Oblikovanje i analiza algoritama
3. predavanje

Saša Singer

singer@math.hr

web.math.pmf.unizg.hr/~singer

PMF – Matematički odsjek, Zagreb

Sadržaj predavanja

- Složenost u praksi — eksperimenti:
  - Uvod u eksperimente.
  - Mjerenje vremena i planiranje eksperimenata.
  - Parcijalne sume reda za $\ln 2$.
  - Ubrzanje algoritma za parcijalne sume alternirajućeg reda.
  - Zbrajanje matrica reda n.
  - Kratka priča o cache memoriji.
  - Množenje matrica reda n.
  - Blokovsko množenje matrica reda n.

Informacije — web stranica

Moja web stranica za Oblikovanje i analizu algoritama je

https://web.math.pmf.unizg.hr/~singer/oaa/

ili, skraćeno

https://web.math.hr/~singer/oaa/

Kopija je na adresi

http://degiorgi.math.hr/~singer/oaa/

Službena web stranica za Oblikovanje i analizu algoritama je

https://web.math.pmf.unizg.hr/nastava/oaa/

Uvod u eksperimente

Sadržaj — algoritmi, složenost, poboljšanja

Model složenosti i eksperimentalna provjera modela na

- tri jednostavna (iterativna) algoritma sa složenostima različitih redova veličina:
  - Parcijalne sume reda za $\ln 2$ — složenost $\Theta(n)$;
  - Zbrajanje matrica reda n — složenost $\Theta(n^2)$;
  - Množenje matrica reda n — složenost $\Theta(n^3)$.

Uz analizu rezultata eksperimenta, napravit ćemo još i

- ubrazanje izloženih algoritama (kad ide):
  - Brzo i točno računanje sume za $\ln 2$;
  - Hijerarhijska građa memorije i pristup podacima;
  - Cache memorija i uloga;
  - Blok–algoritmi za iskorištavanje cachea;

Model složenosti

U svakom od tri algoritma koje ćemo napraviti

- elementarne operacije su osnovne aritmetičke operacije na realnim brojevima (tzv. “floating–point” operacije, IEEE tip double).

Ključna pretpostavka za model složenosti je:

- trajanje tih operacija ne ovisi o operandima.

Dakle, složenost ovih algoritama mjerimo

- brojem floating–point operacija.

Napomena. Gornja pretpostavka ne znači da je trajanje floating–point operacija konstantno.

- Upravo to ćemo testirati!

Model složenosti (nastavak)

Prvi korak — brojanje.

Jednostavnim prebrojavanjem ovih operacija dolazimo do

- ukupnog broja floating–point operacija,
- u funkciji odabranog parametra n, koji mjeri veličinu problema.

Oznaka:

$$F(n) = \text{neka funkcija od } n.$$

Slovo F asocira na "Flops", što je kratica za "floating–point operations" (ali ne "per second").

Dakle, samo broj operacija, a ne brzina izvođenja.

Model složenosti (nastavak)

Drugi korak — brzina.

Kad bi brzina c izvođenja floating–point operacija bila konstantna,

- recimo, $c = 3\,\text{Gigaflopsa}$, tj. $3 \cdot 10^9$ floating–point operacija u sekundi,

onda bi vremenska složenost $T(n)$ bila

$$T(n) = \frac{F(n)}{c},$$

iz $F(n) = c \cdot T(n)$ (broj operacija je brzina puta vrijeme).

A to lako možemo eksperimentalno provjeriti.

Model složenosti (nastavak)

Treći korak — mjerenje vremena i brzine.

Ako programskom "štopericom"

- izmjerimo vremensku složenost $T(n)$,
- i to za razne vrijednosti od n,

onda brzinu izvođenja operacija c možemo odrediti kao funkciju od n, iz modela

$$c(n) = \frac{F(n)}{T(n)}.$$

Iz tablica ili grafova vrijednosti $c(n)$ lako je provjeriti je li to konstantno ili ne.

Što testiramo?

Naš eksperiment se svodi na

- mjerenje brzine izvršavanja algoritma (odnosno, njegovih osnovnih operacija) u funkciji od n.

Pa da vidimo!

Napomena. Za male vrijednosti od n

- vrlo je teško dovoljno točno mjeriti vremena $T(n)$.

Rezolucija timera je, uglavnom, fiksna — i ima veliki korak. Tipične vrijednosti su $\geq 10^{-3}\,\mathrm{s}$, na pr. $\approx 0.06\,\mathrm{s}$.

- Razmislite kako biste algoritamski našli korak rezolucije!

Dakle, čitav eksperiment treba relativno pažljivo isplanirati, da bismo dobili iole razumne rezultate.

Kako testiramo (mjerimo vrijeme)?

Realizacija: za svaki fiksni n

- "isti" posao ponavljamo u petlji puno puta,
- a vrijeme mjerimo jednom, za cijelu petlju.

To, naravno, nije isto kao bez ponavljanja, ali je jedino praktično izvedivo!

Neka je $N(n)$ izabrani broj ponavljanja posla, za dani n.

Brzinu $c(n)$ računamo iz

$$c(n) = \frac{N(n) \cdot F(n)}{T(N(n) \text{ puta ponovi posao za } n)}.$$

Što želimo dobiti?

Prije nego što “vidimo” rezultate na nekom konkretnom računalu,

- nije baš jasno da ćemo dobiti nešto “zanimljivo”.

Hoćemo, hoćemo — obećavam! Bit’ će zanimljivi i na jednom računalu.

Dakle, čemu još rezultati mogu poslužiti?

Ovako dobivene brzine $c(n)$, naravno,

- bitno ovise o brzini računala na kojem testiramo.

Zato je zgodno test napraviti na

- nekoliko raznih računala i usporediti rezultate.

Svrha eksperimenta (nastavak)

Osim za usporedbu računala, stvar može poslužiti i za

- usporedbu različitih programskih produkata, na pr. compilera, kao odgovor na pitanje:
  - koliko efikasan kôd generiraju, posebno s raznim opcijama?

Zato cijeli eksperiment radimo na

- nekoliko računala,
- a usput ćemo usporediti i dva FORTRAN compilera s različitim opcijama za optimizaciju.

“Zvjerinjak” (računala)

Napomena: cijeli eksperiment je proveden 2005–2006. godine.

“Stradalnici” u eksperimentu su (ukratko):

- Pentium 3 na 500 MHz, zvani Klamath5, rođen 1998., (originalno imao Pentium 2 na 333 MHz);
- Pentium 4 na 3.0 GHz, zvani Veliki, rođen 2003., (danas ima P4 na 3.2 GHz);
- Pentium 4/660 na 3.6 GHz, zvan(a) BabyBlue, rođen(a) 2005., “skurena” silnim eksperimentima;
- ovaj notebook s Pentium 4 Mobile na 2.2 GHz, rođen 2003.

Podrobnije “osobne iskaznice” slijede!

FORTRAN compileri i opcije

Zašto FORTRAN (i to 77)?

- Zato što je to, još uvijek, osnovni jezik za numeričko računanje (recimo, BLAS je originalno napisan u FORTRANu),
- i zato što imam tzv. MKL (Math Kernel Library) koji se zove iz FORTRANa.

FORTRAN compileri u testu:

- Compaq Visual Fortran, verzija 6.6C3 (konačna), oznaka CVF,
- Intel Visual Fortran, verzije razne, oznaka IVF.

FORTRAN compileri i opcije (nastavak)

Opcije za optimizaciju:

- puna optimizacija, skraćeno "fast":
  - `/optimize:5` za CVF,
  - `/fast` za IVF.
- normalna optimizacija, bez posebnih opcija, skraćeno "normal".

Parcijalne sume reda

Parcijalne sume reda za $\ln 2$

Problem: Zadan je prirodni broj $n \in \mathbb{N}$. Treba izračunati n-tu parcijalnu sumu alternirajućeg reda za $\ln 2$

$$\begin{aligned} S(n) &= \sum_{i=1}^{n} (-1)^{i-1} \frac{1}{i} \\ &= 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{(-1)^{n-1}}{n}. \end{aligned}$$

Ovu sumu računamo na dva načina:

- zbrajanjem unaprijed,
- zbrajanjem unatrag.

Zbrajanje unaprijed

```
      subroutine sumfwd (n, sum, cumsum)
c
c   Forward sum of ln2 alternat. series, n terms.
c
      implicit none
c
      integer n
      double precision sum, cumsum
c
      integer i, nn
c
c   I loop, inner
c
```

Zbrajanje unaprijed (nastavak)

```
      nn = n
      sum = 0.0d0
      do 10, i = 1, nn
         if (mod(i, 2) .eq. 1) then
            sum = sum + 1.0d0 / i
         else
            sum = sum - 1.0d0 / i
         end if
   10    continue
      cumsum = cumsum + sum
c
      return
      end
```

Zbrajanje unatrag, kumulativna suma

Potprogram dobivamo zamjenom petlje unaprijed:

```
      do 10, i = 1, nn
```

sljedećom petljom unatrag:

```
      do 10, i = nn, 1, -1
```

Sve ostalo (osim imena potprograma) ostaje isto.

Svrha kumulativne sume `cumsum` je “prijevara” optimizacije,

- zato da optimizator ne makne okolnu petlju za ponavljanje poziva.

Bez toga, svi pozivi daju isto $\Rightarrow$ invarijantni su na tu petlju!

Broj operacija

U svakom prolazu kroz petlju imamo dvije operacije

- dijeljenje s `i`,
- zbrajanje ili oduzimanje.

Tome treba dodati jedno zbrajanje na dnu, izvan petlje, za tzv. kumulativnu sumu.

Dakle, ukupan broj operacija u oba algoritma je:

$$F(n) = 2n + 1.$$

Broj ponavljanja je $N(n) = 10^9/n$, tako da dobijemo približno konstantno trajanje "vanjske" petlje kojoj mjerimo vrijeme (one s ponavljanjem poziva).

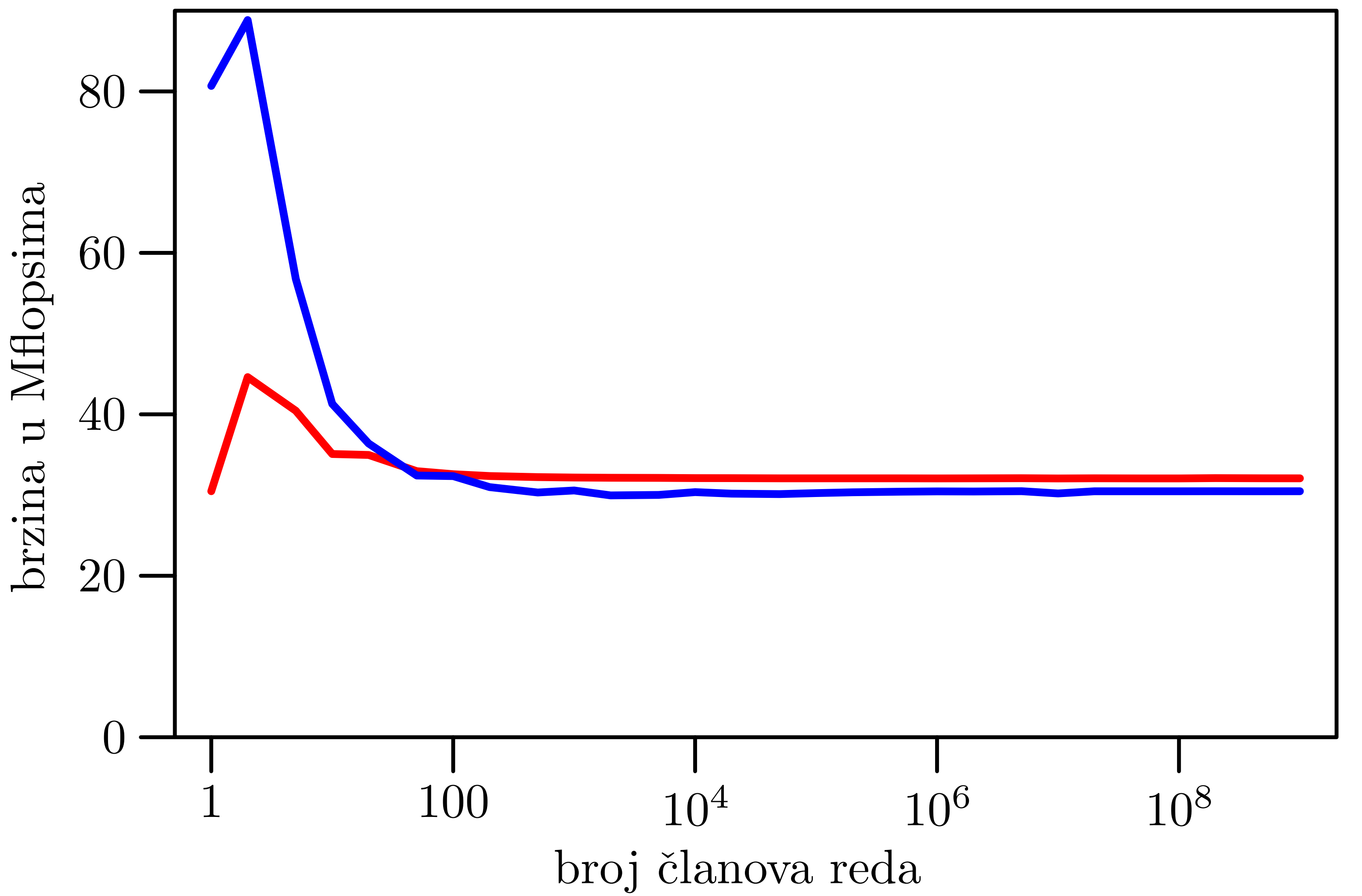
Boje na grafovima

Legenda za čitanje grafova:

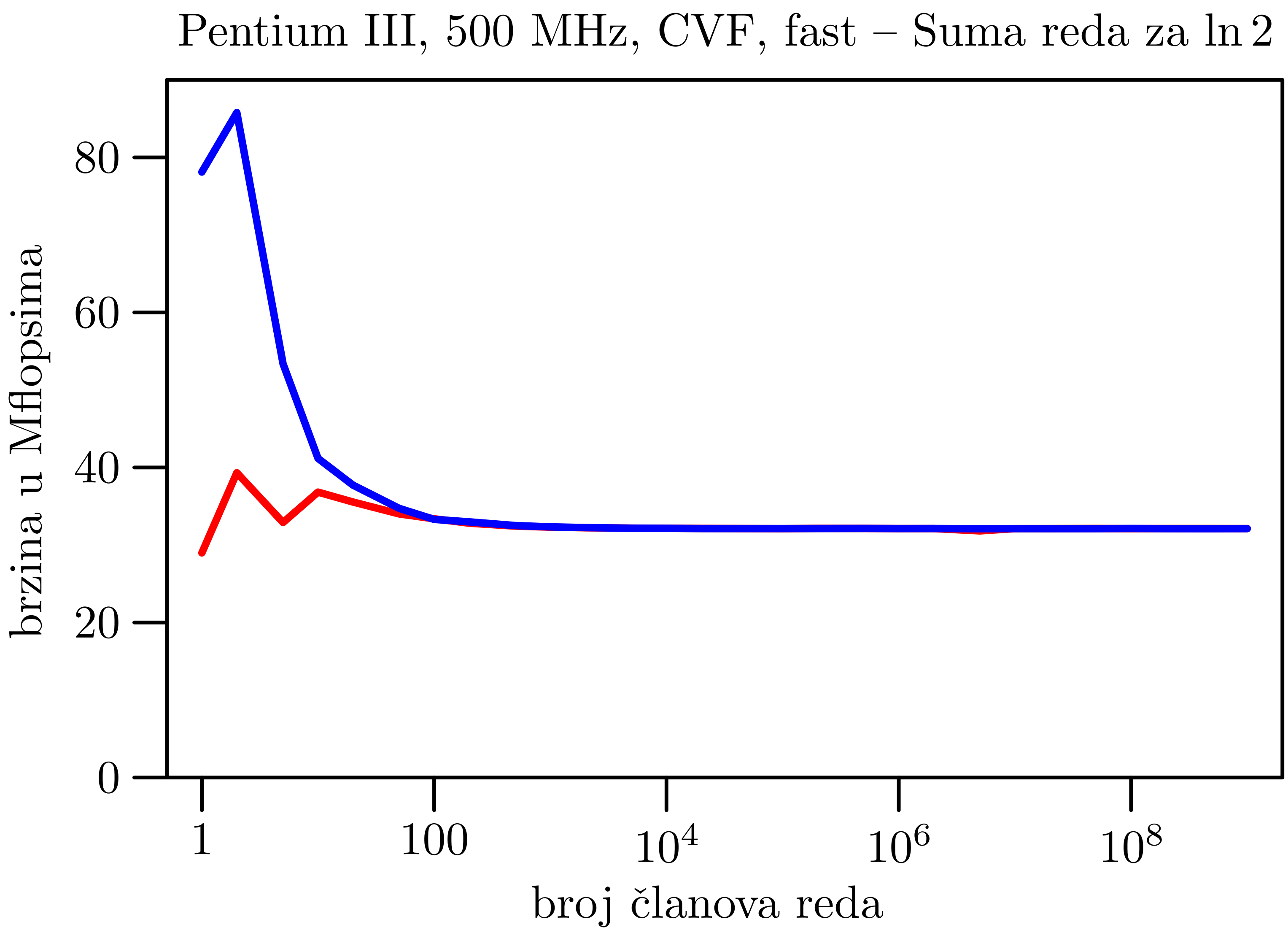
- zbrajanje unaprijed — plavo, brže,
- zbrajanje unazad — crveno, sporije.

