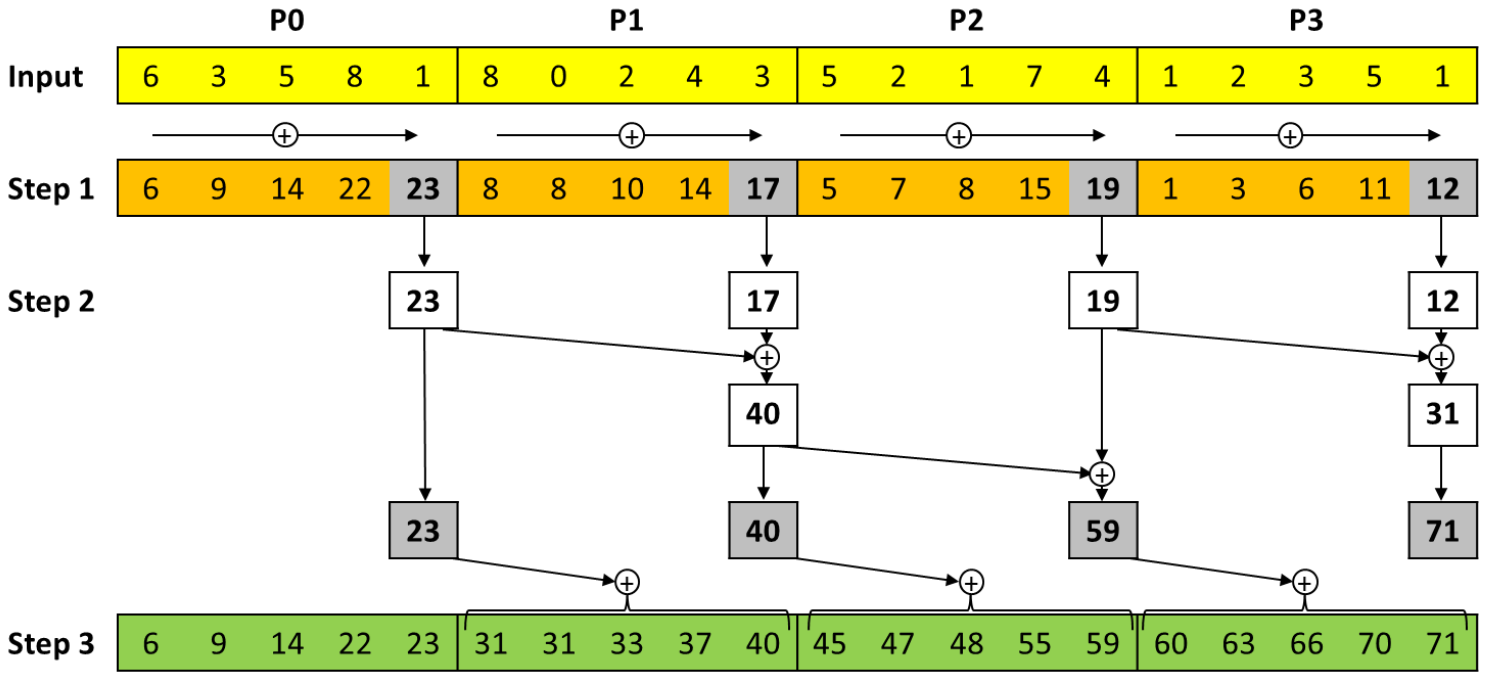
Projektovanje paralelnih programa

Projektovanje paralelnih programa

- **Podela** (engl. *partitioning*)**:**
  - Dati problem treba razložiti na delove primenom paralelizma na nivou podataka, zadataka ili upletenog paralelizma
- **Komunikacija** (engl. *communication*)**:**
  - Izabrana šema podele određuje tip i količinu neophodne komunikacije
- **Sinhronizacija** (engl. *synchronization*)**:**
  - Može se javiti potreba da se niti i/ili procesi sinhronizuju kako bi sarađivali na odgovarajući način
- **Balansiranje opterećenja** (engl. *load balancing*)**:**
  - Posao treba ravnomerno raspodeliti između niti ili procesa kako bi opterećenje sistema bilo balansirano i kako bi se minimizovao prazan hod
- Jedan od pristupa je i **Fosterova metodologija za projektovanje paralelnih algoritama**

Primer – sken (prefiksna suma)

```
for (i=1; i<n; i++)Y[i]=X[i]+Y[i-1];
```



1. **Lokalno sumiranje** od strane svakog od procesora
2. **Računanje prefiksne sume** upotrebom samo krajnje desne vrednosti svakog od lokalnih nizova
3. **Dodavanje** svakom od elemenata lokalnog niza vrednosti izračunate u koraku 2 za levog suseda

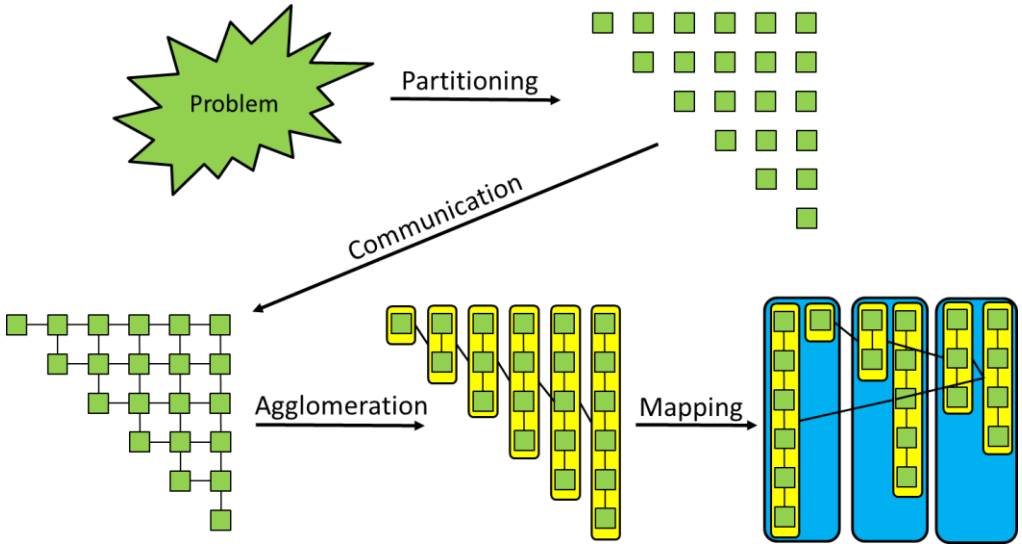
Izvor: https://parallelprogrammingbook.org/

Fosterova metodologija

Fosterova metodologija za projektovanje paralelnih algoritama

1. **Podela** (engl. *Partitioning*)**:** rastaviti problem na veliki broj malih zadataka (fine granularnosti), koji se mogu potom paralelno rešavati
2. **Komunikacija** (engl. *Communication*): odrediti neophodnu komunikaciju između zadataka
3. **Aglomeracija** (engl. *Agglomeration*): spajanje manjih zadataka u veće (grublje granularnosti) kako bi se smanjila komunikacija kroz unapređenu lokalnost podataka
4. **Mapiranje** (engl. *Mapping*): dodela većih zadataka procesima kako bi se minimizovala komunikacija, omogućila konkurentnost i balansiralo opterećenje

PCAM



Izvor: https://parallelprogrammingbook.org/

Primer – Jacobijev metod

- Svaka od vrednosti se menja prosečnom vrednošću svoja četiri suseda – **šablon** (engl. *stencil*), granica ostaje konstantna

$\mathbf{x}[i][j]$

−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1
3	3	3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1

$\mathbf{x}_{new}[i][j]$

2	2	2	2	2	2	2	2	2	2
2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8
2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3
2	2	2	2	2	2	2	2	2	2
1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8
1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3
1	1	1	1	1	1	1	1	1	1
0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8
0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
−0.3	−0.3	−0.3	−0.3	−0.3	−0.3	−0.3	−0.3	−0.3	−0.3

Izvor: https://parallelprogrammingbook.org/

Primer – Jacobijev metod

- Menja se svaki od elemenata date matrice srednjom vrednošću četiri suseda u **svakom koraku iteracije** do postizanja konvergencije:

```
while (not converged) {
    for (int i=1; i<rows-1; i++)
        for (int j=1; j<cols-1; j++)
            buff[i*cols+j] = 0.25f*(data[(i+1)*cols+j] + data[i*cols+j-1]
                            + data[i*cols+j+1] + data[(i-1)*cols+j]);

    memcpy(data,buff,rows*cols*sizeof(float));
}
```

