



УНИВЕРЗИТЕТ МБ  
ПОСЛОВНИ И ПРАВНИ ФАКУЛТЕТ БЕОГРАД

# Паралелно програмирање

- Уводна предавања -

Предавач: Доцент проф. др Боривоје М. Милошевић

2020

# ОСНОВНИ ПОЈМОВИ

Prirodne nauke se zasnivaju na posmatranjima, teoriji i fizičkim eksperimentima. U okvirima klasičnih nauka teorija se gradi iz pretpostavke ili hipoteze koja se dokazuje empirijski (eksperimentalno) ili na osnovu prethodno utvrdjenih zakona i teorema, zbog čega je modeliranje fizičkih procesa u bilo kom obliku suštinski deo svake klasične nauke.

Savremenu nauku karakteriše pristup modernim dostignućima modeliranja, gde su numeričke simulacije najzastupljenije. Razlog zbog kojeg numeričke simulacije imaju sve veći značaj je u uspešnoj imitaciji fizičkih eksperimenata bez kojih ranije nije bilo moguće ostvariti naučni doprinos. Možda najbitnija karakteristika primene numeričkih simulacija je rešavanje onih problema za koje je izrada opreme bila skupa ili nemoguća, ili bi dovela do narušavanja prirodne okoline i zdravlja ljudi.

Neke od naučnih oblasti u kojima numeričke simulacije imaju svakodnevnu primenu su:

- Relativistička fizika i statisticka mehanika;
- Dinamička astronomija, astrofizika i kosmologija;
- Dinamika fluida i turbulencije;
- Nanotehnologije
- Dizajn materijala i superprovodljivost;
- Biologija, farmakologija, genetski inženjering;
- Medicina i modeliranje čovekovih organa i tkiva (npr. kostiju);
- Meteorologija i modeliranje prirodnih procesa u atmosferi.

# ОСНОВНИ ПОЈМОВИ

Prilikom projektovanja i implementacije numeričkih simulacija sve više se susrećemo sa velikim brojem složenih problema čije rešavanje zahteva dobro opremljene racčunare. U nekim situacijama izvršavanje programa isuviše dugo traje, i zato se postavlja pitanje na koji način ubrzati njegov rad. Odgovor leži u konstruisanju paralelnih algoritama za izvršavanje na specijalnim (paralelnim) arhitekturama računarskih sistema.

Pre konstruisanja paralelnog algoritma potrebno je, pored poznavanja softvera, upoznati se i sa tipom arhitekture paralelnog računara na kojem će se dati algoritam implementirati. Pod softverskim znanjem podrazumeva se poznavanje nadogradnje postojećeg programskog jezika ili poznavanje paralelno-programskog jezika specijalno namenjenog za implementaciju paralelnih algoritama. Kao što smo naveli, za pisanje i izvršavanje paralelnog programa potrebno je imati i osnovnu informaciju o tipu mašine na kojoj će izvršavati program, jer nas ova informacija najvipše upućuje na stil paralelizacije koji ćemo primeniti.

Za dva događaja kažemo da su konkurentna ako se realizuju u okviru istog vremenskog intervala. Za naredbe koje se izvršavaju na različitim računarima kazemo da se izvršavaju konkurentno ukoliko se njihova izvršavanja vremenski preklapaju.

# ОСНОВНИ ПОЈМОВИ

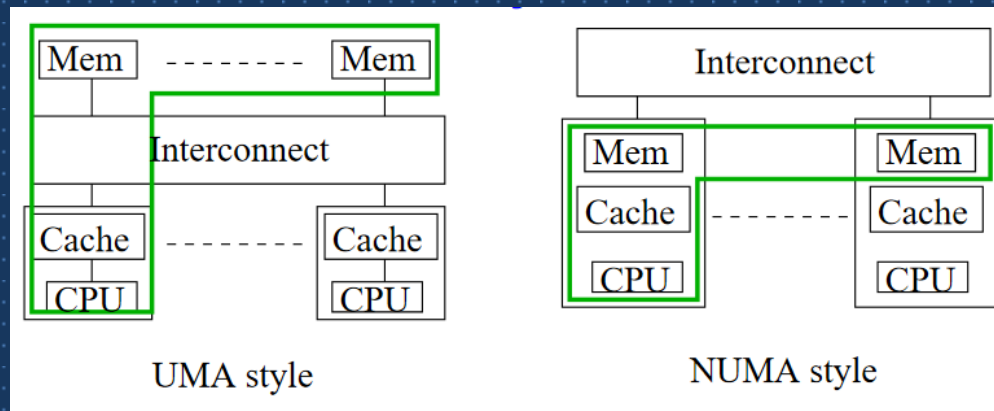
Sta je paralelni računar?

Paralelni računar je višeprosorski računar koji podržava istovremeno (konkurentno) izvršavanje dvaju ili više naredbi. Izdvajamo tri bitne kategorije paralelnih računara:

- ✓ Višeračunarski sistemi- jedan vektorski procesor (eng. multicomputers). Tradicionalni superkompjuter povezan na memorijsku magistralu visokih performansi. Višeračunarski sistem je sistem računara povezanih u računarsku mrežu putem koje procesori različitih računara međusobno komuniciraju distribuirano slanjem i primanjem poruka.
- ✓ Centralizovani višeprosorski računarski sistemi (eng. centralized multiprocessors). Centralizovani višeprosor ili 'simetrični višeprosor', je visoko integrisani računarski sistem u kojem sve centralne prosorske jedinice dele pristup globalnoj memoriji.
- ✓ Današnji superkompjuter - klaster koji se sastoji od čvorova sa po više procesora na svakom VLSI – Efekat integracije 1 M tranzistora dovoljno za potpunu funkcionalnost – Dec Alpha (RISC, devedesete) Danas se ide na više jezgara po čipu. Cena – Veliki broj prosečnih CPU-ova daje bolji odnos FLOPS/\$ u poredjenju sa tradicionalnim superkompjuterima.

# Arhitekture zajedničke memorije

*Arhitekture zajedničke memorije* u kojima se svi procesori mogu fizički obratiti celovitoj memoriji, obično sa podrškom za koherenciju keša - predmemorije (na primer, quad ili oct čip jezgra ili skuplje mašine sa desetinama ili stotinama jezgara).



Jedan adresni prostor, koji dele svi procesori. Zeleni okvir prikazuje memoriju koja se može adresirati bilo kojim procesorom. Arhitekture jedinstvenog pristupa memoriji (UMA) imaju svu memoriju "jednako udaljenu" od svih procesora. NUMA izvođenje se razlikuje zavisno od mesta podataka. NUMA se takođe naziva i Distributed Shared Memory kod koje je memorija fizički raspoređena, ali logički podeljena.

Keš predmemorije kao i obično poboljšavaju performanse, ali izazivaju konzistentnost memorije, što nije prisutno u jednostavnim sistemima sekvencijalne predmemorije: grubo govoreći, kada i kojim redosledom će ažurirati memoriju napravljenu od jednog procesora koja treba postati vidljiva drugima?

# Arhitekture zajedničke memorije

[x==0, y==0]

x = 2;                   LD R1,#2  
                              ST R1,x [to cache]

y = 1;                   LD R1,#1  
                              ST R1,y [to cache]

....

cache write-back of y -> memory

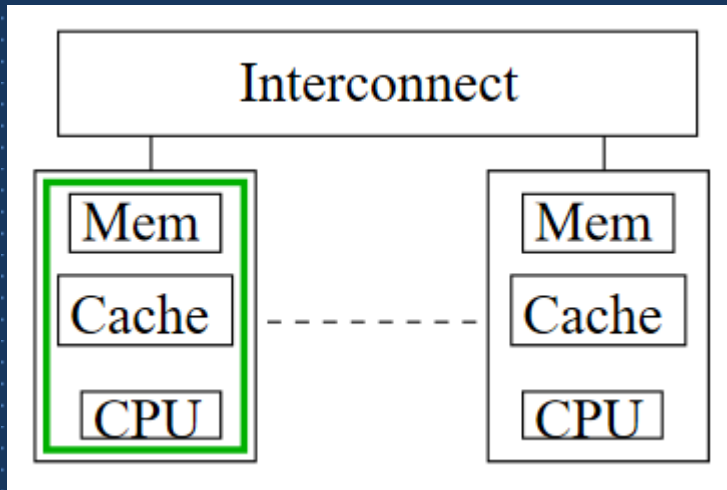
....

if (x<=y)                   LD R1,x [from cache]  
                              LD R2,y [memory back to cache]  
                              BGT R1,R2,target

print("Yes");

# Multikompjuterske Arhitekture

Isti je blok dijagram kao za NUMA deljenu memoriju! Razlika je u nedostatku bilo kakve hardverske integracije između keš predmemorije / memorijskog sistema i međusobnog povezivanja. Svaki procesor pristupa samo vlastitom fizičkom adresnom prostoru, tako da nema doslednosti. Informacije deli eksplicitno uz kooperativno prosledjivanje poruka.



Problemi sa izvodjenjem / ispravnošću uključuju semantiku sinhronizacije i ograničenja za naručivanje poruka. Na primer, uzmimo u obzir sledeće ponašanje proizvođača i potrošača.

P0

```
while (whatever)
x = ...;
send (x, P1);
}
send(-1, P1);
```

P1

```
{ recv(y, P0);
while (y!=-1) {
...= ....y....;
recv(y, P0);
}
```

# Multikompjuterske Arhitekture

NUMA arhitekture čine važnu klasu paralelnih arhitektura. Ovi veliki sistemi organizuju fizički deljenu memoriju kroz nekoliko čvorova povezanih kroz keš predmemorijske veze visokih performansi. Takvim dizajnom *niti* pristupaju memorijskim resursima smeštenim na istom čvoru kao i jezgra koja izvršava *niti* - na lokalnom čvoru ili na drugom čvoru - udaljenom čvoru.

Nekontrolisano postavljanje podataka može dovesti do nesporazuma u prometu kada sve *niti* pristupaju istoj memoriji. Nadalje, daljinski pristup podacima povećava kašnjenje memorije. Sistemi NUMA uvedeni su kao lek za probleme skalabilnosti simetričnih višeprosesora sa jedinstvenim pristupom memoriji (UMA). Nažalost, NUMA *unware* programi koji se izvode na NUMA platformama obično nemaju koristi od dodatne računarske snage i propusnosti memorije koju nude sistemi sa više čvorova.

# Multikompjuterske Arhitekture – NUMA

## -ciklična distribucija jezgra-



# NUMA APIs

Iako se gore spomenute strategije provode na nivou jezgara, korisnicima je dostupno nekoliko NUMA API-a. Operativni sistem Linux Libnuma je NUMA API za Linux. Sadrži funkcije upravljanja memorijom i procesima povezane sa sledećim politikama:

- ✓ Lokalno (Local) – dodela se izvodi na čvoru na kojem se postupak trenutno izvodi. Ovo je zadana politika i takodje se temelji na pravilima prvog dodira.
- ✓ Preferirano (Preferred) – dodela se prvo izvodi na zadanom skupu čvorova, ali ako to ne uspe, izvodi se na najbližim čvorovima.
- ✓ Vezano (Bind) - dodela se strogo izvodi na zadanom skupu čvorova. Ako dodela ne uspe, povratak nije moguć.
- ✓ Isprepleteno (Interleaved) - dodela se izvodi na round-robin način preko NUMA čvorova, na skupu odabranih stranica.

Budući da ovo deluje na granularnost veličine stranice, to je nedelotvorno ako je veličina dodeljenih podataka manja od veličine stranice. Libnuma dolazi zajedno sa numactl, alatom koji pruža iste osobine kao komandna linija. Međutim, promene kroz komand prompt primenjuju se samo na nivou celog programa.

# NUMA APIs

- ✓ *Solaris* Biblioteka lokalne grupe (liblgrp) je uključena u *Solaris*. U ovoj biblioteci set svih grupa sadržan je u više CPU-a i memoriju kojoj se pristupa sa istim kašnjenjem za sve CPU-ove.
- ✓ U deljivim grupama, bilo koji CPU može pristupiti bilo kojoj memoriji, sa varijacijom udaljenosti između CPU-a i tražene memorije.
- ✓ Tada je moguće doći do podataka o datoj grupi (broj resursa, procesori, veličina slobodne memorije itd.). *Niti* i postavljanje memorije mogu se izvoditi sa obzirom na grupe.

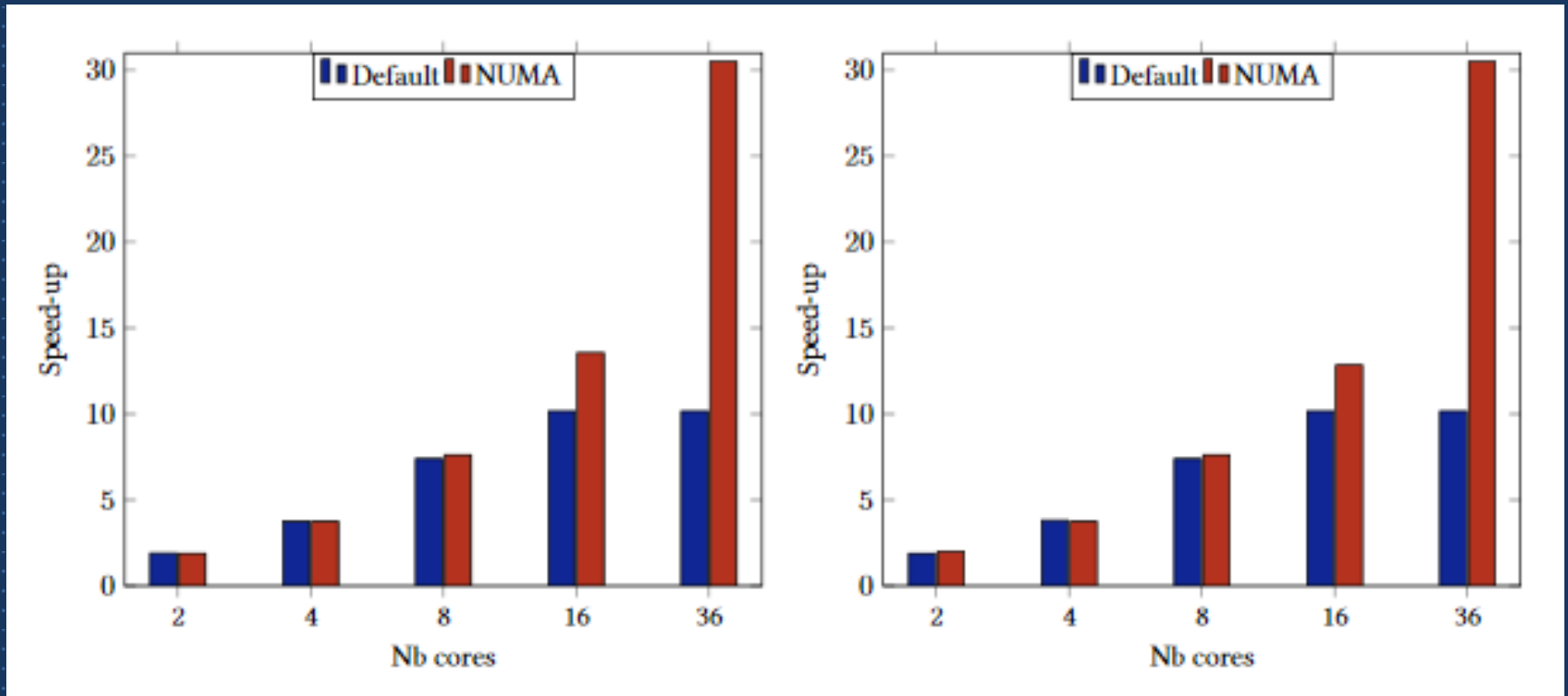
# NUMA APIs

- Pomoću Windows-ovog NUMA API-ja takodje je moguće prihvatiti delove podataka o hardveru i dodeliti memoriju na određenom NUMA čvoru. Međutim, za razliku od libnuma, ne postoje takvih stvari poput dodele memorije.

