koje se računaju algoritmom metode. Dve naizgled pozitivne osobine direktne metode su ***konačan broj koraka*** i ***tačno rešenje*** medjutim, kao što je već rečeno taj konačan broj koraka nije izvodljiv u realnom vremenu, npr. ukoliko se Kramerovim pravilom rešava sistem od 10 jednačina sa 10 nepoznatih biće potrebno izvšiti oko $10 \cdot 10!$ operacija odnosno preko 36 miliona operacija. Što se tiče tačnosti direktnih metoda, ona je apsolutna i zadržana samo ukoliko ne postoji greška računa, koja se vrlo brzo gomila deljenjem što je veoma česta operacija u direktnim metodama, tj tačnost je zadržana ukoliko se decimalni brojevi ne zaokružuju na konačnom broju decimala ili se racionalan zapis broja provlači kroz celu metodu (ukoliko se ne koriste iracionalni brojevi).

Ukoliko se odreknemo apsolutne preciznosti (koja u praksi ni ne postoji jer se za zapis realnih brojeva koristi konačno mnogo bitova) iterativne metode mogu biti mnogo korisnije, jer se u realnom vremenu mogu rešavati sistemi velikih dimenzija - reda veličine nekoliko desetina hiljada, što smisla jer se diskretizacijom različitih problema upravo dobijaju veliki sistemi linearnih jednačina.

4 Realizacija i rezultati

Neka je $Ax = b$ sistem jednačina:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\ \cdots & & & & & & & & \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{nn}x_n & = & b_n \end{array}$$

Prebacivanjem $n-1$ promenljivih sa desne strane i deljenjem sa koeficijentom a_{ii} dobija se:

$$\begin{array}{ccccccccc} x_1 & = & \frac{1}{a_{11}} & (b_1 & & -a_{12}x_2 & - & \cdots & -a_{1n}x_n &) \\ x_2 & = & \frac{1}{a_{22}} & (b_2 & -a_{21}x_2 & & - & \cdots & -a_{2n}x_n &) \\ \cdots & & & & & & & & & \\ x_n & = & \frac{1}{a_{nn}} & (b_n & -a_{n1}x_2 & -a_{n2} & - & \cdots & &) \end{array}$$

Čime su dobijene iterativne formule koje definišu **Jacobi-ev metod iteracije**.

Odnosno u matričnoj reprezentaciji: $Ax = b \iff x = Bx + c$

Dovoljan uslov konvergencije iterativnog algoritma je da koeficijent kontrakcije $q = ||B|| < 1$ za bilo koju normu matrice. Dok se apriorna i aposteriorna ocena greške računaju kao:

$$||x^* - x^{(n)}|| \leq \frac{q^n}{1-q}||x^{(1)} - x^{(0)}||, \qquad ||x^* - x^{(n)}|| \leq \frac{q}{1-q}||x^{(n)} - x^{(n-1)}||.$$

Jedan iterativni korak obuhvata računanje n izraza, ukoliko se u istom iterativnom koraku pri računanju i-tog izraza koriste vrednosti u tom koraku izračunatih x-ova dobija se **Gauss - Saidel-ova** metoda. I jedna i druga metoda konvergiraju pod istim uslovima, medjutim brzina konvergencije je različita s tim da u nekim slučajevima Gauss-Saidel-ova a u nekim Jacobi-eva, konvergira u manjem broju koraka. Može se pokazati da se svaka matrica A može elementarnim transformacijama modifikovati tako da neka od normi matrice B bude uvek manja od 1. Ali ne postoji efektivan algoritam za to, tj algoritam će se svesti na Gausov metod eliminacije a njime se onda može i rešiti sistem.

Dijagonalno dominantne matrice su one matrice kod kojih su elementi na dijagonali veći od zbira apsolutnih vrednosti svih ostalih elemenata u redu. Ukoliko je matrica A dijagonalno dominantna tada će norma matrice B sigurno biti manja od 1. Ovakve matrice su čest proizvod nekih konkretnih problema pa zato ima smisla koristiti ovaj iterativni metod za rešavanje sistema jednačina.

Primer Jacobi-eve i Gauss-Saidelo-ove metode se može dobiti pokretanjem programa *primer04*

```
(6 x 6 | 1)
[ 50.20 4.000 8.000 8.000 10.00 2.000 | 10.00
  4.000 40.60 3.000 6.000 5.000 7.000 | 20.00
  4.000 6.000 55.00 10.00 7.000 8.000 | 6.000
  2.000 7.000 1.000 29.80 2.000 9.000 | 16.00
  2.000 5.000 2.000 2.000 29.80 3.000 | 11.00
  6.000 9.000 7.000 3.000 7.000 46.60 | 16.00 ]

## Jacobi-ev postupak

NORMA JE: 0.704698 qq=2.38636
0     [ 0.199203 ; 0.492610 ; 0.109090 ; 0.536912 ; 0.369127 ; 0.343347 ]   0.536912
1     [ -0.03020 ; 0.280920 ; -0.15367 ; 0.275699 ; 0.195184 ; 0.116159 ]   0.261213
2     [ 0.113864 ; 0.422133 ; -0.01122 ; 0.429927 ; 0.304137 ; 0.268998 ]   0.154228
3     [ 0.027538 ; 0.334851 ; -0.10124 ; 0.328835 ; 0.235476 ; 0.175481 ]   0.101092
4     [ 0.082353 ; 0.389527 ; -0.04472 ; 0.391004 ; 0.278155 ; 0.233797 ]   0.062168
5     [ 0.048256 ; 0.355452 ; -0.07989 ; 0.352108 ; 0.251466 ; 0.197276 ]   0.038895
6     [ 0.069546 ; 0.376742 ; -0.05791 ; 0.376402 ; 0.268119 ; 0.220043 ]   0.024293
7     [ 0.056251 ; 0.363454 ; -0.07163 ; 0.361241 ; 0.257721 ; 0.205824 ]   0.015160
8     [ 0.064551 ; 0.371750 ; -0.06306 ; 0.370707 ; 0.264212 ; 0.214700 ]   0.009466
9     [ 0.059369 ; 0.366571 ; -0.06841 ; 0.364797 ; 0.260159 ; 0.209158 ]   0.005909
10    [ 0.062604 ; 0.369804 ; -0.06507 ; 0.368487 ; 0.262690 ; 0.212618 ]   0.003689
11    [ 0.060584 ; 0.367785 ; -0.06716 ; 0.366184 ; 0.261110 ; 0.210458 ]   0.002303
12    [ 0.061845 ; 0.369046 ; -0.06586 ; 0.367622 ; 0.262096 ; 0.211807 ]   0.001438
13    [ 0.061058 ; 0.368259 ; -0.06667 ; 0.366724 ; 0.261480 ; 0.210965 ]   0.000897
14    [ 0.061550 ; 0.368750 ; -0.06616 ; 0.367284 ; 0.261865 ; 0.211491 ]   0.000560
15    [ 0.061243 ; 0.368443 ; -0.06648 ; 0.366934 ; 0.261625 ; 0.211162 ]   0.000349
16    [ 0.061434 ; 0.368635 ; -0.06628 ; 0.367153 ; 0.261775 ; 0.211367 ]   0.000218
17    [ 0.061315 ; 0.368515 ; -0.06640 ; 0.367017 ; 0.261681 ; 0.211239 ]   0.000136
18    [ 0.061389 ; 0.368590 ; -0.06633 ; 0.367102 ; 0.261740 ; 0.211319 ]   0.000085
19    [ 0.061343 ; 0.368543 ; -0.06637 ; 0.367048 ; 0.261703 ; 0.211269 ]   0.000053
20    [ 0.061372 ; 0.368572 ; -0.06634 ; 0.367082 ; 0.261726 ; 0.211301 ]   0.000033

Jakobijeva metoda je iskonvergirala u 21 iteracija za tacnost 0.0001
```

```
## Gaus-Saidel-ov postupak

NORMA JE: 0.704698 qq=2.38636
0     [ 0.199203 ; 0.472984 ; 0.043005 ; 0.410996 ; 0.245928 ; 0.156489 ]   0.472984
1     [ 0.033939 ; 0.368083 ; -0.06232 ; 0.386496 ; 0.267580 ; 0.212173 ]   0.165263
2     [ 0.056456 ; 0.365000 ; -0.07002 ; 0.367697 ; 0.262758 ; 0.212961 ]   0.022516
3     [ 0.061854 ; 0.368274 ; -0.06685 ; 0.366545 ; 0.261632 ; 0.211401 ]   0.005397
4     [ 0.061558 ; 0.368647 ; -0.06629 ; 0.367005 ; 0.261678 ; 0.211247 ]   0.000560
5     [ 0.061363 ; 0.368578 ; -0.06634 ; 0.367080 ; 0.261716 ; 0.211281 ]   0.000195
6     [ 0.061355 ; 0.368560 ; -0.06636 ; 0.367072 ; 0.261718 ; 0.211289 ]   0.000020

Gauss Saidel-ova metoda je iskonvergirala u 7 iteracija za tacnost 0.0001
```

Broj iteracija će uglavnom zavisiti od norme matrice, ukoliko je dimenzija sistema reda nekoliko stotina a norma matrice bude bliska normi datoj u ovom primeru, broj iteracija će biti sličan, čak će se i smanjivati sa povećanjem dimenzije problema. U tabeli je dat broj iteracija za različite dimenzije sistema

dimenzija	norma	Jacobi	Gauss-Saidel
1	0.00	1	1
2	0.65	9	5
4	0.73	20	6
8	0.77	27	6
16	0.81	32	6
32	0.82	37	6
64	0.82	36	6
128	0.83	35	6
256	0.83	32	5
512	0.83	29	5
1024	0.83	25	4
2048	0.83	22	4
4096	0.83	18	4
8192	0.83	14	4

4.1 Način realizacije

Za rešavanje ovog problema najprihvatljivije je programiranje zasnovano na objektima, jezik koji se koristi je C++, i korišćene su isključivo standardne biblioteke kako bi kod mogao prevesti bilo koji C++ prevodilac. U ovom radu za razvoj korišćen je Gnu-ov C++ prevodilac (verzija 3.3).