- Granične vrednosti su konstantne:

```
x[0][j]
x[n-1][j]
x[i][0]
x[i][n-1]
```

Izvor: https://parallelprogrammingbook.org/

Primer – Jacobijev metod

x[i][j] posle **25** iteracija

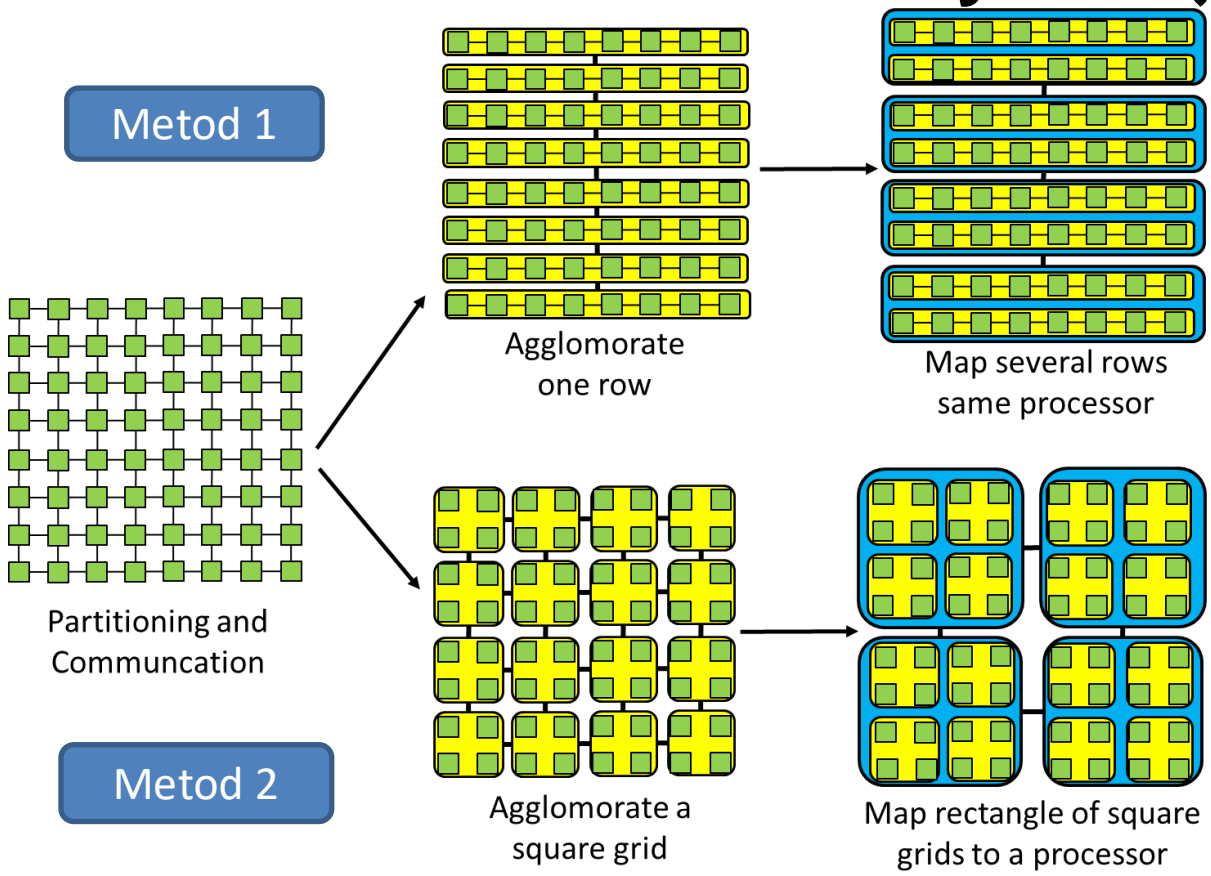
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
3	1.0	0.2	–0.1	–0.3	–0.3	–0.3	–0.3	–0.1	0.2	1.0	3
3	1.6	0.9	0.5	0.4	0.3	0.3	0.4	0.5	0.9	1.6	3
2	1.6	1.2	0.9	0.8	0.7	0.7	0.8	0.9	1.2	1.6	2
2	1.6	1.3	1.1	1.0	0.9	0.9	1.0	1.1	1.3	1.6	2
2	1.5	1.3	1.1	1.0	1.0	1.0	1.0	1.1	1.3	1.5	2
1	1.1	1.0	1.0	0.9	0.9	0.9	0.9	1.0	1.0	1.1	1
1	0.9	0.8	0.7	0.7	0.7	0.7	0.7	0.7	0.8	0.9	1
1	0.6	0.5	0.4	0.3	0.3	0.3	0.3	0.4	0.5	0.6	1
0	0.1	0.0	0.0	–0.1	–0.1	–0.1	–0.1	0.0	0.0	0.1	0
0	–0.3	–0.5	–0.5	–0.5	–0.5	–0.5	–0.5	–0.5	–0.5	–0.3	0
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

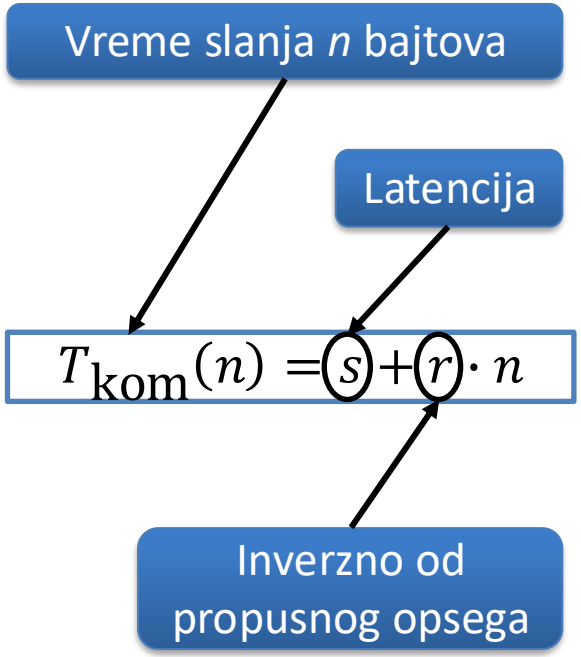
x[i][j] posle **75** iteracija

–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
3	0.9	0.1	–0.3	–0.4	–0.5	–0.5	–0.4	–0.3	0.1	0.9	3
3	1.5	0.7	0.3	0.1	0.0	0.0	0.1	0.3	0.7	1.5	3
2	1.5	1.0	0.6	0.4	0.3	0.3	0.4	0.6	1.0	1.5	2
2	1.5	1.1	0.8	0.6	0.5	0.5	0.6	0.8	1.1	1.5	2
2	1.4	1.0	0.8	0.6	0.5	0.5	0.6	0.8	1.0	1.4	2
1	1.0	0.8	0.6	0.5	0.5	0.5	0.5	0.6	0.8	1.0	1
1	0.8	0.6	0.4	0.4	0.3	0.3	0.4	0.4	0.6	0.8	1
1	0.5	0.3	0.2	0.1	0.1	0.1	0.1	0.2	0.3	0.5	1
0	0.0	–0.1	–0.2	–0.2	–0.3	–0.3	–0.2	–0.2	–0.1	0.0	0
0	–0.4	–0.5	–0.6	–0.6	–0.6	–0.6	–0.6	–0.6	–0.5	–0.4	0
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

Izvor: https://parallelprogrammingbook.org/

Paralelne šeme za Jacobijev metod



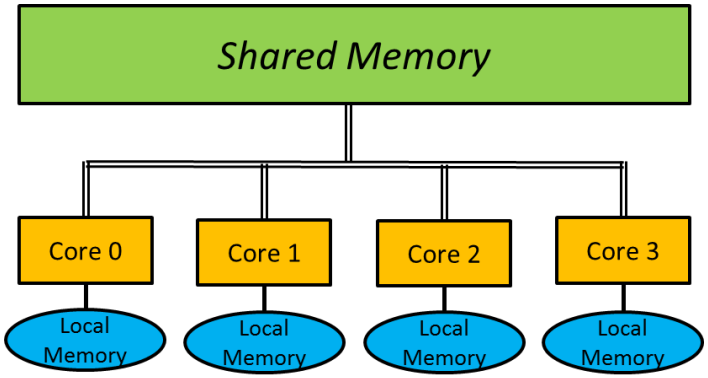


- **Metod 1:** komunikacija između dva procesa (linearni niz) $\approx 2(s + r \cdot n)$
- **Metod 2:** komunikacija između dva procesa (2D meš) $\approx 4\left(s + r\left(\frac{n}{\sqrt{p}}\right)\right)$
- Drugi metod bolji za veliko p pošto se vreme komunikacije smanjuje sa p, dok kod prvog metoda ostaje konstatno

Izvor: https://parallelprogrammingbook.org/

Arhitektura paralelnih sistema

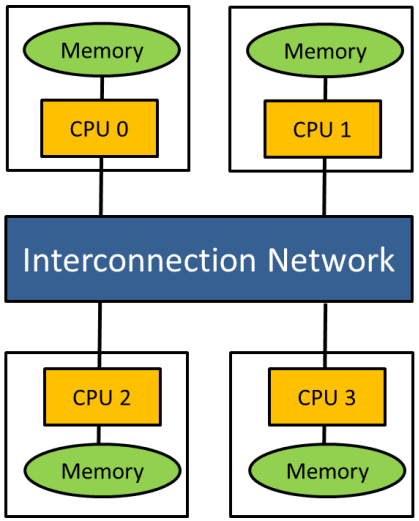
Sistemi sa deljenom memorijom

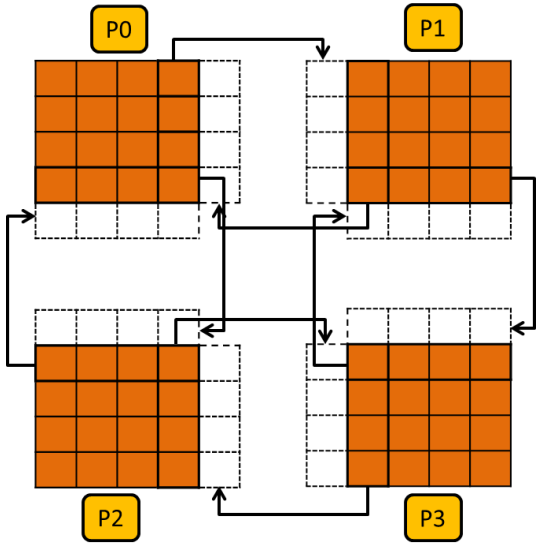


- **Sva jezgra imaju pristup deljenoj memoriji** preko **zajedničke magistrale** (engl. *shared memory*) – primer višejezgarni CPU
- Pored deljene memorije, svako od jezgara može imati svoju manju lokalnu memoriju (npr. keš) kako bi se smanjio broj skupih memorijskih operacija (tzv. *von-Neumann bottleneck*)
  - Savremeni višejezgarni CPU sistemi imaju podršku za ostvarivanje koherentnosti keša (engl. *cache cocherence*) – engl. *ccNUMA: cache coherent non-uniform access architectures*
- Najvažniji programski modeli: **višenitni C++11, OpenMP, CUDA**