Klamath5, CVF, normal — unaprijed, unatrag

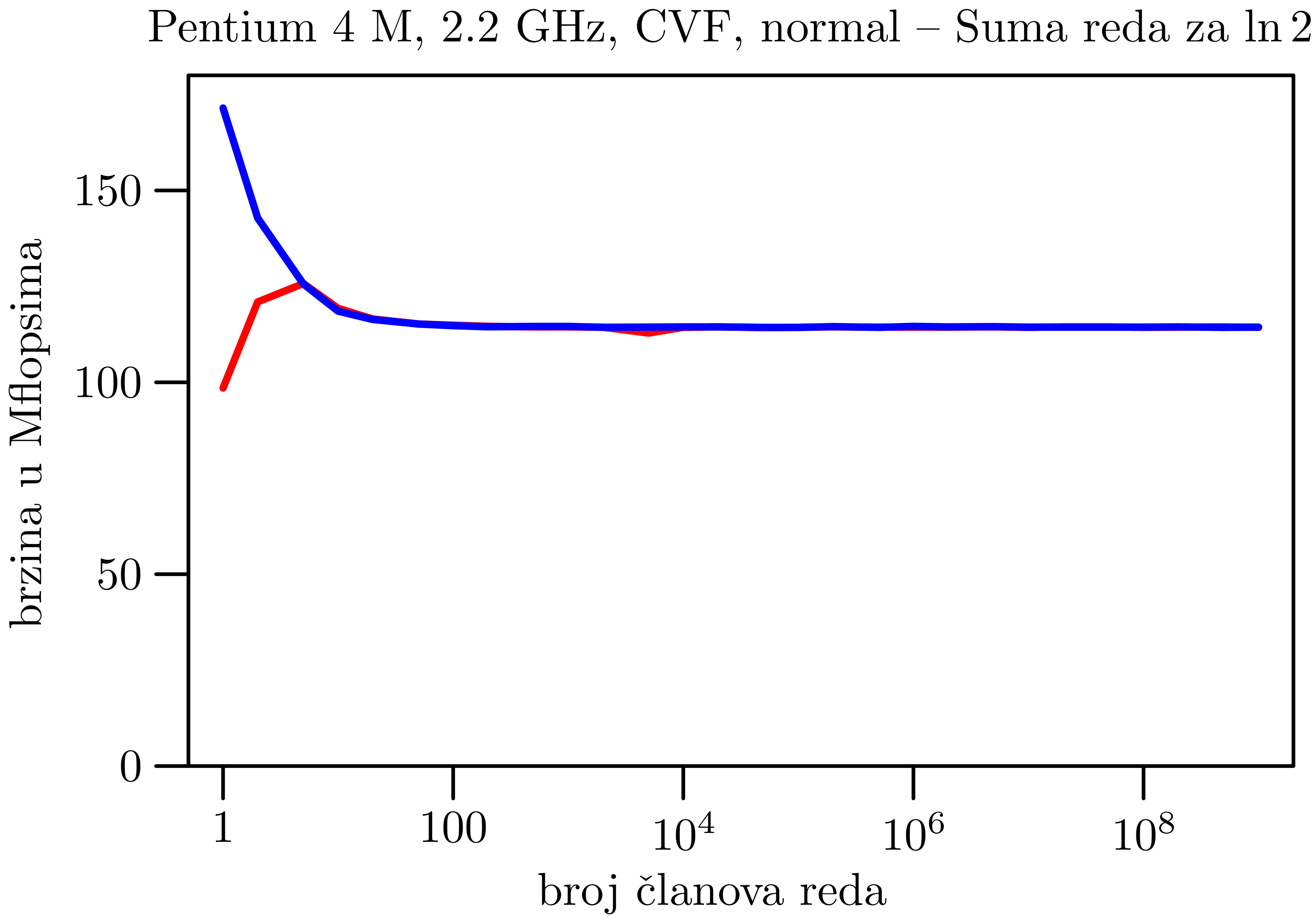
Pentium III, 500 MHz, CVF, normal – Suma reda za $\ln 2$



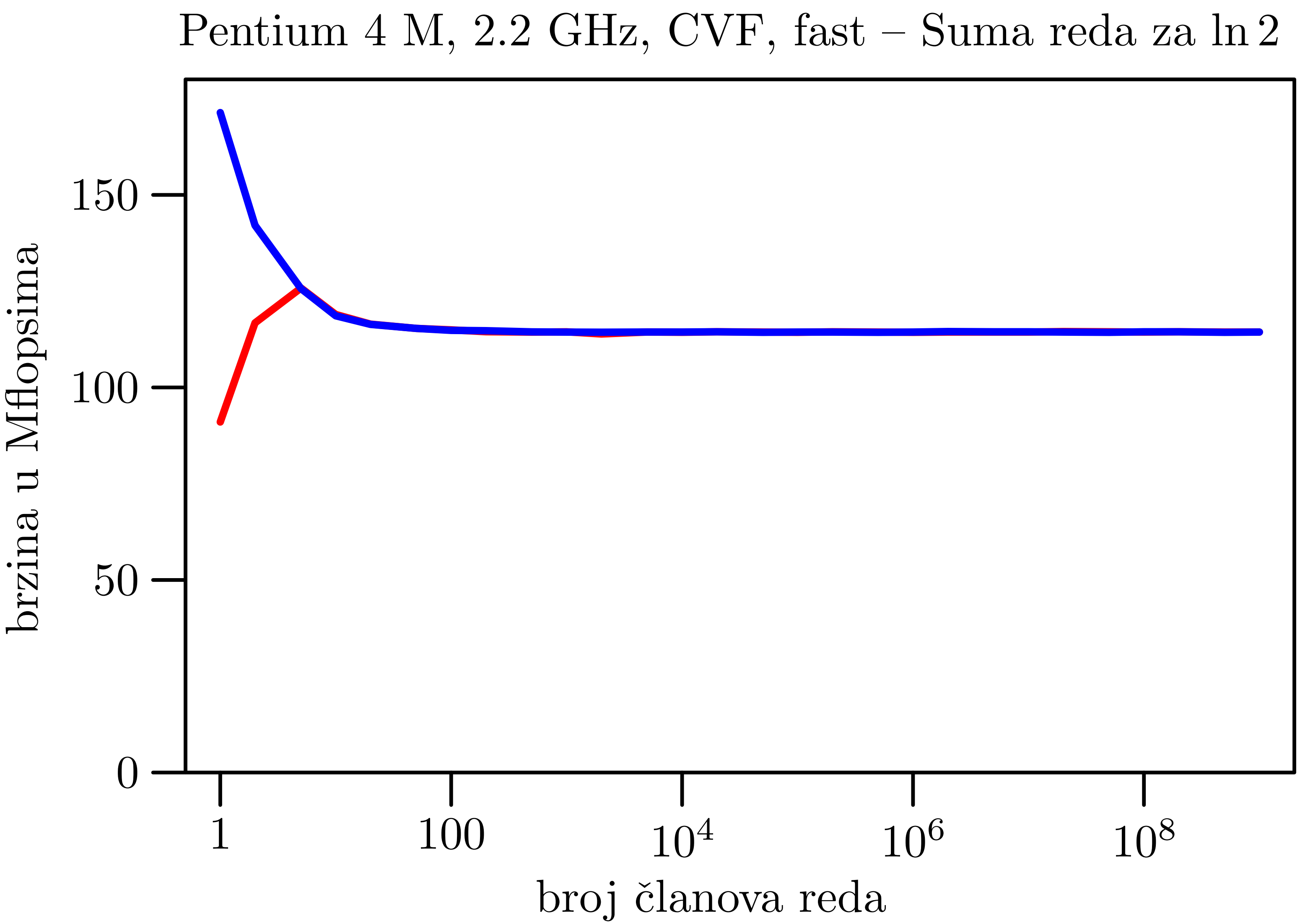
Klamath5, CVF, fast — unaprijed, unatrag



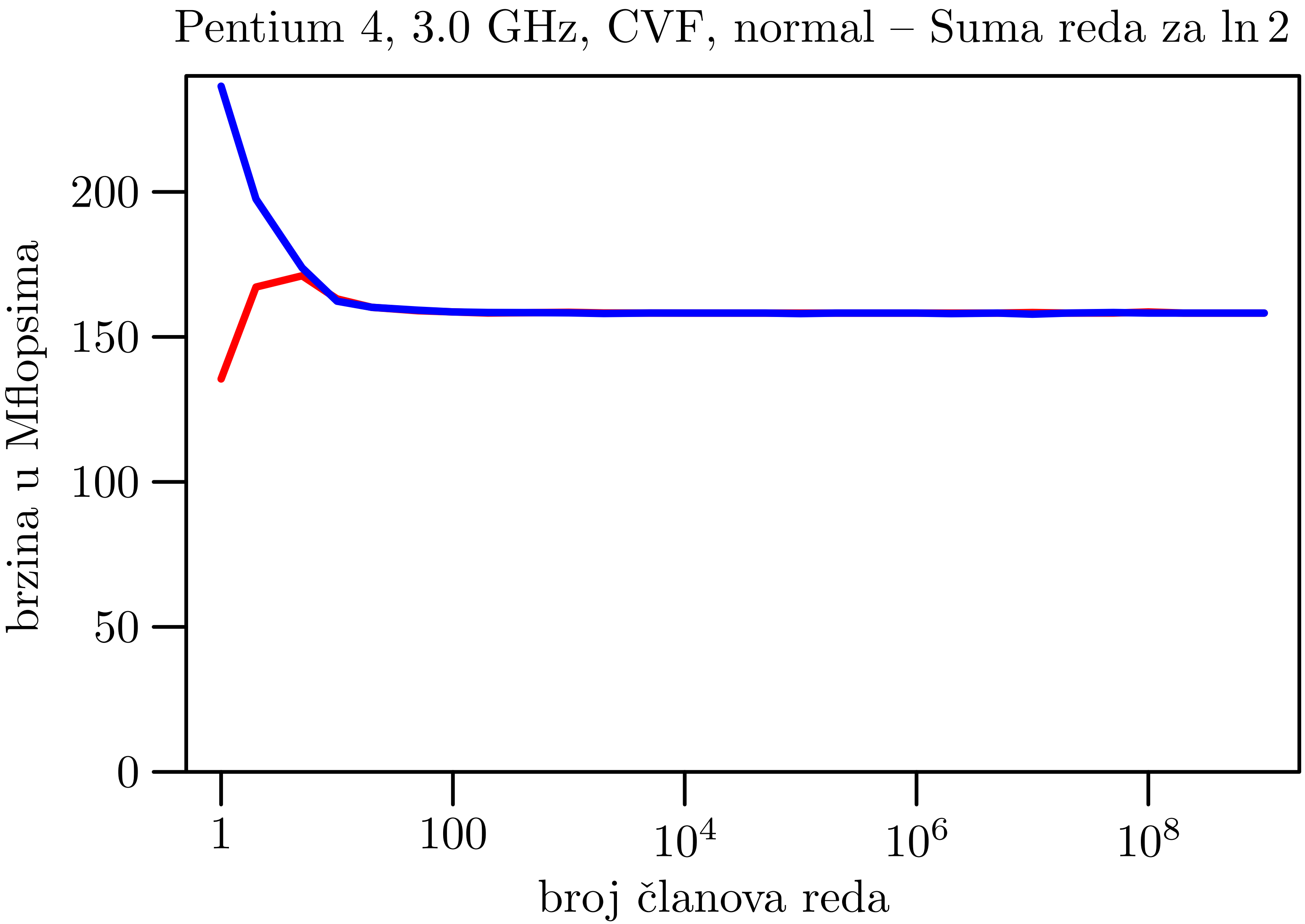
Notebook, CVF, normal — unaprijed, unatrag



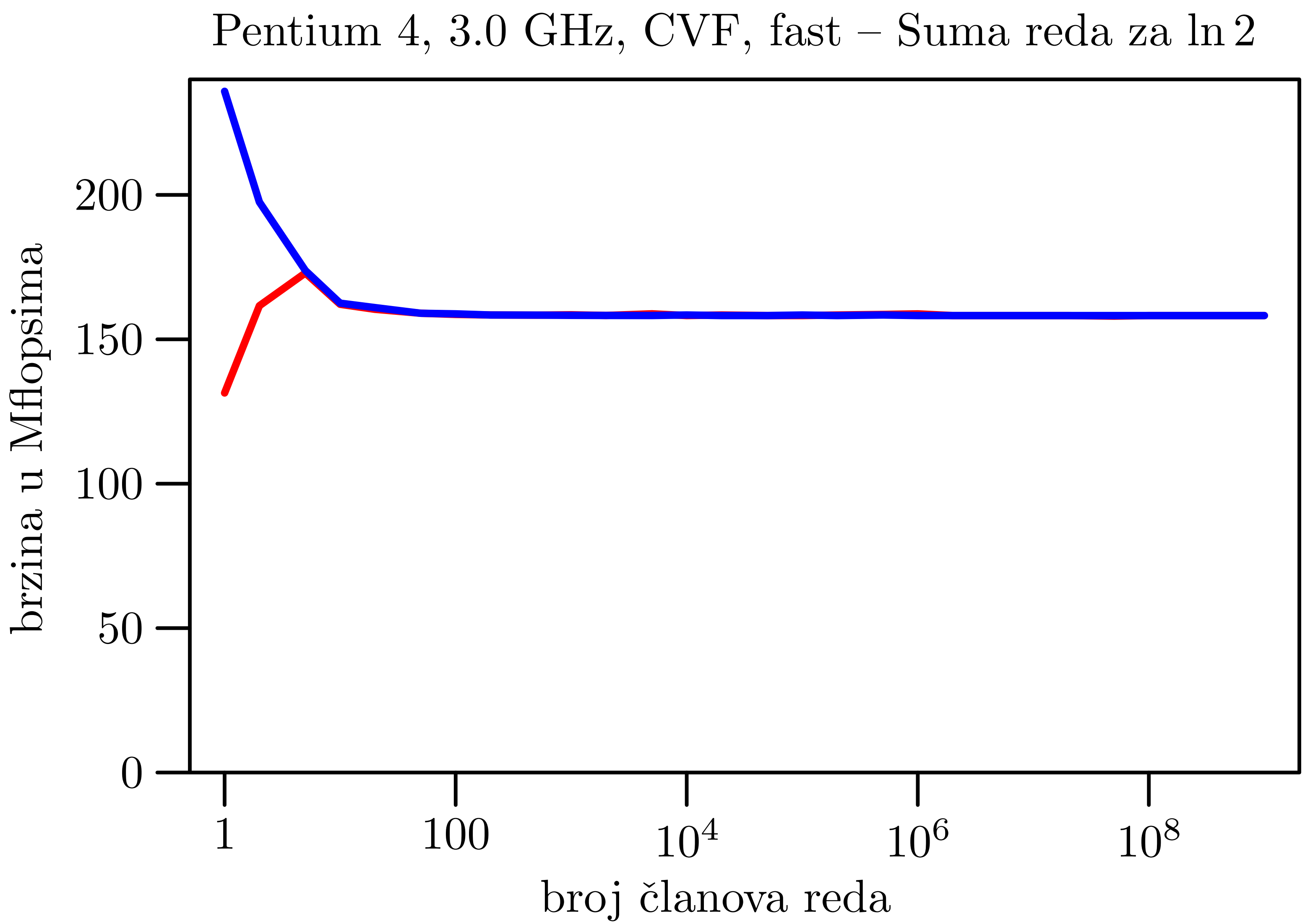
Notebook, CVF, fast — unaprijed, unatrag