Za baznu klasu izabrana je klasa **matrica** u kojoj će biti definisane metode koje su zajedničke za sve tipove matrica. Matrica predstavlja tablicu brojeva (najčešce realnih, ma da mogu biti nad bilo kojim poljem). Tip matrice koji se koristi u matematici su pravougaone matrice, kvadratne matrice i vektori (matrica kolona, ili matrica vrsta). U ovom radu su najviše obradjene kvadratne matrice. U praksi često koriste specifične matrice, specifičnost matrica ogleda se u broju i rasporedu nula u tablici, kako svi brojevi zapisani u tablici zauzimaju isti memorijski prostor, nema smisla čuvati nule pogotovu ako one zauzimaju veliki deo matrice. U zavisnosti od položaja i količina nula najčešće se spominju *guste - dense matrice* (imaju mali broj nula), *retke - sparse matrice* (imaju veliki broj nula proizvoljno rasporedjenih po tablici), *soliteri - skyline matrice* (imaju veliki broj nula grupisanih u kolonama), blok matrice (uz dijagonalu matrice javljaju se kvadratni različitih dimenzija a ostatak je popunjen nulama) itd... Još jedna često korišćena klasa matrica su simetrične matrice $a_{ij} = a_{ji}$, te za pamćenje takvih matrica nema potrebe pamtiti sve podatke.

U ovom radu su implementirane guste i retke matrice. Za guste matrice korišćen je niz u kome se pamte brojevi i kome se pristupa direktno na osnovu ij koordinata elementa koji se traži po

formuli ($j \cdot sirina + i$), dok se za retke matrice koristi binarno drvo koje je implementirano u skupu **map** standardne biblioteke šablona jezika C++. Kao šablon za predragu koristi se par koordinata (x,y) a pretraga se vrši algoritmom složenosti $O(\log(n))$ jer se drvo održava balansiranim pri unosu i brisanju elemenata iz strukture. Implementirana je takodje i simetrična gusta matrica koja zauzima duplo manje memorijskog prostora nego gusta matrica jer važi $a_{ij} = a_{ji}$ medjutim u tom slučaju matrica služi samo za čuvanje podataka jer se bilo kojim transformacijama nad kolonama ili vrstama ove matrice matrica uvek održava u simetričnoj formi što prouzrokuje neispravne transformacije pa ovakav tip matrice nije korišćen u daljem radu ali se može koristiti za neke primene.

Najveći broj metoda za manipulaciju matricama i za razna izračunavanja implementiran je upravo na apstraktnoj klasi matrica. Na izvedenim klasama implementirani su metodi za pristup (čitanje i pisanje) kao i preopterećeni neki od već implementiranih metoda u apstraktnoj klasi (zbog brzine)

4.1.1 Apstraktna klasa matrica

Klasa matrica obezbedjuje osnovne metode za manipulisanje matricom, kako je klasa apstraktna ona ne nosi konkretne podatke osim podatke o dimenziji matrice. Klasa obezbedjuje da sve iz nje izvedene klase moraju imati metode get i put preko kojih se dolazi do konkretnih podataka koji se u memoriji mogu čuvati na različite načine (npr: niz za guste matrice ili binarno drvo (katalog) za retke matrice). U ovoj klasi su implementirane metode za manipulisanje podacima u matrici: zamena kolona/redova, množenje reda (kolone) i dodavanje drugom redu (koloni), računanje norme matrice, svodjenje matrice na donju (gornju) trougaonu matricu ili dijagonalnu matricu gausovim metodom eliminacije, Pripreme matrice za iterativni metod, itd...

Sve metode opisane u ovoj klasi mogu se koristiti na sistemima sa jednim procesorom, samim tim klasa matrica, kao i iz nje izvedene klase gusta, retka, kvadratna i simetrična mogu biti korišćene na bilo kom operativnom sistemu i C++ prevodiocu. Većina metoda je manipulativnog karaktera, tj menja sadržaj tablice u matrici, na primer: metoda gaussDeterminanta će matricu svesti na gornji trougaoni oblik i izračunati determinantu tako što će izmnožiti dijagonalne elemente. Te zbog toga, ukoliko je potrebno da matrica bude sačuvana mora se prvo napraviti njena kopija a zatim na kopiji izračunati determinanta, što će uraditi funkcija determinanta deklarisana u fajlu *funkcije.hh*

Na narednih par stranica biće dat interfejs (deklaracija) klasa. Dok se implementacija klasa može naći u odgovarajucim **cc** fajlovima

```
class matrica {
  private:
    unsigned _sirina;                                        // Sirina matrice
    unsigned _visina;                                        // Visina matrice
  protected:
    void promeni_dimenzije(unsigned sirina, unsigned visina); // promena promenljivih sirina/visina
  public:
    #if defined BROJANJE
      static unsigned long broj_get_operacija;               // staticke promenljive (zajednicke
      static unsigned long broj_set_operacija;               // za sve instance klase matrice )
      static unsigned long broj_sabiranja;                   // koje se koriste za vremensku
      static unsigned long broj_mnozenja;                    // analizu algoritma
    #endif
    matrica(unsigned _sirina, unsigned _visina);             // podrazumevani konstruktor

    // apstraktne metode koje nece biti implementirane u ovoj klasi moraju biti navedene u svakoj
    // podklasi da bi klasa mogla biti instancirana kao objekat

    virtual T get(unsigned x, unsigned y) const = 0;         // get/set - uzimanje/postavljanje
    virtual void set(unsigned x, unsigned y, T broj) = 0;    // broja na koordinate x,y
    virtual unsigned size() const = 0;                       // size vraca realno zauzece memorije
                                                             // koju dati objekat koristi
```

```
    virtual void nule() = 0;                                    // postavlja sve elemente na nulu

    // redimenzionise matricu (promena dimenzije, i alociranje novog sadrzaja)
    virtual void postaviNovuMatricu(unsigned sirina, unsigned visina) = 0;
    inline unsigned sirina() const { return _sirina; }          // vraca dimenzije matrice
    inline unsigned visina() const { return _visina; }

    void randomN(T max=10, T min=0);                            // ispuna matrice slucajnim brojevima
    void napraviJedinicnu();                                    // pravi jedinicnu matricu

    void zameniKolone(unsigned k1, unsigned k2);                // osnovne manipulacije sa
    void zameniRedove(unsigned r1, unsigned r2);                // redovima i kolonama matrice
    void pomnoziIDodajKolonu(unsigned k1, T faktor, unsigned k2);
    void pomnoziKolonu(unsigned k, T faktor);
    void pomnoziIDodajRed(unsigned r1, T faktor, unsigned r2);
    void pomnoziRed(unsigned r, T faktor);

    void izvadiKolonu(unsigned k, matrica & b) const;           // metode potrebne za kramerovo
    void zameniKolonu(unsigned k, const matrica & b);           // pravilo

    T absSumaReda(unsigned r);                                  // racunanje norme matrice
    T absSumaKolone(unsigned k);
    T normaRedova();
    T normaKolona();

    bool normaRedovaZaJacobi( T & norma );
    bool normaKolonaZaJacobi( T & norma );
    bool kvadratna() const;                                     // vraca true ukoliko je matrica kvadratna

    bool gaussDoleNule(matrica * M=0);                          // familija funkcija za gausovu
    bool gaussGoreNule(matrica * M=0);                          // metodu eliminacija
    bool gaussDoleNulePivot(matrica * M=0, short * sgn=0);
    bool gaussGoreNulePivot(matrica * M=0);
    bool gaussDiag(matrica * M=0);
    bool gaussInverzna(matrica & A1);                           // A^-1
    bool gaussResiSistem(matrica * b,matrica * x);              // resava sistem
    T gaussDeterminanta();                                      // racuna determinantu

    void pripremiZaJacobi(matrica &b);                          // funkcije za jacobi-evu metodu iteracija
    void jacobiIteracija(matrica &b, matrica &x, vector<T> & maxRazlike, bool gausSaidel=false) const;

    ostream & ispisi(ostream & ostr, matrica * b=0) const ;     // izlaz (tip matlab)
    ostream & ispisi1(ostream & ostr) const ;                   // izlaz (tip mathematica)
    friend ostream & operator << ( ostream & ostr , const matrica & M );  // ulaz (tip matlab)
};
```

4.1.2 Izvedene klase gustaMatrica, gustaKvadratnaMatrica i gustaSimetrična matrica

Apstraktna metoda sama po sebi nije od neke koristi jer ne nosi podatke, najjednostavnija (i u ovom radu najviše korišćena konkretizacija ove metode je gusta matrica)

```
class gustaMatrica : public matrica {
  private:
    T *data;                                                    // nosac podataka
  public:
    gustaMatrica(unsigned _sirina=1, unsigned _visina=1);       // podrazumevani konstruktor
    gustaMatrica(const gustaMatrica &A);                        // kopi konstruktor
    gustaMatrica & operator=(const gustaMatrica &A);            // operator dodele
    ~gustaMatrica();                                            // destruktor
    inline unsigned sizeOfT() { return sizeof(T); }             // velicina tipa T
    inline unsigned sizeOfData() { return visina()*sirina(); } // broj elemenata u matrici
    inline T *pointerOfData() { return data; }                  // pokazivac na podatke
```

```
    gustaMatrica(T *ddata, unsigned _sirina, unsigned _visina);// konstruktor preko polja
                                                               //  sa podacima
    T get(unsigned x, unsigned y) const;                       // konkretizacija virtualnih
    void set(unsigned x, unsigned y, T broj);                  // metoda iz apstraktne klase
    unsigned size() const;                                     // matrica
    void nule();
    void postaviNovuMatricu(unsigned sirina, unsigned visina);

    gustaMatrica & operator += ( const gustaMatrica & B );     // neki aritmeticki i
    gustaMatrica & operator -= ( const gustaMatrica & B );     // poredbeni operatori
    gustaMatrica & operator *= ( T faktor );
    gustaMatrica & operator /= ( T faktor );
    bool operator == ( const gustaMatrica & B ) const ;
    bool operator != ( const gustaMatrica & B ) const ;
};
```