Izvor: https://parallelprogrammingbook.org/

Sistemi sa slanjem poruka





- **Svaki čvor** ima **sopstvenu privatnu memoriju**. Procesori eksplicitno komuniciraju slanjem poruka putem **sprežne mreže**
  - Najpopularniji standard: **MPI** (npr. MPI_Send, MPI_Recv, MPI_Bcast, MPI_Reduce)
- Primer su računarski (Beowulf) klasteri
  - Skup klasičnih računara povezanih **sprežnom mrežom** (engl. *interconnection network*) (npr. Ethernet ili Infiniband)

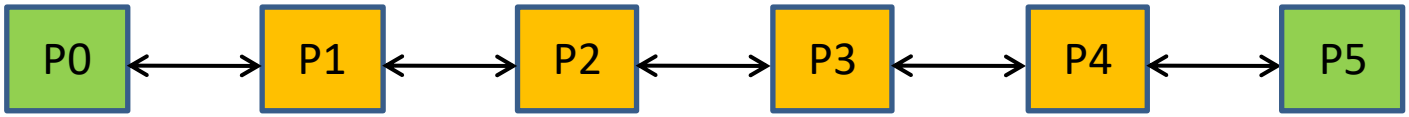
Izvor: https://parallelprogrammingbook.org/

Sprežne mreže

Topologije mreža

- Paralelni računarski sistemi imaju **sprežne mreže** (engl. *interconnection network*) kao važnu komponentu arhitekture
- Dva glavna tipa:
  - **Deljene** (engl. *shared*), primet Ethernet gde se prenosi samo jedna poruka u bilo kom trenutku, mana je ograničena skalabilnost
  - **Prekidačke** (engl. *switched*), mogu istovremeno prenositi više poruka između različitih čvorova, osnova za sve paralelne sisteme visokih performansi sa distribuiranom memorijom
  - **Topologija mreže** je ključna za skalabilnost i performanse paralelne računarske arhitekture
  - Mreža se predstavlja kao **povezani graf**, parametri mreže su **stepen** (engl. *degree*), **prečnik** (engl. *diameter*) i **širina bisekcije** (engl. *bisection width*), željeni cilj – **konstantni stepen**, **mali prečnik** i **velika širina bisekcije**

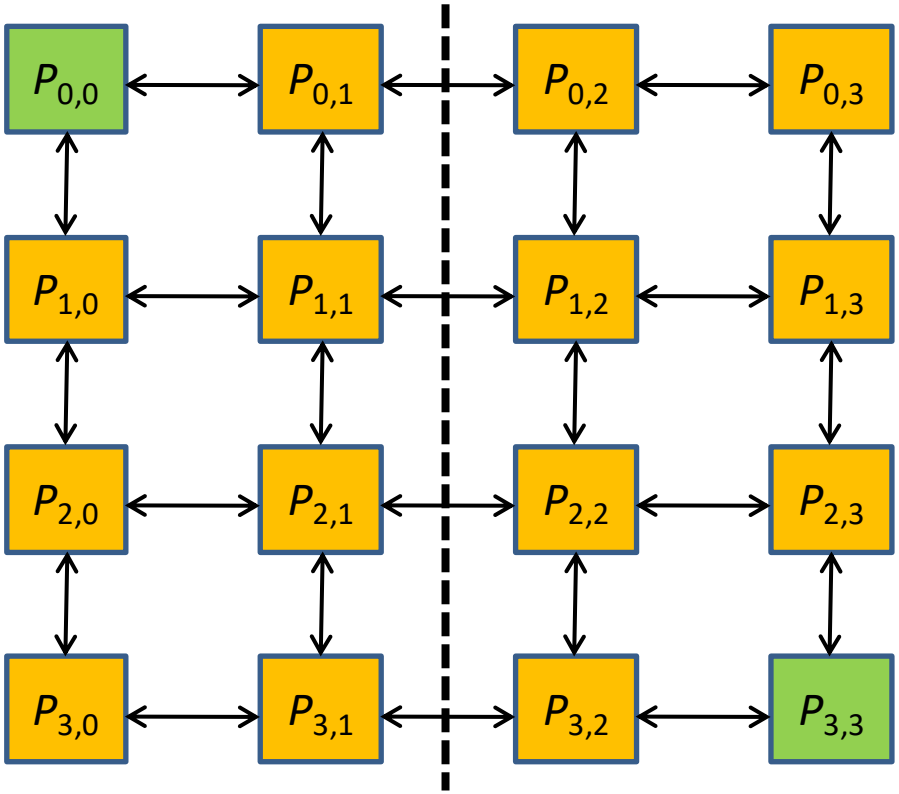
Linearni niz



- **Linearni niz** (engl. *linear array*) L_n sa n procesora
- **Stepen** mreže je maksimalni broj suseda bilo kog procesora
  - $stepen(L_n) = 2$
- **Prečnik** mreže je maksimalna udaljenost između bilo kog para procesora u mreži
  - $prečnik(L_n) = n-1$
- **Širina bisekcije** mreže je minimalni broj potega koje treba ukloniti kako bi se mreža razdvojila na dve dela iste veličine
  - $širina\_bisekcije(L_n) = 1$

Izvor: https://parallelprogrammingbook.org/

2D meš



- Dvodimenzionalni (**2D**) **meš** $M(d, d)$ ima $n = d^2$ procesora
  - *stepen*$(M(d, d)) = 4$

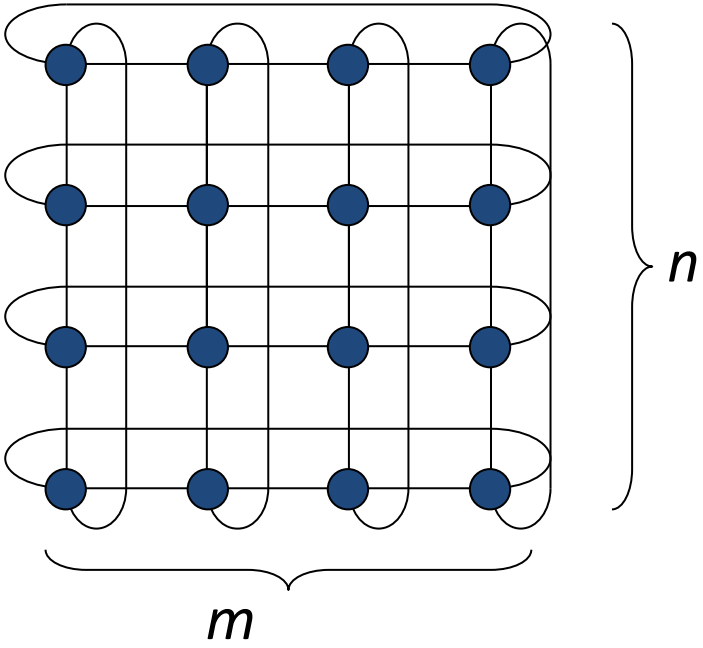
    $\mathbf{O(1)}$
  - *prečnik*$(M(d, d)) = 2(d-1)$