## *Nezavisni od operativnog sistema:*

- ✓ *hwloc* Prenosna hardverska lokacija (*hwloc*) [40] prenosni je softver u nekoliko operativnih sistema, uključujući Linux, Solaris, Windows, BSD ili Darwin / OS X. Njegova svrha je prikupljanje podataka o hardveru putem API interfejsa ili naredbenih redaka. Takodje je moguće povezati niti ili procese pomoću API-ja.
- ✓ *MAi - Minas Framework* uključuje funkcije na visokom nivou za alokaciju podataka i smeštaj u polja. Postoji nekoliko grupa memorije:
  - ✓ Bindgroup ili deli podatke na blokove nezavisno od broja niti ili podatke smešta u jedan skup ograničenih čvorova.
  - ✓ The cyclic grupa distribuira podatke u round-robin krugu po memorijskim stranicama.
  - ✓ U Random slučajnoj grupi memorijske se stranice nasumično postavljaju na CC-NUMA čvorove. Distribucija podataka po čvorovima može se izvršiti pomoću celih polja ili blokovne distribucije. Veličinu za blok distribuciju može odabrati programer ili automatski MAi

# NUMA APIs



Ubrzanje procesa računanja posmatrajući procesor sa jednim i više jezgara

# Čemu služi paralelno i distribuirano računarstvo?

Opšti motivi:

- ✓ Poželjno je iskoristiti kapacitet umreženih radnih stanica, Grid i Cloud resursa
  - ✓ Rešiti računski intenzivne i sofisticirane probleme brže
  - ✓ Nerešivi problemi postaju rešivi upotrebom paralelnog računara
  - ✓ Iskoristiti prednosti velike količine distribuirane memorije
  - ✓ Rešavati veće probleme u istom vremenskom intervalu
  - ✓ Unaprediti preciznost odgovora
  - ✓ Redukovati vreme projektovanja
  - ✓ Iskoristiti sveprisutne višejezgarne procesore i GPU čipove
- 
- ✓ Procesori su milionima puta brži
  - ✓ Veće frekvencije
  - ✓ Povećana konkurentnost
  - ✓ Veći broj funkcionalnih jedinica
  - ✓ Konkurentno izvršenje instrukcija
  - ✓ Spekulativno izvršavanje instrukcija
  - ✓ Sistemi su sada brži trilion puta
  - ✓ Sami procesori se ubrzavaju
  - ✓ Kombinuju se hiljade procesora u superračunaru

# Zašto su superkompjuteri nedostupni?

- ✓ Superkompjuter  $\neq \sum \text{CPUs}$
- ✓ Brzina računanja  $\neq$  protok (#jobs/time)
- ✓ Spora interkonekcija
- ✓ Neadekvatni I/O
- ✓ Posebno prilagodjene komponente istovremeno povlače inerciju u prihvatanju novih čipova sa najpovoljnijim odnosom cena=performanse
- ✓ Fokus na high-end tržište povlači nedostatak redukcije cena

## *Softver*

- ✓ Neadekvatni operativni sistemi
- ✓ Neadekvatna programska okruženja

# Traženje konkurentnosti

*Grafovi zavisnosti:* Usmereni graf. Čvorovi predstavljaju zadatke. Veze predstavljaju zavisnosti.

## *Paralelizam podataka:*

- ✓ Nezavisni taskovi primenjuju istu operaciju na različitim elementima skupa podataka. U redu je da se ove operacije obave konkurentno
- ✓ Povećanje brzine: potencijalno  $p$  puta, gde je  $p = \text{\#procesora}$

Primer

```
for i = 0 to 99 do
```

```
  a[i] = b[i] + c[i]
```

```
endfor
```

## *Funkcionalni paralelizam:*

- ✓ Nezavisni taskovi primenjuju različite funkcije nad istim ili različitim elementima podataka
- ✓ Brzina limitirana brojem konkurentnih podzadataka
- ✓ U primeru prvi i drugi izraz pa treći i četvrti izraz

Primer

```
a = 2
```

```
b = 3
```

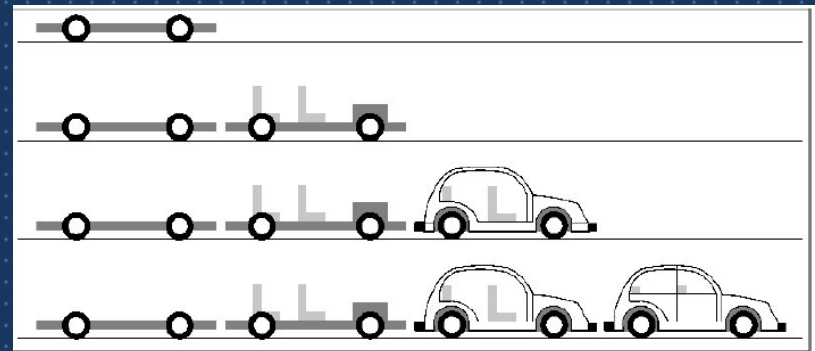
```
m = (a + b) / 2
```

```
s = (a^2 + b^2) / 2
```

```
v = s - m^2
```

## *Pajplajning:*

- ✓ Čitav proces podeliti na faze
- ✓ Proizvoditi nekoliko objekata istovremeno
- ✓ Ubrzanje: Limitirano brojem konkurentskih pod-taskova = #broj faza pajplajna



# Programiranje paralelnih računara

*Proširenje kompajlera:* prevodjenje sekvencijalnih programa u paralelne

- Paralelizujući kompajler
- Detektovanje paralelizma u sekvencijalnim programima
- Produkcija paralelnog izvršnog fajla
- Akcenat na Fortran kompajlerima

Prednosti:

- ✓ Može se primeniti na milione linija već postojećeg koda
- ✓ Štedi vreme i napor
- ✓ Ne zahteva se dodatna obuka programera
- ✓ Sekvencijalno programiranje je lakše od paralelnog

Mane:

- ✓ Paralelizam se može nepovratno izgubiti kada se program napiše sekvencijalno
- ✓ Ova tehnologija radi isključivo kod jednostavnih konstrukcija,
- ✓ kao što su petlje
- ✓ Za sada ne postoji nijedno production-quality rešenje ovog tipa

# Programiranje paralelnih računara i jezici

| Contributions                 | Characteristics of supported parallel programs           | Year |
|-------------------------------|----------------------------------------------------------|------|
| <b>Intermediate languages</b> |                                                          |      |
| [93] (Erbium)                 | Streaming applications                                   | 2010 |
| [99] (PLASMA IR)              | SIMD parallelism                                         | 2010 |
| POP, COHEN [105]              | Directive-based                                          | 2011 |
| [141] (PIR)                   | Habanero-Java                                            | 2011 |
| [77] (SPIRE)                  | Multithreaded, distributed, PGAS                         | 2013 |
| [73] (INSPIRE)                | Multithreaded, distributed, with accelerators            | 2013 |
| [24] (PENCIL)                 | DSLs                                                     | 2015 |
| [117] (Tapir)                 | Multithreaded                                            | 2017 |
| <b>Graphs</b>                 |                                                          |      |
| [125] (EFG)                   | PCF Fortran                                              | 1992 |
| [116] (PPG)                   | Deterministic multithreaded                              | 1994 |
| [84] (CCFG)                   | Multithreaded                                            | 1998 |
| [119] (MPI-CFG)               | MPI                                                      | 2000 |
| [129] (MPI-ICFG)              | MPI                                                      | 2006 |
| [85] (OMPCFG)                 | OpenMP                                                   | 2008 |
| [51] (SPIR)                   | Streaming applications                                   | 2009 |
| [32] (Kimble IR)              | Multithreaded                                            | 2010 |
| [42] (Delite IR)              | DSLs                                                     | 2011 |
| <b>SSA form</b>               |                                                          |      |
| [124] (PSSA)                  | PCF Fortran                                              | 1993 |
| [84] (CSSA)                   | Multithreaded with event-based synchronizations          | 1998 |
| [98] (CSSAME)                 | Multithreaded with synchronizations and mutual exclusion | 1998 |
| [55]                          | Multithreaded with synchronizations and mutual exclusion | 1999 |
| [45]                          | HPF-like                                                 | 2001 |
| <b>Polyhedral model</b>       |                                                          |      |
| [28]                          | OpenMP                                                   | 2011 |
| [101]                         | MPI                                                      | 2012 |
| [140]                         | X10                                                      | 2013 |
| [46]                          | Multithreaded                                            | 2015 |
| [47]                          | Multithreaded                                            | 2016 |
| [57]                          | Multithreaded                                            | 2016 |
| [52]                          | OpenStream [105]                                         | 2016 |

## System-oriented languages

1. sequential languages : CFT [56], PTRAN [60]
2. hardware specific languages : CFD [63, 54, 55], MPL [47]  
: DAP-Fortran [55]
3. architecture class oriented languages
 

|            |                                   |
|------------|-----------------------------------|
| MIMD-based | : Occam [35, 13]                  |
| SIMD-based | : C* [59, 58], Parallaxis [9, 10] |

## Problem-oriented languages

### explicitly parallel

4. task parallel languages : PVM [27], Linda [28],  
: Ada [2, 6, 12]
5. data parallel languages : Modula-2\* [34], Vector C [46]  
: Actus [53, 55], Fortran D [26]  
: HPF [43]

### implicitly parallel

6. functional languages : Haskell [37, 38],  
: extended ML [5]
7. data-flow languages : VAL [49], Id [21, 51],  
: Sisal [48, 15]
8. equational languages : ASL [33, 22], EPL [64],  
: Crystal [39]
9. logical languages : Concurrent Prolog [62]  
: Parlog [32, 18, 16], GHC [66]

# Programiranje paralelnih računara i jezici

*Dodavanje paralelnog programskog sloja:*

*Donji sloj*

- ✓ Kompjuciono jezgro
- ✓ Proces manipuliše sopstvenom porcijom podataka da bi proizveo porciju rezultata (objektno)

*Gornji sloj*

- ✓ Kreiranje i sinhronizacija procesa
- ✓ Particionisanje podataka me. u procesima

*Paralelni programski jezik:*

- ✓ Razviti paralelni jezik “from scratch”
- ✓ Primer je jezik Occam
- ✓ Dodati paralelne konstrukcije u već postojeći jezik
- ✓ Fortran 90, Fortran 95
- ✓ High Performance Fortran
- ✓ C\*

| Intermediate languages | Language support               | Code generation | Compilers              |
|------------------------|--------------------------------|-----------------|------------------------|
| [93] (Erbium)          | OpenStream [105]               | -               | GCC [92]               |
| Pop and Cohen [105]    | OpenMP, OpenACC                | -               | GCC                    |
| [99] (PLASMA IR)       | OpenMP, Matlab, StreamIT       | CUDA            | PLASMA                 |
| [24] (PENCIL)          | VOBLA [30]                     | OpenCL, CUDA    | DSL compilers          |
| [77] (SPIRE)           | OpenMP, MPI, Chapel, OpenSHMEM | OpenMP          | LLVM [78], PIPS [76]   |
| [141] (PIR)            | Habanero-Java [44]             | -               | Habanero-Java compiler |
| [73] (INSPIRE)         | OpenMP, MPI, Cilk, OpenCL      | OpenMP, MPI     | Insieme [131, 100]     |

# Programiranje paralelnih računara

## Šta je paralelno programiranje?

Paralelno programiranje je programiranje u nekom programskom jeziku koje nam omogućuje da eksplicitno navedemo koji će se delovi programa izvršavati konkurentno (istovremeno). Konkurentnost je vrlo korisna apstrakcija za bolje razumevanje načina izvršavanja programa na jednoprocorskim i višeprocorskim računarima, budući da izraz 'paralelno', kao opšti termin, sada već ima istorijsku konotaciju.

Svaki konkurentni program sadrži više sekvencijalnih delova, te kako se sekvencijalni programi sastoje od više niti (eng. thread), nitna struktura se prenosi i na konkurentni program što obezbeđuje izvršavanje više procedura u isto vreme. Imajući u vidu ovakvu organizaciju jasno je da konkurentno programiranje omogućuje razne načine korišćenja računara. Može se reći da postoje tri načina kojima se realizuje konkurentno programiranje - višenitno, distribuirano i paralelno.

- ✓ Višenitni softverski sistemi su karakteristični za jednoprocorske računare i koriste se kod multitaskinga, odnosno, za simultano korišćenje više aplikacija na računaru (eng. multitasking).
- ✓ Distribuirani sistemi, odnosno sistemi više računara povezanih u lokalnu i globalnu računarsku mrežu, obezbeđuju rad sa udaljenim podacima (na primer, preko Web servisa). Grid-computing je poznati primer za ovakve sisteme.
- ✓ Paralelni računarski sistemi obuhvataju paralelne računare. Paralelni programi omogućuju podelu podataka ili zadataka izmedju više komponenti paralelnog računara.

# ОСНОВНИ ПОЈМОВИ

## Окружење паралелних računarskih sistema

Паралелни računarski sistem je računarski sistem koji sadrži dva ili više procesora, окружење u kojem se mogu projektovati паралелни programi i паралелни program koji će se izvršavati.

Za adekvatno окружење je potrebno imati:

- Operativni sistem;
- Sisteme za pravilno uspostavljanje komunikacije medju procesima pomoću:

Razmene poruka:

- ✓ MPI (Message Passing Interface)
- ✓ PVM (Parallel Virtual Machine)

Deljene memorije

- ✓ OpenMP
- ✓ Shmem

Programiranje паралелних računara možemo podeliti u sledeće faze razvoja:

1. Faza identifikacije delova sekvencijalnog algoritma ili podataka kao nezavisnih procesa.
2. Faza distribucije procesa koji će se u isto vreme izvršavati na različitim procesorima.
3. Faza koordinacije rada i komunikacije izmedju procesa.

Pre projektovanja паралelnog koda za konkretan паралелни računarski sistem moramo razmotriti njegov tip arhitekture i način komunikacije. Ova dva uslova predstavljaju jedan od ključnih parametara pomoću kojih razvrstavamo veliki broj današnjih паралелних računarskih sistema.

# ОСНОВНИ ПОЈМОВИ

*Wikipedia says: “Parallel computing is a form of computation in which many calculations are carried out simultaneously”*

Paralelna obrada je jednovremena obrada istog zadatka na dva ili više procesora kako bi se postigli brži rezultati. Računarski resursi mogu uključivati jedan računar sa više procesora ili nekoliko računara povezanih mrežom, ili kombinaciju oba. Procesori pristupaju podacima putem zajedničke memorije. Neki sistemii paralelnih računarskih paralelnih obrada imaju stotine hiljada mikroprocesora.

Pomoću paralelne obrade, može se odjednom izvršiti određeni broj računanja, čime se smanjuje vreme potrebno za dovršetak projekta. Posebno je korisno u projektima koji zahtevaju složene proračune, poput modeliranja vremena i digitalnih specijalnih efekata. Uzmimo primer iz stvarnog života da bismo razumjeli korisnost ove vrste obrade.