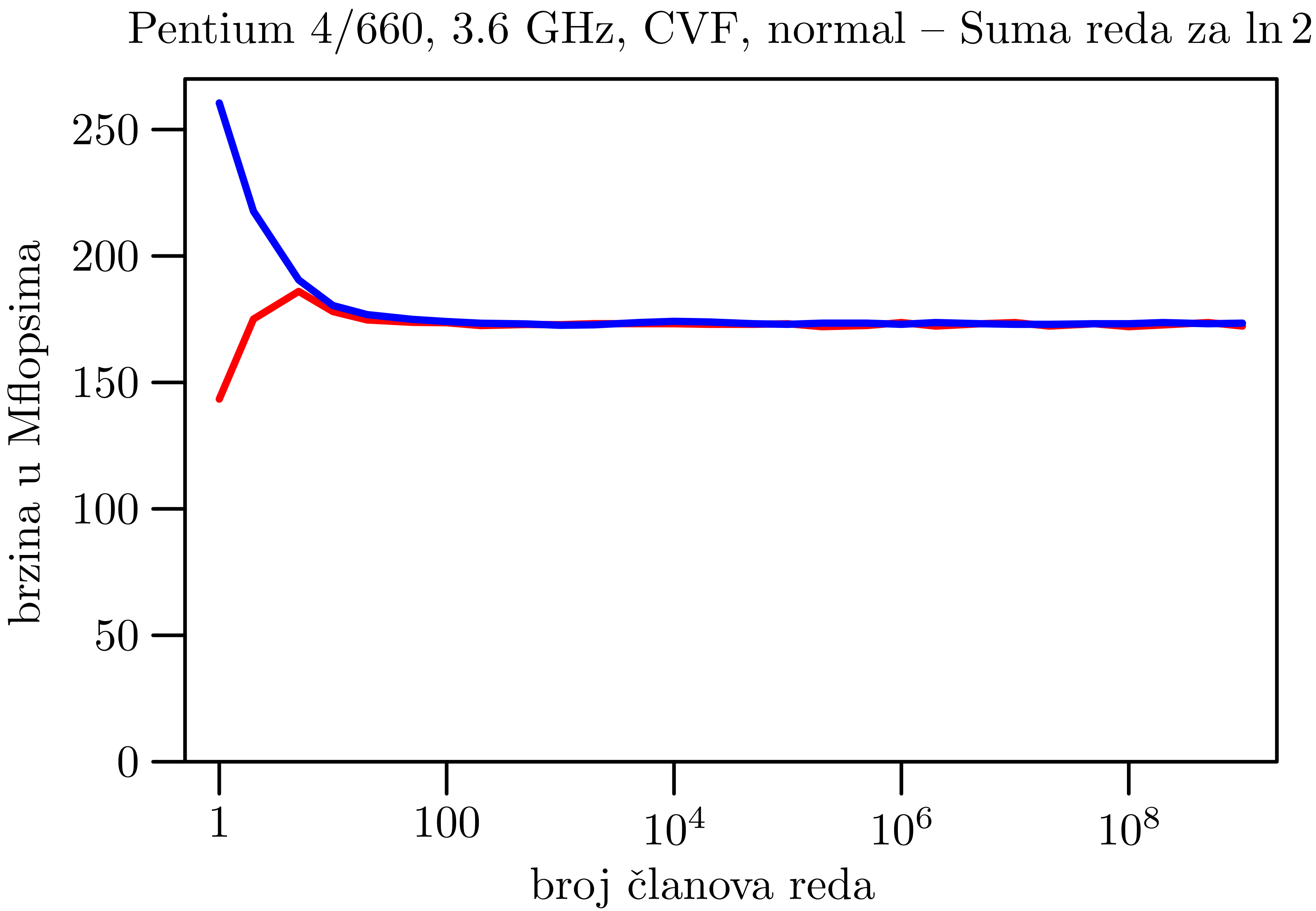
Veliki, CVF, normal — unaprijed, unatrag



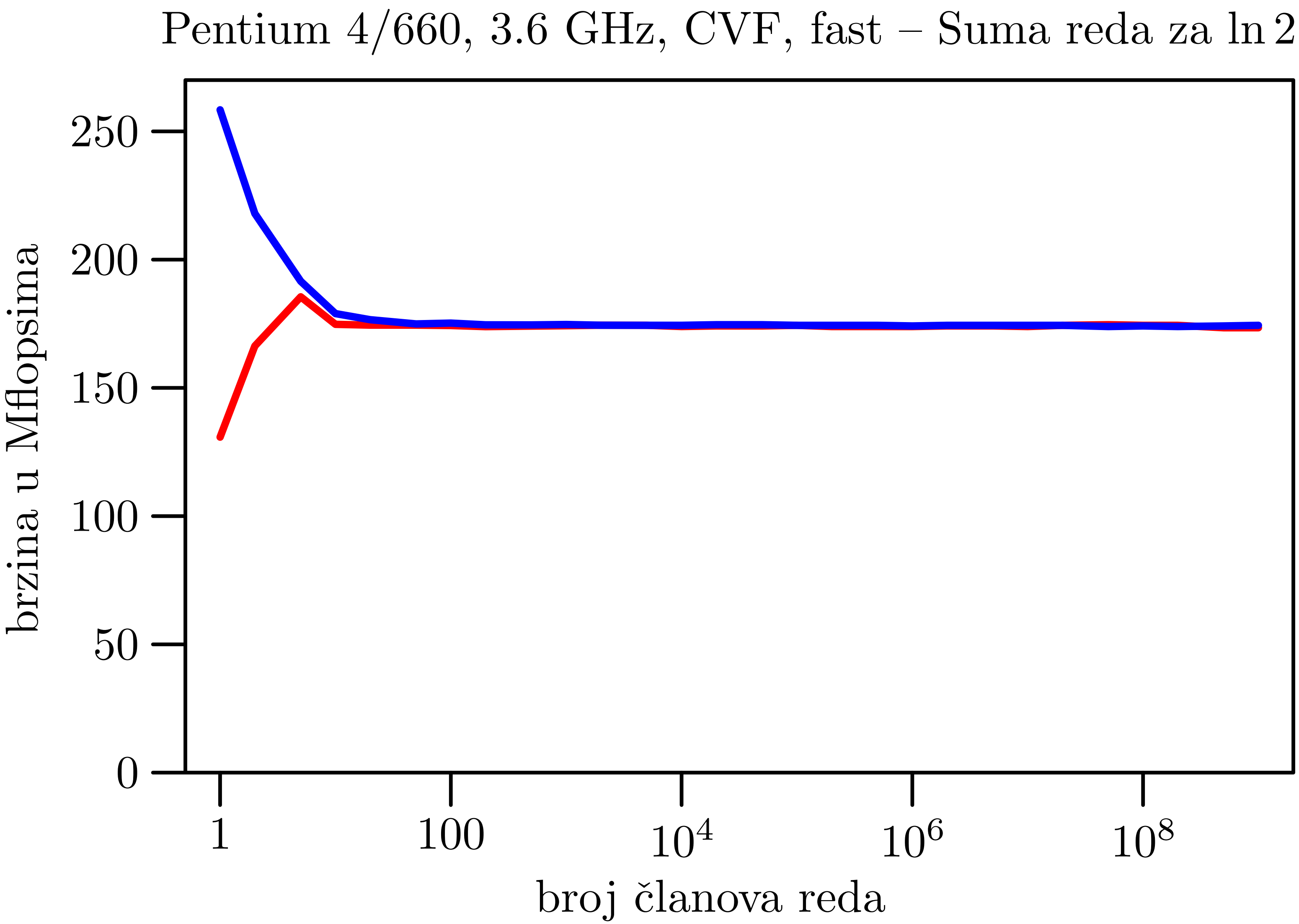
Veliki, CVF, fast — unaprijed, unatrag



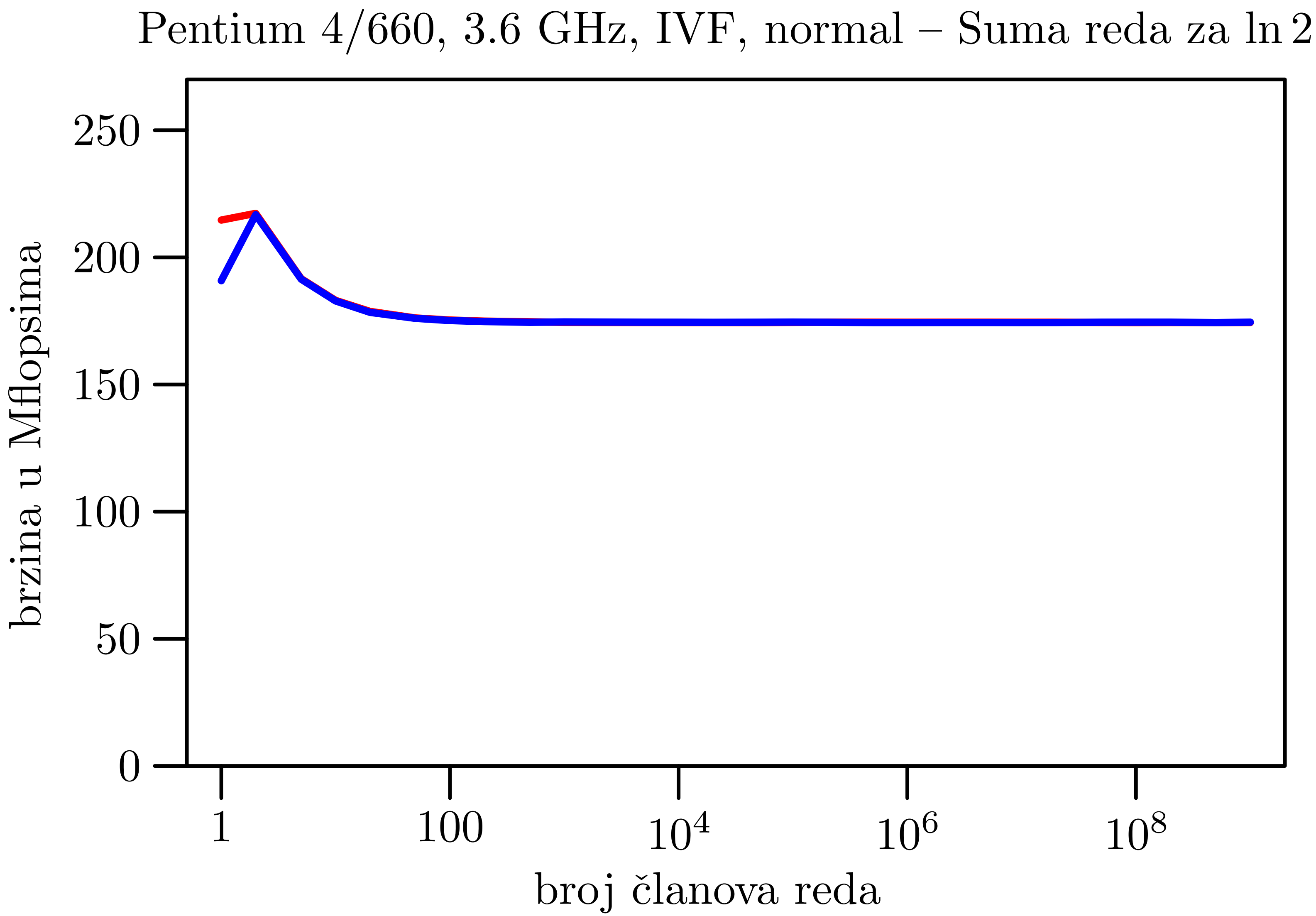
BabyBlue, CVF, normal — unaprijed, unatrag



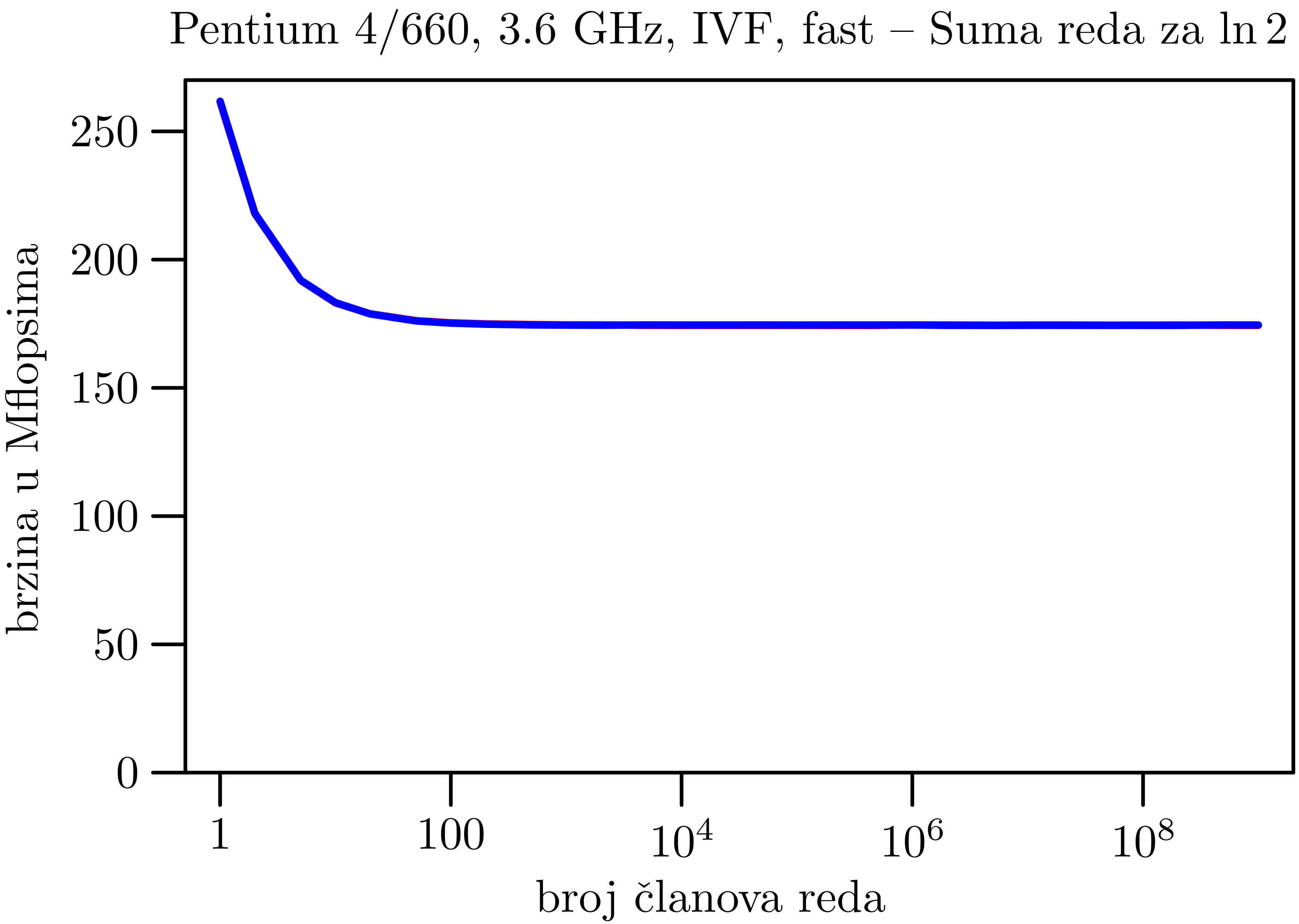
BabyBlue, CVF, fast — unaprijed, unatrag



BabyBlue, IVF, normal — unaprijed, unatrag



BabyBlue, IVF, fast — unaprijed, unatrag



Tablica brzina za velike n

Vidimo da nema neke razlike u brzinama između:

- zbrajanja unaprijed i unazad,
- fast i normal opcija compilera.

Usporedba brzina (u Mflops) za razna računala:

Računalo	Brzina
Klamath5	32.1
Notebook	114.4
Veliki	158.2
BabyBlue	174.4

Brzina raste sporije od frekvencije!

Komentar rezultata

Podatke o brzinama treba gledati relativno (za usporedbu), a ne apsolutno — i to samo za velike n.

Naš model složenosti (brojimo floating–point operacije)

$$F(n) = 2n + 1$$

je poprilično neprecizan. Zato dobijemo "čudno" ponašanje grafova brzine za male n.

Izmjerena vremena imaju puno "pitomije" ponašanje.

- Trajanje vanjske petlje s ponavljanjem je praktički konstantno, i dobivamo "razuman" broj u sekundama,

što znači da smo dobro izabrali broj ponavljanja $N(n)$.