    $\mathbf{O(\sqrt{n})}$
  - *širina_bisekcije*$(M(d,d)) = d$

    $\mathbf{O(\sqrt{n})}$

Izvor: https://parallelprogrammingbook.org/

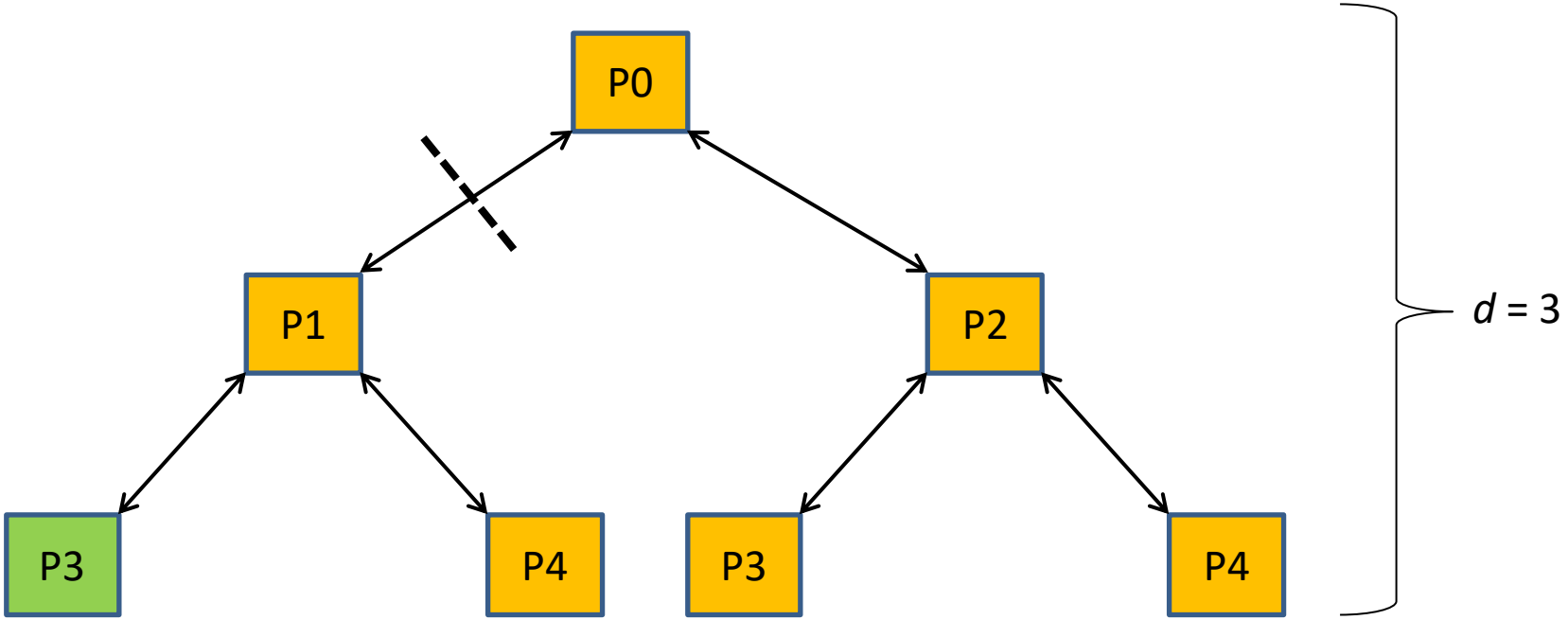
2D torus



- **Torus** $T(c,d)$ je meš proširen „obmotavajućim“ potezima na granici meša
- $T(c, d)$ ima $\boldsymbol{n = c \cdot d}$ procesora
  - *stepen*$(T(c, d)) = 4$
  - *prečnik*$(T(c, d)) = d/2 + c/2$
  - *širina_bisekcije*$(T(c, d)) = \min\{2 \cdot c, 2 \cdot d\}$

Izvor: https://parallelprogrammingbook.org/

Binarno stablo

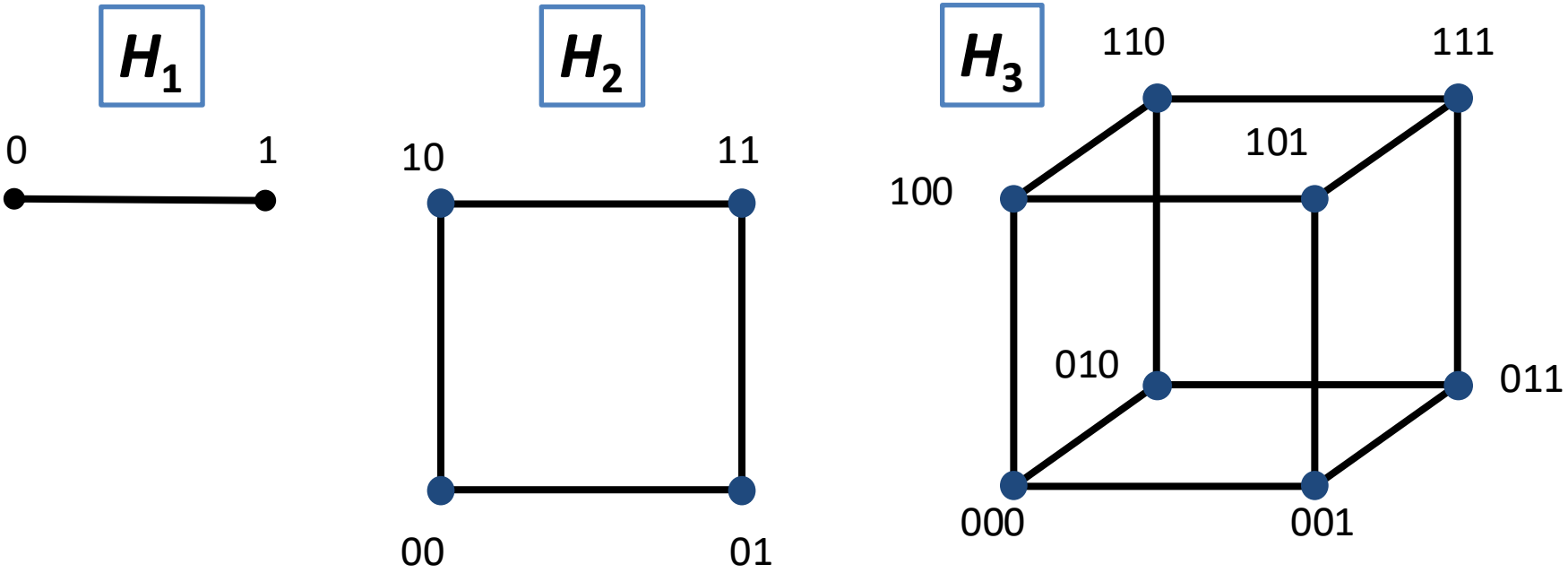


- *BS*(*d*) ima $n = 2^d - 1$ procesora
  - *stepen*(*BT*(*d*)) = 3 **O(1)**
  - *prečnik*(*BT*(*d*)) = 2(*d*–1) **O(log(*n*))**
  - *širina_bisekcije*(*BT*(*d*)) = 1 **O(1)**

Izvor: https://parallelprogrammingbook.org/

Hiperkocka

- ***Hiperkocka***, u oznaci H_d ($d \geq 1$), je graf čiji čvorovi predstavljaju 2^d bitova stringa dužine d
- Dva čvora su **susedna** ako i samo ako se bit stringovi koje predstavljaju razlikuju u **tačno** jednoj bitskoj poziciji

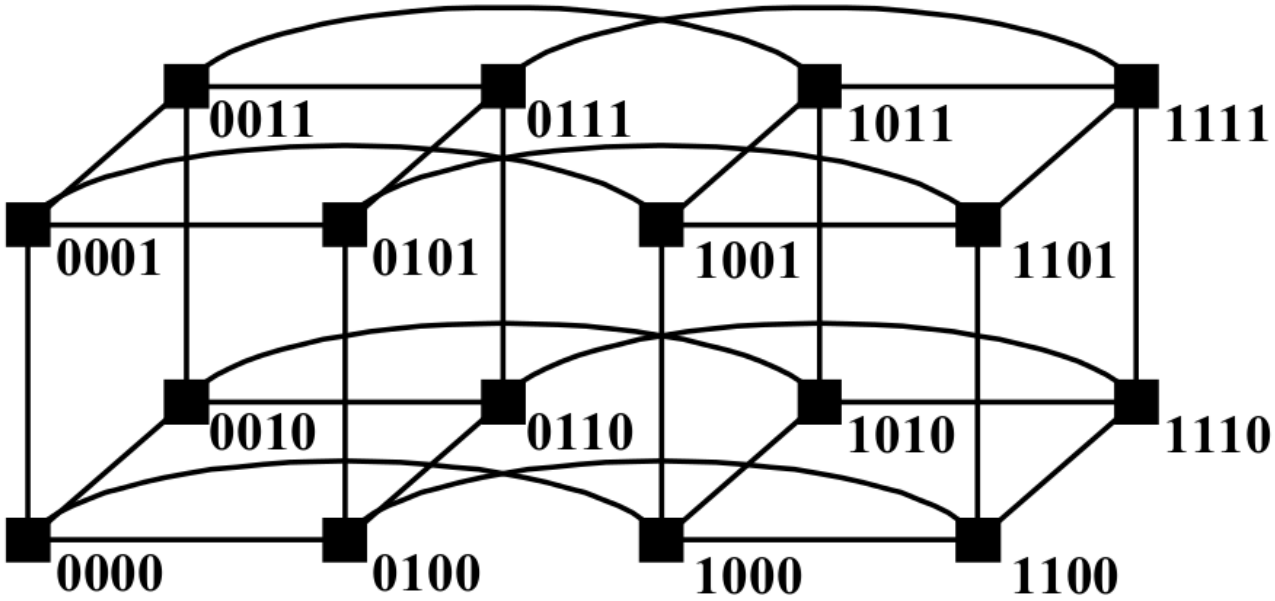


Izvor: https://parallelprogrammingbook.org/

Hiperkocka

H,K(d) = hiperkocka stepena d, broj procesora: $n = 2^d$

- *stepen*(*HK*(d)) = d **O(log(*n*))**
- *prečnik*(*HK*(d)) = d **O(log(*n*))**
- *širina_bisekcije*(*HK*(d)) = $n/2$ **O(*n*)**



Izvor: https://parallelprogrammingbook.org/

Kriterijumi za ocenu topologija

topologija	stepen	prečnik	širina bisekcije
Linearni niz	$O(1)$	$O(n)$	$O(1)$
2D meš/torus	$O(1)$	$O(\sqrt{n})$	$O(\sqrt{n})$
3D meš/torus	$O(1)$	$O(\sqrt[3]{n})$	$O(n^{2/3})$
Binarno stablo	$O(1)$	$O(\log(n))$	$O(1)$
Hiperkocka	$O(\log(n))$	$O(\log(n))$	$O(n)$