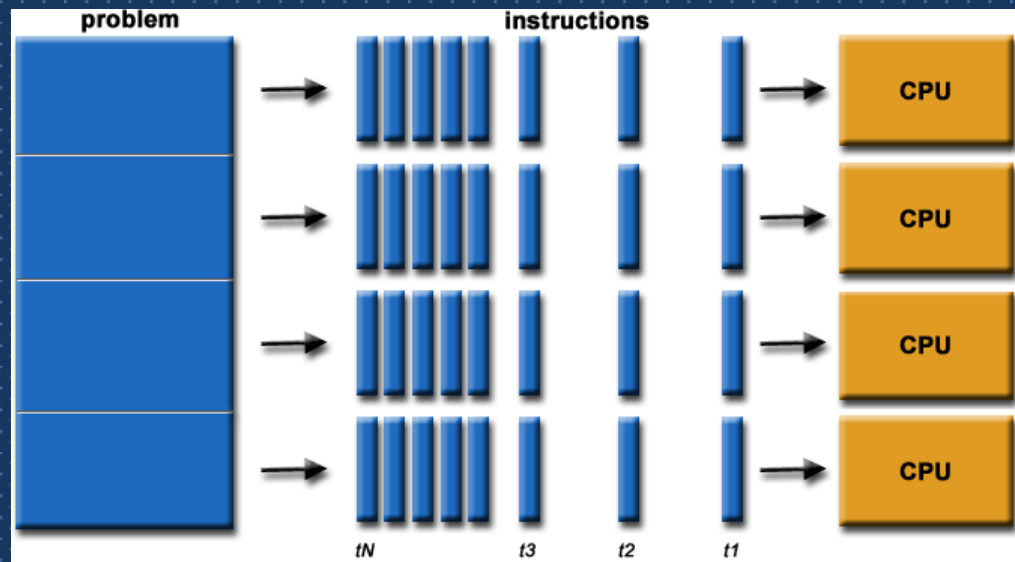
Brzina se računa u jedinicama **FLOPS** – (Floating Point Operations Per Second)

# ОСНОВНИ ПОЈМОВИ

Standardno, programi se pišu za serijsko izvršavanje:

- ✓ Izvršavaju se na jednom računaru na jednoj procesorskoj jedinici
- ✓ Problem je razložen na diskretne serije instrukcija
- ✓ Instrukcije se izvršavaju jedna nakon druge
- ✓ Samo jedna instrukcija se može izvršavati u bilo kom momentu

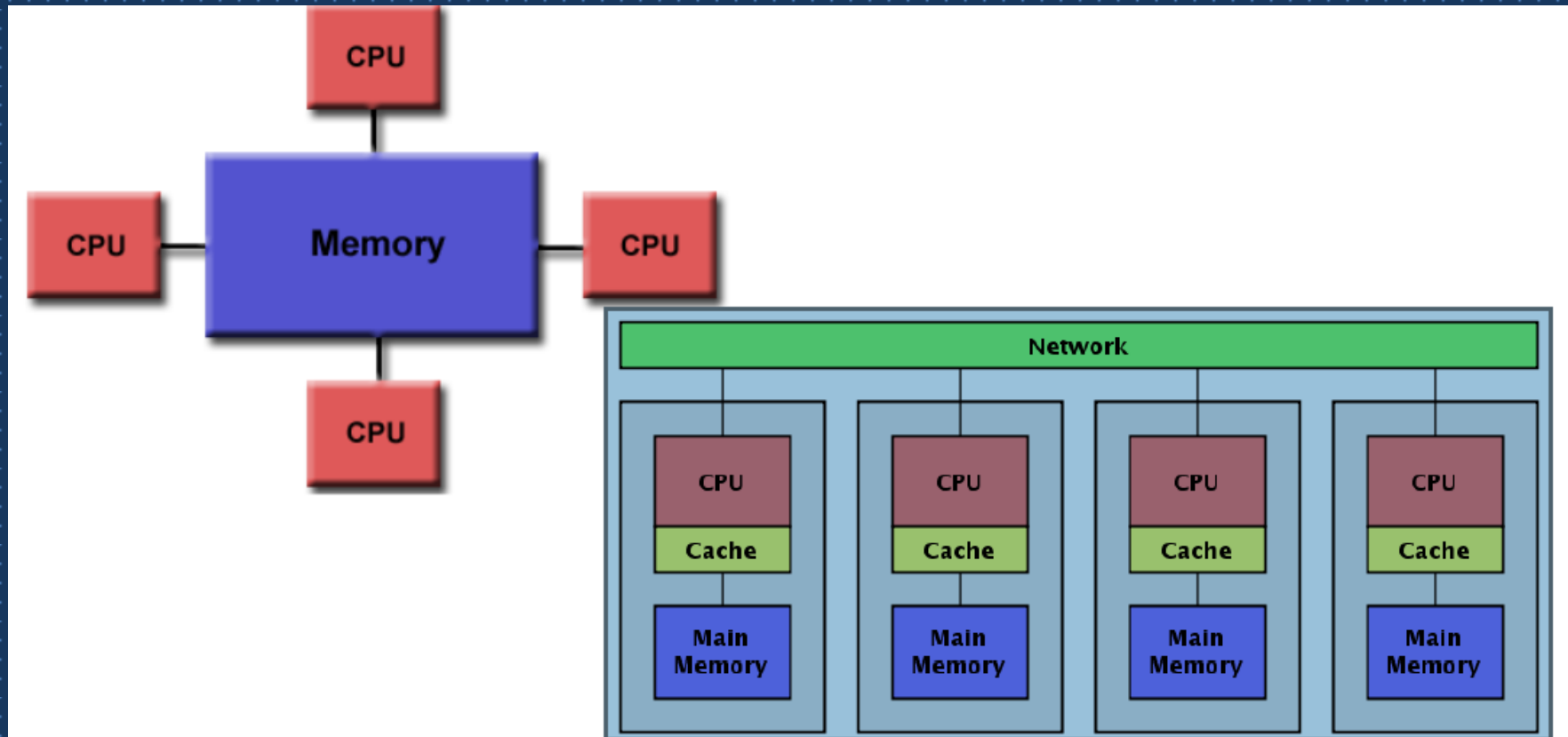
Paralelno programiranje predstavlja simultano korišćenje više računara (ili CPU jedinica) za rešavanje problema



# ОСНОВНИ ПОЈМОВИ

Kako ovo može biti korisno?

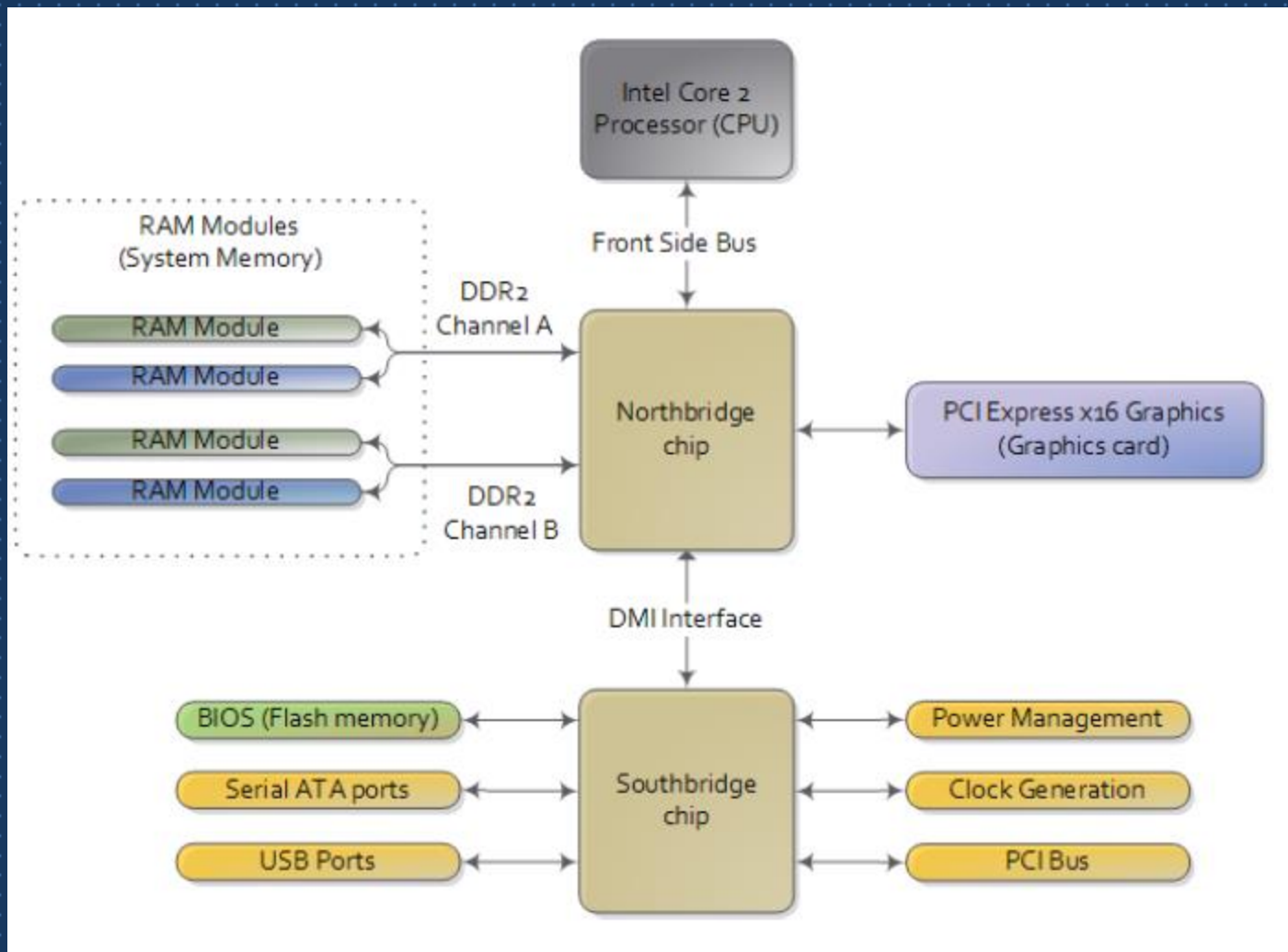
- Brži proračuni
- Može se koristiti manje moćan hardver?
- Mnogo računara ili "čvorova" mogu biti kombinovani u klaster
- Ali, to je mnogo složenije za implementaciju



# Zbog čega se performanse računara povećavaju?

- ✓ *Klimatsko modeliranje.* Da bi bolje razumeli klimatske promene, potrebni su nam mnogo precizniji računarski modeli, modeli koji uključuju interakcije između atmosfere, oceana, zemlje i ledenih kapica na polovima. Moramo takođe biti u mogućnosti napraviti detaljne studije o tome kako različite intervencije mogu uticati na globalnu klimu.
- ✓ *Slaganje proteina.* Veruje se da pogrešno složeni proteini mogu izazvati bolesti poput Huntingtona, Parkinsona i Alzheimerove bolesti, ali naša sposobnost da proučavamo konfiguracije složenih molekula poput proteina je strogo ograničena na trenutnu računarsku snagu.
- ✓ *Otkrivanje novih lekova.* Mnogo je načina na koje se povećana računarska snaga može koristiti u istraživanju novih medicinskih tretmana i lekova. Na primer, ima mnogo lekova koji su primenjeni u lečenju relativno malog dela onih koji pate od neke bolesti. Moguće je da možemo smisliti alternativne načine lečenja pažljivom analizom genoma osoba za koje je poznato lečenje nekorisno. Međutim, ovo će uključivati opsežnu računarsku analizu genoma.
- ✓ *Energetska istraživanja.* Povećana računarska snaga omogućuje programiranje puno detaljnijih tehnoloških modela poput vetrenjača, solarnih ćelija i baterija. Ovi programi mogu pružiti informacije potrebne za izgradnju daleko korisnijih čistih izvora energije.
- ✓ *Analiza podataka.* Generišemo ogromne količine podataka. Po nekim procenama količina podataka se u svetu udvostručuje svake dve godine, ali velika je većina beskorisna ako se ne analizira. Primer je, poznavanje sleda nukleotida u ljudskoj DNK. Razumevanje kako ova sekvenca utiče na razvoj i kako može izazvati bolest zahteva opsežnu analizu. Pored genomike, ogromne količine podataka generišu se sudarima čestica poput Velikog hadronskog sudarača u CERN-u, medicinskim slikama, astronomskim istraživanjima i web pretraživačima.

# Основни појмови

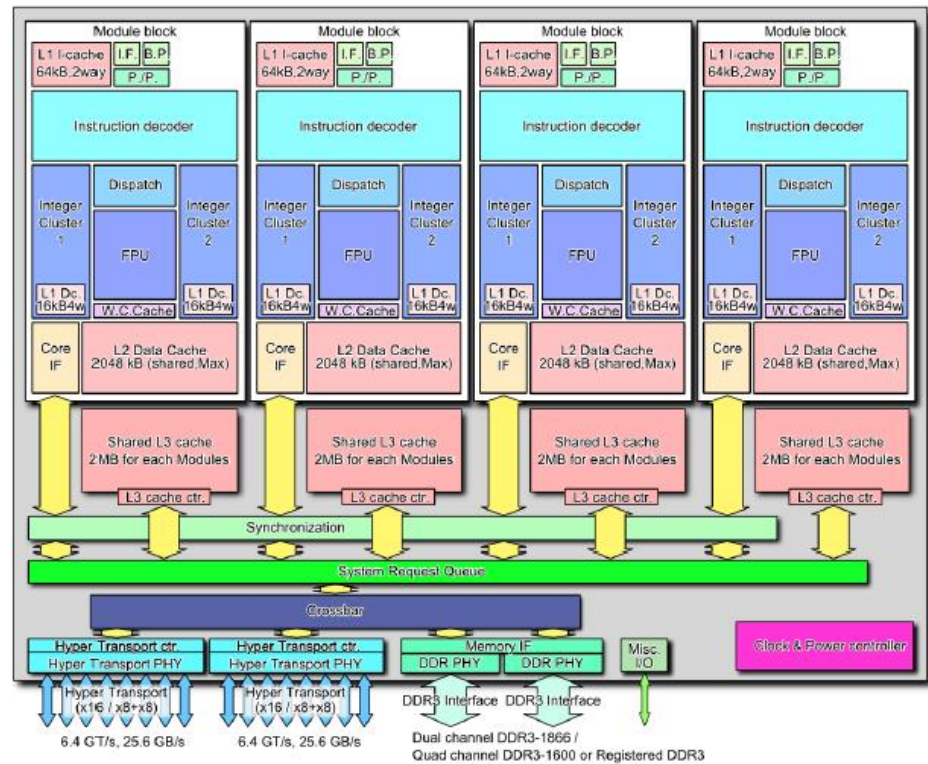
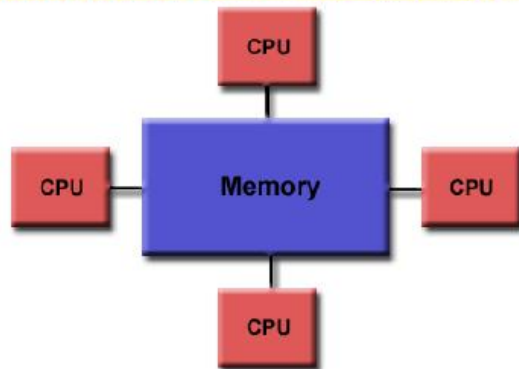
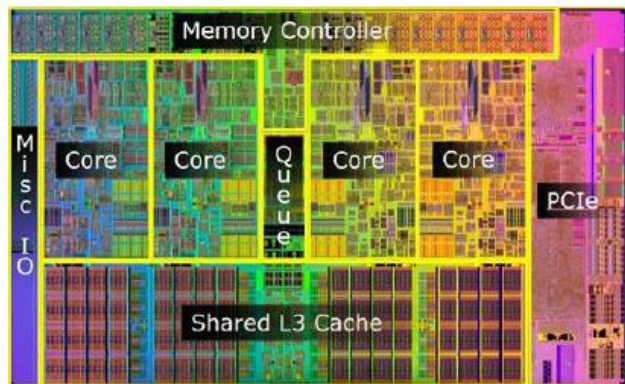


# ОСНОВНИ ПОЈМОВИ

Većina programa koje ljudi pišu i izvode iz dana u dan su serijski programi. Serijski program radi na jednom računaru, obično na jednom procesoru. Naredbe u programu se izvršavaju jedna za drugom, u nizu, i istovremeno se izvršava samo jedna instrukcija.

Paralelno računanje je oblik računanja koji omogućava da se mnoge naredbe u programu istovremeno pokreću, paralelno. Da bi se to postiglo, program se mora podeliti na nezavisne delove tako da svaki procesor može izvršavati svoj deo programa istovremeno s ostalim procesorima. Paralelno računanje se može izvršavati na jednom računaru sa više procesora, većim brojem pojedinačnih računara povezanih mrežom ili kombinacijom ovih dvaju.