Sledeća konkretizacija predstavlja kvadratnu matricu kao specijalizaciju pravougaone matrice

```
class gustaKvadratnaMatrica : public gustaMatrica {
  public:
    gustaKvadratnaMatrica(unsigned dim=1);                     // podrazumevani konstruktor
    void transponuj();                                         // transponovanje matrice
};
```

Sledi interfejs simetrične matrice. Simetrična matrica predstavlja specijalizaciju kvadratne matrice, ali se sa gledišta programera simetrična matrica pre svega odlikuje po načinu pakovanja podataka pa će zato biti izvedena direktno iz klase matrica.

```
class gustaSimetricnaMatrica : public matrica {
  private:
    T *data;                                                   // nosac podataka
    unsigned sz;                                               // velicina "data" niza

  public:
    gustaSimetricnaMatrica(unsigned dim);                      // podrazumevani konstruktor
    gustaSimetricnaMatrica(const gustaSimetricnaMatrica &A);   // konstruktor kopije
    gustaSimetricnaMatrica & operator=(const gustaSimetricnaMatrica &A); // dodela
    ~gustaSimetricnaMatrica();                                 // destruktor

    T get(unsigned x, unsigned y) const;                       // konkretizacija virtualnih
    void set(unsigned x, unsigned y, T broj);                  // metoda iz apstraktne klase matrica
    unsigned size() const;                                     // metode get i set su kljucne u ovoj
    void nule();                                               // klasi

    void randomN(T max=10, T min=0);                           // preopterecenje metode randomN

    gustaSimetricnaMatrica & operator += ( const gustaSimetricnaMatrica & B );
    gustaSimetricnaMatrica & operator -= ( const gustaSimetricnaMatrica & B );
    gustaSimetricnaMatrica & operator *= ( T faktor );
    gustaSimetricnaMatrica & operator /= ( T faktor );
    bool operator == ( const gustaSimetricnaMatrica & B ) const ;
    bool operator != ( const gustaSimetricnaMatrica & B ) const ;
};
```

4.1.3 Izvedena klasa retkaMatrica

Sledeća grupa nosača podataka koja je prikazana su retke (*sparse*) matrice, koje koriste šablon klasu *map* (asocijativni kontejner) iz standardne biblioteke šablona, kao i šablon klasu *pair* koja predstavlja ključ u asocijativnom kontejneru.

```
typedef pair<unsigned,unsigned>       pos;                 // definisanje pozicije
typedef map<pos,T>::iterator          positer;             // iterator za poziciju
typedef map<pos,T>::const_iterator    cpositer;            // iterator za citanje pozicije
```

```
class retkaMatrica : public matrica {
  private:
    map <pos,T> data;                                          // nosac podataka

    bool postoji( pos & p, cpositer & iter ) const;            // da li postoji broj na poziciji pos
    void obrisi( pos & p );                                    // brise broj sa pozicije pos
  public:
    retkaMatrica( unsigned s=1, unsigned v=1);                 // podrazumevani konstruktor
    retkaMatrica(const retkaMatrica &A);                       // kopi konstruktor
    retkaMatrica & operator=(const retkaMatrica &A);           // operator dodele
    ~retkaMatrica();                                           // destruktor
    void sviElementi() const;                                  // ispisuje elemnte ('debug' metoda)

    T get(unsigned x, unsigned y) const;                       // dohvatanje podataka sa lokacije
    void set(unsigned x, unsigned y, T broj);                  // postavljanje podataka na lokaciju
    unsigned size() const;                                     // velicina nosaca
    void nule();                                               // brisanje svih elemenat
    void postaviNovuMatricu(unsigned sirina, unsigned visina);// postavljanje nove matrice
};
```

4.2 Paralelni i distribuirani računarski sistemi

Već duže vreme oblast paralelnog izračunavanja u središtu razvoja računarstva. U upotrebi je više tipova paralelnih računara, sa veoma velikim brojem procesora. Nije moguće usvojiti jedan opšti model izračunavanja koji bi obuhvatio sve paralelne računare. Prema broju procesora može se napraviti sledeća klasifikacija: **jednoprocesorski sistemi** koji ne mogu vršiti stvarna paralelna izračunavanja, ali je zahvaljujući brzini procesora pseudo paralelizam na ovim sistemima već duže vreme prisutan; **čvrsto vezani višeprocesorski sistemi** se sastoje od skupa procesora koji dele zajedničku glavnu memoriju i nalaze se pod integralnom kontrolom i upravljanjem operativnog sistema; **labavo vezani višeprocesorski sistemi** se sastoje od skupa relativno autonomnih sistema koji imaju svoje posebne procesore, memoriju i ulazno izlazne kanale. Ovakav sistem se naziva Multiračunar (*multicomputer*) ili Klaster (*cluster*) doslovno prevedeno "gomila".

Prema broju tokova instrukcija i broju tokova podataka može se napraviti sledeća podela:

- **SISD** *Single Instruction Single Data* jedna instrukcija, jedan podatak;
- **SIMD** *Single Instruction Multiple Data* jedna instrukcija, više podataka (ovakvi sistemi imaju jednu jedinicu za obradu instrukcija i više jedinica za obradu podataka, tzv. *vektorski procesori*);
- **MISD** *Multiple Instruction Single Data* više instrukcija, jedan podatak (ovakvi sistemi prenose podatke do skupa procesora koji istovremeno izvršavaju razlicite instrukcije - u praksi ovakav sistem nije zaživeo);
- i na kraju najinteresantnija **MIMD** *Multiple Instruction Multiple Data* više instrukcija, više podataka (ovi sistemi istovremeno (sa više procesora) izvršavaju različite instrukcije nad različitim podacima).

Postoje dva tipa **MIMD** Arhitekture, sa deljenom i disribuiranom memorijom, tj čvrsto vezani procesorski sistem i labavo vezani procesorski sistemi. Čvrsto vezani procesorski sistem predstavlja bolje rešenje jer se komunikacija vrši putem deljene memorije što je ubedljivo najbrži način. Medjutim labavo vezani sistemi imaju odredjene prednosti, glavna od njih je fleksibilnost u broju procesora jer broj procesora kod čvrsto vezanih sistema može biti ograničen mogućnostima matične ploče računara. Osobine labavo vezanog procesorskog sistema su:

- svaki od procesora poseduje sopstveni nezavistan memorijski sistem;

- računarski sistemi ne moraju biti identični, snaga procesora može značajno da se razlikuje izmedju računara u sistemu;
- procesori se nalaze pod kontrolom zajedničkog operativnog sistema (moguće je da se ista verzija operativnog sistema izvršava na svakom od procesora);
- svaki od računara u sistemu je potpun računarski sistem koji je sposoban da obavlja operacije nezavisno od ostalih;
- podaci izmedju računara razmenjuju se putem prenošenja poruka;
- ovakav računarski sistem je krajnje fleksibilan, unapredjenje sistema se može izvršiti dodavanjem novih računara (node-ova);

4.3 Posix threads - paralelni programi na čvrsto vezanim procesorima

4.3.1 Implementacija Jacobi-eve metode

Korišćenjem **Posix** niti (*pthreads*) Jacobi-eva iterativna metoda se može vrlo lako paralelizovati na čvrsto vezanim višeprocesorskim sistemima. Paralelizacija se može izvesti na više nivoa. Najelementarniji bi bio da se paralelno izvršavaju osnovne računske operacije i ukoliko ne bi postojali gubitci pri komunikaciji izmedju procesa (procesora) paralelizacija na najnižem nivou bi bila najefikasnija, medjutim u praksi to nije slučaj. Kao druga krajnost može se uzeti paralelno rešavanje više različitih sistema. Ova metoda se u praksi vrlo lako sprovodi, ali za njom pre svega mora postojati mogućnost i potreba, tj da za rešenje nekog problema treba rešiti dva ili više sistema linearnih jednačina pri čemu se rezultati jednog ne koriste u postavci nekog drugog (ovakav način se može ilustrovati rešavanjem jednog sistema metodom determinanti, kod koje je potrebno izračunati $n+1$ determinantu različitih matrica). Medjutim često je potrebno rešiti samo jedan sistem linearnih jednačina, tako da je optimalno rešenje je kao i uvek negde izmedju dve krajnosti.

Nemoguće je izvršiti više od jedne iteracije istovremeno jer izračunavanje naredne iteracije zavisi od rezultata dobijenih u prethodnoj iteraciji. Medjutim dobra strana Jakobi-jeve metode je da u jednoj iteraciji treba izračunati više suma koje medjusobno ne zavise jedna od druge što nije slučaj kod Gauss-Saidel-ove modifikacije ove metode.

Obzirom na prethodno, najoptimalnije rešenje je da se jedna iteracija paralelizuje i to na taj način što bi se dobijene iterativne formule nezavisno rešavale. U zavisnosti od broja procesora na sistemu potrebno je odrediti broj niti u kojima će se izračunavati iterativne formule. Jedna krajnost u ovom izboru je da se svakoj jednačini dodeli poseban proces, ali kako je metoda numerički zahtevna deljenje više procesa na jednom procesoru nema mnogo smisla, odakle sledi da se jednoj iterativnoj jednačini dodeli jedan procesor, što je u praksi nemoguće jer je najčešće na raspolaganju mnogo manji broj procesora nego dimenzija problema (koja uostalom može biti proizvoljna). Kao neko realno rešenje nameće se da se dimenzija sistema jednačina podeli sa brojem procesora i da se na taj način rešavanje jedne iteracije razdeli na moguće procese, tj da svaki od procesora rešava unapred zadati skup iterativnih formula. Medjutim ni to rešenje nije najbolje jer se u napred ne može znati da li je za rešavanje k prvih formula i nekih k drugih formula potrebno isto vreme jer neke mogu imati više nula od drugih, naravno ovo se može izbeći analizom sistema ali ispostavlja se da to nije potrebno jer u razmatranju nije uzeto u obzir da neki procesori mogu biti opterećeniji drugim pozadinskim procesima koje operativni sistem kontroliše u pozadini.