BabyBlue, IVF, fast — Izračunate brzine

```
n =          1   261.764   261.579
n =          2   218.102   217.972
n =          5   191.968   191.899
n =         10   183.223   183.125
n =         20   178.828   178.883
n =         50   176.088   176.221
...
n =   10000000   174.385   174.506
n =   20000000   174.379   174.528
n =   50000000   174.376   174.462
n =  100000000   174.361   174.480
n =  200000000   174.360   174.523
n =  500000000   174.533   174.375
n = 1000000000   174.514   174.369
```

BabyBlue, IVF, fast — Izmjerena vremena

```
n =          1    11.461    11.469
n =          2    11.463    11.469
n =          5    11.460    11.464
n =         10    11.461    11.468
n =         20    11.464    11.460
n =         50    11.472    11.463
...
n =   10000000    11.469    11.461
n =   20000000    11.469    11.459
n =   50000000    11.469    11.464
n =  100000000    11.470    11.463
n =  200000000    11.470    11.460
n =  500000000    11.459    11.470
n = 1000000000    11.460    11.470
```

Parcijalne sume reda
Ubrzanje algoritma

Leibnizov kriterij za alternirajuće redove

Zamislimo da je zadana tražena točnost ε i treba naći sumu reda S, s greškom manjom od ε, tj. hoćemo

$$|S - S(n)| \le \varepsilon.$$

Koliko članova treba zbrojiti? $(n = ?)$

Za alternirajuće redove

$$S = \sum_{i=1}^{\infty} (-1)^{i-1} f_i, \quad f_i \text{ istog znaka},$$

koristimo Leibnizov kriterij konvergencije:

- Ako članovi reda f_i monotono teže u nulu, onda
- red konvergira, suma S ima predznak prvog člana f_1 i vrijedi $|S| < |f_1|$.

Leibnizov kriterij za ocjenu greške

Kod nas je $f_i = 1/i$, pa red za $\ln 2$ zadovoljava pretpostavku za Leibnizov kriterij.

No, ako članovi reda f_i monotono teže u nulu, onda to isto vrijedi i za "ostatak" reda

$$S - S(n) = \sum_{i=n+1}^{\infty} (-1)^{i-1} f_i.$$

Zaključak: Greška $S - S(n)$ ima predznak prvog zanemarenog člana $(-1)^n f_{n+1}$ i vrijedi ocjena greške

$$|S - S(n)| < |f_{n+1}|.$$

Dakle, ako uzmemo n tako da je $|f_{n+1}| \leq \varepsilon$, onda je sigurno

$$|S - S(n)| < \varepsilon.$$

Brzina konvergencije reda za $\ln 2$

Konkretno, u redu za $\ln 2$, znamo da je $|\ln 2 - S(n)| < \varepsilon$, ako za prvi zanemareni član vrijedi

$$\frac{1}{n+1} \leq \varepsilon \quad \text{ili} \quad n \geq \frac{1}{\varepsilon} - 1.$$

Ako želimo $\varepsilon = 10^{-18}$, onda treba uzeti $n = 10^{18}$ članova reda!

To će "potrajati" ...

- Znamo čak i koliko (v. tablicu) — oko $11.47 \cdot 10^9$ sekundi. To je preko 300 godina!

Srećom, ima i puno brži način.

Ideja usrednjavanja parcijalnih suma

Greška $S - S(n)$ ima predznak prvog zanemarenog člana $(-1)^n f_{n+1}$ i još vrijedi ocjena greške $|S - S(n)| < |f_{n+1}|$.

No, članovi alterniraju po predznaku, pa onda

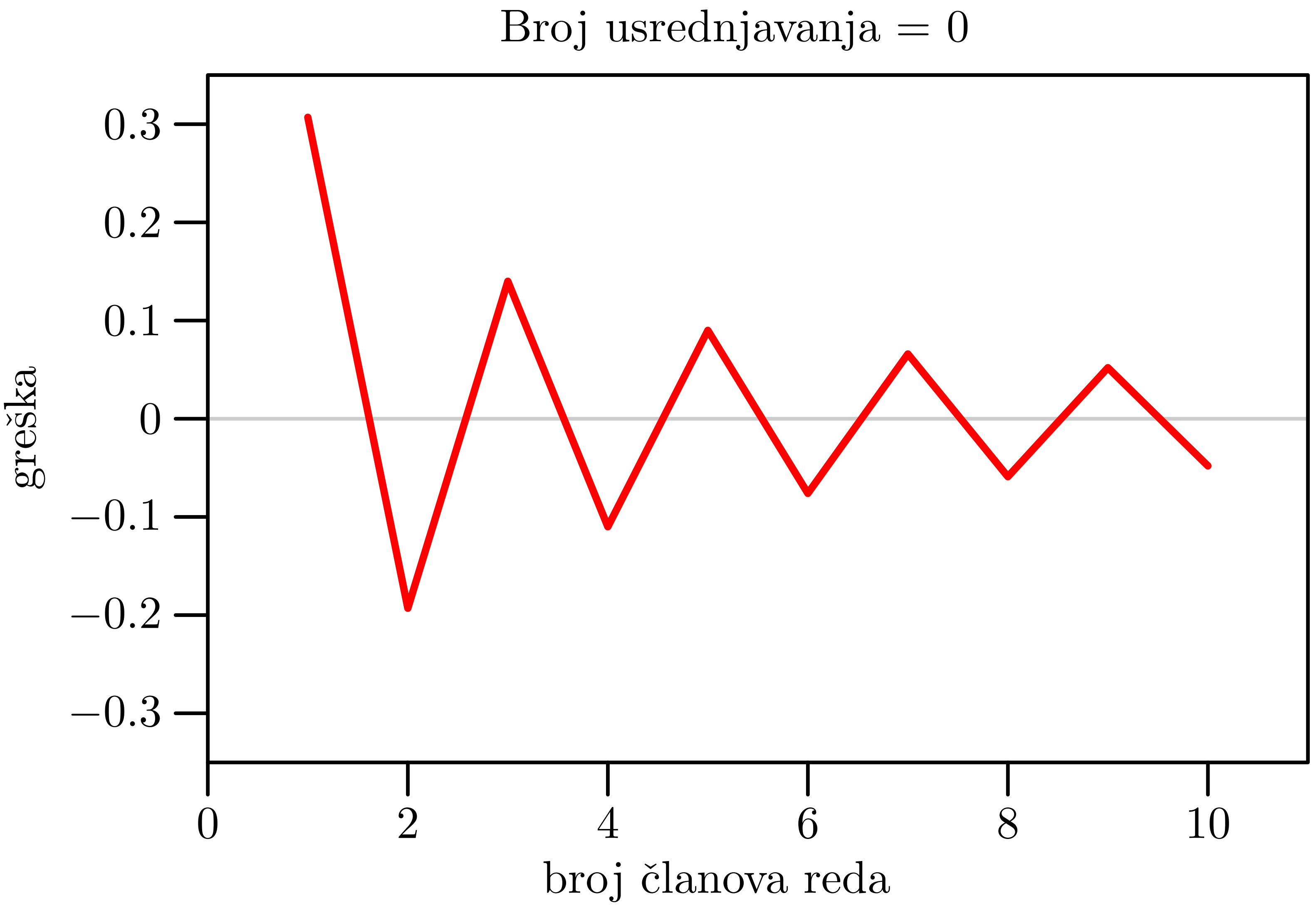
- greške alterniraju po predznaku, odnosno,
- parcijalne sume $S(n)$ "osciliraju" oko prave sume S (iznad–ispod–iznad–...).

Kad pogledamo dvije susjedne parcijalne sume $S(n)$ i $S(n+1)$ (jedna je iznad S, a druga ispod, ili obratno), odmah vidimo da je njihova srednja vrijednost

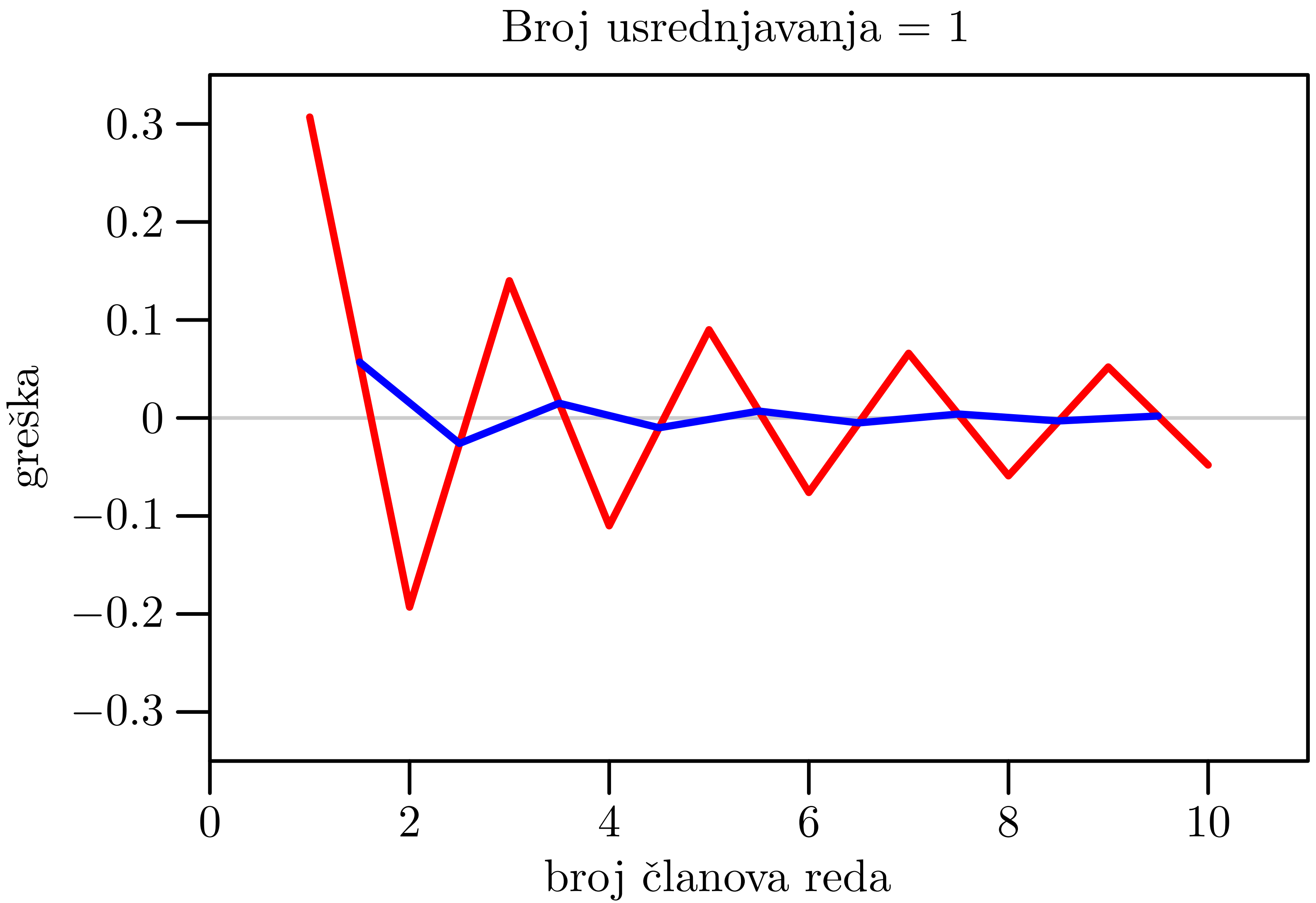
$$S(n,1) := \frac{S(n) + S(n+1)}{2}$$

mnogo bolja aproksimacija za S — bar 2 puta bolja od $S(n)$.

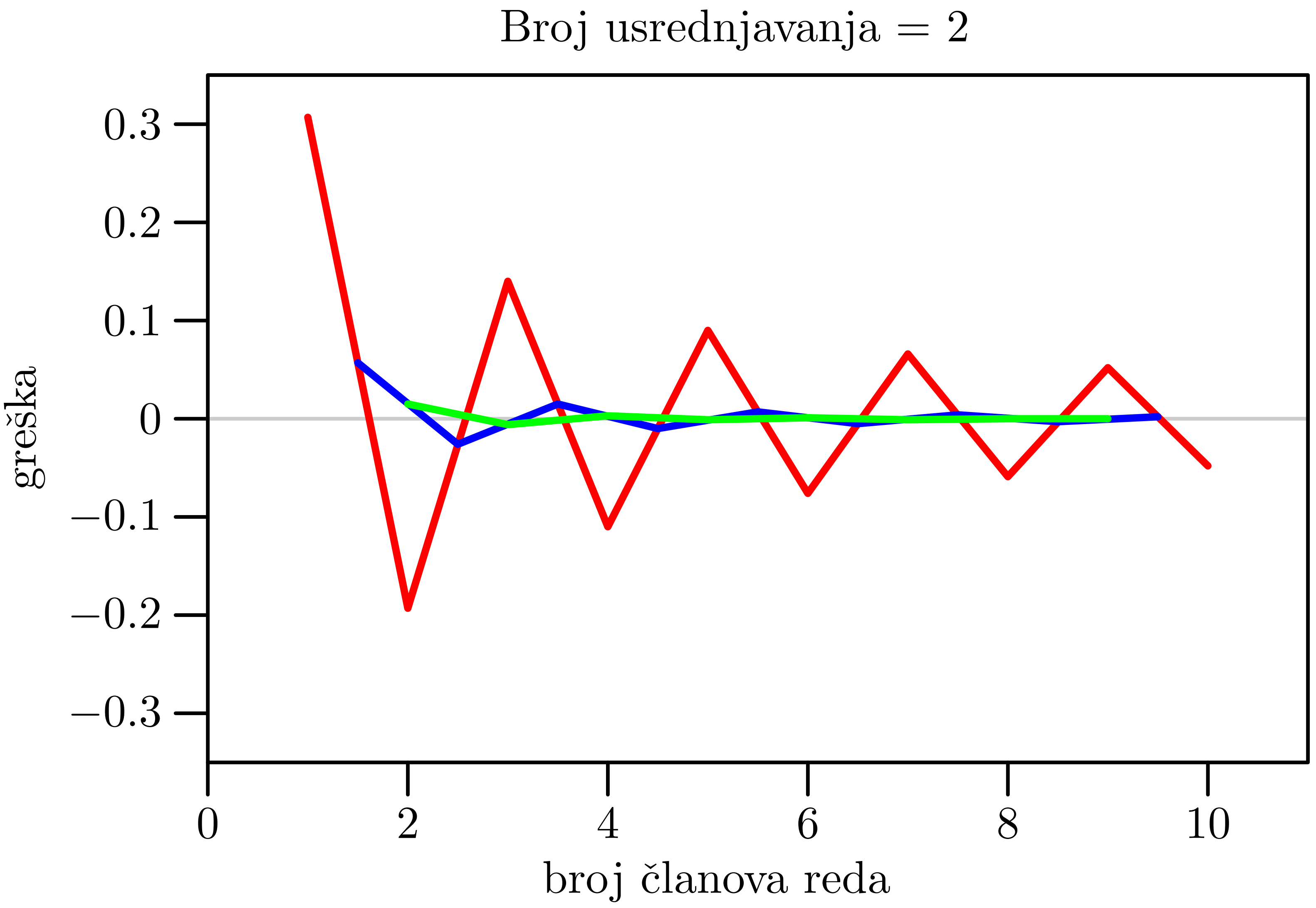
Greške nakon 0 usrednjavanja



Greške nakon 1 *usrednjavanja*



Greške nakon 2 usrednjavanja



Ponovljeno usrednjavanje

Ako članovi polaznog reda "ravnomjerno" teže u nulu, tj.

- ako greške usrednjenih parcijalnih suma $S(n, 1)$ imaju isti oblik ponašanja kao i greške za $S(n)$,

onda ima smisla ponoviti usrednjavanje (kao u primjeru)!

Općenito, iz $k - 1$-og niza, uz $S(n, 0) := S(n)$, usrednjavanjem dobivamo k-ti niz usrednjenih parcijalnih suma

$$S(n, k) := \frac{S(n, k-1) + S(n+1, k-1)}{2}.$$

Iz N članova polaznog reda možemo izračunati trokutastu tablicu usrednjenih parcijalnih suma

$$S(n, k), \quad n = 1, \ldots, N - k, \quad k = 0, \ldots, N - 1.$$

Tablica grešaka nakon 9 usrednjavanja

Vrijednosti grešaka $S - S(n,k)$ za prvih $N = 10$ članova reda:

k	1	2	3	4	5	6	7	8	9	10
0	-0.307	0.193	-0.140	0.110	-0.090	0.076	-0.066	0.059	-0.052	0.048
1	-0.057	0.026	-0.015	0.010	-0.007	0.005	-0.004	0.003	-0.002	
2	-0.015	0.006	-0.003	0.001	-0.001	0.001	-0.000	0.000		
3	-0.005	0.001	-0.001	0.000	-0.000	0.000	-0.000			
4	-0.002	0.000	-0.000	0.000	-0.000	0.000				
5	-0.001	0.000	-0.000	0.000	-0.000					
6	-0.000	0.000	-0.000	0.000						
7	-0.000	0.000	-0.000							
8	-0.000	0.000								
9	-0.000									

Ocjena greške, algoritam, ubrzanje, Madelung — v. `alt_red`.

Kad je usrednjavanje dobro?

U prijevodu — što znači da

- članovi polaznog reda "ravnomjerno" teže u nulu?