# Основни pojmovi



Višeprocorski sistem

# Основни појмови

- Паралелно рачунарство се дефинише као употреба паралелног рачунара у циљу умањења времена потребног за решавање неког проблема.
- Паралелно и дистрибуирано рачунарство представља брже решавање проблема коришћењем већег броја процесора.
- Код паралелног рачунарства у ужем смислу имамо дељену меморију између више процесора, док код дистрибуираног рачунарства сваки процесор поседује своју локалну меморију, тј. свој локални адресни простор.
- Главни проблеми који се решавају су партиционисање, синхронизација, зависност, као и балансирање оптерећења.
- Паралелни рачунар је рачунарски систем са више процесора који подржава паралелно програмирање.

# Основни појмови

Постоје две категорије паралелних рачунара:

- **Кластер (мултикомпјутер)** - паралелни рачунар састављен од више рачунара повезаних мрежом. Процесори са више различитих рачунара међусобно могу да комуницирају искључиво разменом порука.
- **Централизовани мултипроцесор** (симетрични мултипроцесори, SMP) - интергисани системи у којима сви процесори имају приступ једној главној, глобалној, меморији која подржава комуникацију и синхронизацију између процесора.

# Основни појмови

**Паралелно програмирање** је програмирање у језику који дозвољава да се експлицитно зада како ће се разложити делови израчунавања, тј. како ће бити извршени конкурентно на различитим процесорима.

Највећу мотивацију за развој паралелних рачунара представљају тзв. ***Grand Challenge*** проблеми. То су фундаментални проблеми науке и инжењерства, који су толико комплексни да њихово решавање путем нумеричких симулација захтева изузетно брзе и сложене рачунаре. Неки од њих су:

- честична физика,
- прогноза времена и климе,
- глобално загревање,
- дизајн материјала,
- суперпроводљивост,
- нано-уређаји,
- космичке технологије,
- моделовање ткива и органа,
- разбијање шифара и обавештајне услуге,
- обрада података прикупљених преко друштвених мрежа,
- фармацеутска индустрија,
- нафтна индустрија
- земљин магнетизам
- предвиђање кретања урагана
- акцелератори честица...

# Основни појмови

Са друге стране, садашње стање развоја интегрисаних кола указује да убрзавање микропроцесора једноставним повећањем радне фреквенције, смањење дисипације и густине паковања више неће бити могуће.

Томе доприносе физичка ограничења минијатуризације, трансмисиона граница бакарног проводника итд. Услед ове чињенице, данас је развој микропроцесора пре свега усмерен на увећање броја процесорских језгара, тиме увећавајући значај паралелног програмирања и конкурентне обраде података уопште.

Уколико се не деси неки већи продор у индустрији микроелектронике (као нпр. продор у квантном рачунарству и нанотехнологији), паралелизација остаје једини начин да се приступи било каквој масовној обради.

# Основни појмови – супер рачунари

**Супер-рачунар** је рачунар чији је рачунарски капацитет за неколико редова величине већи од рачунара опште намене у датом временском тренутку.

Одатле директно следи да сама дефиниција супер-рачунара зависи од времена. Неко практично правило налаже да рачунарска машина мора бити барем 100 пута бржа од стандардног персоналног рачунара да би ушла у категорију супер-рачунара. Када се говори о супер-рачунарима, често се срећу и следеће скраће-нице:

- **HPC** - *High Performance Computing*,
- **HEC** - *High End Computing* и
- **CI** - *Cyberinfrastructure*.

Перформансе супер-рачунара мере се у јединицама под називом FLOPS (*Floatingpoint OPerations per Second*) уместо стандардног MIPS-а (*Million Instructions per Second*).

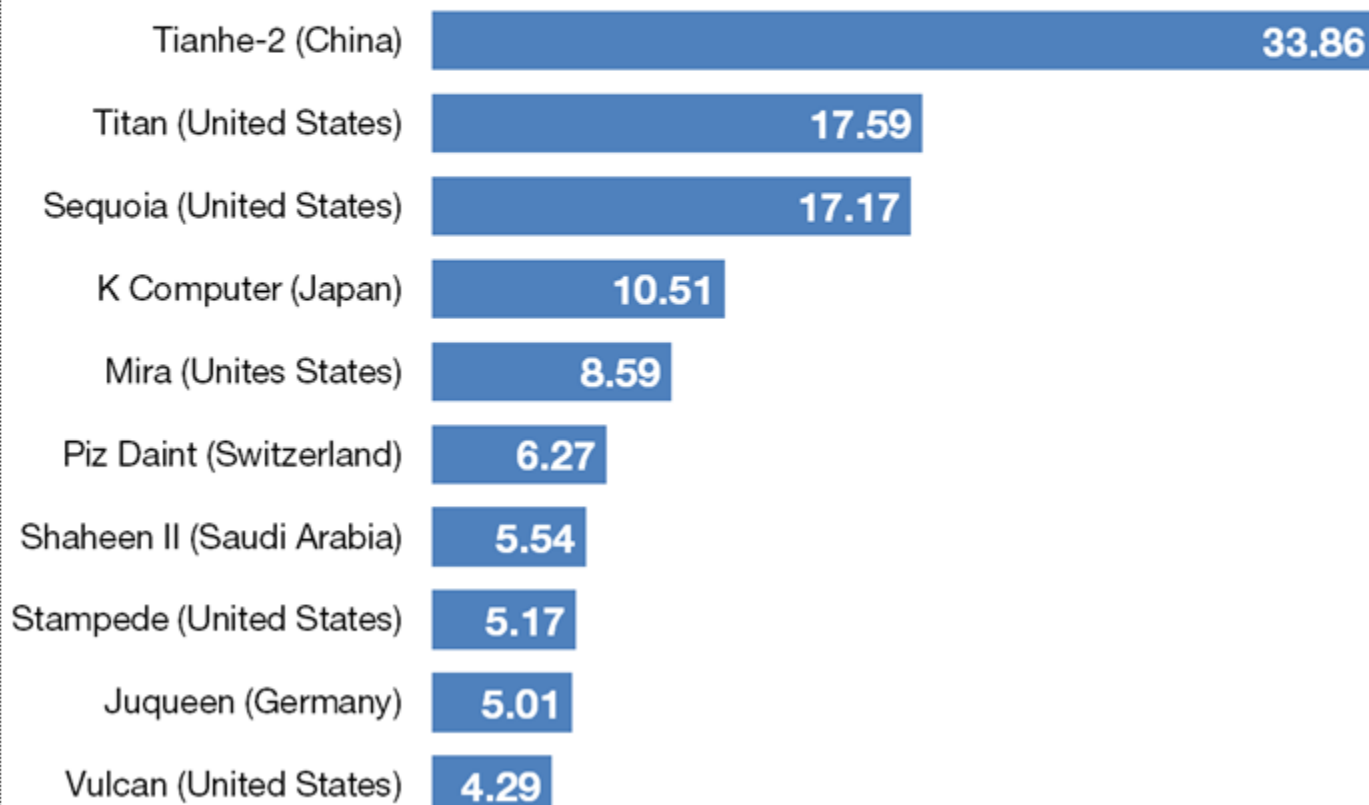
# World's Top Ten Fastest Supercomputers – June 2018

## Updated list

| # | System                                                  | Entity                                                                           | Processor                                                                                                  | TFlop/s (Peak)   | Power (kW)   | #  | System                    | Entity                                                            | Processor                                                                                                      | TFlop/s (Peak)   | Power (kW)  |
|---|---------------------------------------------------------|----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|------------------|--------------|----|---------------------------|-------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|------------------|-------------|
| 1 | <a href="#">Summit</a>                                  | IBM & US Department of Energy<br>United States                                   | IBM Power9 processors + 6 x Nvidia Tesla V100 GPU<br>2,282,544 Cores<br>2,801,664 GB RAM                   | 187,659 TFlop/s  | 8,805.50 kW  | 6  | <a href="#">Piz Daint</a> | Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland | Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100<br>361,760 Cores<br>340,480 GB RAM | 25,326.3 TFlop/s | 2,271.99 kW |
| 2 | <a href="#">Sunway TaihuLight</a>                       | National Supercomputing Center in Wuxi<br>China                                  | Sunway SW26010 260C 1.45GHz<br>10,649,600 Cores<br>1,310,720 GB RAM                                        | 125,436 TFlop/s  | 15,371.00 kW | 7  | <a href="#">Titan</a>     | Cray Inc. DOE/SC/Oak Ridge National Laboratory United States      | Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x<br>560,640 Cores<br>710,144 GB RAM            | 27,112.5 TFlop/s | 8,209.00 kW |
| 3 | <a href="#">Sierra</a>                                  | National Institute of Advanced Industrial Science and Technology (AIST)<br>Japan | Xeon Gold 6148 20C 2.4GHz, 391,680 cores<br>417,792 GB RAM                                                 | 32,576.6 TFlop/s | 1,649.25 kW  | 8  | <a href="#">Sequoia</a>   | IBM DOE/NNSA/LLNL United States                                   | BlueGene/Q, Power BQC 16C 1.60 GHz<br>1,572,864 Cores<br>1,572,864 GB RAM                                      | 20,132.7 TFlop/s | 7,890.00 kW |
| 4 | <a href="#">Tianhe-2 (MilkyWay-2)</a>                   | National Super Computer Center in Guangzhou<br>China                             | Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P<br>2,277,376 GB RAM<br>4,981,760 Cores | 100,679 TFlop/s  | 18,482.00 kW | 9  | <a href="#">Trinity</a>   | Cray Inc. DOE/NNSA/LANL/SNL United States                         | Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect<br>979,968 Cores                                 | 43,902.6 TFlop/s | 3,843.58 kW |
| 5 | <a href="#">AI Bridging Cloud Infrastructure (ABCI)</a> | National Institute of Advanced Industrial Science and Technology (AIST)<br>Japan | Xeon Gold 6148 20C 2.4GHz processor<br>391,680 Cores<br>417,792 GB                                         | 32,576.6 TFlop/s | 1,649.25 kW  | 10 | <a href="#">Cori</a>      | Cray Inc. DOE/SC/LBNL/NERSC United States                         | Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect<br>622,336 Cores<br>878,592 GB RAM               | 27,880.7 TFlop/s | 3,939.00 kW |

# Top 10 supercomputers

Petaflop/s on the Linpack benchmark



# 1. Summit



The \$200 million supercomputer is an IBM AC922 system utilizing 4,608 compute servers containing two 22-core IBM Power9 processors and six Nvidia Tesla V100 graphics processing unit accelerators each. Summit is also (relatively) energy-efficient, drawing just 13 megawatts of power, compared to the 15 megawatts TaihuLight pulls in. Summit boasts a peak performance of 200,000 trillion calculations per second and will be eight times more powerful than America's current top-ranked system, Titan. It took four years to build, according to Romett

## 2. Sunway TaihuLight



Currently ranked as the fastest supercomputer in the world, the Sunway TaihuLight supercomputer measures in at **125 petaFLOPS** (theoretical peak)—five times as fast as the supercomputer in second place. Housed in the National Supercomputing Center in Wuxi, it is comprised of **10.6 million cores** and is being used for climate research, earth systems modeling and data analytics. On top of being the fastest supercomputer in the world, the Sunway TaihuLight is currently ranked as the fourth most energy-efficient one as well, requiring substantially fewer megawatts per megaFLOPS

### 3. Sierra



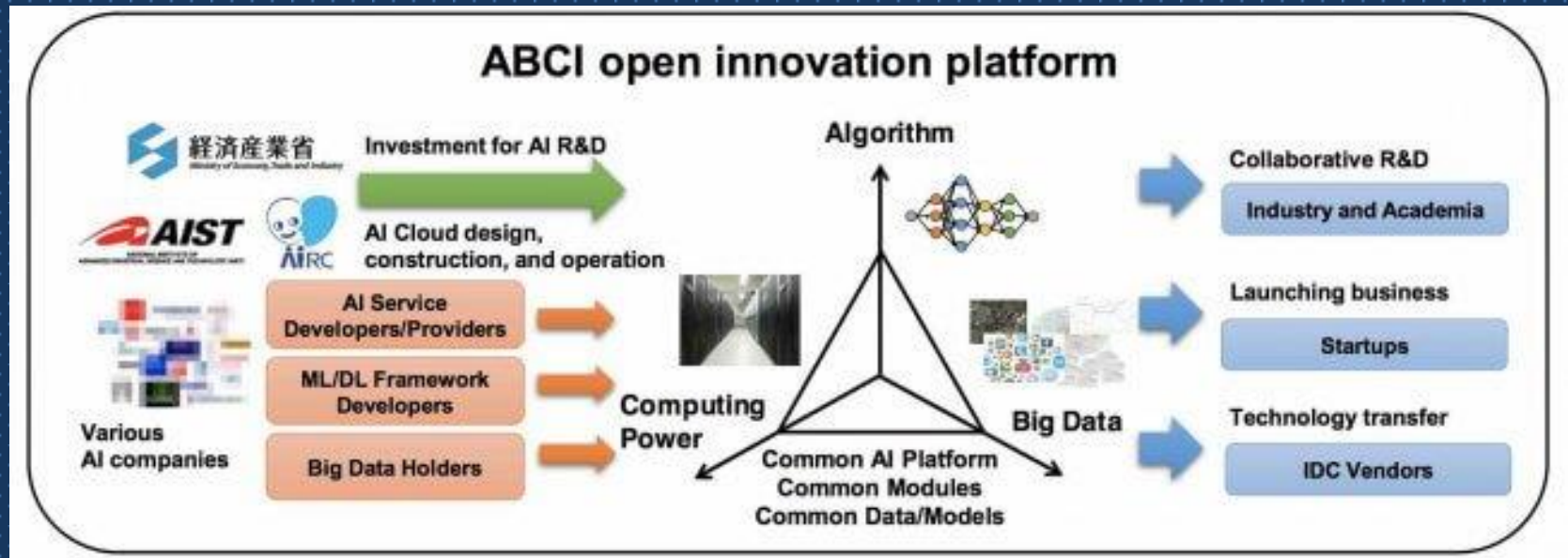
Sierra, LLNL's next-generation supercomputer, is projected to provide four to six times the sustained performance and be at least seven times more powerful than Sequoia, with a **125 petaFLOP/s peak** based on IBM's Power9 chip

## 4. Tianhe-2 (MilkyWay-2)



In late 2016, the Piz Daint supercomputer in Lugano, Switzerland gained a huge hardware upgrade. That new power tripled its computing performance and brought its theoretical peak performance up to **19.6 petaFLOPS** (their own measurements pin it currently at 25.3), making it the fastest supercomputer outside Asia. Named after a mountain in the Swiss Alps, the Piz Daint also creates advanced visualizations and high-resolution imaging simulations. It will soon provide processing power to the Large Hadron Collider at CERN, helping it analyze huge amounts of data.

# 5. AI Bridging Cloud Infrastructure (ABCI)



AI Bridging Cloud Infrastructure (ABCI) is a 130-petaflops supercomputer costing \$173 million (£131 million). Japan's new machine will be used in the field of Artificial Intelligence, which explains its rather boring name: "AI Bridging Cloud Infrastructure," or ABCI. According to Satoshi Sekiguchi, a director-general at Japan's National Institute of Advanced Industrial Science and Technology, where the computer will be built, the system will also be used to "tap medical records to develop new services and applications."