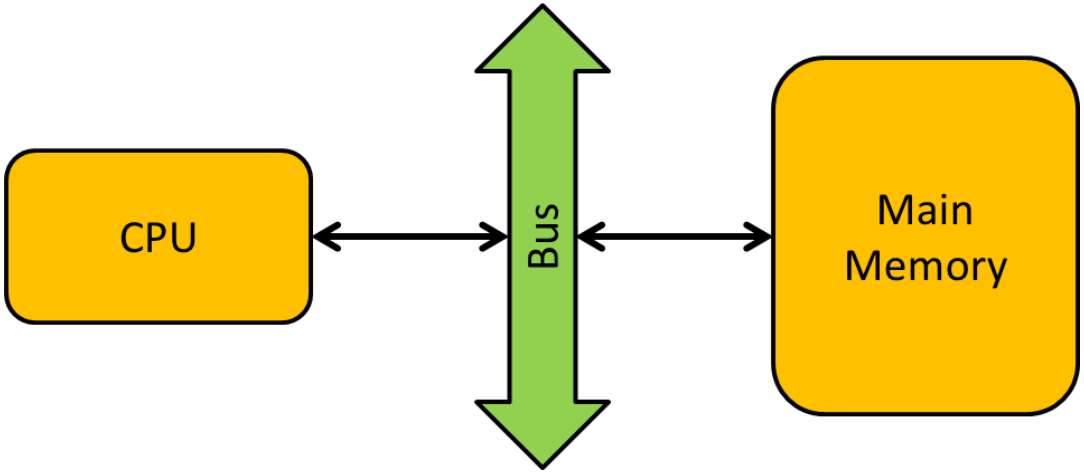
- **Mali prečnik** kako bi komunikacija između bilo koja dva procesora bila efikasna
- **Velika širina bisekcije**:
  - Mala bisekciona širina može usporiti mnogo operacija kolektivne komunikacija i s toga ozbiljno ograničiti performanse aplikacija. Postizanje velike širine bisekcije može zahtevati stepen mreže koji nije konstanta
- **Konstantni stepen** (nezavisan od veličine mreže) omogućava da mreža skalira na veliki broj čvorova bez potrebe za dodavanjem preteranog boja veza

Izvor: https://parallelprogrammingbook.org/

Hijerarhija memorije

Klasična von Neumannova arhitektura

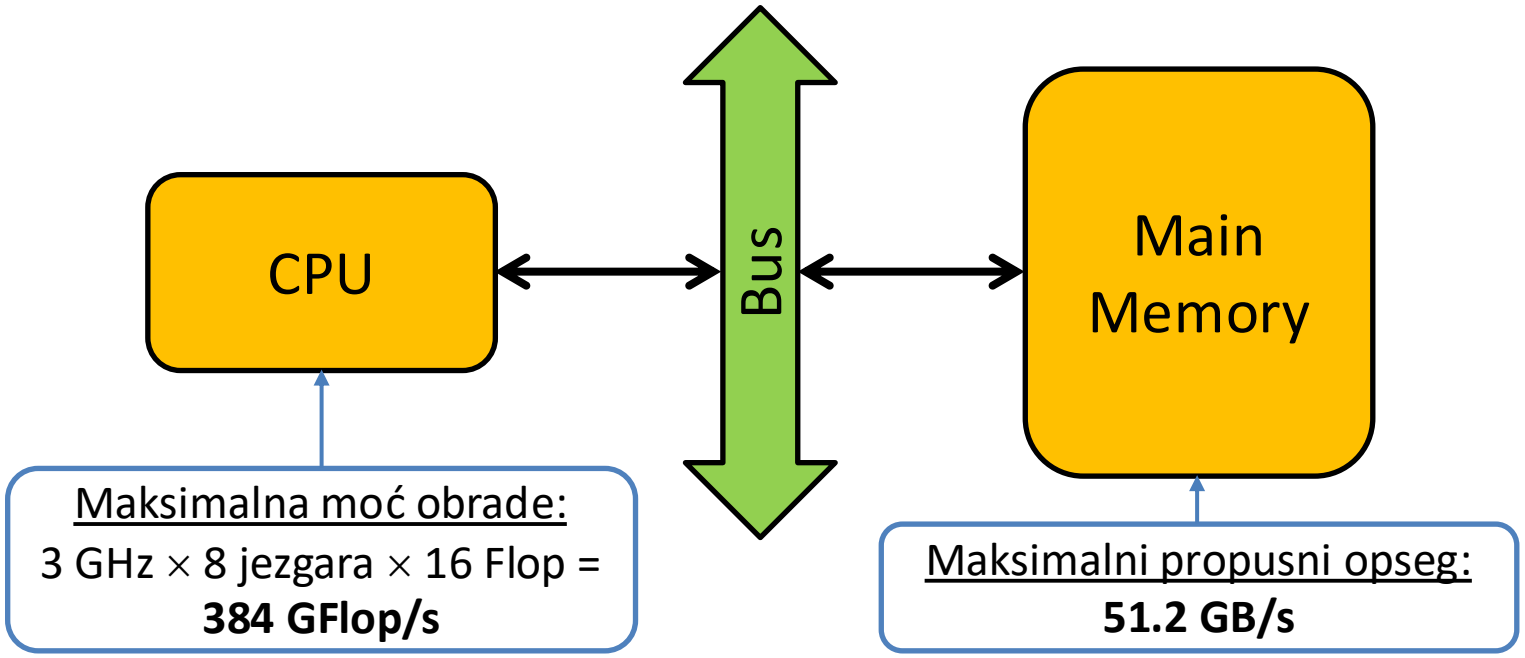
- Kod ranih digitalnih računara, **vremena** potrebna **za pristup memoriji** i **izračunavanja** bila su **istog reda veličina**
- **Brzina izračunavanja rasla** je **mnogo brže** od brzine pristupa memoriji što je dovelo do **velike razlike u performansama ove dve operacije**
- **von Neumannovo usko grlo** (engl. *bottleneck*) je **razlika između brzine izračunavanja** procesora (CPU) i **brzine pristupa glavnoj memoriji** (DRAM)



Izvor: https://parallelprogrammingbook.org/

von Neumannovo usko grlo

- Uprošćen model kako bi se utvrdila **gornja granica performansi** kod izračunavanja **skalarnog proizvoda** (engl. *dot product*) dva vektora *u* i *v*, koji sadrže po *n* brojeva u dvostrukoj preciznosti smeštenih u glavnoj memoriji
- Model daje **teoretski maksimum**



Izvor: https://parallelprogrammingbook.org/

Primer – performanse skalarnog proizvoda

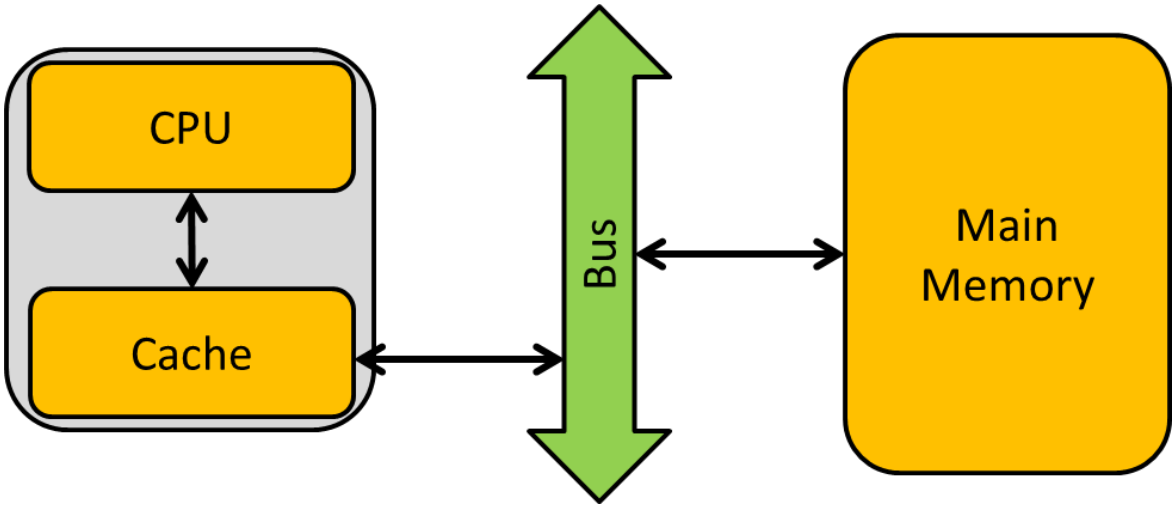
```
// skalarni proizvod dva vektora
double dotp = 0.0;
for (int i = 0; i<n; i++)
    dotp += u[i] * v[i];
```

- Primer: $n = 2^{30}$
- Vreme izračunavanja: $\boldsymbol{t_{comp}} = \frac{2\ \text{GFlop}}{384\ \text{GFlop/s}} = 5.2\ \text{ms}$
  - Ukupno operacija: $2 \cdot n = 2^{31}$ Flops = 2 GFlops
- Vreme prenosa podataka: $\boldsymbol{t_{mem}} = \frac{16\ \text{GB}}{51.2\ \text{GB/s}} = 312.5\ \text{ms}$
  - Ukupno podataka za prenos: $2 \cdot 2^{30} \cdot 8\ \text{B} = 16\ \text{GB}$
- Vreme izvršavanja: $\boldsymbol{t_{exec}} \geq max(5.2\ \text{ms}, 312.5\ \text{ms}) = 312.5\ \text{ms}$
  - Postignute performanse: $\frac{2\ \text{GFlop}}{312.5\ \text{ms}} = \mathbf{6.4\ GFlop/s}$ (<2% od teoretskog maksimuma)
- Zaključak: skalarni proizvod je **memorijski ograničen** (engl. *memory bound*)