Članovi reda f_i (bez znaka) su vrijednosti $f_i = f(i)$, $i \in \mathbb{N}$, tzv. potpuno monotone funkcije f — one za koju vrijedi:

- $f \in C[1, \infty)$, tj. f je neprekidna na $[1, \infty)$,
- $f \in C^\infty(1, \infty)$, tj. f je beskonačno puta derivabilna,
- derivacije od f su fiksnog znaka i ti predznaci alterniraju, tj. $(-1)^k f^{(k)}(x) \geq 0$, za $x \geq 1$ i za svaki $k = 0, 1, 2, \ldots$.

U redu za $\ln 2$ je $f(x) = 1/x$ — očito je potpuno monotona.

Što sporije konvergira polazni red, to je usrednjavanje bolje!

Zbrajanje matrica

Zbrajanje matrica

Problem: Zadan je prirodni broj $n \in \mathbb{N}$ i 3 realne matrice A, B i C, reda n. Treba izračunati izraz

$$C := C + A + B.$$

Akumulacija ("nazbrajavanje") zbroja $A + B$ u matrici C ovdje ima samo jednu svrhu:

- "prevariti" optimizaciju compilera,

kod višestrukog ponavljanja eksperimenta.

Ova realizacija napravljena je po ugledu na množenje matrica (v. kasnije).

Zbrajanje matrica — formula

"Matematička" realizacija matrične operacije

$$C := C + A + B$$

po elementima je trivijalna:

$$c_{ij} := c_{ij} + a_{ij} + b_{ij},$$

za sve indekse

$$i = 1, \ldots, n, \quad j = 1, \ldots, n.$$

Dakle, "programski" — treba "zavrtiti" dvije petlje.

Zbrajanje matrica — potprogram

```
      subroutine addij (lda, n, a, b, c)
c
c   Matrix addition
c   C(n, n) = C(n, n) + A(n, n) + B(n, n).
c
      implicit none
c
      integer lda, n
      double precision a(lda, lda), b(lda, lda),
     $                 c(lda, lda)
c
      integer i, j, nn
```

Zbrajanje matrica — potprogram (nastavak)

```
c
c   IJ loop, inner
c
      nn = n
      do 20, i = 1, nn
         do 10, j = 1, nn
            c(i, j) = c(i, j) + a(i, j) + b(i, j)
   10       continue
   20    continue
c
      return
      end
```

Permutacija petlji

Prvu varijantu zovemo `ij` — po poretku (indeksa) petlji, izvana prema unutra.

Ove dvije petlje možemo permutirati, tj. zamijeniti im poredak, pa dobivamo `ji` varijantu:

```
c
c    JI loop, inner
c
      nn = n
      do 20, j = 1, nn
         do 10, i = 1, nn
            c(i, j) = c(i, j) + a(i, j) + b(i, j)
   10       continue
   20    continue
```

Broj operacija

U svakom prolazu kroz unutarnju petlju imamo dvije operacije

- zbrajanja matričnih elemenata.

Obje petlje imaju (svaka) točno n prolaza.

Dakle, ukupan broj operacija u obje varijante algoritma je:

$$F(n) = 2n^2.$$

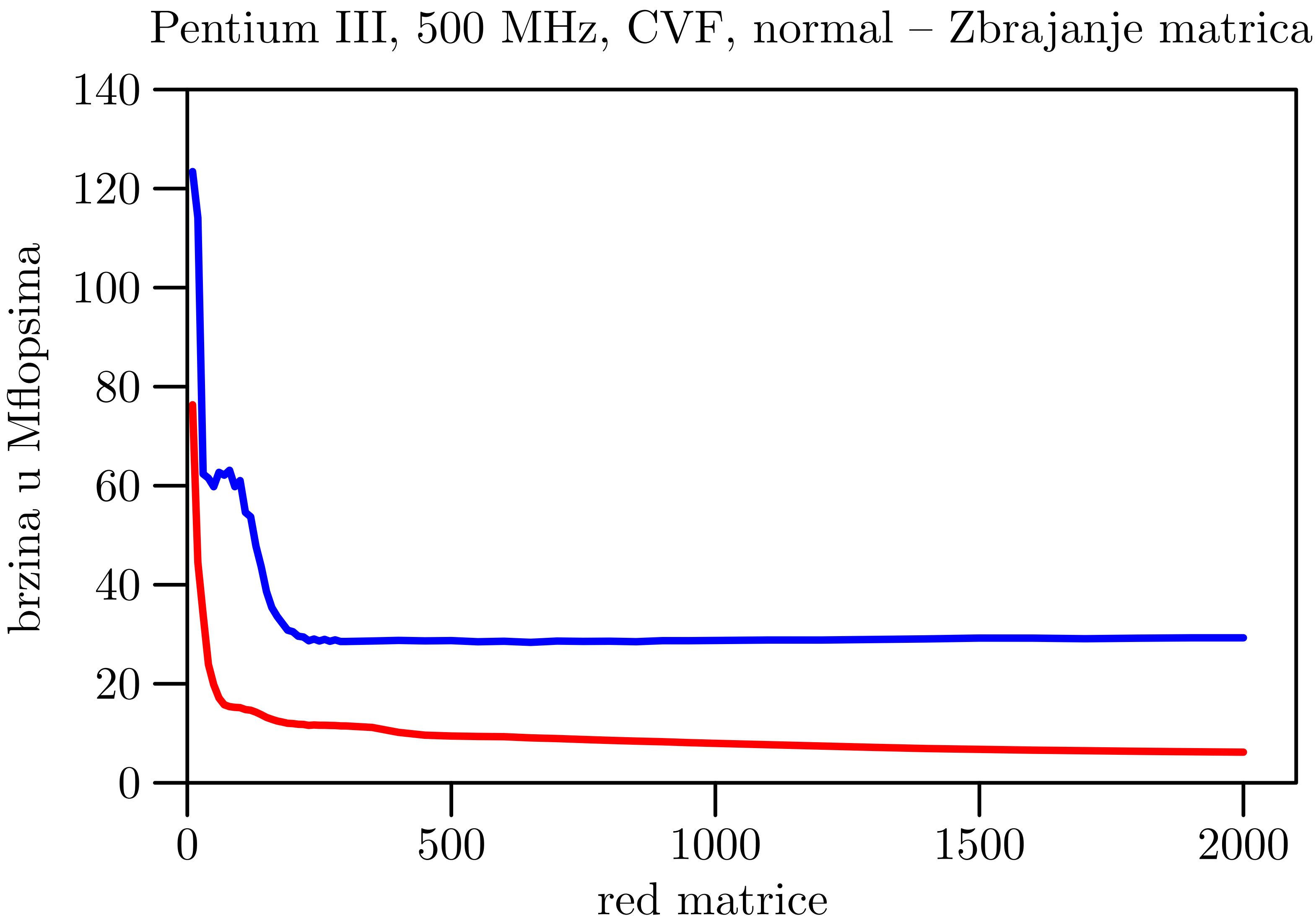
Broj ponavljanja $N(n)$ izabran je tako da dobijemo približno konstantno trajanje "okolne" petlje (s ponavljanjem) kojoj mjerimo vrijeme.

Boje na grafovima

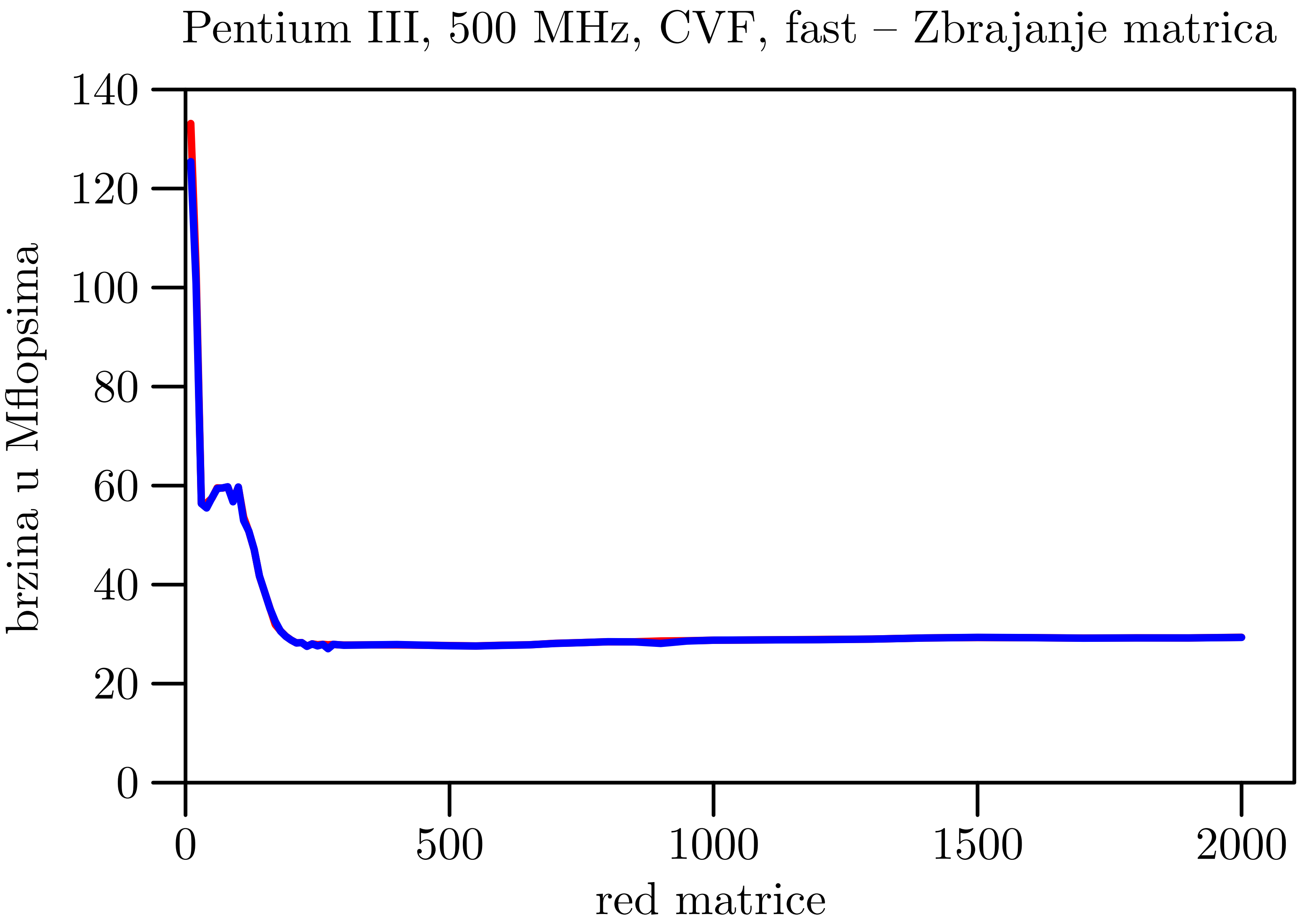
Legenda za čitanje grafova:

- petlja `ij` — crveno, sporo;
- petlja `ji` — plavo, brzo.