## Specifications

**Manufacturer:** Fujitsu

**Cores:** 391,680

**Memory:** 417,792 GB

**Processor:** Xeon Gold 6148 20C 2.4GHz

**Linpack Performance (Rmax):** 19,880 TFlop/s

**Theoretical Peak (Rpeak):** 32,576.6 TFlop/s

**Power:** 1,649.25 kW (Submitted)

**Measured Cores:** 10,880

**Operating System:** Linux

# Основни појмови

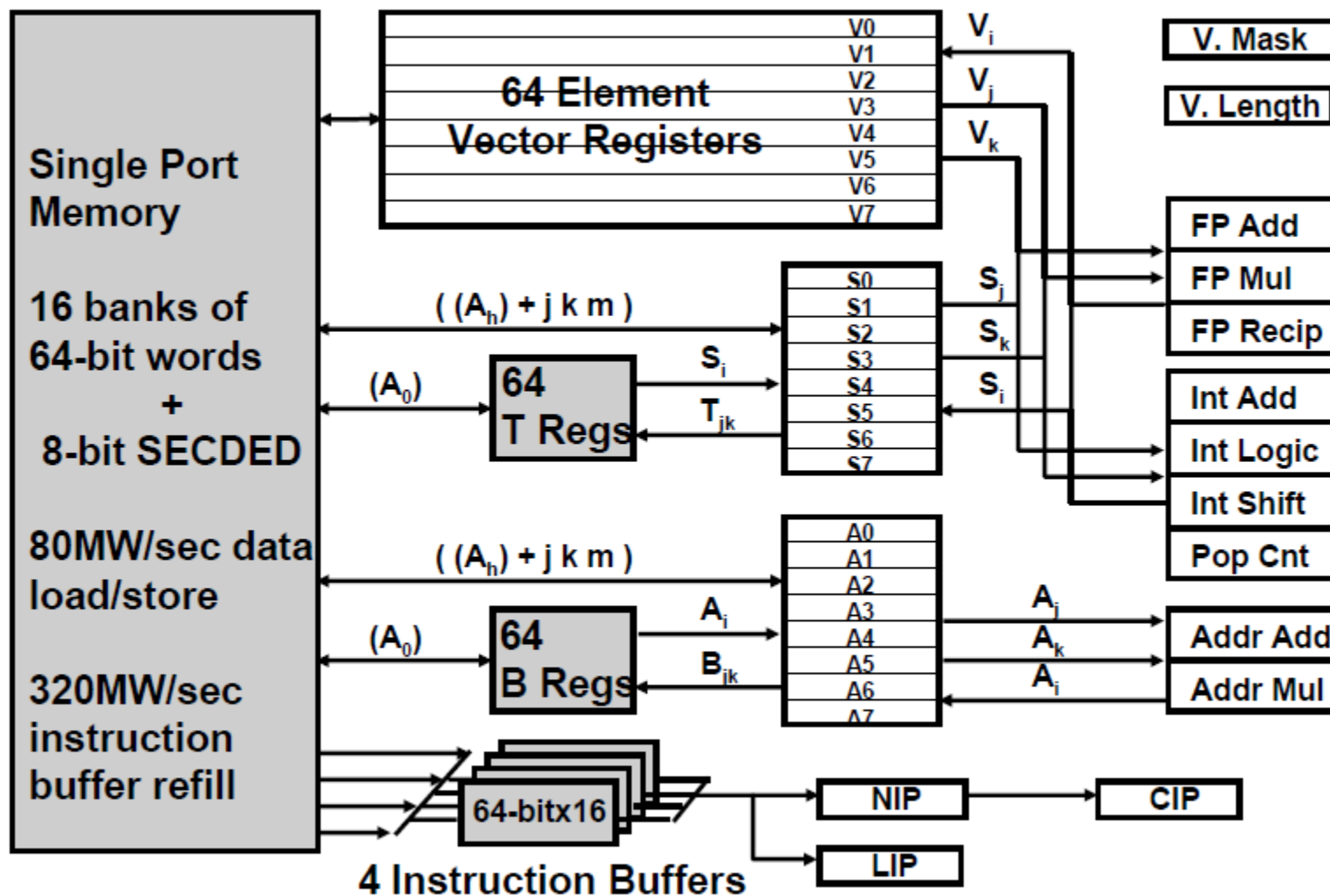


Супер-рачунари који су доступни (2016. године) имају капацитет од више десетина PFLOPS (петафлопс).

Увек ажурна листа најмоћнијих супер-компјутера данашњице може се наћи на веб локацији:  
[www.top500.org](http://www.top500.org).

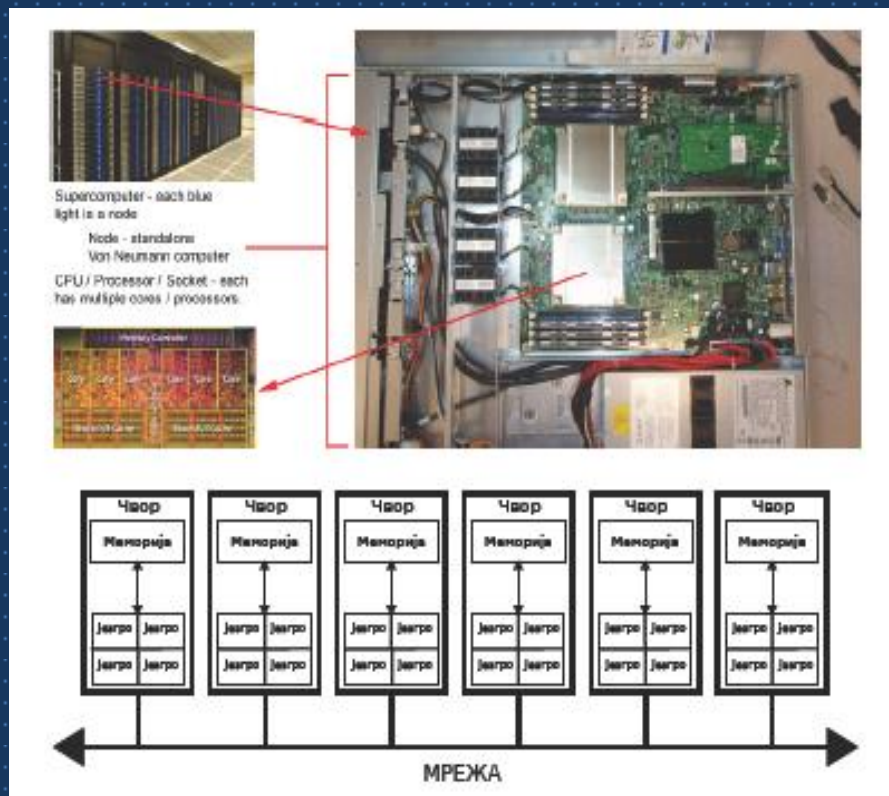
CRAY-1 - пример традиционалног супер-рачунара.  
Конструисан 1975. године

# Cray-1 (1976)



*memory bank cycle 50 ns    processor cycle 12.5 ns (80MHz)*

# Основни појмови



- Традиционални супер-рачунар изграђен је као један једини векторски процесор повезан на меморијску магистралу високих перформанси.
- Са друге стране, савремени супер-рачунар је паралелни рачунар са хиљадама процесора, спакован у кластер који се састоји од чворова са по више процесора у сваком од њих.
- Овај историјски прелаз је омогућен развојем микропроцесора и напредком VLSI (*Very Large Scale of Integration*) технологије и наравно развојем нанотехнологије. За изградњу савремених супер-рачунара се користи велики број сасвим просечних, на тржишту широко доступних микропроцесора.

Архитектура савременог супер-рачунара

# Основни појмови

Са друге стране, супер-рачунар ипак није једноставна сума својих процесора. На пример, ако супер-рачунар поседује 1000 процесора, то, на жалост, никако не значи да је 1000 пута бржи.

Разлог томе је, логично, спора интерконекија, неадекватно решени улазно/излазни подсистеми, као и неадекватни оперативни системи и програмерска окружења.

Све у свему, комерцијални паралелни системи су били прескупи, а услед брзог развоја хардвера су каскали на разним пољима.

Са савременог аспекта, синоним за традиционалне супер-рачунаре су примитивна програмерска окружења и искључиви фокус на комерцијални домен и владине институције. Научници су, логично, потражили алтернативу.

# Модерни паралелни рачунари

Модерни паралелни рачунари су, уместо на специјалним процесорима, засновани на стандардним микропроцесорима који се уграђују у персоналне рачунаре и сервере опште намене. Дато је неколико примера са почетака модерног паралелног рачунарства, чији је заједнички именитељ да су засновани на процесорима опште намене:

- **Caltech Cosmic Cube** (1981. година) - састављен од 64 Intel 8086 микропроцесора потпомогнутим 8087 копроцесорима за операције са покретним зарезом. Имао је 128кВ меморије по процесору, између 5 и 10 мегафлопса и био дупло јефтинији, а 5-10 пута бржи од тадашњег супер-рачунара који се могао набавити по истој цени.
- **nCube** (1983. година) - комерцијална копија претходног, произвео *Intel*, 512 процесора.
- **Connection Machine (CM2, CM5)** (1987. година) - произвео *Thinking Machines Corporation*, VLSI, 65000 4-битних процесора. CM2-SIMD, CM5-MIMD су подваријанте .

# Предности паралелног и дистрибуираног рачунарства

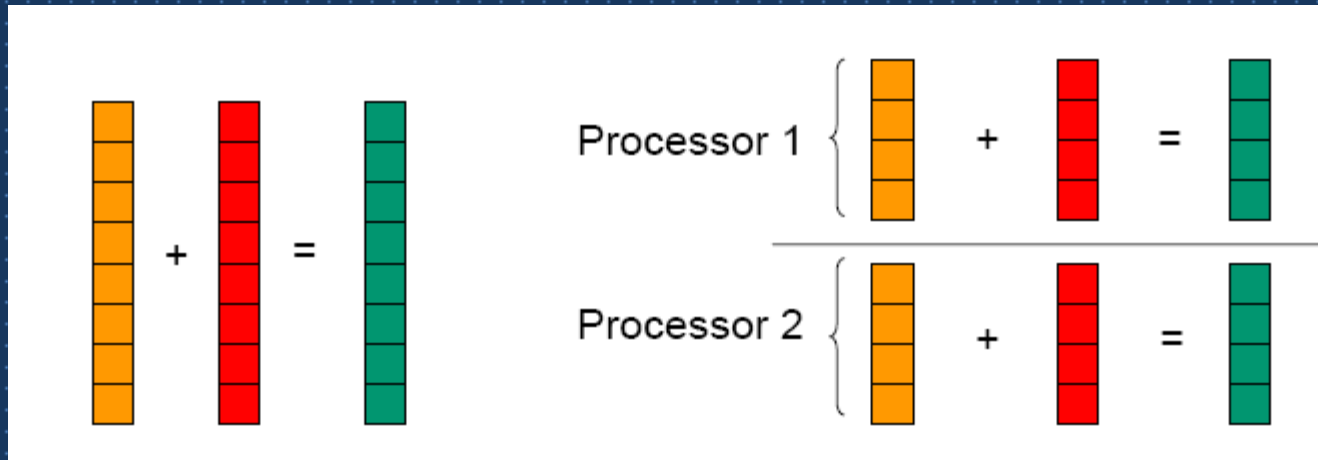
Предности рада на паралелним и дистрибуираним системима могу се сублимирати у следећим ставкама:

1. Решити проблеме брже.
2. Решити обимније проблеме за исто време – BIG DATA.
3. Искористити велику количину дистрибуиране меморије.
4. Повећати прецизност решења, visokobitni procesori
5. Искористи капацитет умрежених радних станица и Грид ресурса.
6. Коришћењем вишејезгарне процесоре и графичке процесоре.
7. Коришћењем *dataflow* архитектуре .

# Osnovne postavke

Primena konkurentnog računanja

- Internet: Klijent i server su nezavisni, interakcija se ostvaruje preko aplikacija,
- Pretraga jednog elementa: pretraga distribuirane baze podataka preko višestrukih procesora
- Dodavanjem dva polja za integer:



# Osnovne postavke

- Skalarni produkt:

$$s = \sum_{i=0}^{N-1} a[i] * b[i]$$

- Paralelni algoritam

$$\begin{aligned} s &= \sum_{i=0}^{N/2-1} (a[i] * b[i]) + \sum_{i=N/2}^{N-1} (a[i] * b[i]) \\ &= \underbrace{\sum_{i=0}^{N/2-1} (a_{local}[i] * b_{local}[i])}_{rank=0} \quad + \quad \underbrace{\sum_{i=0}^{N/2-1} (a_{local}[i] * b_{local}[i])}_{rank=1} \end{aligned}$$

– zahtev za komunikaciju izmedju procesa

# Osnovne postavke

## PP za filmove

- Special effects are highly compute intensive
- Example: Lord of the Rings
  - company: Weta Digitals
  - 3200 processor cluster
  - a single scene contains:
    - per second 24 frames
    - per frame: 4996 x 3112 points with 32- or 64 bit color encoding
    - each object means a separate compute cycle
- Number of computer-added special effects in movies:
  - Jurassic Park (I): 75
  - Lord of the Rings (I): 540
  - each of the following episodes of Lord of the Rings doubled the number of special effects
  - last episode of Star-Wars: 2000-2500



# Optimizacija tehnike

Sa programskog gledišta, Interpolacija i Inverzni Helmholtz mogu se implementirati sledeći formulu za interpolaciju:

$$v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot u_{lmn}.$$

I za inverzni Helmholtz:

$$t_{ijk} = \sum_{l,m,n} A_{kn}^T \cdot A_{jm}^T \cdot A_{il}^T \cdot u_{lmn}$$

$$p_{ijk} = D_{ijk} \cdot t_{ijk}$$

$$v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot p_{lmn}$$

Predstavljeni su dva nivoa transformacije programa neophodne za ubrzavanje. Imamo na umu da zbog sažetosti više ne izražavamo dimenzije petlji i tenzora uključenih u mrežu.

```
1  for (e = 0; e < M; e++)
2  for (i = 0; i < N; i++)
3  for (j = 0; j < N; j++)
4  for (k = 0; k < N; k++)
5  for (l = 0; l < N; l++)
6  for (m = 0; m < N; m++)
7  for (n = 0; n < N; n++)
8  v[e][i][j][k] = A[k][n] * A[j][m] * A[i][l] * u[e][l][m][n];
```

(a) Interpolation

```
1  for (e = 0; e < M; e++) {
2  for (i = 0; i < N; i++)
3  for (j = 0; j < N; j++)
4  for (k = 0; k < N; k++)
5  for (l = 0; l < N; l++)
6  for (m = 0; m < N; m++)
7  for (n = 0; n < N; n++)
8  t[e][i][j][k] = A[n][k] * A[m][j] * A[l][i] * u[e][l][m][n];
9
10 for (i = 0; i < N; i++)
11 for (j = 0; j < N; j++)
12 for (k = 0; k < N; k++)
13 p[e][i][j][k] = D[e][i][j][k] * t[e][i][j][k];
14
15 for (i = 0; i < N; i++)
16 for (j = 0; j < N; j++)
17 for (k = 0; k < N; k++)
18 for (l = 0; l < N; l++)
19 for (m = 0; m < N; m++)
20 for (n = 0; n < N; n++)
21 v[e][i][j][k] = A[k][n] * A[j][m] * A[i][l] * p[e][l][m][n];
22 }
```