Izvor: https://parallelprogrammingbook.org/

Struktura procesora sa jednim kešom

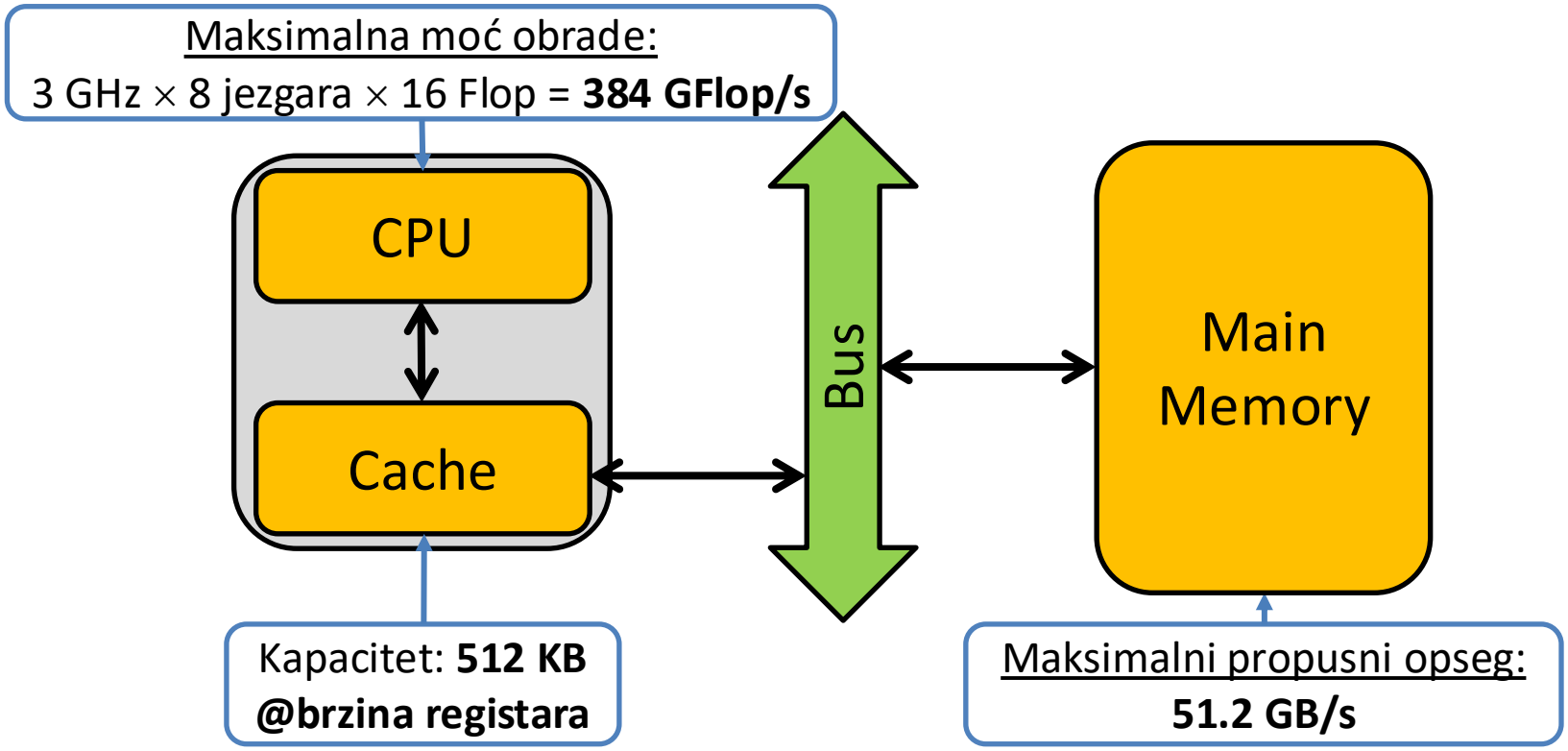
- Procesori uglavnom imaju **hijerarhiju sa tri nivoa keša** (L1, L2, L3)
  - Grafički procesori imaju dva nivoa keša
- **Veći propusni opseg** (engl. *bandwidth*) i **kraća latencija** u odnosu na glavnu memoriju, ali i značajno **manji kapacitet**
- Razmena kapaciteta za brzinu i obratno
  - Npr. L1 keš je mali, ali brz, dok je L3 keš relativno veliki, ali spor
- Keševi mogu biti privatni za jedno jezgro ili deljeni između više jezgara



Izvor: https://parallelprogrammingbook.org/

Keš memorija – Primer

- Uprošćen model kako bi se utvrdila **gornja granica performansi** kod izračunavanja proizvoda matrica $W = U \times V$, pri čemu je svaka od matrica veličine $n \times n$ i čuva se u glavnoj memoriji – model daje **teoretski maksimum**



Izvor: https://parallelprogrammingbook.org/

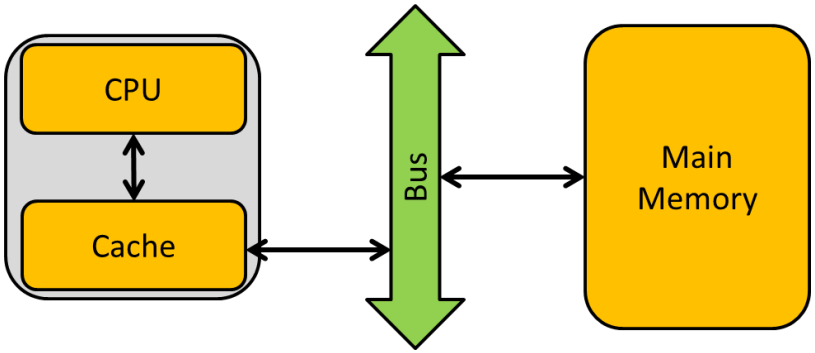
Primer – performanse množenja matrica

```
// mnozenje dve matrice
for (int i = 0; i<n; i++)
    for (int j = 0; j < n; j++) {
        double dotp = 0;
        for (int k = 0; k<n; k++)
            dotp += U[i][k]*V[k][j];
        W[i][j] = dotp;
    }
```

- Primer: $n = 128$
- Vreme prenosa podataka: $\boldsymbol{t_{\mathrm{mem}}} = \frac{384\ \mathrm{KB}}{51.2\ \mathrm{GB/s}} = 7.5\ \mu\mathrm{s}$
  - Prenos podataka (iz/ka kešu): $n = 128$: $128^2 \times 3 \times 8\mathrm{B} = 384$ KB (staje u keš)
- Vreme izračunavanja: $\boldsymbol{t_{\mathrm{comp}}} = \frac{2^{22}\ \mathrm{Flop}}{384\ \mathrm{GFlop/s}} = 10.4\ \mu\mathrm{s}$
  - Ukupno operacija: $2 \cdot n^3 = 2 \cdot 128^3 = 2^{22}$ Flops
- Vreme izvršavanja: $\boldsymbol{t_{\mathrm{exec}}} \geq 7.5\ \mu\mathrm{s} + 10.4\ \mu\mathrm{s} = 17.9\ \mu\mathrm{s}$
  - Postignute performanse: $\frac{2^{22}\ \mathrm{Flop}}{17.9\ \mu\mathrm{s}} = \mathbf{223\ GFlop/s}$ (≈ 60% od teoretskog maksimuma)
- Dosta podataka se koristi više puta prilikom množenja matrica. Šta se dešava ako matrice ne staju u keš?

Izvor: https://parallelprogrammingbook.org/

Algoritmi za keširanje



Koji podaci se učitavaju iz glavne memorije?

Gde u kešu smeštamo podatke?

Ako je keš pun, koje podatke izbacujemo?

- **Korisnik ne upravlja eksplicitno sadržajem keš memorija**
- Kešom se upavlja putem skupa keš polisa (**algoritama za keširanje**) koje određuju koji podaci se keširaju prilikom izvršavanja programa
- **Pogodak keša** (engl. *cache hit*): zahtev za podacima se može opslužiti čitanjem iz keša bez potrebe za prenosom iz glavne memorije
- **Promašaj keša** (engl. *cash miss*): u suprotnom
- **Procenat pogodaka** (engl. *hit ratio*): procenat zahteva za podacima koji dovode do pogotka keša

Izvor: https://parallelprogrammingbook.org/

Algoritmi za keširanje – prostorna lokalnost

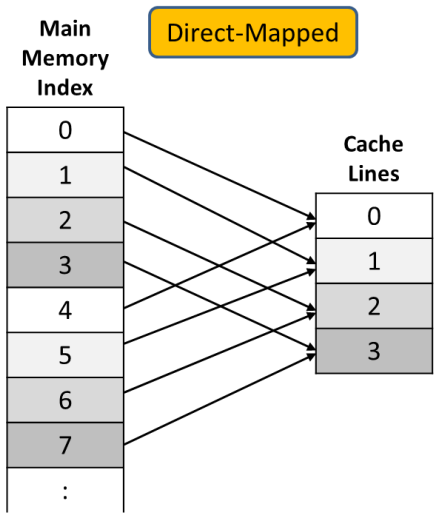
Koji podaci se učitavaju iz glavne memorije?

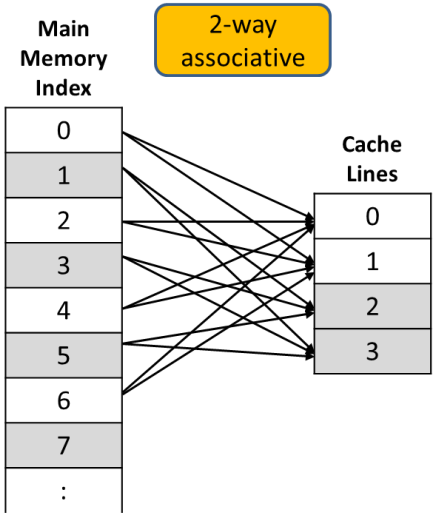
```
//maksimum niza (elementi smesteni kontinualno)
for (i = 0; i<n; i++)
    maximum = max(a[i], maximum);
```

- **Keš linija**: više pojedinačnih podataka kao jedna memorijska lokacija
- Umesto zahteva za pojedinačnim podatkom, čitava keš linija se puni vrednostima sa susednih adresa
- **Primer:** Keš linija veličine 64 B, vrednosti dvostruke preciznosti
  - Prva iteracija: traži se `a[0]`, što dovodi do promašaja keša
  - Osam uzastopnih vrednosti `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]`, `a[5]`, `a[6]`, `a[7]` se pune na istu keš liniju
  - Sledećih sedam iteracija će onda rezultovati pogocima keša
  - Tek zahtev za `a[8]` će ponovo dovesti do promašaja keša, itd.
  - Ukupni **procenat pogodaka keša** u posmatranom primeru je 87.5%

Izvor: https://parallelprogrammingbook.org/

Algoritmi za keširanje – vremenska lokalnost





Gde u kešu smeštamo podatke?