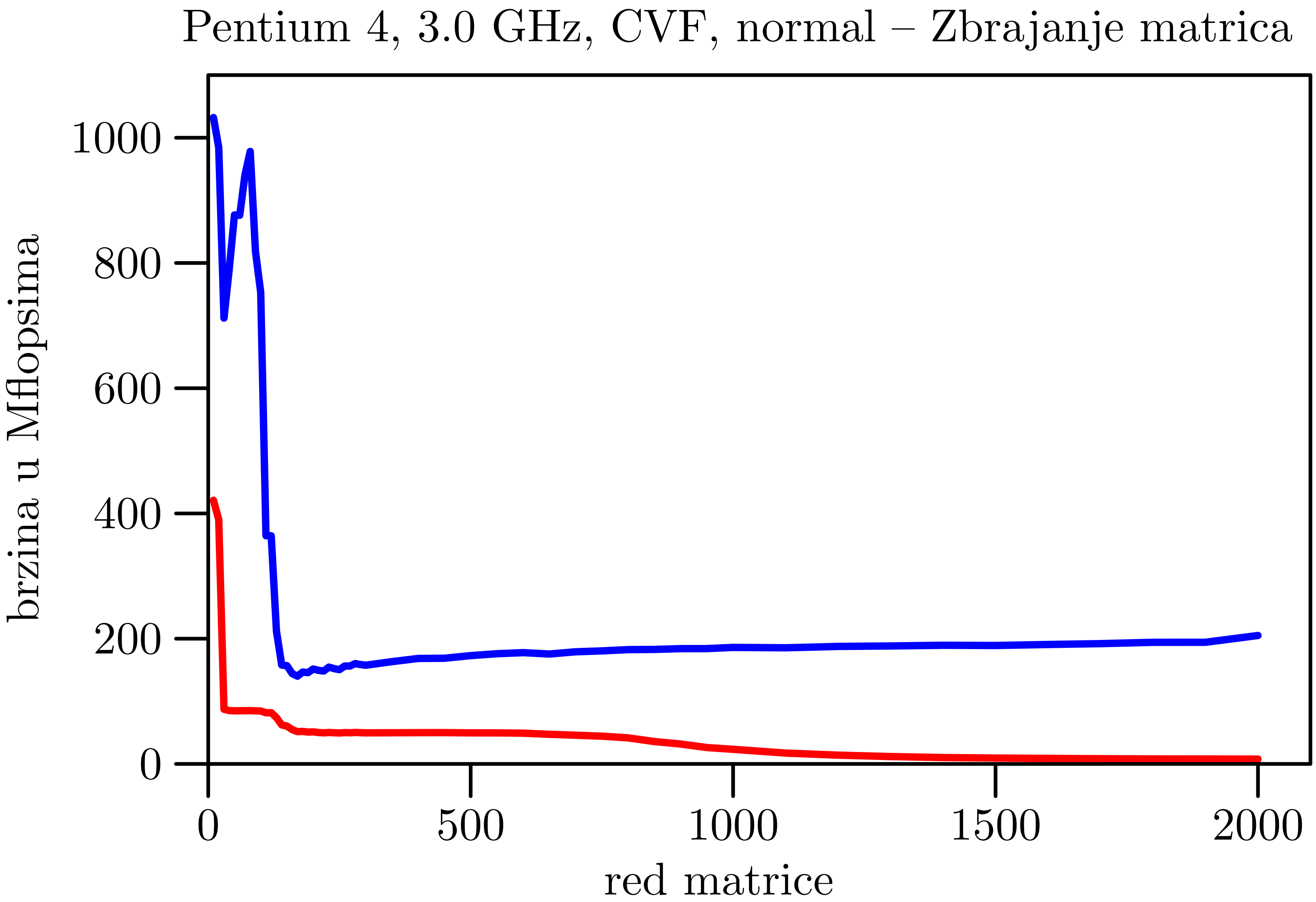
Klamath5, CVF, normal — ij, ji



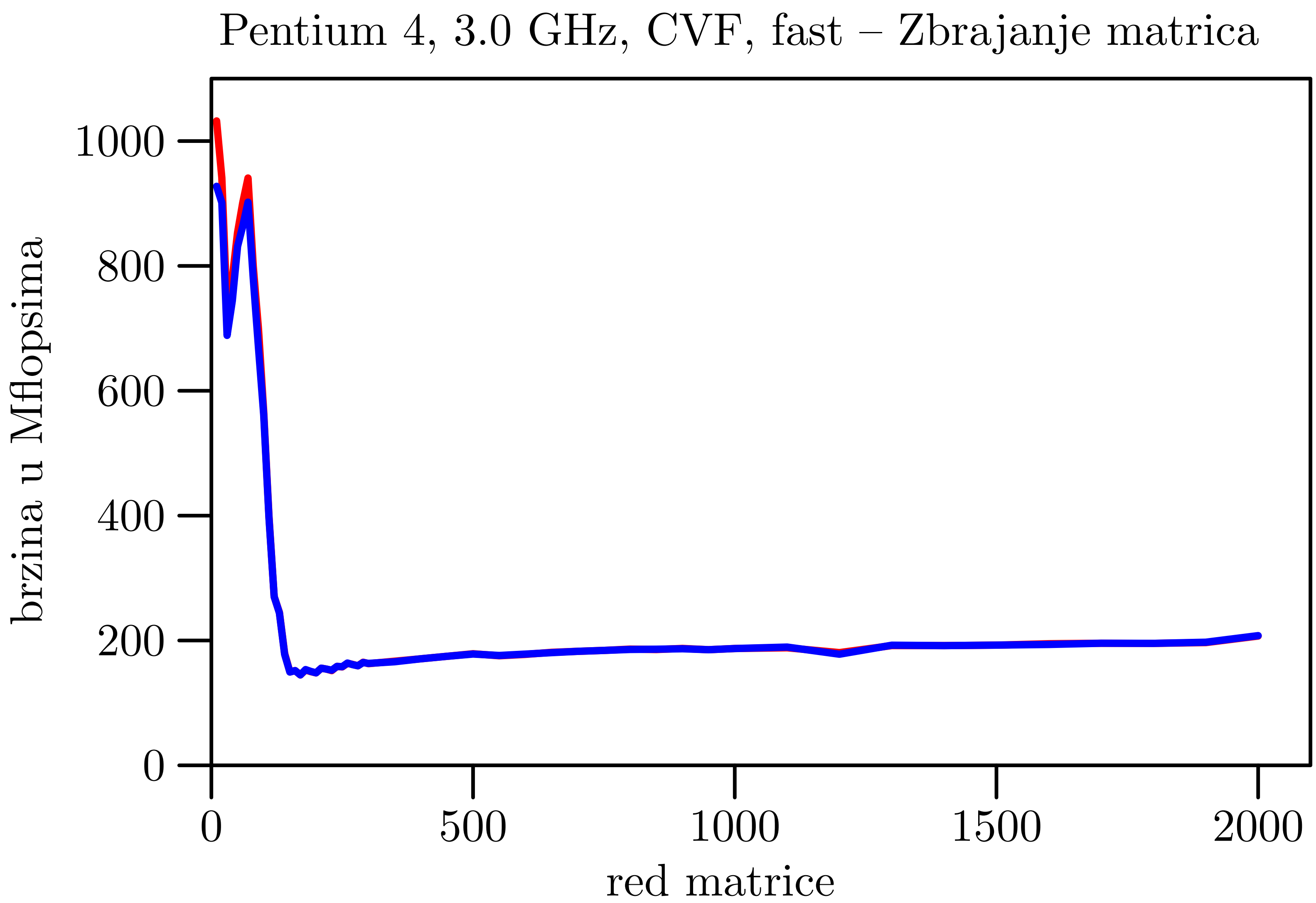
Klamath5, CVF, fast — `ij`, `ji`



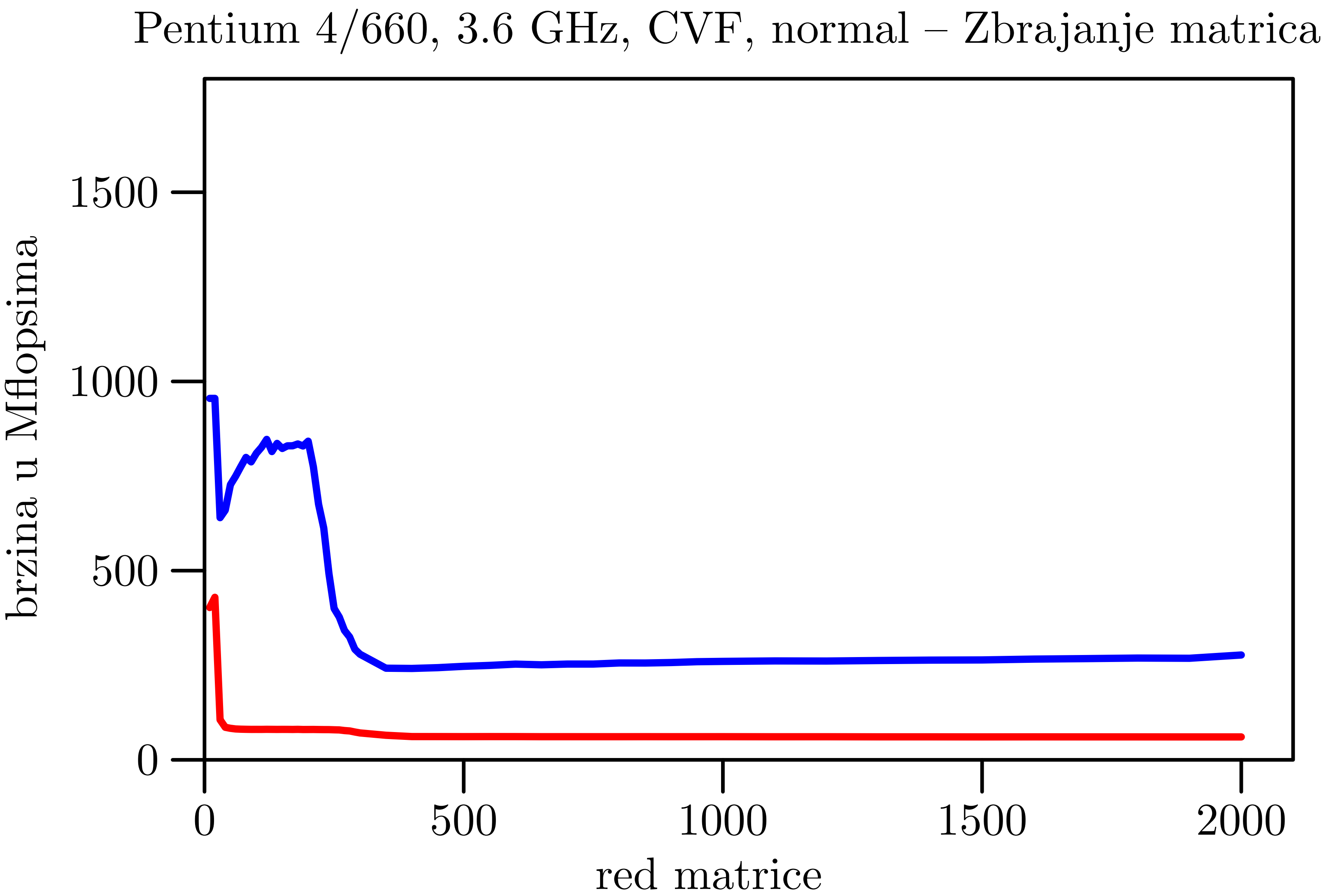
Veliki, CVF, normal — `ij`, `ji`



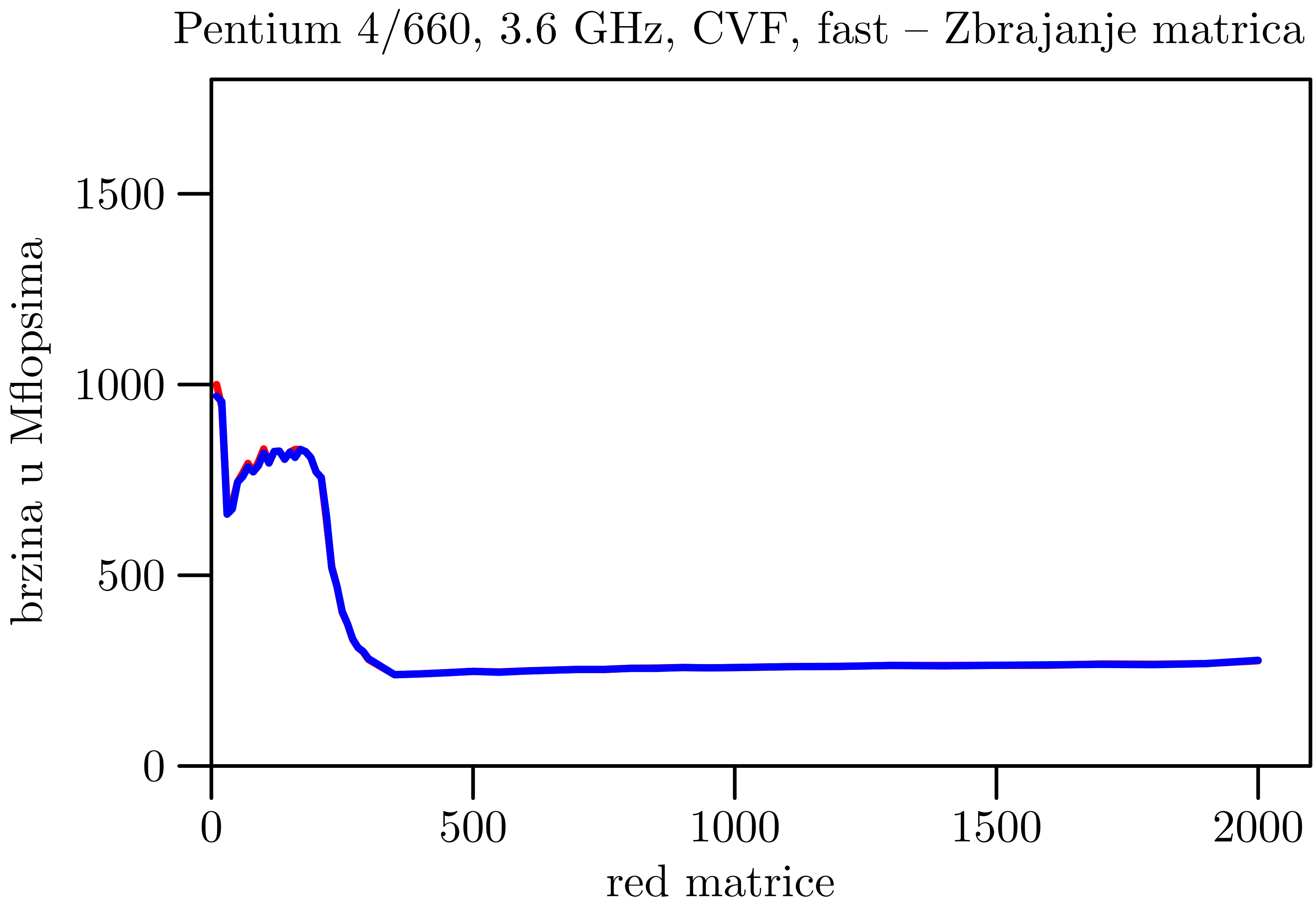
Veliki, CVF, fast — `ij`, `ji`



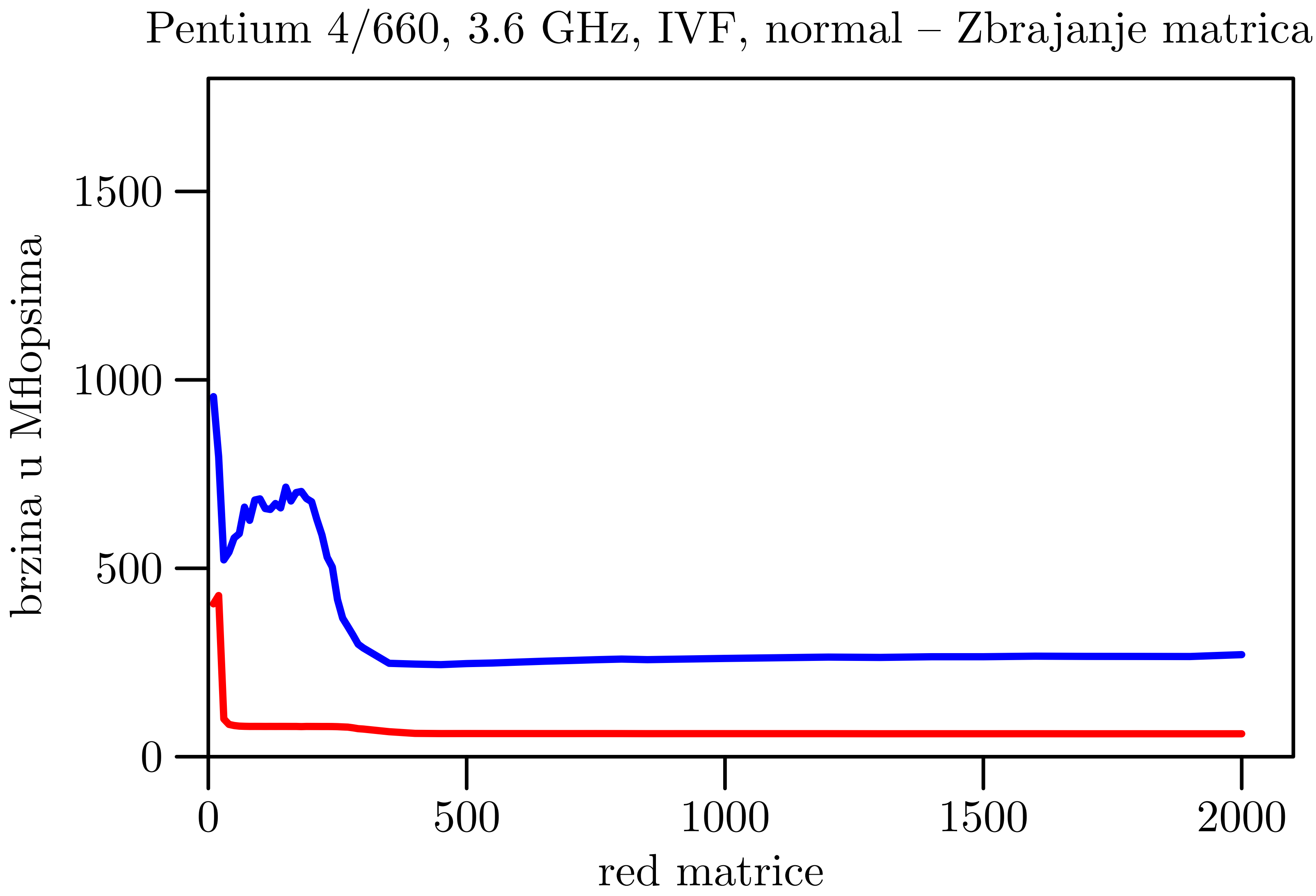
BabyBlue, CVF, normal — ij, ji



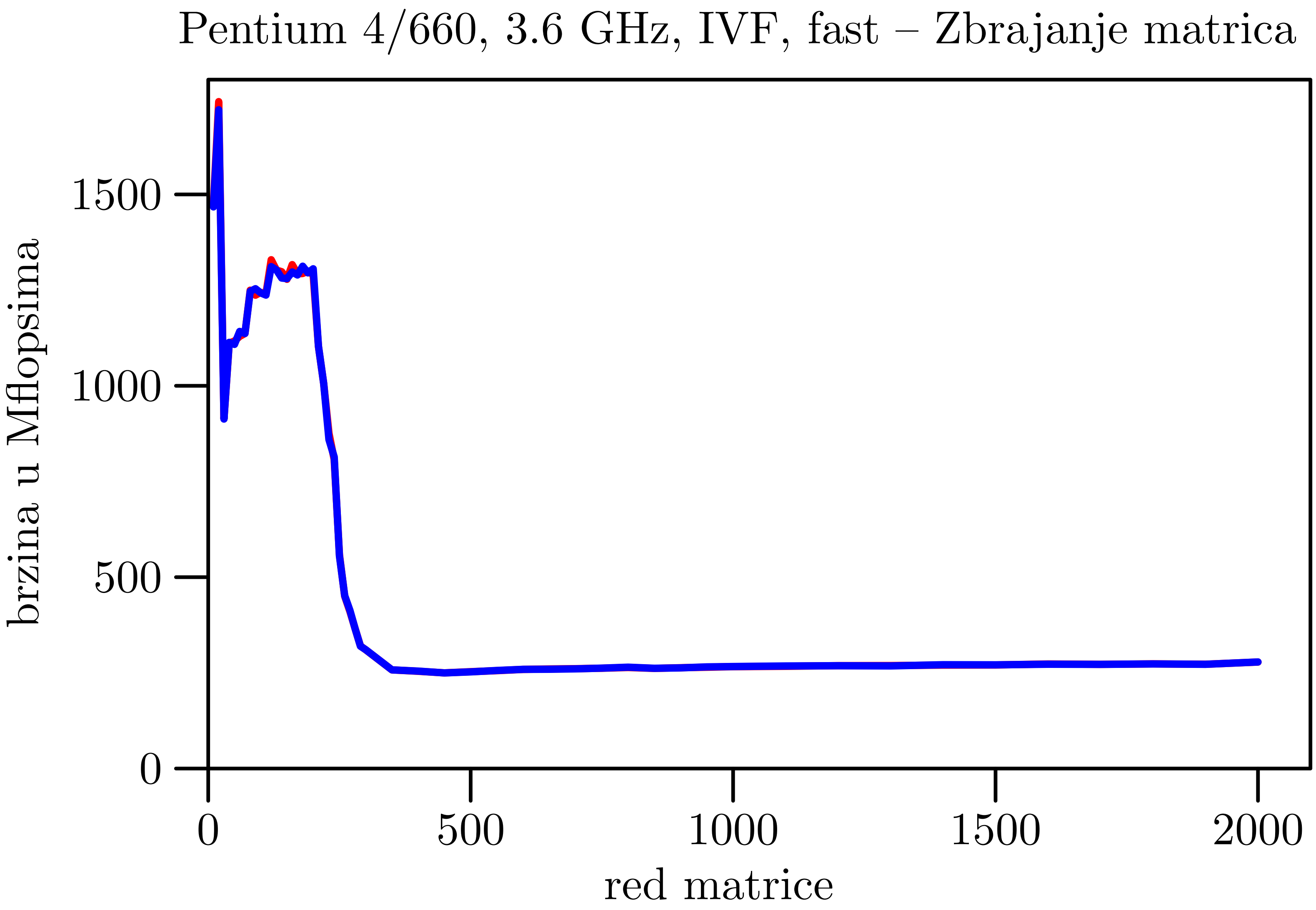
BabyBlue, CVF, fast — `ij`, `ji`



BabyBlue, IVF, normal — `ij`, `ji`



BabyBlue, IVF, fast — ij, ji



Tablica brzina za velike n

Razlika u brzini između brže `ji` petlje i sporije `ij` petlje je ogromna, a nema neke razlike u brzinama između:

- fast i normal opcija compilera za bržu `ji` petlju.