Rešenje koje je primenjeno u ovom radu zasniva se na redovima za čekanje. Na početku je potrebno instancirati potreban broj procesa. Najoptimalnije je da broj procesa bude jednak broju procesora, medjutim pokazuje se da gubitci nisu veliki ukoliko se instancira više procesa, ali naravno od toga nema koristi. Procesi bi redom uzimali prvu neizračunatu iterativnu formulu i izračunavali je sve

dok se ne iscrpu sve formule u jednoj iteraciji. Koja će formula kada biti izračunata i od strane kog procesa ničim ne može biti determinisano.

Slede dekleracije funkcija koje opisuju metodu

```
// funkcija preko koje korisnik pokrece metodu
unsigned jacobiPthreads(const gustaMatrica & _A, const gustaMatrica & _b,
                        gustaMatrica & _x, unsigned num_of_threads);

void * thrPripremiZaJacobi (void *voidargs);            // nit u kojoj se priprema matrica
void pripremiZaJacobiPthreads(unsigned num_of_threads); // f-ja koja pokrece niti za pripremu
void * thrJacobiObradi(void *voidargs);                 // nit koja obradjuje iteracije
void iteracija(unsigned num_of_threads);                // f-ja koja pokrece niti za obradu
```

4.3.2 Analiza rezultata

U sledećoj tabeli dati su vremenski rezultati rešavanja sistema linearnih jednačina raznih dimenzija u zavisnosti od broja niti koji se koristi. Merenje je vršeno na sistemu sa dva procesora

dimenzija (n x n)	1 nit (sec)	2 niti (sec)	3 niti (sec)	4 niti (sec)	broj iteracija (jacobi)
1	0.0002	0.0003	0.0003	0.0003	1
2	0.0003	0.0004	0.0004	0.0005	8
4	0.0003	0.0005	0.0006	0.0007	19
8	0.0005	0.0006	0.0007	0.0008	26
16	0.0006	0.0008	0.0009	0.0010	31
32	0.0012	0.0013	0.0015	0.0017	36
64	0.0030	0.0032	0.0034	0.0035	35
128	0.0102	0.0104	0.0106	0.0107	34
256	0.0586	0.0587	0.0596	0.0598	31
512	0.2052	0.2034	0.2103	0.2005	28
1024	0.7118	0.5811	0.5877	0.5849	24
2048	2.6056	1.5633	1.5575	1.5853	21
4096	8.5705	4.8865	4.8951	4.9147	17
8192	29.914	16.265	16.471	16.203	13

Merenje je radjeno na računaru ***sokocalo.engr.ucdavis.edu*** sa procesorima navedenih karakteristika: 2 x (AMD Athlon MP 2400+, 2 Ghz, 256 KB cache, 3948 mips). Dobijeni rezultati predstavljaju potrebno vreme za rešavanje datih sistema, ta vremena medjutim zavise od još nekoliko faktora. Pogodnije bi bilo da vremena nisu zadata sa 4 decimale ali se tada ne bi videlo vreme potrebno za rešavanje sistema malih dimenzija.

Iz ovog merenja može se zaključiti nekoliko činjenica:

- za sisteme malih dimenzija ubrzanje deljenjem na više procesora ne postoji, čak je prisutno i blago usporenje koje je izazvano medjuprocesnom komunikacijom;
- za sisteme većih dimenzija $n > 1000$ primetno je ubrzanje koje se ubrzo fiksira na onoliko puta koliko sistem poseduje procesora
- programe sa intenzivnim numeričkim izračunavanjima kao što je rešavanje sistema jednačina nema svrhe paralelizovati na više niti nego što je broj procesora koji je na raspolaganju jer ne postoje nikakvi značajni dobitci, medjutim ni gubitci kada postoje nisu značajni tako da paralelizacija na više niti nije štetna.

4.3.3 Implementacija paralelnog množenja matrica

Množenje dve matrice će biti paralelizovano po istom principu kao i jedna iteracija Jacobi-eve metode. Na početku je potrebno instancirati potreban broj niti koje će iz reda poslova uzimati red matrice A i kolonu matrice B i zatim vršiti potrebna sabiranja i množenja kako bi se dobio jedan element matrice C. Implementacija je slična kao i kod prethodne metode. Sledi tabela rezultata merenja množenja dve matrice na dvoprocesorskom sistemu ***sokocalo.engr.ucdavis.edu***

dimenzija (n)	A dim	B dim	C dim	1 nit (sec)	2 niti (sec)	3 niti (sec)	4 niti (sec)
1	3x1	2x3	2x1	0.0001	0.0001	0.0002	0.0002
2	6x2	4x6	4x2	0.0001	0.0002	0.0002	0.0002
4	12x4	8x12	8x4	0.0001	0.0002	0.0002	0.0002
8	24x8	16x24	16x8	0.0002	0.0003	0.0003	0.0003
16	48x16	32x48	32x16	0.0009	0.0009	0.0009	0.0010
32	96x32	64x96	64x32	0.0061	0.0062	0.0062	0.0059
64	192x64	128x192	128x64	0.0532	0.0396	0.0380	0.0371
128	384x128	256x384	256x128	2.7270	1.3885	1.4381	1.4051
256	768x256	512x768	512x256	23.330	12.001	12.071	11.965
512	1536x512	1024x1536	1024x512	187.50	95.893	96.286	96.187

Iz date tabele mogu se izvući isti rezultati kao i iz prethodne, na koje se može dodati i to da u ovom algoritmu vreme izvršavanja počinje naglo da raste, zbog kubne složenosti algoritma, dok se u prethodnom algoritmu vreme izvršavanja povećavalo linearno.

4.4 Programiranje slanjem poruka - MPI

MPI (*Message Passing Interface*) je standard razvijen od strane MPI-Foruma. i predstavlja interfejs za pisanje programa zasnovanih na slanju poruka. Standard uključuje: Point-to-point komunikaciju; Zajedničke operacije; Grupe procesa; Konteksti komunikacije; Topologije procesa; Ugradjenu implementaciju za Fortran i C; Ispitivanje i organizovanje okruženja i interfejs za profilisanje.

Za prevodjenje i pokretanje MPI baziranih programa korišćen je **MPICH** prevodilac koji predstavlja besplatnu (*Open Source*) verziju implementacije MPI standarda. Program se prevodi komandom mpicc, mpiCC, mpif77 ili mpif90 u zavisnosti od jezika na kome je program pisan. Dok se pokretanje programa vrši pomoću skripte mpirun gde je glavni parametar broj node-ova (računara) na kojima će se izvršavati program i ime programa koji se pokreće. Potom se u instancira potreban broj procesa na node-ovima kao i na glavnom računaru i pokreće se program. Da bi program ispravno radio potrebno je da zadovolji odredjenu formu propisanu standardom.

Trivijalan program na **C++**-u imao bi sledeću formu

```
#include <iostream>
#include <mpi.h>

using namespace std;

int main(int argc, char **argv) {
  MPI::Init(argc, argv);
  int mojID       = MPI::COMM_WORLD.Get_rank();
  int brojProcesa = MPI::COMM_WORLD.Get_size();
  cout << "Pozdrav od procesa " << mojID << " od ukupno "
       << brojProcesa << " procesa" << endl;
  MPI::Finalize();
}
```

Na početku dela programa u kojem će se koristiti paralelno izračunavanje potrebno je inicijalizovati MPI funkcijom `MPI_Init` kojoj se prenose argumenti komandne linije. Proces se na kraju završava funkcijom `MPI_Finalize()`, dok će se sve što se nalazi izmedju te dve komande izvršavati paralelno na zadatom broju node-ova. Funkcija `MPI_Comm_size`; daje informaciju o ukupnom broju procesa, dok funkcija `MPI_Comm_rank`; daje identifikaciju tekućeg procesa, identifikacija - rang (rank) procesa je jedinstven broj koji predstavlja glavni parametar pri komunikaciji izmedju procesa. U ovom jednostavnom primeru se instancira zadat broj procesa i svaki od njih radi isto tj, ispisuje svoj rang i ukupan broj procesa. Ako se program kompajlira i pokrene dobiće se *otprilike* sledeći izlaz:

```
$ mpicc hello.c -o hello
$ mpirun -np 5 ./hello
  Pozdrav od procesa 0 od ukupno 5 procesa
  Pozdrav od procesa 2 od ukupno 5 procesa
  Pozdrav od procesa 3 od ukupno 5 procesa
  Pozdrav od procesa 1 od ukupno 5 procesa
  Pozdrav od procesa 4 od ukupno 5 procesa
```

reč *otprilike* nije slučajno napisana jer redosled po kojima će se procesi javljati ničim nije unapred utvrdjen. Ponovnim pokretanjem istog programa na istom klaster računaru može se dobiti drugačiji izlaz, no ovo naravno nije nikakav problem jer se putem slanja poruka program može vrlo jednostavno sinhronizovati.