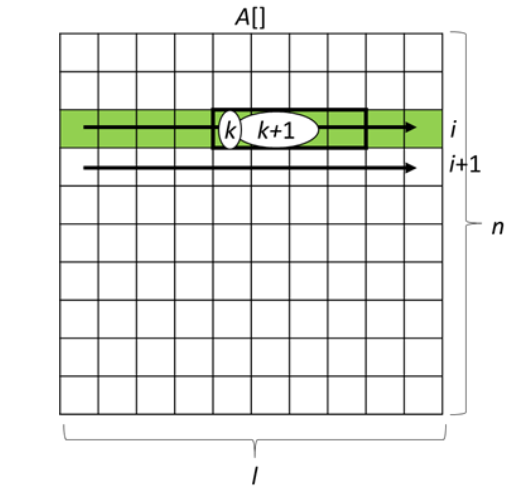
Ako je keš pun, koje podatke izbacujemo?

- Keš se organizuje u veći broj **keš linija**
- Strategija za mapiranje keša određuje u koju lokaciju keša će kopija određene vrednosti iz glavne memorije biti smeštena
- **Direktno-mapirani keš**: svaki blok iz glavne memorije može se smestiti na tačno jednu keš liniju (veliki procenat promašaja)
- **Asocijativni keš sa *n*-linija**: svaki blok iz glavne memorije može se smestiti na jednoj od *n* mogućih keš linija (veći procenat pogodaka uz povećanu složenost)
- **Najdavnije korišćen** (engl. *Least Recently Used – LRU*): često korišćena polisa zasnovana na vremenskoj lokalnosti

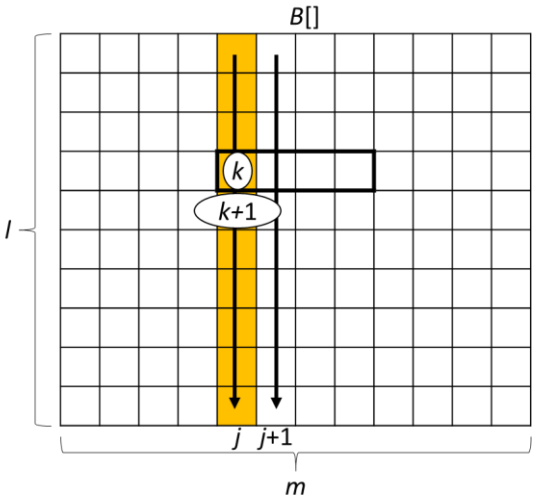
Izvor: https://parallelprogrammingbook.org/

Optimizacija pristupa kešu – „naivni“ pristup

```
//“naivno“ mnozenje matrica
for (int i = 0; i<n; i++)
    for (int j = 0; j < m; j++) {
        float acc = 0;
        for (k = 0; k < l; k++)
            acc += A[i*l+k]*B[k*n+j];
        C[i*m+j] = acc;
    }
```



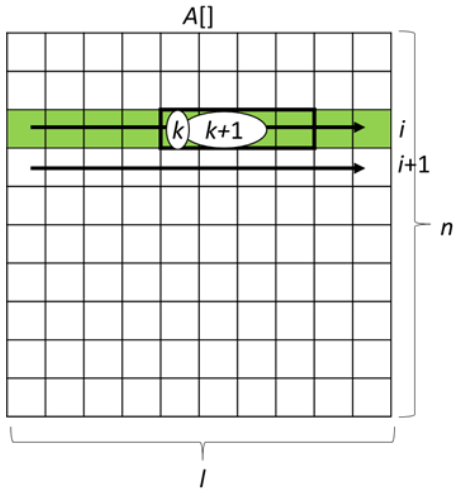
- Množenje matrica: $A_{n\times l} \cdot B_{l\times m} = C_{n\times m}$
- Matrice se čuvaju po vrstama u linearnoj memoriji
- Obrazac pristupa *A* – **kontinualan**: (*i*,*k*), (*i*,*k*+1)
- Obrazac pristupa *B* – **nekontinualan**: (*k*,*j*), (*k*+1,*j*)
  - Udaljeni *l*×sizeof(float) u glavnoj memoriji, ne smeštaju se na istu keš liniju
  - Moguće izbacivanje keš linije iz L1 keša, mali procenat pogodaka za veliko *l*

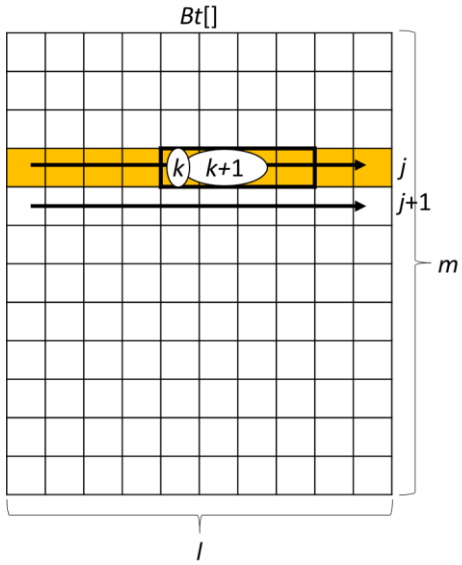


Izvor: https://parallelprogrammingbook.org/

Optimizacija pristupa kešu – „T-i-P“ pristup

```
// transponuj-i-mnozi
for (k=0; k<l; k++)
    for (j = 0; j<m; j++)
        Bt[i*l+k] = B[k*n+j];
for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
        float acc = 0;
        for (k=0; k<l; k++)
            acc += A[i*l+k] * B[j*l+k];
        C[i*m + j] = acc;
    }
```





- **Transponuj-i-množi** („T-i-P“) pristup:
  1. $Bt_{m\times l} = (B_{l\times m})^T$
  2. $A_{n\times l} \cdot Bt_{m\times l} = C_{n\times m}$
- Obrazac pristupa *A* – **kontinualan**: (*i*,*k*), (*i*,*k*+1)
- Obrazac pristupa *Bt* – **kontinualan**: (*j*,*k*), (*j*,*k*+1)

Izvor: https://parallelprogrammingbook.org/

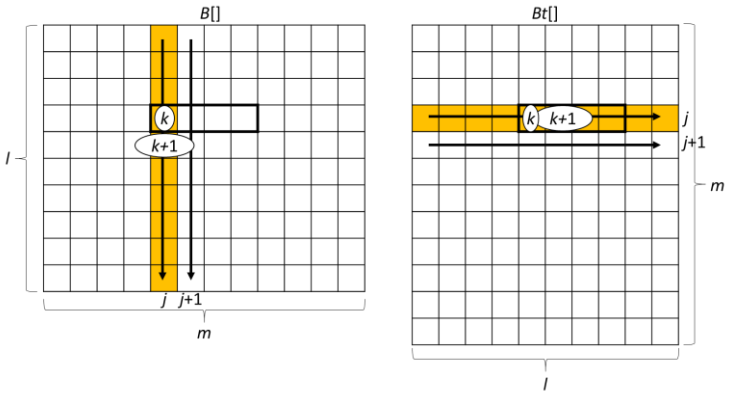
Optimizacija pristupa kešu

Vreme izvršavanja na i7-6800K za $m = n = l = 2^{13}$
`#elapsed time (naive_mult):  5559.5s`
`#elapsed time (transpose):  0.8s`
`#elapsed time (transpose_mult):  497.9s`

Ubrzanje: 11.1×

Vreme izvršavanja na i7-6800K za $m = n = 2^{13}$, $l = 2^8$
`#elapsed time (naive_mult):  28.1s`
`#elapsed time (transpose):  0.01s`
`#elapsed time (transpose_mult):  12.9s`