Usporedba brzina (u Mflops) sporije i brže petlje za razna računala (za velike n):

Računalo	`ij` petlja	`ji` petlja
Klamath5	6.2	29.3
Veliki	7.7	205.1
BabyBlue, CVF	60.5	277.1
BabyBlue, IVF	60.8	270.8

Komentar rezultata — brža i sporija petlja

Opažanje 1. Bez optimizacije dobivamo

- ogromnu razliku u brzini između brže i sporije petlje,

za bilo koje redove n.

(O razlici između malih i velikih n — malo kasnije.)

Razlog: Brzina je bitno veća kad

- podacima pristupamo redom, kako su spremljeni u memoriji

(Računanje adresa, "blok"–transfer podataka.)

Kako se spremaju matrice u pojedinim programskim jezicima?

Komentar rezultata — brže/sporije (nastavak)

FORTRAN: matrice se spremaju po stupcima, tj.

- "brže" se mijenja prvi indeks `i` (za retke).

Za sekvencijalni pristup podacima

- indeks stupca `j` mora biti izvana, a indeks retka `i` unutra.

Zato je `ji` petlja brža od `ij` petlje!

C i Pascal: matrice se spremaju po recima, tj.

- "brže" se mijenja drugi indeks `j` (za stupce).

Tamo je obratno — `ij` petlja je brža od `ji` petlje!

Komentar rezultata — “mali” i “veliki” n

Opažanje 2. Za male redove n dobivamo

- bitno veće brzine, u usporedbi s onima za velike n,

i to bez obzira na optimizaciju.

Razlog: “Krivac” za ovo povećanje brzine je cache memorija.

Međutim, to povećanje brzine

- ne pripada problemu zbrajanja matrica, jer svaki ulazni podatak koristimo samo jednom,
- već dolazi samo od višestrukog ponavljanja eksperimenta (pa matrice ostaju u cacheu).

Posljedica: brzinu za velike n ne možemo “popraviti” promjenom algoritma.

Stvarni “izgled” računala

Sadržaj

- Stvarni “izgled” računala:
  - Registri modernog procesora (IA–32).
  - Primjer matične ploče, blok–dijagram.
  - Hijerarhijska struktura memorije (cache).
  - “Priča o cacheu”.

Standardni kućni procesori

Standardni kućni procesori bazirani su na tzv. IA–32 arhitekturi (Intel ili AMD, svejedno mi je). Osnovna svojstva:

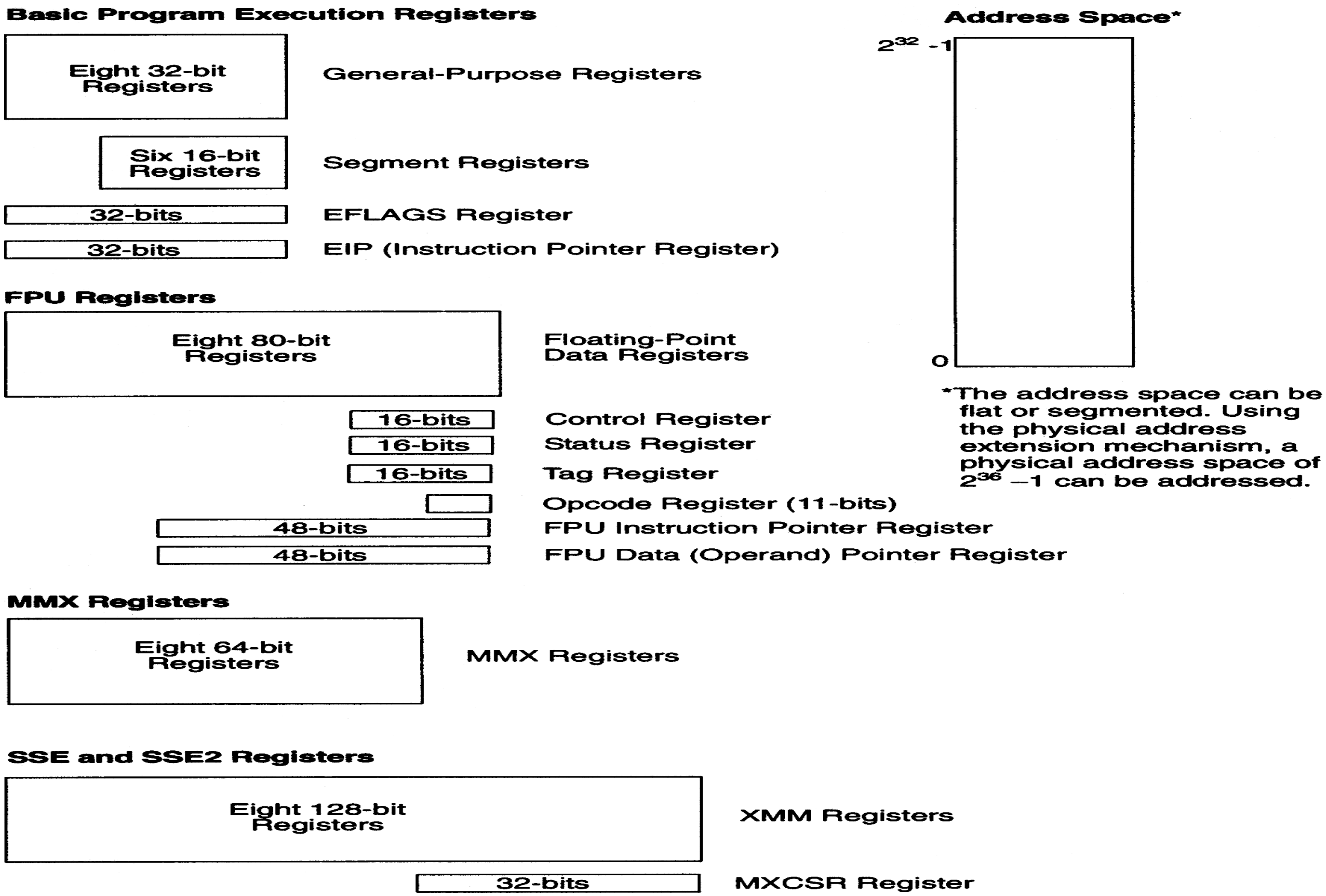
- riječ = 32 bita = 4 B, (toliki je tip `int` u C-u),
- adresa = 32 bita (x86) ili, modernije, 64 bita (x64).

Ovi procesori imaju gomilu registara, raznih namjena, koji sadrže razne vrste podataka i instrukcija (ili dijelova instrukcija).

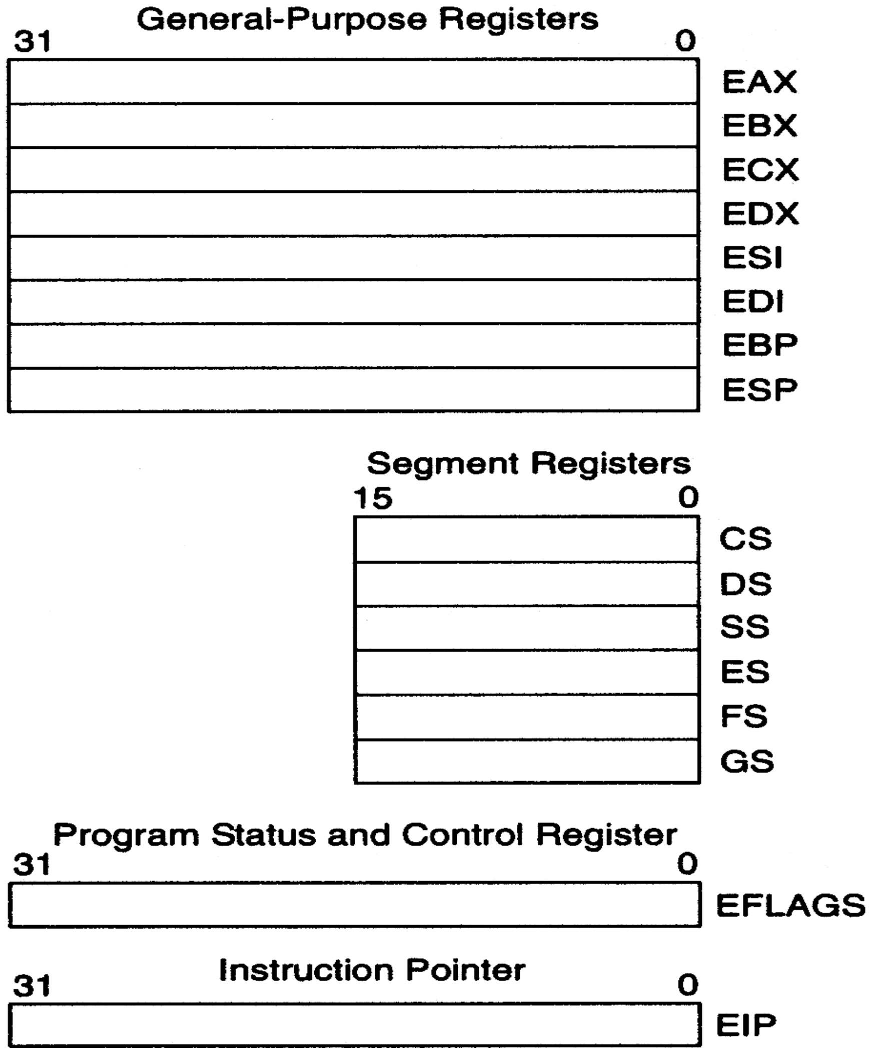
Shematski izgled svih registara, a onda samo registara opće namjene dan je na sljedeće dvije stranice.

Napomena: slike odgovaraju IA–32 procesoru Pentium 4, serija Northwood, podnožje 478 (danas već zastarjelom).

IA–32 — Svi registri i adresni prostor



IA–32 — Osnovni izvršni registri

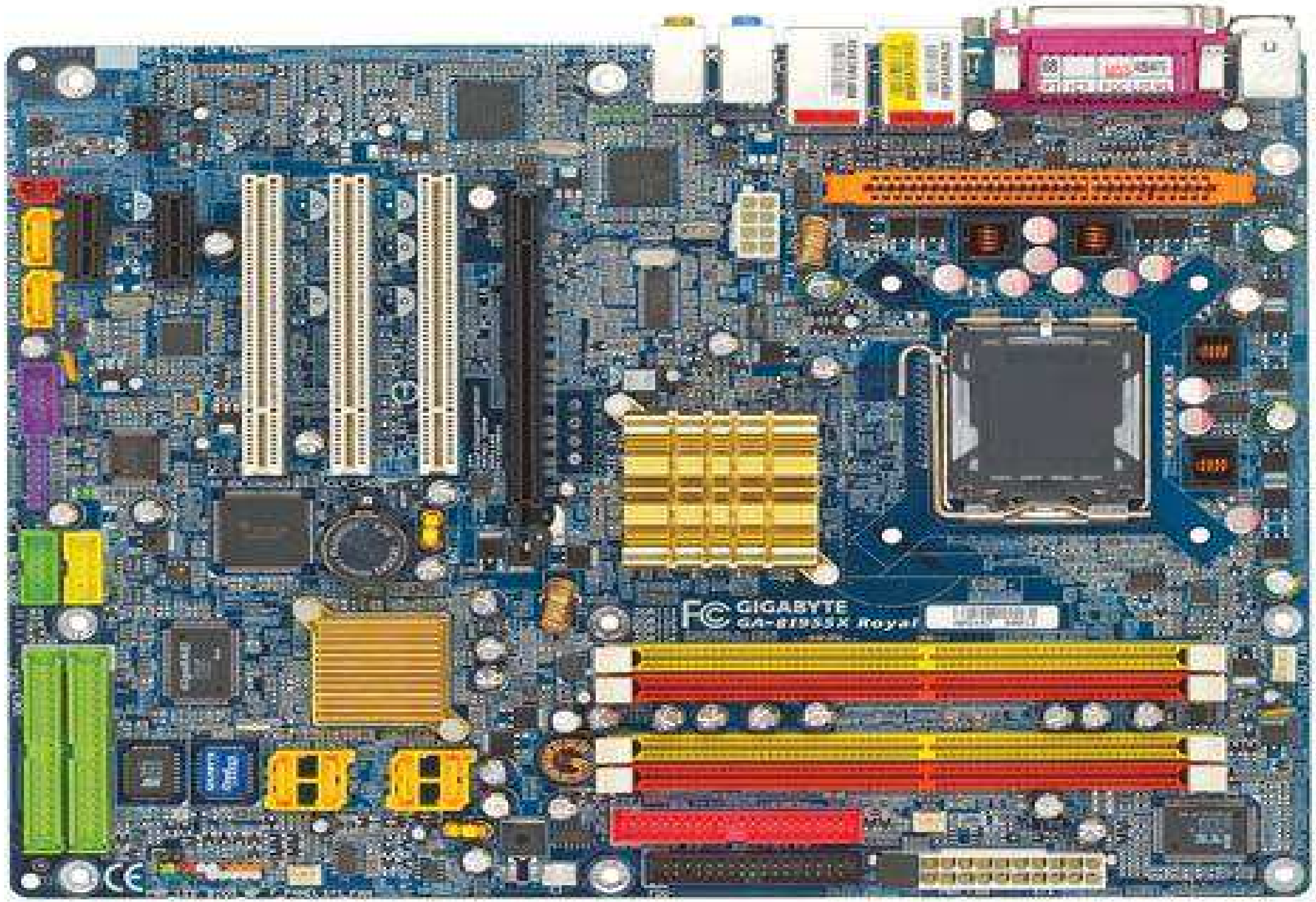


Izgled matične ploče računala

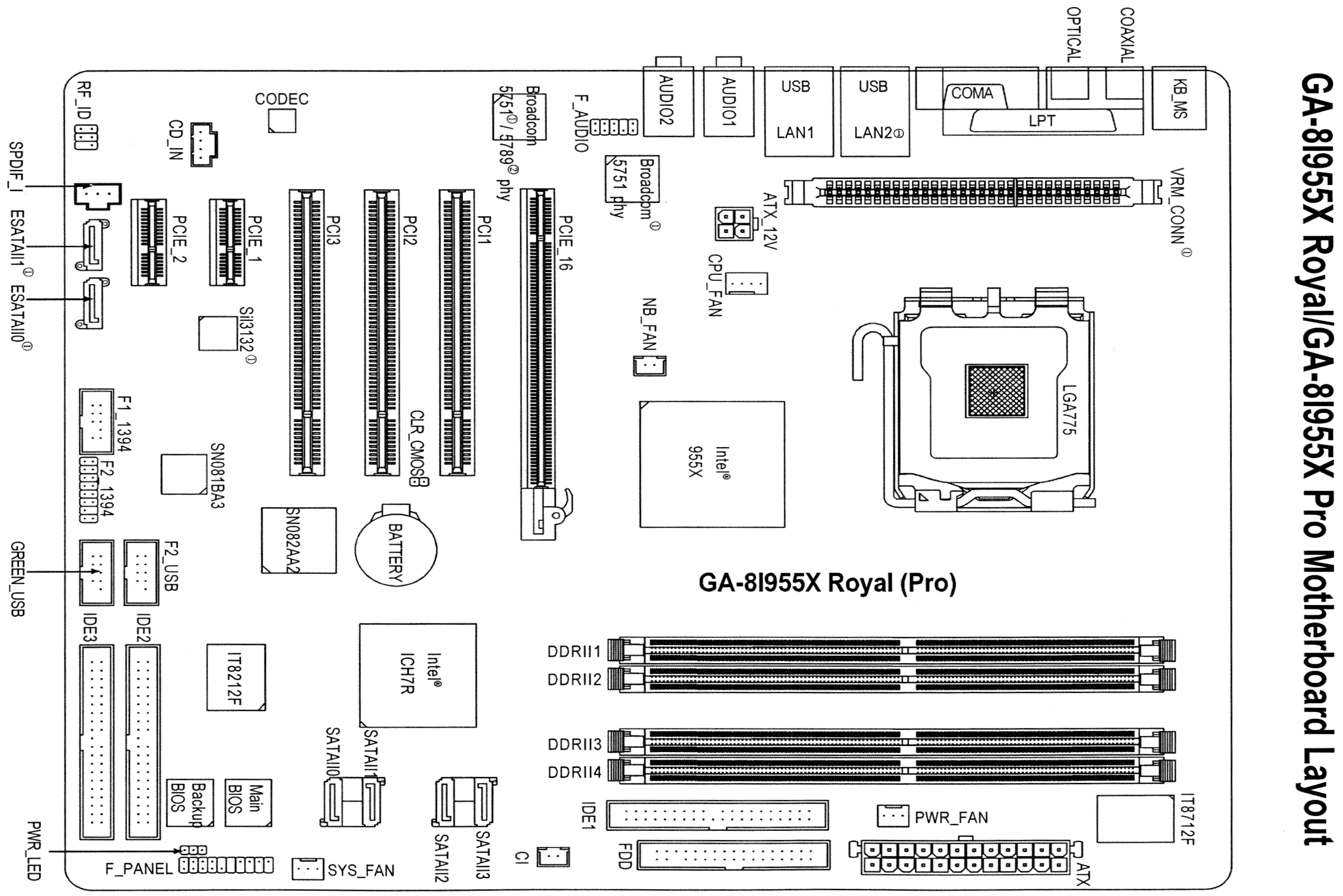
Moderna "kućna" računala, naravno, imaju sve standardne dijelove računala.