4.4.1 Tipovi podataka

Tipovi podataka koji se prenose metodama koje nudi interfejs za slanje poruka (MPI), uglavnom su saglasni sa tipovima podataka koji se sreću C/C++, oni su nabrojani (zajedno sa širinom) u sledećoj tabeli:

tip podataka	broj bajtova
MPI::PACKED	1
MPI::BYTE	1
MPI::CHAR	1
MPI::UNSIGNED_CHAR	1
MPI::SIGNED_CHAR	1
MPI::WCHAR	2
MPI::SHORT	2
MPI::UNSIGNED_SHORT	2
MPI::INT	4
MPI::UNSIGNED	4
MPI::LONG	4
MPI::UNSIGNED_LONG	4
MPI::FLOAT	4
MPI::DOUBLE	8
MPI::LONG_DOUBLE	16
MPI::CHARACTER	1
MPI::LOGICAL	4
MPI::INTEGER	4
MPI::REAL	4
MPI::DOUBLE_PRECISION	8
MPI::COMPLEX	2x4
MPI::DOUBLE_COMPLEX	2x8

4.4.2 Interprocesorska komunikacija na Računaru sa distribuiranom memorijom

Kod sistema sa distribuiranom memorijom svaki procesor može pristupati samo svojoj memoriji (memoriji na računaru na kome se nalazi taj procesor). Ne postoji deljena memorija kojoj više procesora mogu pristupati. Jedini način da procesor sa jednog računara pristupi memoriji na drugom računaru je eksplicitno slanje poruke. Dva osnovna metoda za rad sa porukama koje nudi MPI su **MPI_Send** i **MPI_Recv**

```
int MPI::Comm.Send( void* sendBuf, int count, MPI_Datatype datatype,
                    int destinationRank, int tag );

int MPI::Comm.Recv( void* recvBuf, int count, MPI_Datatype datatype,
                    int sourceRank, int tag);
```

- **Comm** - komunikacioni domen, najčešće se koristi COMM_WORLD koji sadrži sve procesore
- **sendBuf** - pokazivač na memorijski prostor gde se nalaze podaci koje treba poslati
- **recvBuf** - pokazivač na memorijski prostor gde će podaci biti smesteni
- **count** - broj podataka (1 ukoliko se radi o primitivnom podatku ili **n** ukoliko se šalje niz podataka)
- **datatype** - tip podataka koji se šalje MPI::INT, MPI::FLOAT, ...
- **destinationRank** - ID procesa kome se podatak šalje
- **sourceRank** - ID procesa od koga se podatak prima
- **tag** - je identifikator koji se koristi da bi označio vrstu poruke, jer dva procesa mogu razmenjivati više različitih vrsta poruke, definisanje tag-a je ostavljeno programeru

Metode **MPI_Send** i **MPI_Recv** će blokirati izvršavanje procesa sve dok poruka u celini ne bude poslata odnosno prihvacena

Primer: *Komunikacija izmedju procesa* ***mpiPrimeri/komunikacija.tgz***

4.4.3 Metode za kolektivnu komunikaciju

Ukoliko je potrebno da se jedan isti podatak prosledi svim procesima u grupi, moguće je upotrebiti metodu Bcast.

```
int MPI::Comm.Bcast( void *poruka, int count, MPI_Datatype datatype, int tag);
```

Sintaksa Bcast metode je potpuno ista u glavnom procesu i u procesima koji primaju podatak

Primer: *Slanje iste poruke svim procesima* ***mpiPrimeri/broadcast.tgz***

4.4.4 Dekompozicija računa na više procesa i redukovanje rezultata

Ukoliko se neko izračunavanje može podeliti na više nezavisnih delova (procesora) tada se rezultat koji je svaki procesor izračunao može sklopiti pomoću metode **MPI_Reduce**.

```
MPI::Comm.Reduce(void *sendBuf, void *recvBuf, 1, MPI_Datatype, MPI_Op, int root);
```

- **operation** - Operacija koja će se izvršiti po sklapanju rezultata
- **sourceRank** - ID root procesa (najčešće 0)

Po izračunavanju svog dela, svaki proces komandom **MPI_Reduce** šalje svoj rezultat, ukoliko je ID procesa 0 **MPI_Reduce** će primiti finalan rezultat koji je nastao primenom MPI_Op operacije nad svim izračunatim rezultatima. Jednostavan primer je integracija: Oblast koju treba integrisati na neki način treba podeliti na podoblasti koje će svaki proces integrisati nezavisno. Zatim se metodom **MPI_Reduce** spajaju izračunati delovi i u root procesu se može očitati rezultat. Naravno sve se može izvršiti i metodama Send i Recv ali je **MPI_Reduce** jednostavnija

Primer: *računanje broja* π *integracijom* $\pi = \int_0^1 \frac{4}{1+x^2}$ ***mpiPrimeri/parallelPi.tgz***

4.5 MPI Implementacija Kramerovog pravila

Kramerovo pravilo predstavlja jednu direktnu metodu koja nije primenljiva u praksi, ali može poslužiti kao dobra ilustracija rada na distribuiranom sistemu. Rešenje sistema jednačina ovom metodom svodi se na rešavanje n različitih determinanata i deljenjem dobijenih rezultata sa determinantom sistema. Potrebno je izračunati $n+1$ determinanta, kako je računanje determinante veoma kompleksne složenosti $O(n!)$ sama metoda neće biti primenljiva za veće sisteme ali će dobro poslužiti za poredjenje brzine računanja sistema jednačina na različitom broju procesora u distribuiranom sistemu. Za samo računanje determinante biće korišćen Gausov sistem eliminacija koji će značajno smanjiti kompleksnost računa sa $O(n!)$ na $O(n^3)$ što je minimum za direktnu metodu računanja determinante.

Sve matrice za koje je potrebno izračunati determinantu grade se od polazne matrice sistema na taj način što se i-ta kolona matrice zameni sa vektorom slobodne promenljive b. Odatle se zaključuje da je svakom računaru u sistemu potrebno dostaviti matricu sistema A i vektor slobdnih promenljivih b. Ovde se može primeniti **Broadcast** metoda. Slanjem poruka od **master** procesa ka **slave** procesima biće dirigovano na kojoj koloni treba ubaciti vektor b i kao odgovor treba očekivati determinantu date matrice. Zatim se u **master** procesu računaju vredosti nepoznatih.

4.5.1 Implementacija

Razvijena je klasa `distrGustaMatrica` koja nasledjuje klasu `gustaMatrica` dodajući joj metode za komunikaciju preko MPI protokola i neke dodatne atribute (koji će biti potrebni za Jakobijev iterativni metod i biće opisani nešto kasnije). Razlikuju se dve funkcije **master()** i **slave()**, funkcija master() se poziva ukoliko je rank instance programa jednak nuli, ova funkcija je ujedno i dirigent u metodi, ali pored odredjivanja poslova ona i sama izvodi računske operacije.

Sledi interfejs klase distribuirane matrice (klasa je specijalno prilagodjena za Jacobi-evu metodu iteracija)

```
class distrGustaMatrica : public gustaMatrica {
  private:
    unsigned prvaJednacina;                                    // oznaka dela jednacine
    unsigned n;                                                // sirina sistema
    unsigned m;                                                // sirina matrice slobodnih vektora
  public:
    distrGustaMatrica(unsigned prva=NEDEFINISANO , unsigned _s=0, unsigned _v=0) :
              gustaMatrica( _s, _v ) { prvaJednacina=prva; }  // podrazumevani konstruktor

    void MPI_Send(unsigned destination) const;                 // slanje objekta drugom cvoru
    void MPI_Recv(unsigned source);                            // primanje objekta
    void MPI_Bcast(unsigned rank);                             // slanje objekta svim cvorovima

    inline void set_nm(unsigned _n, unsigned _m) { n=_n; m=_m; } // postavljanje promenljivih n i m
    inline unsigned nm() { return n*m; }                       // velicina n*m
    inline unsigned get_n() { return n; };                     // n
    inline unsigned get_m() { return m; };                     // m
```

```
    inline unsigned prva() const { return prvaJednacina; }    // broj prve jednacine u matrici
    T normaRedova_A_zaJacobi();                                // racunanje norme
};
```

Funkcija master generiše slučajnu matricu i slobodan vektor i BroadCast metodom ih šalje ostalim čvorovima. Zatim iteracijom po dimenziji sistema funkcija se sukcesivno javlja svakom od čvorova i zadaje mu zadatak koji on treba da reši, tj koju determinantu treba da izračuna. Tada čvor zamenjuje slobodan vektor na odgovarajuće mesto u matrici sistema, računa determinantu i rezultat pamti u lokalni niz. Kada se master() funkcija jednom obrati svim čvorovima ona preuzima sledeći posao i radi isto što su radili i ostali čvorovi (zbog kompleksnosti računanja determinante $O(n^3)$ u odnosu na količinu podataka koja se stalno prenosi (nekoliko bajtova)) može se smatrati da će (ukoliko su čvorovi sličnih karakteristika), izračunavanje biti zavrvšeno u isto vreme. Zatim se novi poslovi šalju čvorovima sve dok se ne iscrpi sve kolone.

Na kraju procesa slave() funkcije šalju rezultate master() funkciji koja ih uparuje i deli odgovarajuće determinante sa determinantom sistema i ispisuje rezultat!