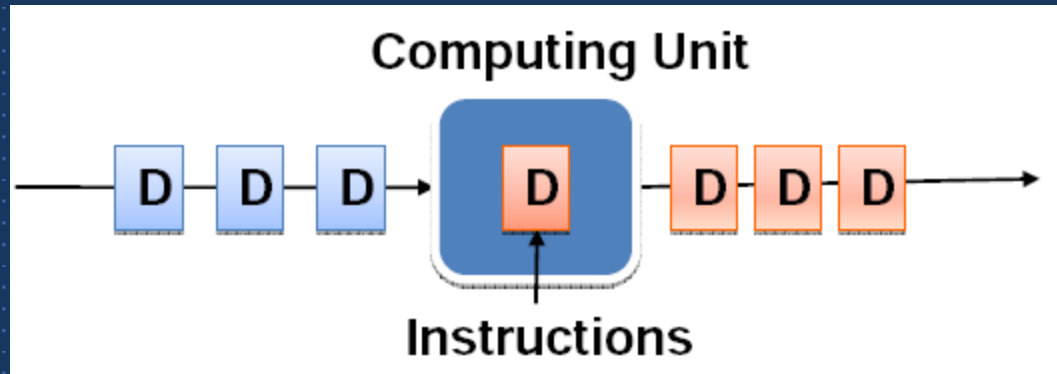
(b) Inverse Helmholtz

# Osnovne postavke

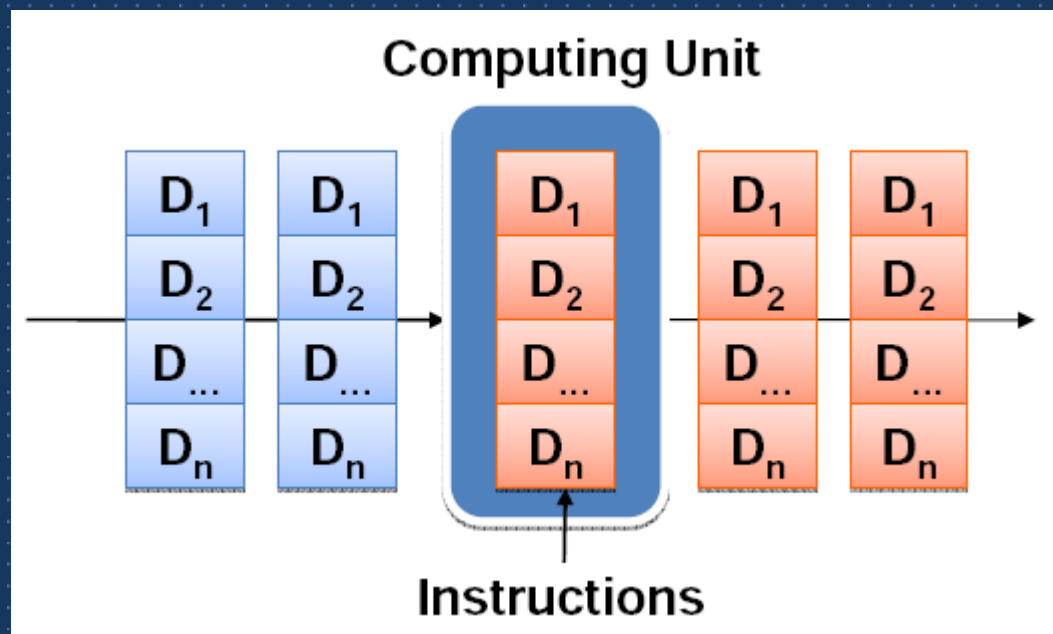
## Klassifikacija Paralelne Arhitekture

- **SISD:** Single instruction single data
  - Classical von Neumann architecture
- **SIMD:** Single instruction multiple data
- **MISD:** Multiple instructions single data
  - Non existent, just listed for completeness
- **MIMD:** Multiple instructions multiple data
  - Most common and general parallel machine

## Single Instruction Single Data - SISD

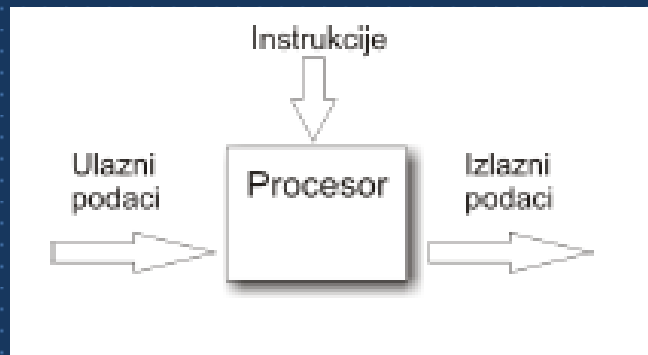


## Single Instruction Multiple Data - SIMD



# Single Instruction Single Data - SISD

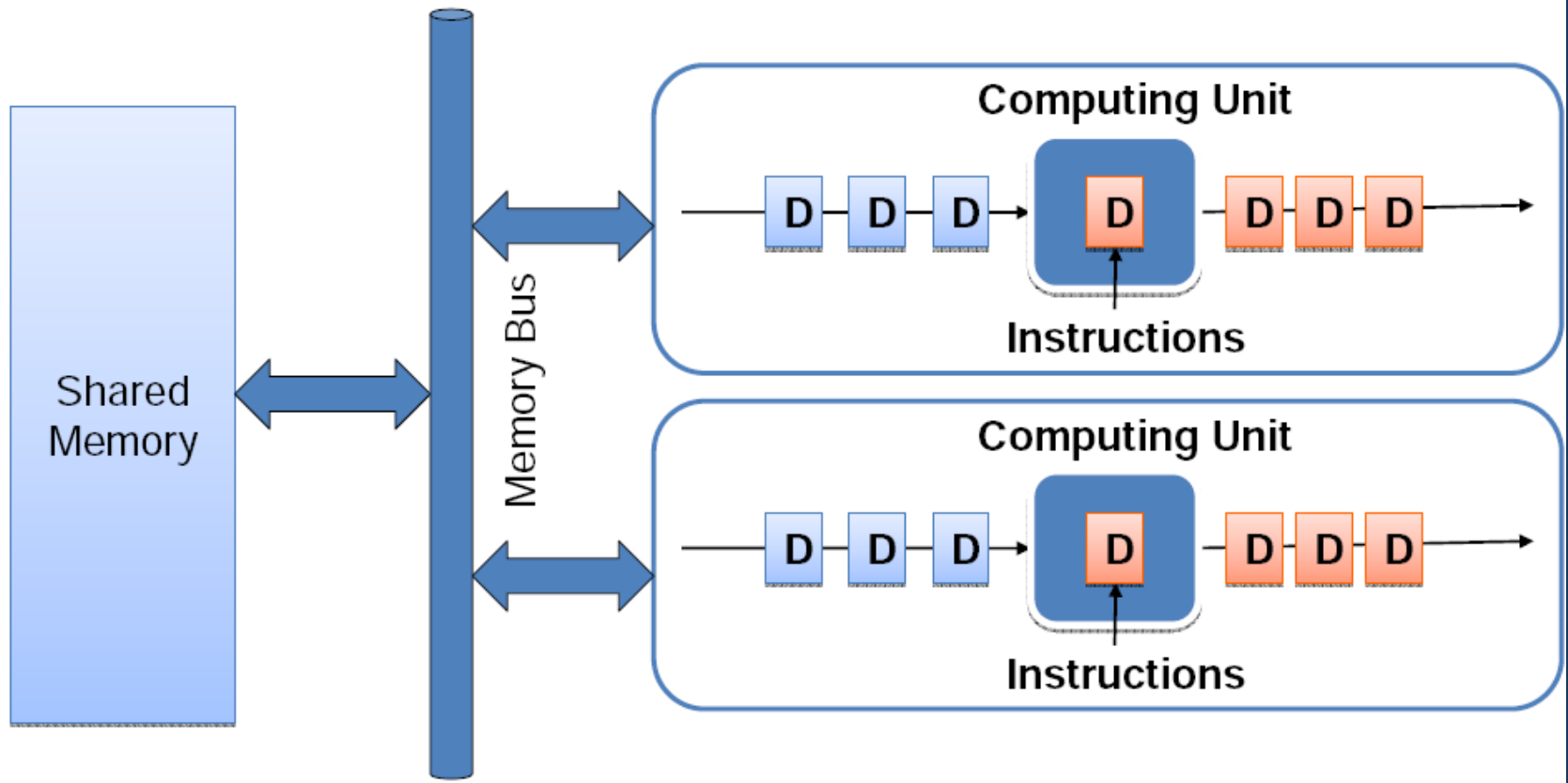
SISD model procesora je sposoban da izvršava jedan tok instrukcija i operiše nad jednim tokom podataka, te u ovu grupu procesora spadaju jednoprocesorski računarski sistemi. U SISD procesorima instrukcije se izvršavaju jedna za drugom (sekvencijalno), zbog čega ih još nazivamo i sekvencijalnim računarima. Najveći broj današnjih računara sadrži SISD model procesora. Instrukcije i podaci se smeštaju u lokalnu memoriju (videti sliku 1.2), te je brzina jednog procesorskog elementa u SISD modelu ograničena brzinom kojom računar prebacuje podatke.



# Multiple Instructions Multiple Data - MIMD

Shared-memory

Multiple CPUs with a shared Memory

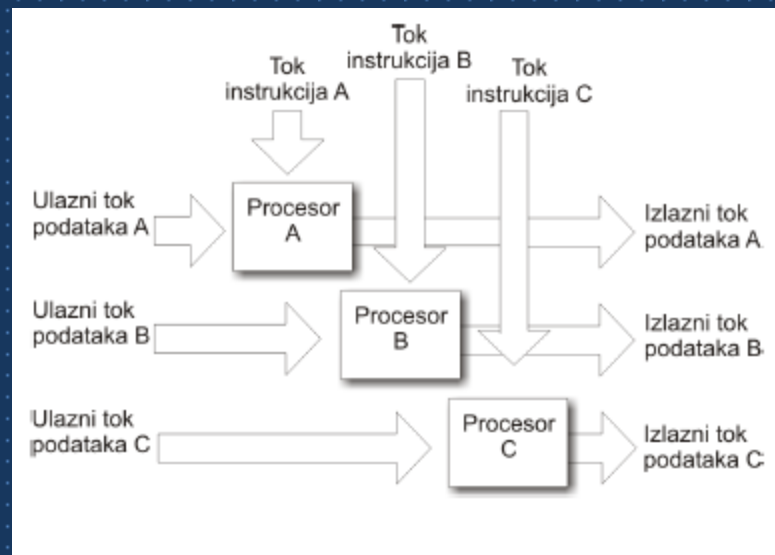


# Multiple Instructions Multiple Data - MIMD

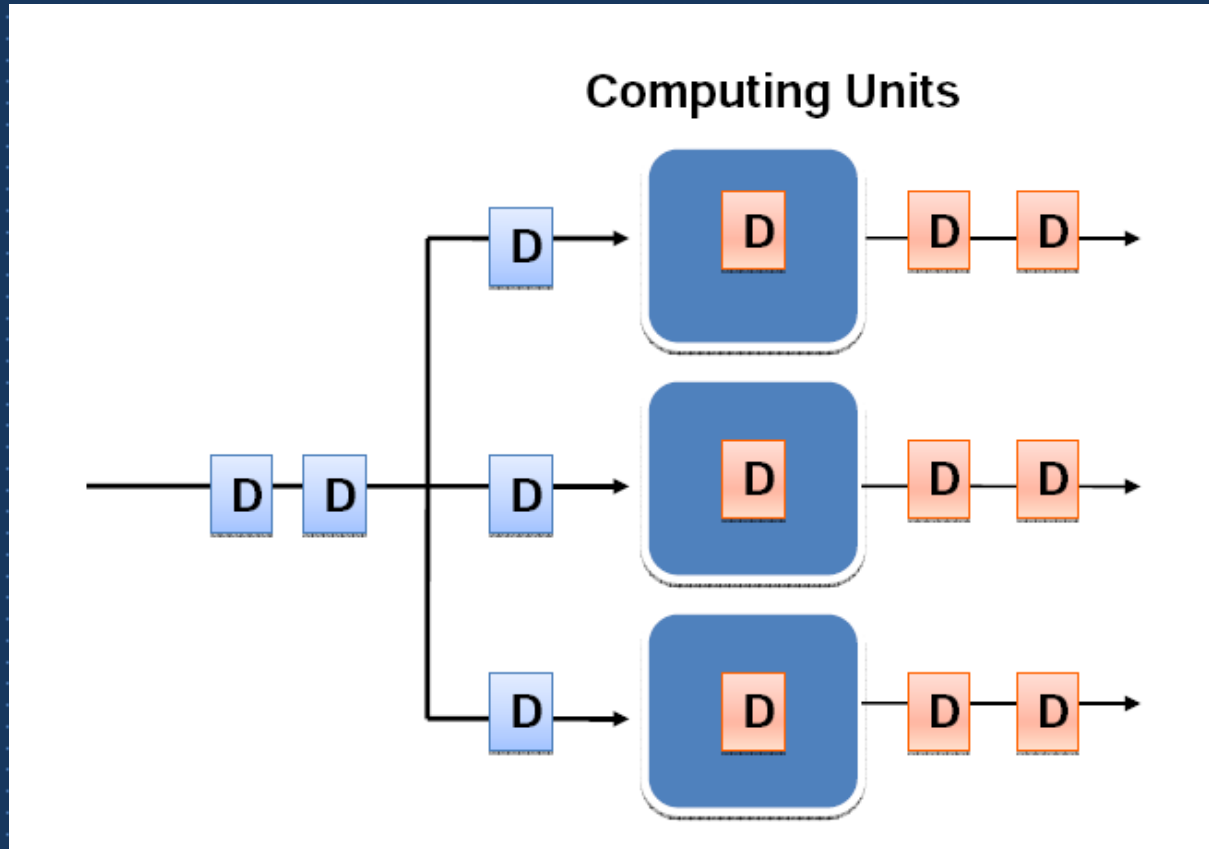
MIMD računarski sistemi su najsloženiji u kome svaki procesor poseduje sopstvenu lokalnu memoriju. MIMD višeprocorske mašine su sposobne da izvrše više instrukcija nad različitim skupom podataka. Način na koji se instrukcije izvršavaju je asinhron, što znači da se u istom trenutku mogu izvršavati različite instrukcije nad različitim podacima.

U kategoriju MIMD modela spada najveći broj današnjih superračunara. Prednosti MIMD modela u odnosu na ostale modele iz Flinove podele su:

- MIMD procesori mogu da izvršavaju više različitih poslova za isto vreme;
- Svaki MIMD procesor je u stanju da bilo koju operaciju izvršava nezavisno od rada drugih procesora.



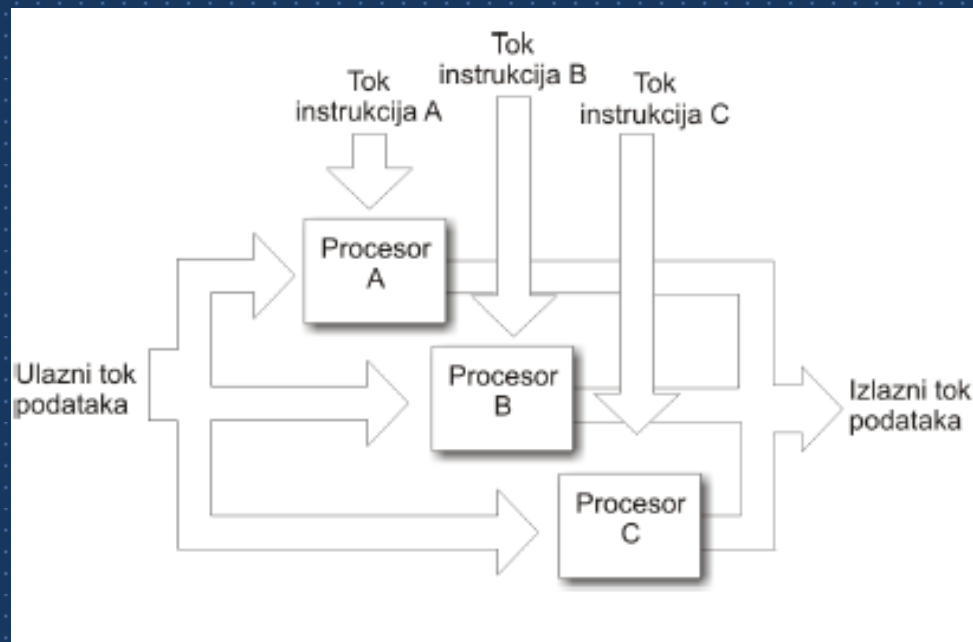
# Multiple Instructions Single Data - MISD



# Multiple Instructions Single Data - MISD

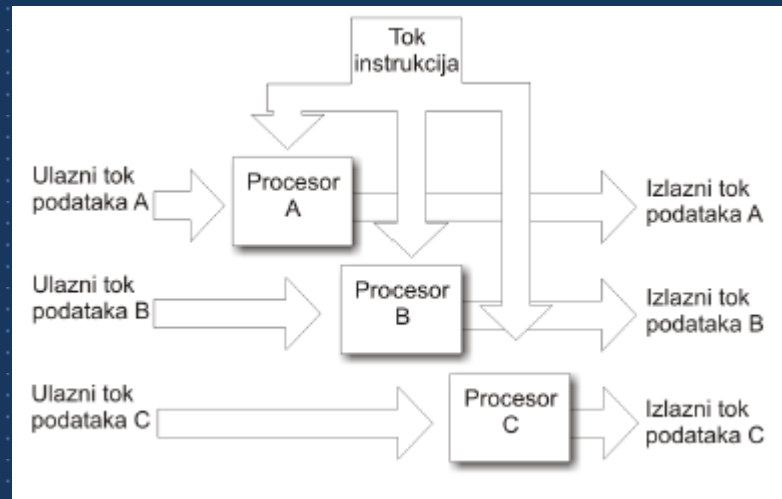
MISD je model višeprocorskog sistema gde  $n$  procesora, svaki sa po jednom kontrolnom jedinicom, dele isti tok podataka. Svi procesori, u svakom koraku, obradjuju istovremeno jedan podatak prihvaćen iz zajedničke memorije, prema instrukciji koju svaki procesor prima sa svoje upravljačke jedinice. Primena ovakvog tipa računara je pogodna za:

1. Primenu filtra za izdvajanje signala iz ulaznog niza podataka;
2. Izvršavanje različitih kriptografskih algoritama za razbijanje šifri.