- Međutim, zbog "multimedijalne" namjene, ta računala imaju mogućnost priključivanja velikog broja raznih uređaja ("ulaz–izlaz").
- Gomila toga je već ugrađena na modernim tzv. matičnim pločama (engl. motherboard).
- Procesor zauzima relativno "mali" dio površine (ili prostora), a najuočljiviji dio na njemu (nakon ugradnje) je hladnjak.
- Utori za memorijske "chipove", također, ne zauzimaju previše prostora.

Matična ploča GA–8I955X Royal — izgled

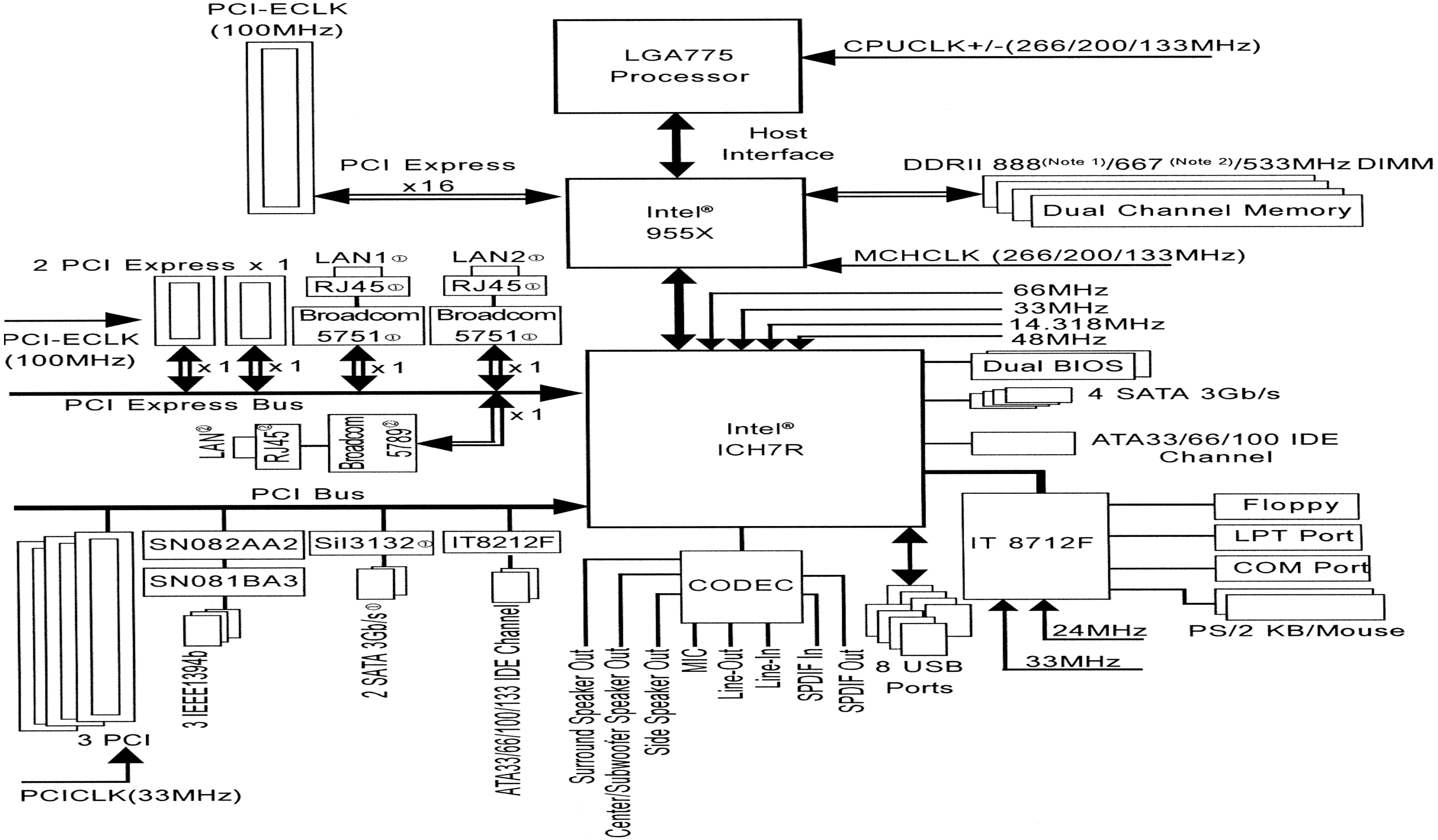


Matična ploča — raspored



Matična ploča — blok dijagram

Block Diagram



(Note 1) DDR II memory can be overclocked to 888MHz (must be used with an 1066MHz FSB processor) through overclocking in BIOS. Go to GIGABYTE's website for more information about the supported DDR II memory modules for this feature.
(Note 2) To use a DDR II 667 memory module on the motherboard, you must install an 800/1066MHz FSB processor.

Izgled matične ploče računala (nastavak)

Zbog bitno različite brzine pojedinih dijelova računala, postoje još dva bitna "chipa" koji povezuju razne dijelove i kontroliraju komunikaciju — prijenos podataka između njih. To su:

- Tzv. "northbridge" (sjeverni most), koji veže procesor s "bržim" dijelovima računala. Standardni brzi dijelovi su:
  - memorija,
  - grafika (grafička kartica).
- Tzv. southbridge (južni most), na kojem "visi" većina ostalih "sporijih" dijelova ili vanjskih uređaja.

Izgled matične ploče računala (nastavak)

- Tipični uređaji vezani na southbridge su:
  - diskovi (koji mogu biti i na dodatnim kontrolerima),
  - DVD i CD uređaji,
  - diskete,
  - komunikacijski portovi,
  - port za pisač (printer),
  - USB (Universal Serial Bus) portovi,
  - tzv. Firewire (IEEE 1394a, b) portovi,
  - mrežni kontroleri,
  - audio kontroleri,
  - dodatne kartice u utorima na ploči (modem), itd.

Izgled matične ploče računala (nastavak)

Veze između pojedinih dijelova idu tzv. “magistralama” ili “sabirnicama” (engl. bus, koji nije autobus).

- Ima nekoliko magistrala, raznih brzina.
- Na istoj magistrali može biti više uređaja, i oni su, uglavnom, podjednakih brzina.

Uočite hijerarhijsku organizaciju komunikacije pojedinih dijelova:

- najsporiji su vezani na ponešto brže,
- ovi na još brže,
- i tako redom, do najbržeg — procesora.

Ova hijerarhija je ključna za efikasnu komunikaciju!

Hijerarhijska struktura memorije

Nažalost, ova hijerahija komunikacije nije dovoljna za efikasnost modernog računala. Grubo govoreći, fali joj vrh, koji se ne vidi dobro na izgledu matične ploče.

- Pravo i najgore usko grlo u prijenosu podataka je komunikacija između procesora i memorije.

Gdje je problem?

Podsjetimo: bilo koje operacije nad bilo kojim podacima možemo napraviti samo u procesoru — preciznije, u registrima procesora. To znači da

- prije same operacije, podatak moramo "dovući" iz obične memorije u neki registar procesora.

Baš to je sporo!

Hijerarhijska struktura memorije (nastavak)

Na primjer, ako procesor radi na 3.6 GHz, a memorija na 533 MHz, onda će

- prijenos podatka u registar trajati okruglo 6 puta dulje od operacije na njemu.

Nažalost, isti tehnološki problem se javlja kod svih modernijih računala.

- Obična radna memorija je bitno sporija od procesora.

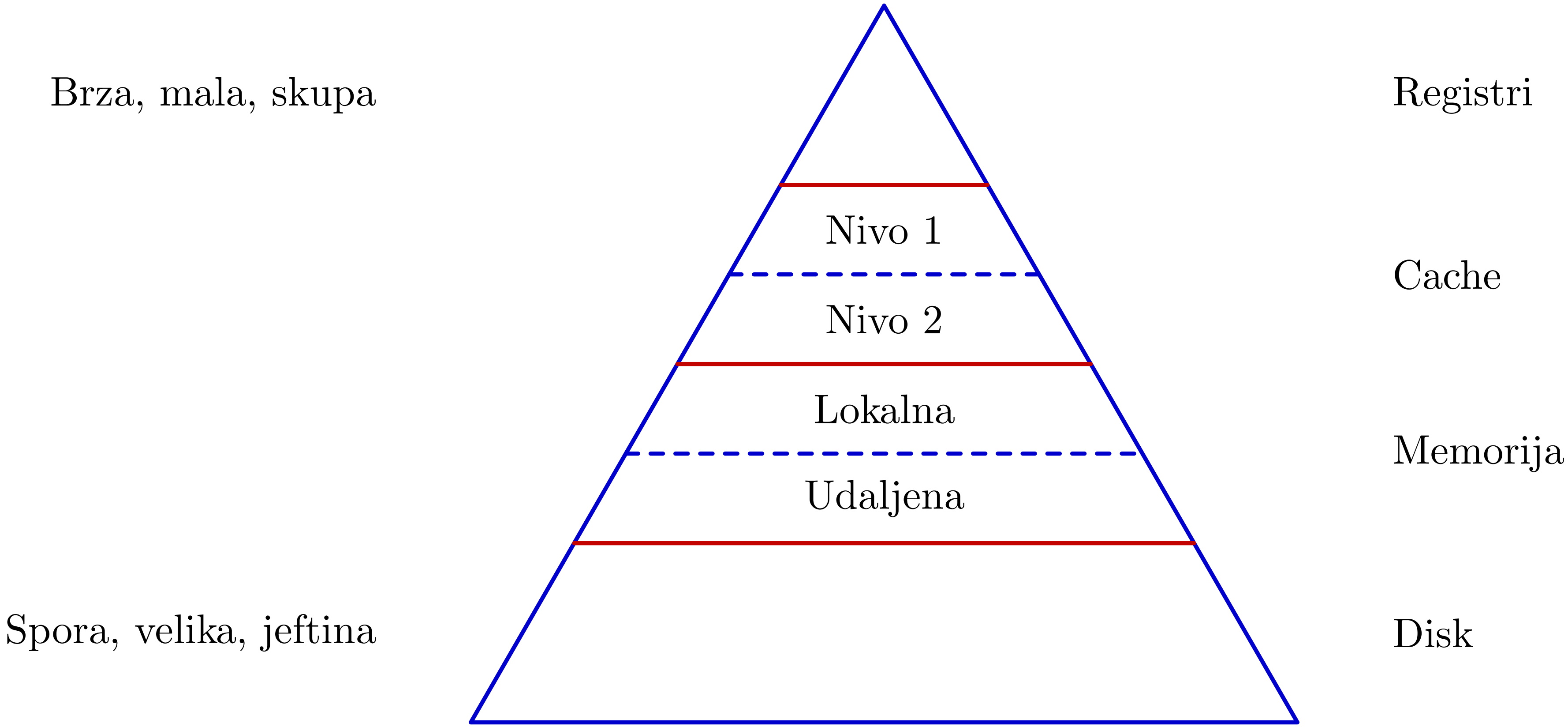
Kako se to izbjegava, ili, barem ublažava?

- Dodatnom hijerhijskom strukturom memorije, između obične radne memorije (RAM) i registara procesora.

Ta "dodatna" memorija se tradicionalno zove cache.

Hijerarhijska struktura memorije (nastavak)

Globalna struktura memorije u računalu ima oblik:



Cache memorija

Dakle, cache je mala i brza “lokalna” memorija — bliža procesoru od obične memorije (RAM). Gdje se nalazi?

- Obično, na samom procesorskom chipu, da bude što bliže registrima.

Nadalje, i taj cache je hijerarhijski organiziran. U modernim procesorima postoji nekoliko nivoa (razina) cache memorije.

- L1 cache za podatke i instrukcije — najbrži, veličina (trenutno) u KB.
- L2 cache za podatke — nešto sporiji, danas obično na frekvenciji procesora, veličina već u MB.
- Katkad postoji i treća razina — L3 cache.

Cache memorija (nastavak)

Na primjer, moj “notebook” ima Intel Pentium 4–M procesor koji na sebi ima (bez pretjeranih tehničkih detalja):

- L1 cache za podatke — 8 KByte-a,
- L1 cache za instrukcije — 12 K tzv. mikro–operacija,
- L2 cache — 512 KByte-a, na frekvenciji procesora.

Ovo su tipični omjeri veličina za Intelove procesore.

Za usporedbu, na AMDovim procesorima omjeri su bitno drugačiji:

- L1 cache je veći,
- L2 cache nešto manji (i, katkad, sporiji).

(Ne ulazimo u to što je bolje!)

Cache memorija (nastavak)

Kako (ugrubo) radi cache?

Kad računalo (tj. njegov operacijski sustav) izvršava neki naš program, onda

- uglavnom, imamo kontrolu sadržaja obične memorije koju taj naš program koristi za podatke i naredbe.

Za razliku od toga,

- nemamo nikakvu izravnu kontrolu nad sadržajem cache memorije.

Naime, cache nije izmišljen zato da bude mala, brža kopija obične memorije i tako ubrza ukupni rad računala.

Cache memorija (nastavak)

Puno je efikasnije da

- cache sadrži podatke koji se češće koriste.

Isto vrijedi i za instrukcije. Dakle, osnovna ideja je:

- “Skrati put do onog što ti često treba”.

Naravno, ključna stvar za efikasnost je:

- Što znači “češće” korištenje nekog podatka ili instrukcije?

Dobra globalna ili prosječna efikasnost postiže se samo ako se to odnosi na sve što računalo izvršava u nekom trenutku, tj. na sve pokrenute korisničke programe i dijelove operacijskog sustava.

Cache memorija (nastavak)

U tom svjetlu, kad malo bolje razmislite,

- zaista bi bilo nepraktično da svaki programer određuje što i kada treba ići u koju cache memoriju,

jer prosječna efikasnost nipošto ne ovisi samo o njegovom programu. Zato nema posebnih naredbi za

- učitavanje podataka u cache, ili
- pisanje podataka iz cachea u običnu memoriju.

Umjesto toga, sadržajem cachea upravljaju posebni cache kontroleri, koji

- raznim tehnikama "asocijacije" na više načina povezuju nedavno korištene podatke i instrukcije s onima koje tek treba iskoristiti i izvršiti.

Cache memorija (nastavak)

Bez puno tehničkih detalja, ova asocijacija se realizira otprilike ovako:

- Za svaki sadržaj (podatak ili instrukciju) u cacheu, dodatno se pamti i adresa (iz RAM-a), s koje je taj sadržaj stigao.
- Ako procesor (uskoro) zatraži sadržaj s te adrese, on se "čita" iz cachea (tj. ne treba po njega ići u RAM).
- Po istom sistemu, u cacheu se pamte i stvari koje se "pišu" u običnu memoriju (na putu u RAM).
- Tada se iz cachea brišu podaci koji su najstariji, odnosno, najmanje korišteni (u zadnje vrijeme, otkad su u cacheu).

Cache memorija (nastavak)

Dakle, sadržaj cachea se stalno obnavlja, tako da

- cache čuva najčešće nedavno korištene sadržaje koji bi uskoro mogli trebati.

Iskustvo pokazuje da se isti sadržaji vrlo često koriste više puta, pa se ovo isplati.

Očiti primjer:

- instrukcije u petljama se ponavljaju puno puta!

Ne zaboravimo da je upravo to svrha programiranja i osnovna korist računala.

Cache memorija (nastavak)

Malo kompliciranije je s podacima.

- Ako naš algoritam ne koristi iste podatke puno puta, onda nam cache neće ubrzati postupak.
- U suprotnom, isplati se preurediti algoritam tako da iste podatke koristi puno puta, ali u kratkom vremenskom razmaku — da ne "izlete" iz cachea. (To je neizravna kontrola nad sadržajem cachea.)

Primjeri iz linearne algebre:

- zbrajanje matrica, $C = A + B$ — cache ne pomaže puno;
- množenje matrica, $C = C + A * B$ — dobro korištenje cachea može ubrzati množenje matrica i za 5 puta.