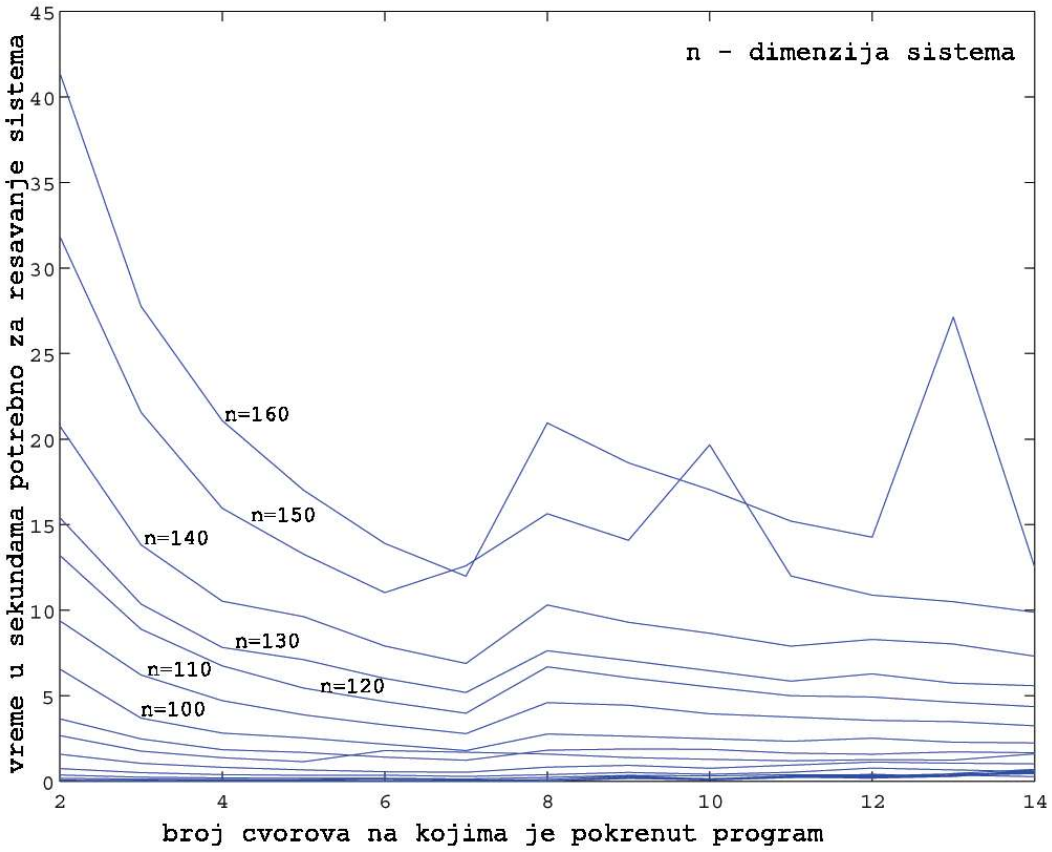
4.5.2 Testiranje programa

Za testiranje programa na raspolaganju je bio klaster računar sa 7 čvorova. Čvorovi predstavljaju računare koji poseduju procesore sa radnim taktom od 400 MHz, i sa po 128 Mb memorije. Obzirom da to nisu reprezentativni računari (jer je kao sistem sa čvrsto vezanim procesorima korišćen računar sa dva procesora koja rade na po 2000 MHz) vremenski neki rezultati su lošiji nego na sistemu manje čvrsto vezanih procesora. Ta činjenica ipak ne ugrožava donošenje ispravnih zaključaka jer se vremena posmatraju relativno (jedna u odnosu na druga).

GeoWulf.engr.ucdavis.edu - računar na kome sam testirao program

dim \ np	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0.0663	0.0581	0.0270	0.1023	0.1589	0.0967	0.0081	0.1832	0.1256	0.2795	0.1993	0.3414	0.4503
2	0.0331	0.0027	0.0077	0.0079	0.0078	0.0073	0.0086	0.1674	0.0873	0.2675	0.4153	0.2803	0.5897
3	0.0346	0.0382	0.0078	0.0067	0.0072	0.0091	0.0081	0.1705	0.0957	0.2500	0.2094	0.3705	0.4794
4	0.0338	0.0392	0.0392	0.0078	0.0078	0.0074	0.0090	0.2451	0.1462	0.2742	0.3084	0.3503	0.5391
5	0.0409	0.0368	0.0334	0.0396	0.0072	0.0073	0.0086	0.2500	0.0969	0.2604	0.1795	0.4570	0.7014
10	0.0319	0.0043	0.0083	0.0083	0.0085	0.0085	0.0093	0.2000	0.1557	0.2485	0.2489	0.2868	0.6611
20	0.0444	0.0100	0.0116	0.0114	0.0111	0.0109	0.0278	0.1749	0.0642	0.2218	0.2895	0.3599	0.2666
30	0.0498	0.0448	0.0437	0.0144	0.0146	0.0155	0.0282	0.2176	0.1593	0.2361	0.1874	0.2778	0.6873
40	0.1779	0.1202	0.1018	0.0865	0.1042	0.0697	0.0990	0.3142	0.1351	0.3208	0.3283	0.4293	0.4760
50	0.3622	0.2411	0.1927	0.1746	0.1769	0.1281	0.2252	0.3503	0.3108	0.3871	0.3877	0.2889	0.5187
60	0.7390	0.5062	0.3927	0.3452	0.3612	0.2794	0.3917	0.5176	0.4087	0.5305	0.7770	0.6650	0.5683
70	1.5849	1.0451	0.8102	0.6690	0.5631	0.5222	0.8222	0.9270	0.7649	0.9399	1.1220	1.0565	1.0127
80	2.6657	1.7614	1.3813	1.1438	1.7966	1.7024	1.5917	1.3854	1.2823	1.1953	1.2621	1.2367	1.5985
90	3.6509	2.4731	1.8500	1.6895	1.4102	1.2282	1.8112	1.8884	1.8728	1.6403	1.5809	1.7221	1.6629
100	6.5563	3.6891	2.8208	2.5434	2.1660	1.7920	2.7648	2.6329	2.4891	2.3341	2.5226	2.2844	2.2436
110	9.3803	6.2132	4.7150	3.8817	3.2953	2.7870	4.5955	4.4433	3.9445	3.7556	3.5634	3.4923	3.2444
120	13.201	8.8883	6.7420	5.4495	4.6638	3.9728	6.6920	6.0595	5.5121	4.9999	4.9287	4.6223	4.3639
130	15.411	10.357	7.8231	7.1140	6.0083	5.1914	7.6362	7.0572	6.4558	5.8543	6.2800	5.7344	5.5817
140	20.766	13.824	10.532	9.6201	7.9071	6.8864	10.309	9.2940	8.6519	7.8973	8.2877	8.0267	7.3110
150	31.842	21.559	15.960	13.278	11.031	12.595	15.637	14.084	19.671	11.996	10.875	10.498	9.8823
160	41.392	27.756	21.078	17.012	13.909	11.986	20.946	18.610	17.044	15.200	14.268	27.136	12.516

Već je napomenuto da Kramerovo pravilo za rešavanje sistema linearnih jednačina nije efikasno, to se može videti i sa rezultata testa. Medjutim najveća mana ovog metoda je ta što se efektivno ne mogu ni izračunati sistemi većih dimenzija. Matrice sistema su generisane slučajno i to sa veoma malim brojevima. Matrica sistema sadržala je samo brojeve iz skupa $\{1, 2, 3\}$ ali ipak je determinanta sistema dimenzije 150x150 bila veoma veliki broj $5.76 \cdot 10^{117}$ koji se u osmobajtnom double polju pamti ali ne baš najpreciznije, dok se determinante većih sistema ne mogu ni izračunati ukoliko se koristi double tip podataka, problem se delimično može rešiti uvodjenjem šesnaestobajtnog ili tridesetdvobajtnog floating point tipa. Pored toga, vremenski program sa porastom dimenzije veoma usporava svoj rad (bez obzira na broj računara u mreži) jer je kompleksnost algoritma $O(n^3)$. Detaljnija analiza podataka biće opisana zajedno sa analizom podataka Iterativne metode



4.6 MPI Implementacija Jacobi-eve metode

Jakobijeva iterativna metoda za rešavanje sistema linearnih jednačina, već je paralelizovana na sistemima sa čvrsto vezanim procesorima. Kao najbolje rešenje u tom slučaju pokazalo se rešenje sa redovima za čekanje. Matrica sistema pamćena je u zajedničkoj memoriji dok su procesi prilazili toj matrici red po red i obradjivali ga računajući delove jedne iteracije. Karakteristično za ovu metodu je da komunikacija izmedju procesa praktično ne postoji, osim u tome što procesi pristupaju istoj promenljivoj ogradjenoj mutex-om koja označava red matrice koji je sledeći za obradu. Kod labavo vezanih sistema kod kojih svaki procesor ima svoju autonomnu memoriju ne bi bilo praktično da se matrica sistema čuva u memoriji jednog računara a da procesor tog računara putem poruka šalje koeficijente jednačine koje mu zahteva odredjen procesor (zbog gubitka vremena pri transportu podataka kroz mrežu). Jedno od rešenja bi bilo da se **Broadcast** metodom učitan sistem pošalje svim računarima - tada bi moglo biti primenjeno rešenje slično rešenju na sistemima sa čvrsto vezanim procesorima, ali bi se memorija neracionalno trošila. Racionalnija potrošnja memorije bila bi kada bi se svakom računaru u mreži na samom početku dodelio odredjen broj jednačina, ovakav metod je ranije komentarisan kao loš upravo zbog moguće različitosti procesora u mreži kao i zbog eventualne preopterećenosti nekih računara nekim drugim poslom. Ali ukoliko se uzme u obzir vreme potrebno za slanje velikih količina podataka ovaj način se pokazuje dobar.

4.6.1 Implementacija

Implementacija Jacobi-eve metode izvedena je na sličan način kao i u prethodnom primeru. U izvedenoj klasi koja predstavlja distribuiranu matricu, pored metoda za MPI komunikaciju dodati su i atributi kojima se precizno odredjuje položaj podmatrice u matrici sistema [1].

master() funkcija poziva funkciju za učitavanje matrice sa standardnog ulaza koji se, naravno, može preusmeriti iz proizvoljnog fajla. Ulaz matrice mora biti saglasan sa ispisom matrice koju generiše metod ispis() u baznoj apstraktnoj klasi matrica. Pre učitavanja matrice poznat je broj čvorova na kojima će se izvršiti dalji proračun te funkcija automatski deli pakete čvorovima čim ih učita (na taj način dolazi do uštede memorije). Takodje, zbog vremenskih testova napravljena je i funkcija koja generiše random dijagonalno dominantnu matricu i zatim rešava sistem linearnih jednačina.

Kada je matrica jednom učitana i distribuirana svim čvorovima pristupa se rešavanju sistema. Kada čvor dobije paket on ga odmah prilagodi numeričkoj metodi i ulazi u beskonačnu petlju u kojoj će čekati komande instance programa koja pogreće master() funkciju. Beskonačna petlja neće nepotrebno uzimati resurse jer je prva stvar blokirajuci MPI_Recv metod koji propušta program tek kad poruka bude primljena u celosti. Kao poruke, master() funkcija šalje komande iza kojih šalje odgovarajuće argumente. slave() funkcija, a samim tim i tekuća instanca programa, završava rad kada dobije komandu CMD_TERMINATE.