# Single Instructions Multiple Data – SIMD

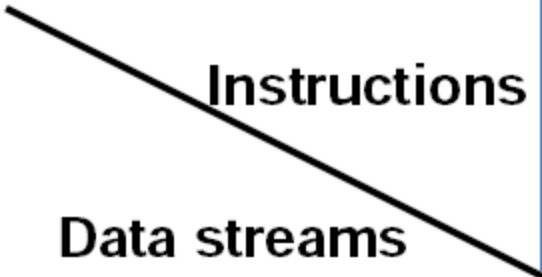
SIMD model procesora se odnosi na računare u kojima svaki od  $N$  identičnih procesora sa zajedničkom kontrolnom jedinicom izvršava isti program nad svojim lokalnim podacima. Izvršavanje je sinhrono, odnosno, svaki procesor u isto vreme izvršava istu instrukciju ali za drugačiji tok podataka. Poznati primer SIMD procesora je vektorski procesor.



Mašine bazirane na SIMD modelu su korisne za rad u naučnim oblastima u kojima se često koriste operacije nad vektorima i matricama. Primer takve konstrukcije je niz procesora sa jednom procesorskom kontrolnom jedinicom i nizom podređenih procesora. Najstarija paralelna mašina ILIAC je imala SIMD arhitekturu i sastojala se od 32 procesorska elementa povezanih u prsten, od kojih se svaki element sastojao od jednog procesora i lokalne memorije. Iako se SIMD koristio uglavnom na vektorskim superračunarima, poput onih koje je 70-tih godina popularisao Cray, potreba za obradom multimedijalnih podataka je dovela do dodavanja SIMD instrukcija u arhitekturu procesora opšte namene. Ovaj trend je izrodio tehnologije kao što su: PowerPC-jev AltiVec; Intel-ovi MMX, SSEx; AMD-ov 3Dnow, i mnoge druge.

# Osnovne postavke

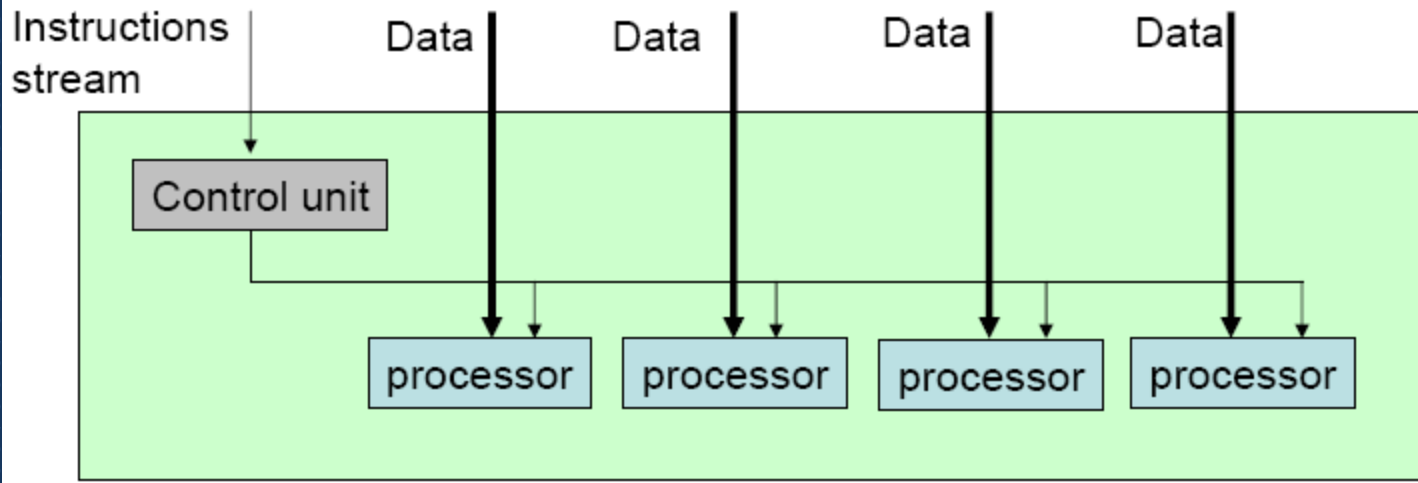
klasifikacija kompjuterskih arhitektura  
Data & Operand (instructions)

|  Instructions<br>Data streams | Single Instruction<br>(SI)                | Multiple Instruction<br>(MD)                 |
|----------------------------------------------------------------------------------------------------------------|-------------------------------------------|----------------------------------------------|
|                                                                                                                | Single Data<br>(SD)                       | Multiple Data<br>(MD)                        |
| Single Data<br>(SD)                                                                                            | <b>SISD</b><br>Single-threaded<br>process | <b>MISD</b><br>Pipeline architecture         |
| Multiple Data<br>(MD)                                                                                          | <b>SIMD</b><br>Vector processing          | <b>MIMD</b><br>Multi-threaded<br>programming |

# Osnovne postavke

## Single Instruction Multiple Data

- Poznato kao Polje procesora: Array-processors
- Poseban tok instrukcija se prenosi na više procesora, koji poseduju svoj poseban tok podataka
  - Danas se koristi kodgrafickih katrica



# Osnovne postavke

## Multiple Instructions Multiple Data (I)

- Svaki procesor ima sopstveni tok instrukcija i ulazne podatke
- Generalni slucaj
  - svaki drugi scenario moze biti mapiran sa MIMD
- Budici razvoj MIMD obicno se bazira na organizaciji memorije
  - Deljeni memorijski sistemi
  - Distribuirani memorijski sistemi

# Osnovne postavke

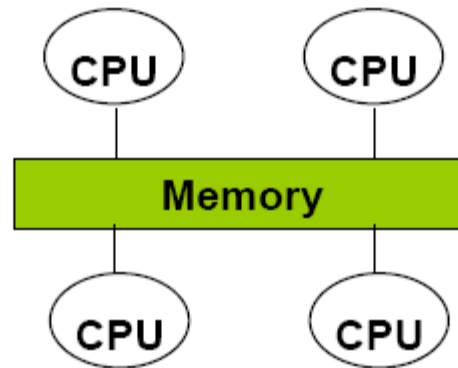
## Shared memory systems (I)

- Svi procesori imaju pristup istom adresnom prostoru
  - E.g. PC sa više od jednog procesora
- Razmena podataka izmedju procesa je pisanjem/citanjem deljivih promenljivih
  - Shared memory system je relativno lako programirati
  - Tekuci standard u programiranju: OpenMP
- Dve verzije shared memory sistema prisutne danas su
  - Symmetric multiprocessors (SMP)
  - Non-uniform memory access (NUMA) architectures

# Osnovne postavke

## Symmetric multi-processors (SMPs)

- Svi procesori dele istu glavnu fizicku memoriju

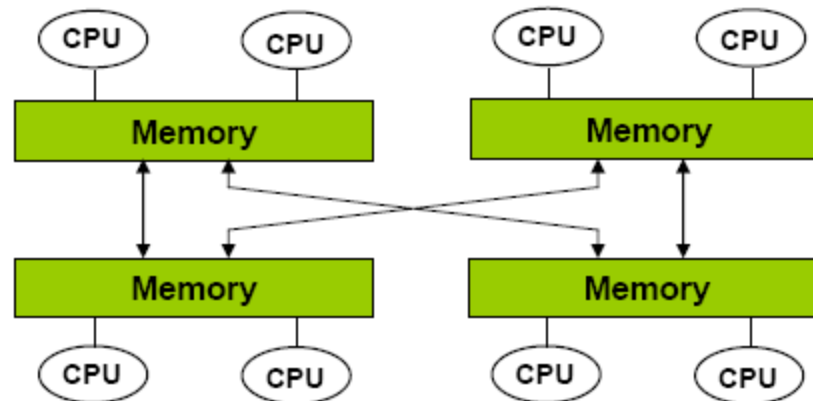


- Opseg memorije po procesoru limitirajuci je faktor za ovaj tip arhitekture
- Tipicna velicina: 2-32 processors

# Osnovne postavke

## NUMA arhitektura (I)

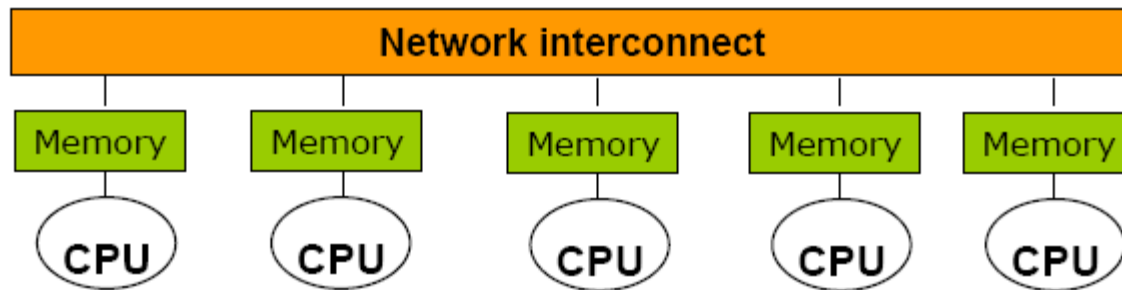
- Odredjena memorija je bliza izvesnom procesoru nego druga memorija
  - Kompletan memorija je adresibilna za sve procesore
  - U zavisnosti od toga koji deo podataka procesor zahteva, vreme pristupa moze vrlo varirati



# Osnovne postavke

## Distributed memory machines (I)

- Svaki procesor ima svoj vlastiti adresni prostor
- Komunikacija izmedju procesa preko razmene eksplicitnih podataka
  - Sockets
  - Message passing
  - Remote procedure call / remote method invocation



# Osnovne postavke

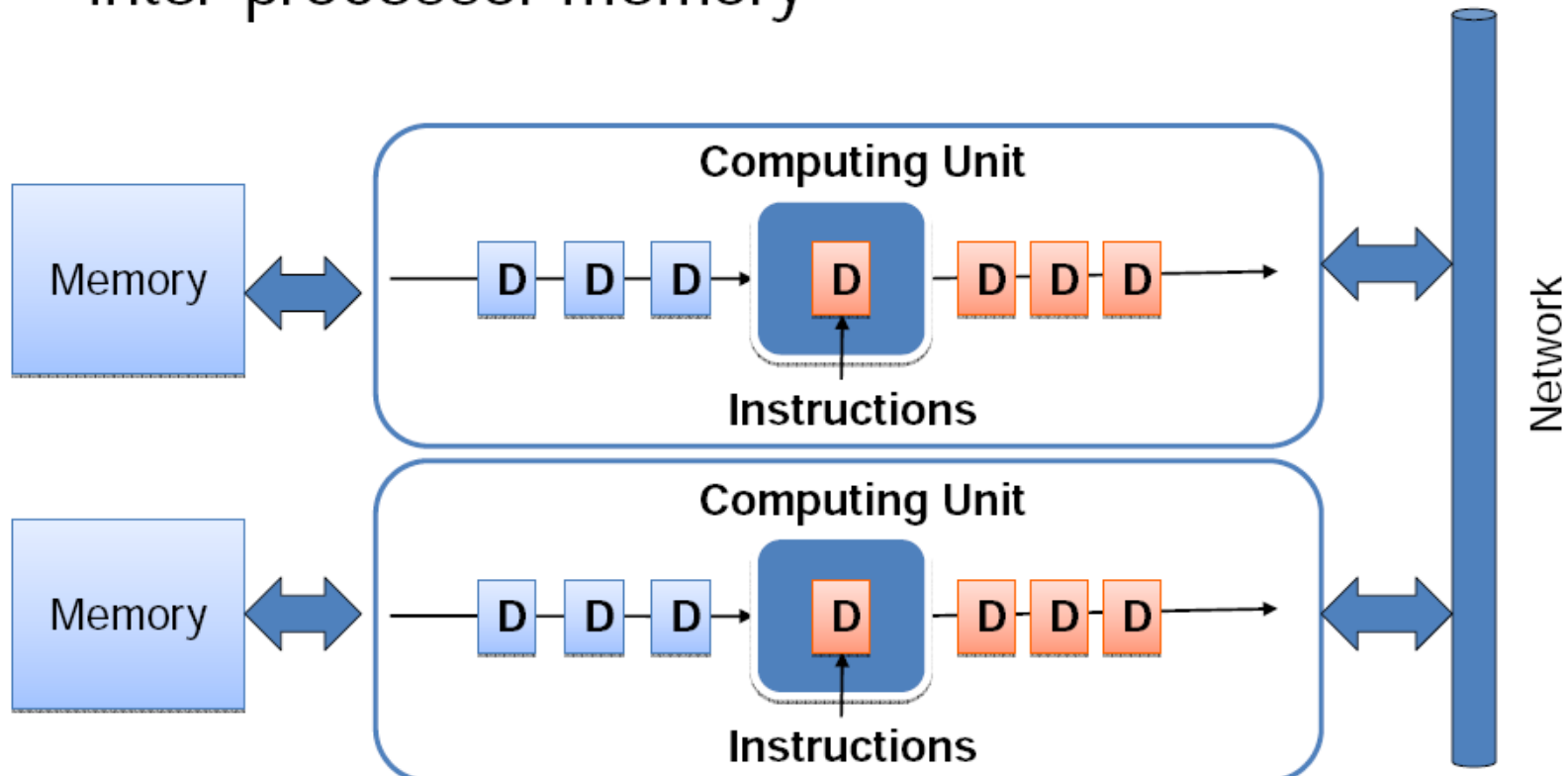
## Distributed memory machines (III)

- Imamo dve klase distributed memory masina:
  - Massively parallel processing systems (MPPs)
    - Tesno kuplovano okruzenje
    - Jedinstvena slika sistema (specialni OS)
  - Klasteri
    - Of-the-shelf hardverske i softverske komponene kao što su:
      - Intel P4, AMD Opteron etc.
      - Standard operating systems such as LINUX, Windows, BSD UNIX C++ C Sharp