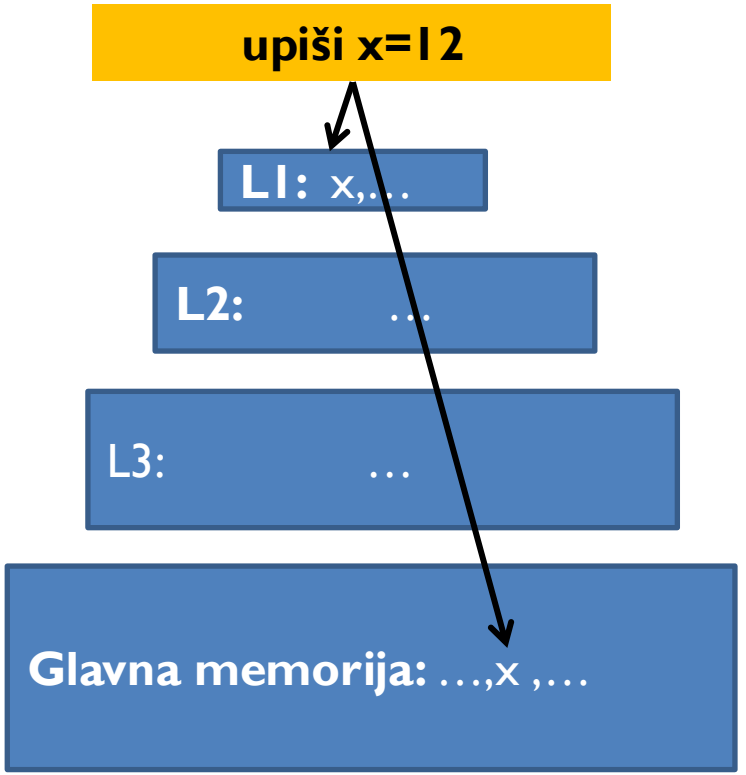
Ubrzanje: 2.2×



Izvor: https://parallelprogrammingbook.org/

Pravila upisa u keš

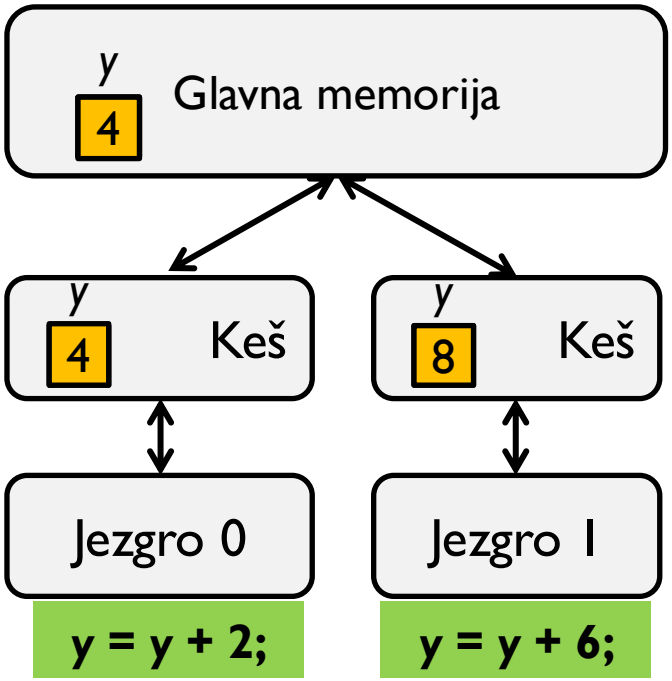
- Kada procesor upiše podatak u keš, ta vrednost može postati **nekonzistentna** sa vrednošću u glavnoj memoriji
- Pravilo **piši-kroz** (*write-through*)
  - Podatak se ažurira u glavnoj memoriji istovremeno sa njegovim upisom u keš
- Pravilo **piši-nazad** (*write-back*)
  - U kešu se podatak obeleži kao „prljav“ (engl. *dirty*)
  - Kada se keš linija zameni novom keš linijom iz memorije, „prljava“ linija se upisuje u glavnu memoriju



Izvor: https://parallelprogrammingbook.org/

Problem koherentnosti keša – Primer

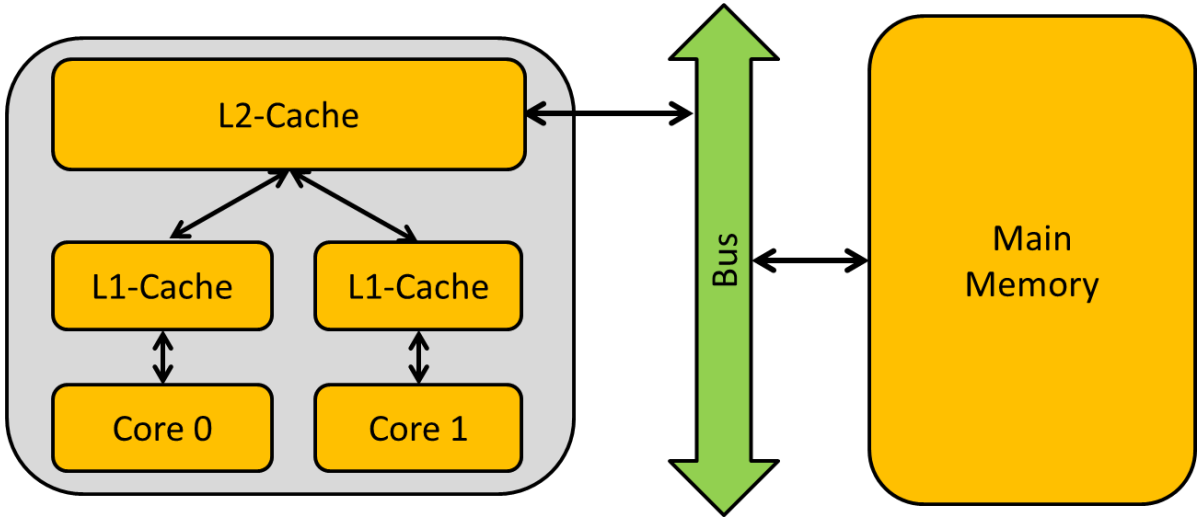
- **Nekonzistentnost keša** se javlja kada dve keš memorije čuvaju različite vrednosti za isti podatak



Izvor: https://parallelprogrammingbook.org/

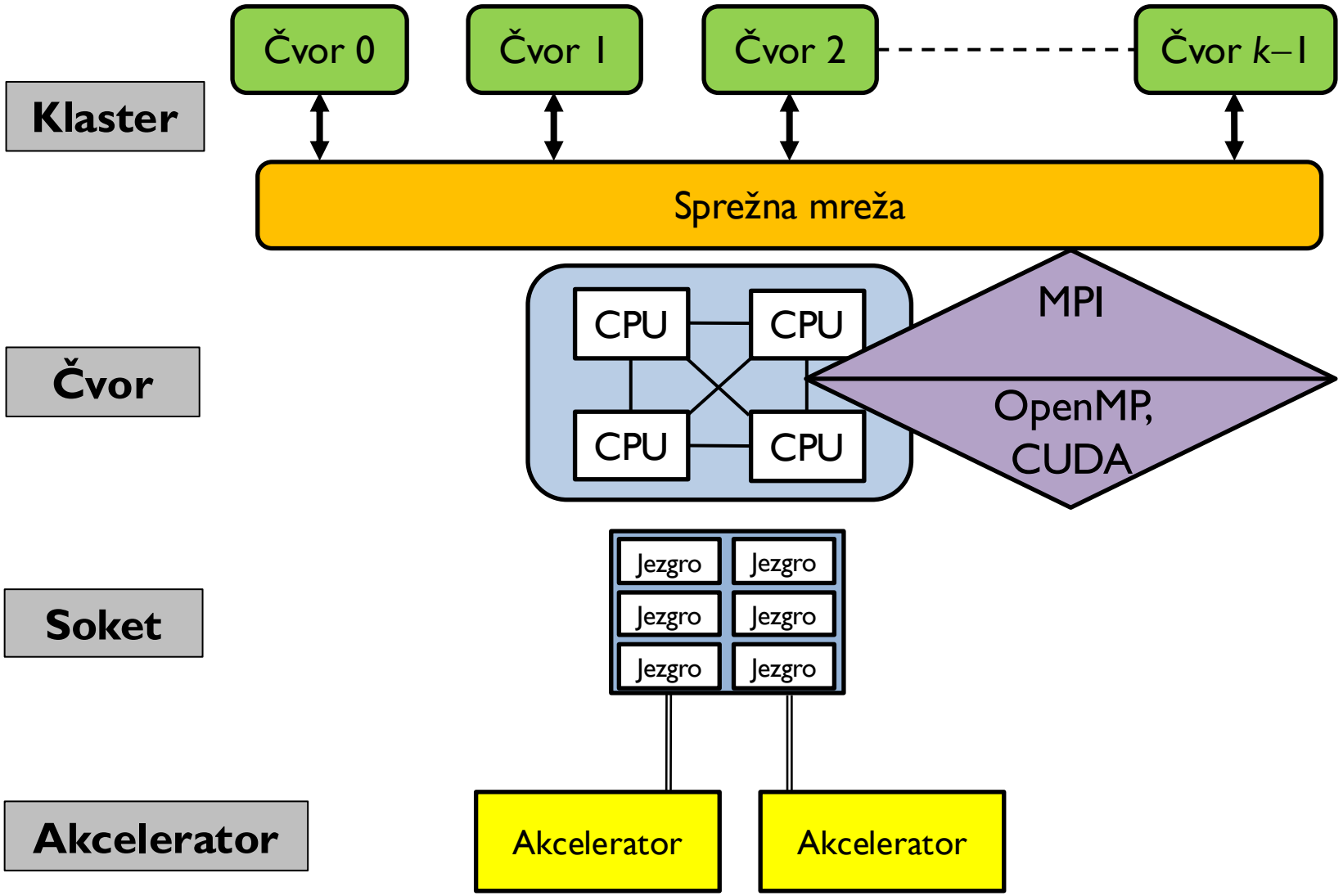
Koherentnost keša

- Savremeni višejezgarni procesori sadrže više nivoa keš memorija
  - Svako jezgro ima mali i brzi privatni keš nižeg nivoa
  - Sva jezgra dele veći, ali sporiji zajednički keš višeg nivoa
- Može postojati više kopija deljenih podataka
  - npr. jedna kopija u L1 kešu jezgra 0 i jedna kopija u L1 kešu jezgra 1
- **Nekonzistentost keša** (engl. *cache inconsistency*)
  - Ako jezgro 0 upiše na pridruženu keš liniju, ažuriraće se samo vrednost u L1 kešu jezgra 0, ali ne i jezgra 1
- Zbog prethodnog su neophodni **protokoli za održavanje koheretnosti keša**



Izvor: https://parallelprogrammingbook.org/

Arhitektura heterogenih paralelnih sistema



Izvor: https://parallelprogrammingbook.org/