Pored toga što master() funkcija šalje zahteve slave() funkcijama koji se izvršavaju u instancama na raznim čvorovima ona i sama rešava deo numeričkog izračunavanja. Paralielizovanje je izvršeno na nivou jedne iteracije, kao u primeru sa čvrsto vezanim procesorima (razlika je samo u tome što sada svaki čvor rešava uvek iste jednačine). Prva komanda koju master() funkcija šalje slave() funkcijama je komanda za računanje norme matrice sistema i potom master funkcija sama računa normu za deo matrice koji se nalazi u memoriji lokalnog čvora. Ukoliko je norma matrice zadovoljavajuća master() funkcija ulazi u iterativnu petlju koja će biti prekinuta i ukoliko je postignuta zadovoljavajuća tačnost ili ukoliko je postignut maksimalan broj iteracija. Posle svake iteracije master() funkcija poziva funkciju koja prikuplja izračunate vrednosti sa ostalih čvorova i vrši proveru da li je

[1] Matrica je pravougana jer se i matrica sistema i matrica slobodnih vektora sada čuvaju u jednoj matrici, pre svega zbog pojednostavljivanja komunikacije

dostignuta željena tačnost. Ukoliko je tačnost dostignuta ili je iscrpljen maksimalni broj iteracija program završava sa radom, i master() funkcija šalje terminirajuću komandu slave() funkcijama.

4.6.2 Analiza i zaključci

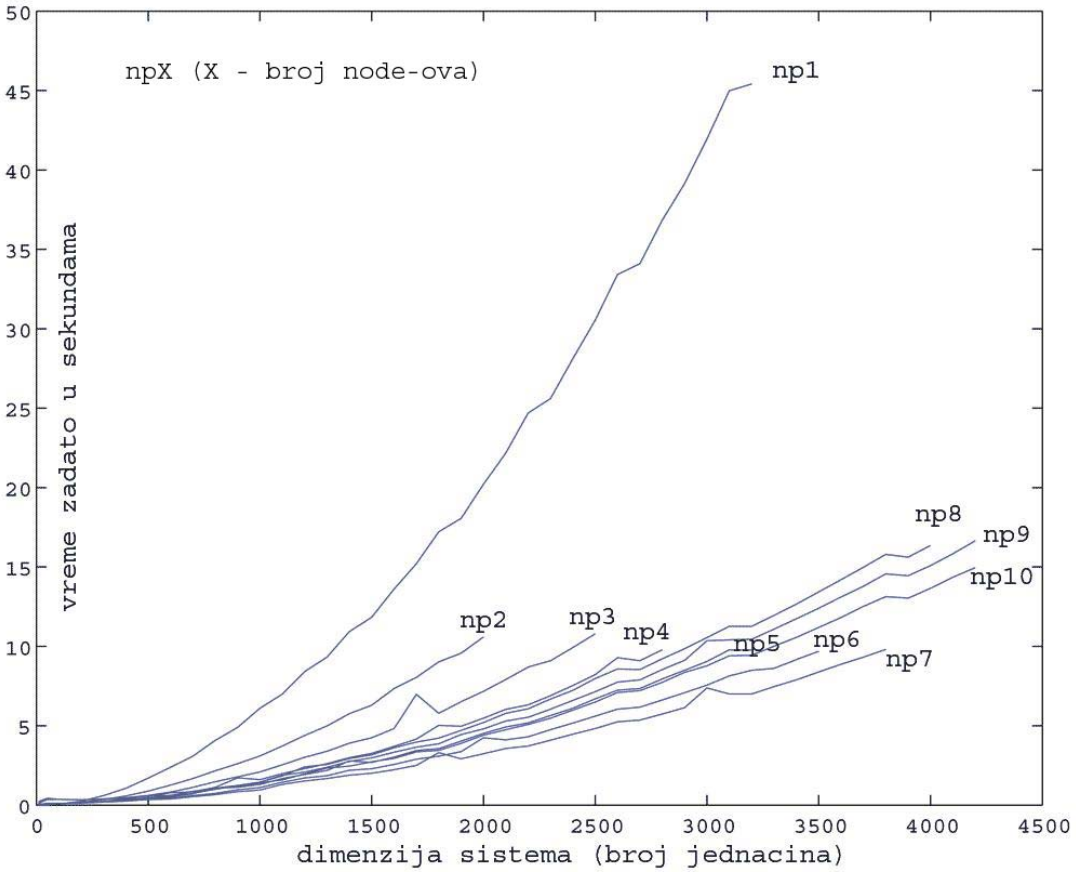
Merenja su vršena na slučajno generisanim dijagonalno-dominantnim matricama. Za svako merenje rešavana su 3 sistema jednačina koja su imala istu matricu koeficijenata A, dok su im slobodni vektori bili različiti slučajno generisani vektori. Dimenzija sistema varirala je izmedju 100x100 i 4200x4200 a takodje su prikazani rezultati za neke manje sisteme (10x10,20x20 i 50x50). Zbog malo raspoložive memorije na node-ovima kao osnovni tip podataka korišćen je tip float koji zauzima 4 bajta za jedan broj. Pored dimenzije sistema data je i količina memorije potrebna da se matrica A zapamti. Vremena prikazana u tabeli odnose se isključivo na metodu[2]

dim \ np	kb	1	2	3	4	5	6	7	8	9	10
10	0.39	0.00043	0.02254	0.02868	0.03852	0.04383	0.05008	0.04566	0.06694	0.10766	0.21196
20	1.5	0.00239	0.03507	0.04811	0.06122	0.06690	0.06151	0.07121	0.08254	0.18961	0.27365
50	9.7	0.01430	0.05407	0.06667	0.08919	0.09266	0.08291	0.08796	0.10804	0.36536	0.43651
100	39	0.06203	0.08671	0.09185	0.10810	0.11234	0.11267	0.10964	0.11742	0.36567	0.36633
200	156	0.24442	0.18129	0.15964	0.15230	0.14376	0.14060	0.13889	0.14852	0.33348	0.32084
300	351	0.59401	0.35269	0.24713	0.22260	0.23347	0.19733	0.19087	0.22150	0.37762	0.39453
400	625	1.06420	0.57582	0.39560	0.34425	0.32186	0.28387	0.26477	0.27995	0.45655	0.46536
500	976	1.70290	0.88592	0.61394	0.47277	0.42269	0.37891	0.36388	0.44463	0.56421	0.61307
600	1406	2.38886	1.25787	0.82607	0.61860	0.51319	0.43409	0.39941	0.56452	0.78129	0.61508
700	1914	3.08588	1.67883	1.11599	0.84761	0.71062	0.59576	0.53539	0.74458	0.83794	0.86613
800	2500	4.07652	2.16407	1.46758	1.11413	1.08456	0.74481	0.67425	1.01220	1.06948	1.06072
900	3164	4.89060	2.61151	1.78580	1.72468	1.13326	0.96927	0.84161	1.20451	1.23206	1.20504
1000	3906	6.09867	3.09167	2.08422	1.59788	1.31025	1.11076	0.95408	1.44458	1.45452	1.40185
1100	4726	6.98592	3.71629	2.54426	1.97920	1.67051	1.44325	1.31063	1.84120	1.88236	1.58977
1200	5625	8.40791	4.38715	3.01005	2.30137	1.93650	1.70884	1.52225	2.40541	2.07192	1.97473
1300	6601	9.31953	4.98430	3.38867	2.61162	2.18280	1.85518	1.67452	2.56051	2.38313	2.33549
1400	7656	10.9291	5.75910	3.90372	2.98722	2.78851	2.20053	1.88284	2.94460	2.75067	2.45520
1500	8789	11.8309	6.28915	4.24541	3.26352	2.68808	2.28959	2.00211	3.16036	2.97592	2.72751
1600	10000	13.6012	7.33886	4.81801	3.69412	3.02924	2.56616	2.22971	3.61419	3.33225	2.95174
1700	11289	15.2011	8.04369	6.97591	4.15512	3.43581	2.89408	2.50445	3.97436	3.65599	3.35926
1800	12656	17.2144	9.02297	5.78179	5.01779	3.55279	3.07840	3.30327	4.21776	3.86929	3.45384
1900	14101	18.0526	9.56333	6.51192	4.96717	4.02518	3.36911	2.92266	4.70070	4.44169	3.88619
2000	15625	20.2054	10.5833	7.16186	5.47540	4.51156	4.24411	3.23433	5.20309	4.79870	4.41065
2100	17226	22.1732		7.91710	6.03285	4.91134	4.11077	3.56320	5.78578	5.29655	4.74782
2200	18906	24.6957		8.69101	6.31882	5.17484	4.29655	3.71812	6.06193	5.53004	5.08690
2300	20664	25.6010		9.09636	6.89405	5.63558	4.74916	4.08793	6.67742	6.03985	5.47519
2400	22500	28.1166		9.92197	7.53332	6.10184	5.16511	4.46859	7.22147	6.59209	5.97947
2500	24414	30.5407		10.7923	8.21988	6.69392	5.60056	4.82366	7.96807	7.13883	6.48916
2600	26406	33.4097			9.28157	7.23682	6.04017	5.25430	8.57260	7.74946	7.08880
2700	28476	34.0950			9.09529	7.34110	6.17653	5.35062	8.53706	7.88724	7.20798
2800	30625	36.8388			9.77569	7.95188	6.62661	5.73406	9.21629	8.53635	7.73177
2900	32851	39.1473				8.48562	7.08239	6.13255	9.85746	9.12278	8.36205
3000	35156	41.9419				9.05069	7.55645	7.37291	10.5536	10.3458	8.77325
3100	37539	44.9883				9.77699	8.13847	7.00550	11.2556	10.4117	9.40242
3200	40000	45.4148				9.77324	8.48918	7.00589	11.2614	10.4449	9.43336
3300	42539	48.5909					8.61437	7.46057	11.9468	11.0806	10.0096
3400	45156	51.9339					9.16362	7.88499	12.6613	11.7292	10.5597
3500	47851	54.8340					9.69141	8.38363	13.4209	12.3983	11.1835
3600	50625	57.9959						8.87363	14.1964	13.1096	11.8108
3700	53476	61.6245						9.32333	14.9715	13.7773	12.5162
3800	56406	65.1677						9.80422	15.7979	14.5576	13.1337
3900	59414	67.1889							15.6127	14.4428	13.0389
4000	62500	68.4166							16.3484	15.0787	13.6504
4100	65664	71.7105								15.8267	14.3545
4200	68906	75.6071								16.6479	14.9538

Primećuje se da su neka polja u tabeli rezultata prazna. Radi se o slučajevima kada sistem nije moguće rešiti na zadatom broju node-ova zbog nedostatka memorije pri komunikaciji. Na primer sistem dimenzije 2100x2100 zauzima oko 17 MB memorije, kada je potrebno sistem rešiti korišćenjem dva računara tada se komunikacionim kanlom treba prebaciti oko 9 MB memorije. Sa sistemom na je radjeno taj transfer nije bio moguć [3]. Medjutim ukoliko se sistem rešava korišćenjem 3 čvora tada treba poslati po 6 MB svakom čvoru što je bilo izvodljivo.

[2]Pri pokretanju programa na više čvorova potrebno je neko konstantno vreme za podizanje pojedinačne instance tog programa na svakom čvoru, zatim potrebno je vreme da se podaci iz učitane ili generisane matrice proslede čvorovima, koje je takodje konstanta u odnosu na broj iteracija (ova dva vremena nisu uzimana u obzir)

[3]Zbog količine raspoložive memorije u čvorovima



5 Analiza i zaključci

Analizom dobijenih rezultata, kao i analizom rezultata prethodne metode može se doći do sledećih zaključaka:

- kod sistema malih dimenzija (čak i do 100x100) paralelizacijom se dobija drastično usporenje (zbog velike količine podataka koje treba prebacit izmedju čvorova u odnosu na broj računskih operacija koje svaki čvor uradi na svom paketu podataka). Za direktnu - Kramerovu metodu taj prag je dosta niži (već se za sisteme dimenzija većih od 5x5 dobija ubrzanje)

- kod sistema dimenzija većih od 1000x1000 komunikacija postaje zanemarljiva u odnosu na vreme potrebno za izračunavanje na svakom čvoru, tj. najoptimalniji rezultati ove metode se dobijaju na sistemima dimezija većih ili jednakih 1000x1000. Kao i u prethodnom zaključku kod Kramerovog pravila granica se pomera dosta niže.

- kada se u proračun ubaci osimi čvor što je moguće iako fizički postoji samo sedam, ali tada će jedan čvor biti proces koji će se izvršavati na računaru koji već obradjuje jednu količinu jednačina, dobija se drastično usporenje. Usporenje je posledica toga da svi čvorovi sadrže isti broj jednačina i da je jednom od njih potrebno duplo više vremena da reši 2 puta više jednačina nego ostali čvorovi, pa ga ostali čvorovi moraju čekati kako bi iteracija bila kompletirana. Sličan problem (za ovako paralelizovan metod) javio bi se kada bi čvorovi u klaster računaru bili drastično različitih performansi, (tj tada bi se sistem prilagodio računaru najlošijih peformansi). Rešavanje ovog problema moglo bi se izbeći detaljnijom analizom (o čemu

je bilo reči pri paralelizaciji na čvrsto vezanim procesorskim sistemima). Ovo odstupanje se najjasnije vidi na grafiku kod Kramerove metode (gde se kod svih slučajeva) vidi oštar skok kada se proračun izvodi na 8 čvorova

- daljim povećavanjem broja čvorova vreme rešavanja se stabilizuje, jer je podela poslova nešto ravnomernija
- iz slučaja 3900x3900 jednačina primećuje se da korišcenje više čvorova nego što ima raspoloživih računara ima smisla, jer se tada 60 MB podataka izdeli na manje porcije koje mogu nesmetano doći do čvorova.
- prethodni zaključak iskorišćen je za rešavanje većih sistema. Na primer sistem 5400x5400 (113 MB) moguće je rešiti tek na 14 čvorova (u datoj konfiguraciji) za šta je potrebno 17.02 sekundi, dok je za rešavanje istog sistema na jednom čvoru potrebno čak 358.18 sekundi. Za još veći sistem 10000x10000 jednačina koji je zauzimao preko 390 MB i čije rešavanje nije ni moguće na jednom računaru (koji ima 128 MB RAM memorije)[4] bilo je potrebno instancirati 63 čvorova koji su zajedno rešili sistem za 55.04 sekunde.
- za Rešavanje velikih (gustih) sistema jednačina u prihvatljivom vremenu je pored brzih računara pre svega potrebno dosta memorije. Na primer za sistem od 30000 jednačina (kada bi se koristio double tip podataka) zahtevao bi $30000^2 \cdot 8(bajtova) \cdot \frac{1}{1024}(kB) \cdot \frac{1}{1000}(MB) \cdot \frac{1}{1000}(GB) = 7GB$ što računar sa 32-obitnom arhitekturom ne može da adresira (jer je ograničenje $2^{32} = 4GB$) ali mnogi praktični problemi koriste retke (*sparse*) matrice čime je problem memorije a i transporta dosta pojednostavljen.

5.1 Primene

Višeprocesorski sistemi u različitim oblicima već se duže vreme koriste u praksi, pre svega za rešavanje raznih numeričkih problema, medjutim i druge grane računarstva koriste višeprocesorske sisteme. To su na primer: velike baze podataka od kojih se zahteva izrazito brz odgovor (npr pretraživači poput google-a), Programi za koje je vrlo bitno da ne prestanu sa radom (jer je verovatnoća da će svi računari u mreži otkazati istovremeno vrlo mala - osim ako se ne radi o nestanku napajanja), Gde su podaci duplirano smešteni na više računara, te pri padu jednog računara posao preuzima drugi, zatim kod sistema za rendering 3d slike i pravljenje animiranih filmova, itd ...

U zavisnosti od obima i kompleksnosti posla koje jedan procesor treba da obavi brzina mreže distribuiranog sistema može ali i ne mora igrati bitnu ulogu. Ukoliko je za deo posla potrebno odvojiti više minuta, pa i nekoliko sati a ulazni podaci nisu preveliki tada se može koristiti i globalna mreža pa čak i računari koji su na istu privezani modemima. Jedan od poznatijih projekata koji koristi ovaj način rada je **SETI** projekat (*Search for Extraterrestrial Intelligence*), koji analizom podataka dobijenih skeniranjem kosmosa najvećim radio teleskopom na svetu (Arecibo teleskop u Porto Riku) pokušava da otkrije signale postojanja van zemaljske inteligencije. Da bi podaci bili obradjeni potrebno učešće moćnih mainframe računara, medjutim UC Berkeley SETI tim je razvio sistem **seti@HOME** koji umesto mainframe računara koristi klijentske računare na univerzitetu koji većinu vremena nisu iskorišćeni, takodje svaki računar koji je permanentno ili povremeno povezan na internet može biti deo velike mašinerije koja obradjuje podatke. Nakon povezivanja na mrežu program preuzima podatke iz baze neobradjenih podataka i kada je računar u tzv *idle time* modu program izvršava analizu signala, što se korisniku može manifestovati kao screen saver, ili se pak program može stalno izvršavati na računaru pri čemu mu treba spustiti prioritet kako bi računar mogao da se koristi [5]. Klijentski program i više informacija mogu se pronaći na adresi *http://setiathome.berkeley.edu/*

[4] Izračunavanje nije moguće bez upotrebe memorije na disku, što višestruko usporava proces

[5] Dok pišem ovaj tekst na mom računaru se obradjuju sledeći podaci (Sky coordinates: 29.431 R.A., 23.050 Dec).

Još jedan (nešto prizemniji projekat) pokrenuo je za sada najpoznatiji svetski pretraživač google. Projekat je nazvan *Google Compute*. Ideja je da google-ov server bude most izmedju klijentskih računara koji bi vršili razna izračunavanja pažljivo izabranih naučnih projekata za koja su ta izračunavanja potrebna. Za uzvrat klijent na svom računaru dobija google toolbar koji mu može olakšati pretragu interneta i koji će u pozadini izvršavati potrebne proračune. Jedan od prvih projekata u planu je **Folding@home** koji predstavlja ne-profitno akademsko istraživanje na Stanford Univerzitetu sa ciljem da se prouči struktura proteina kako bi se bolje odredjivali tretmani za razne bolesti.

6 Literatura i software

6.1 Literatura

- Dr. Desanka Radunović, Numeričke Metode, Beograd, 1998.
- Desanka Radunović, Aleksandar Samardžić, Numeričke metode - zbirka, Beograd, 2002.
- Miodrag Živković, Algoritmi. Matematički fakultet, Beograd, 2000.
- Nenad Mitić, Osnovi računarskih sistema. Matematički fakultet, Beograd, 2002.
- C++ Izvornik, Stanley B. Lippman, Addison-Wesley, 2000.
- Posix threads Programming
- Yukiya Aoyama, Jun Nakano, Practical MPI Programming, IBM www.redbooks.ibm.com

6.2 Internet lokacije

- Lokacija na kojoj se može naći ovaj rad: `http://www.jwork.net/programiranje/jacobi`
- Matematički fakultet - Beograd: `http://www.matf.bg.ac.yu/`
- mpi forum: `http://www.mpi-forum.org`
- GeoWulf: `http://sokocalo.engr.ucdavis.edu/~jeremic/GeoWulf/`
- SETI projekat: `http://setiathome.berkeley.edu/`

6.3 Software

- gcc (GCC) 3.3 20030226 (prerelease) (SuSE Linux)
- MPICH
- LaTeX(Web2C 7.4.5)
- MatLab 6.5

Zahvaljujem se prof. Borisu Jeremiću (univerzited Davis - Kalifornija) za pružanje računarskih resursa potrebnih da napišem ovaj rad.