# Osnovne postavke

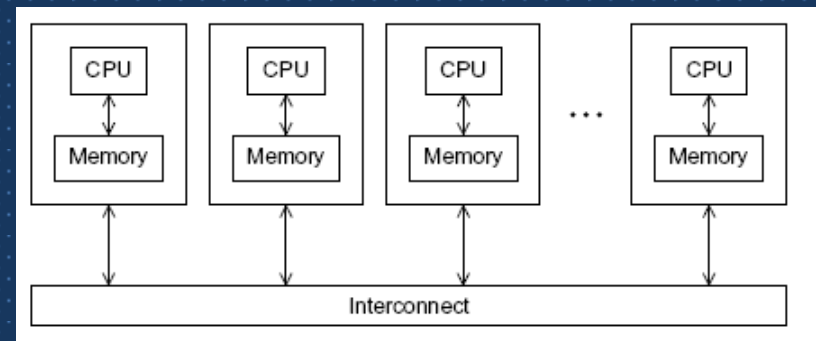
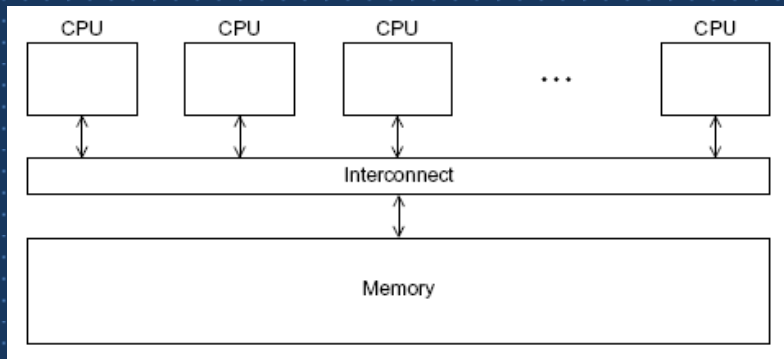
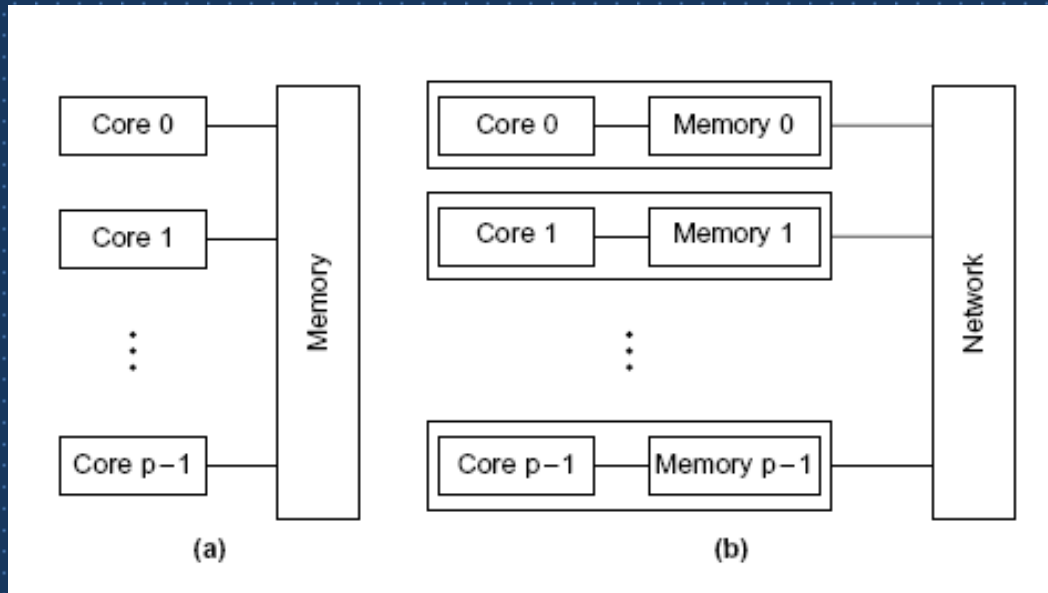
## MIMD, Distributed Memory

- Require a communication network to connect inter-processor memory



# Shared-memory system (a) i (b)

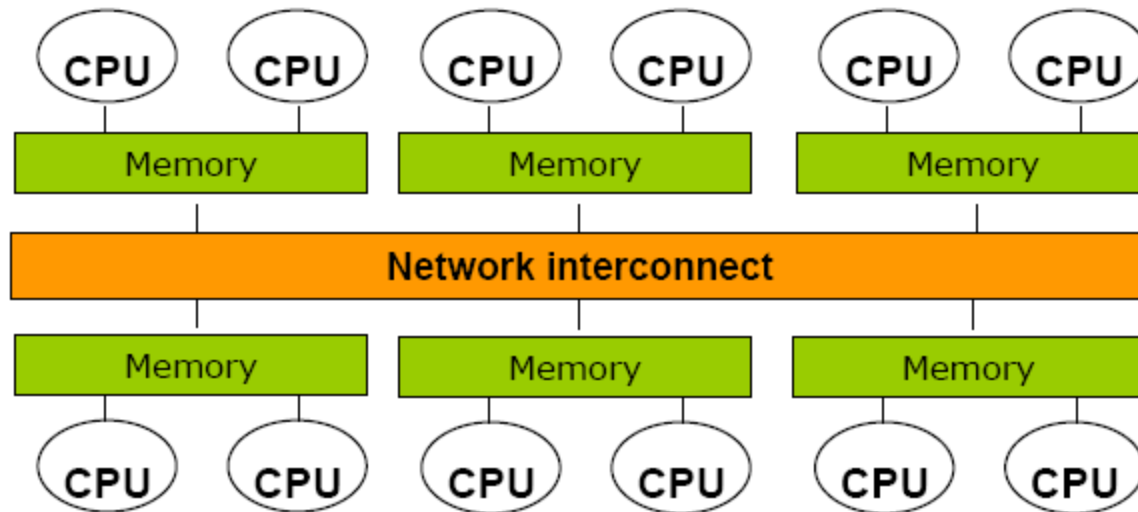
## Distributed-memory system



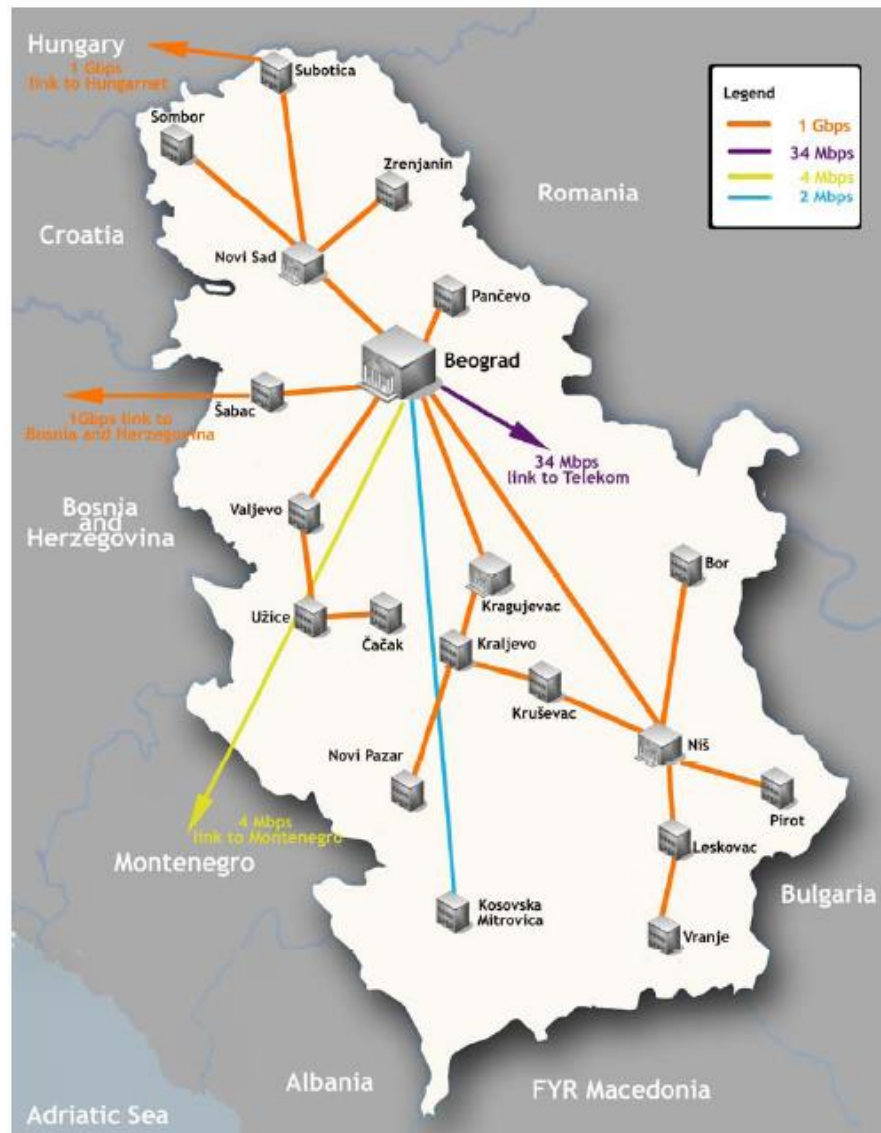
# Osnovne postavke

## Hibridni sistemi

- Klasteri multi-procesorskih nodova



# Kičma Akademske mreže



# Еволуција супер-рачунара

Крајем Другог светског рата у Сједињеним Америчким Државама је конструисан ENIAC (*Electronic Numerical Integrator And Computer*), како би се брже и тачније срачунавале тзв. артиљеријске табеле, тј. пројектоване путање пројектила високог домета.

Током Хладног рата, јаки рачунари су коришћени за пројектовање нуклеарног наоружања, разбијање шифара, космички програм и сличне намене.

Данас се углавном не развија ново нуклеарно наоружање, већ се користе нумеричке симулације да би се одржавао постојећи арсенал нуклеарног наоружања.

Данас супер-рачунари имају далеко ширу примену, што говори у прилог чињеници да је велика потражња покретач за развој оваквих машина.

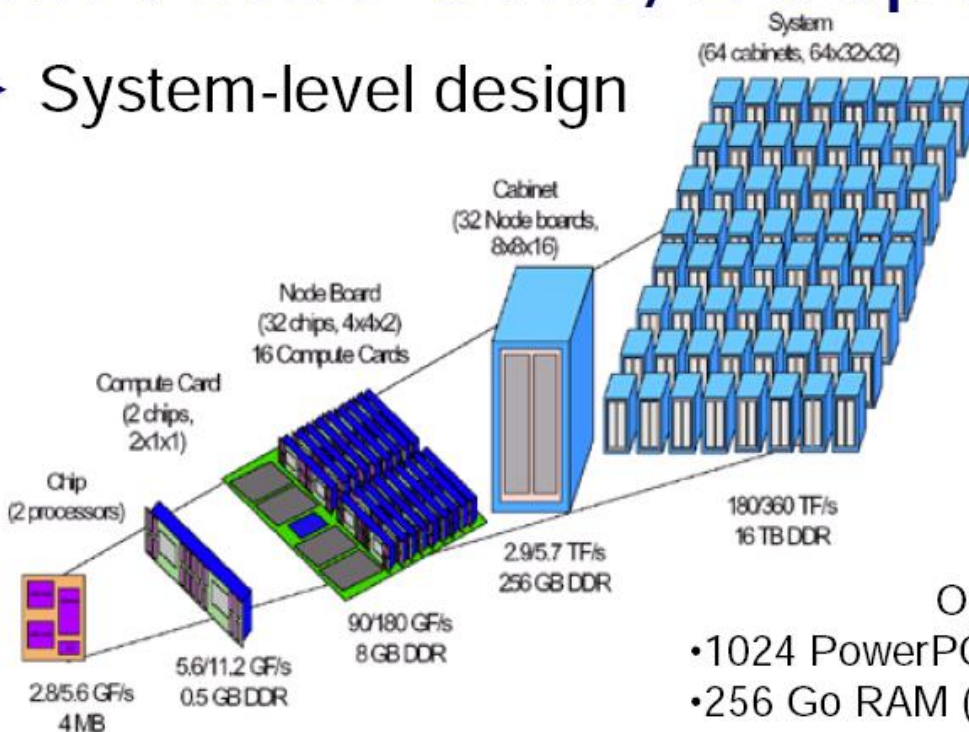
| MFLOPS         | GFLOPS         | TFLOPS            | PFLOPS            | EFLOPS            |
|----------------|----------------|-------------------|-------------------|-------------------|
| $10^6, 2^{20}$ | $10^9, 2^{30}$ | $10^{12}, 2^{40}$ | $10^{15}, 2^{50}$ | $10^{18}, 2^{60}$ |

|                   |              |                      |
|-------------------|--------------|----------------------|
| Cray Titan (2015) | 17,59 PFLOPS | } Трилион пута бржи! |
| ENIAC (1946)      | 350 op/s     |                      |

# Еволуција супер-рачунара

## IBM Blue Gene/L SuperComputer

### ► System-level design



#### One Cabinet

- 1024 PowerPC 700Mhz
- 256 Go RAM ( up to 2Go ) /
- 5.7 teraflops of processing power
- IBM version of a Linux Kernel on processing nodes
- Novell Linux on Management Nodes

# Еволуција супер-рачунара



НАСА - Силикон графикс кластер

# Еволуција супер-рачунара



Veliki [Linux](#) klaster na Chemnitz University of Technology, Germany

# Еволуција супер-рачунара



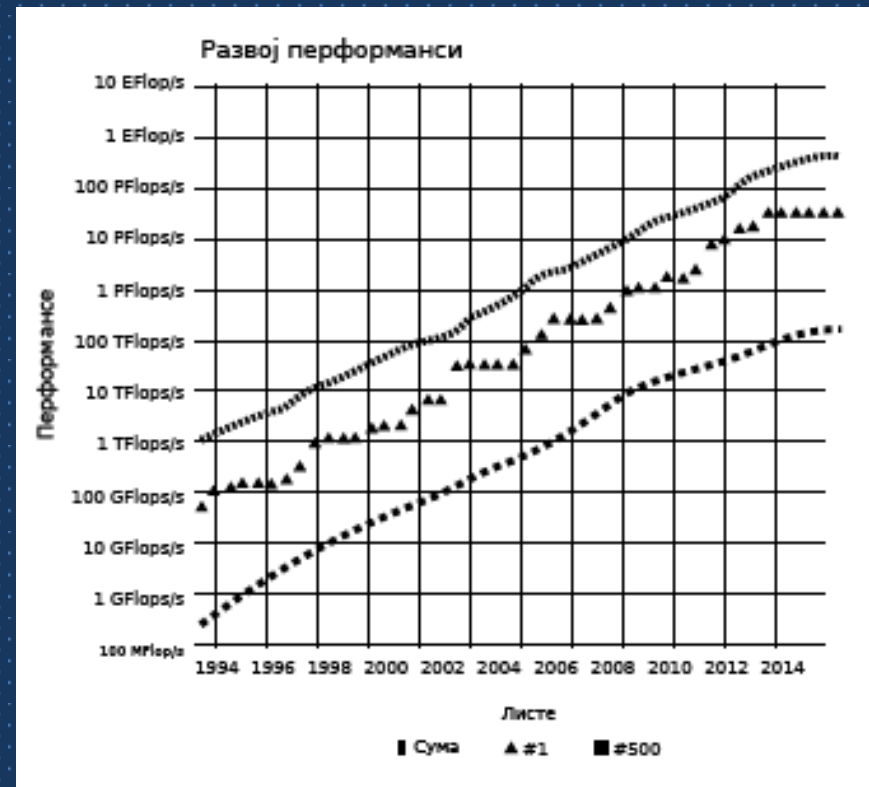
Sun Microsystems Solaris Cluster

# Еволуција супер-рачунара

Пошто су се и „обични“ рачунари изузетно брзо развијали, почетком 90-их година прошлог века, NASA-ини научници су искористили широко доступне процесоре, уобичајену интерконекцију и бесплатан софтвер и склопили први тзв. **Beowulf** кластер.

За релативно мала улагање су добили веома високе перформансе. Њихов систем се састојао од 16 стандардних рачунара са Intel 486DX4 процесорима који су повезани уобичајеном Ethernet мрежом од 10Mbps. На чворовима овог кластера био је инсталиран **Linux** оперативни систем, са батеријом GNU компајлера и библиотеком која подлеже MPI стандарду за интерпроцесну комуникацију и паралелизам.

*Beowulf* кластер постиже значајно бољи однос перформансе/цена од традиционалних супер-рачунара.



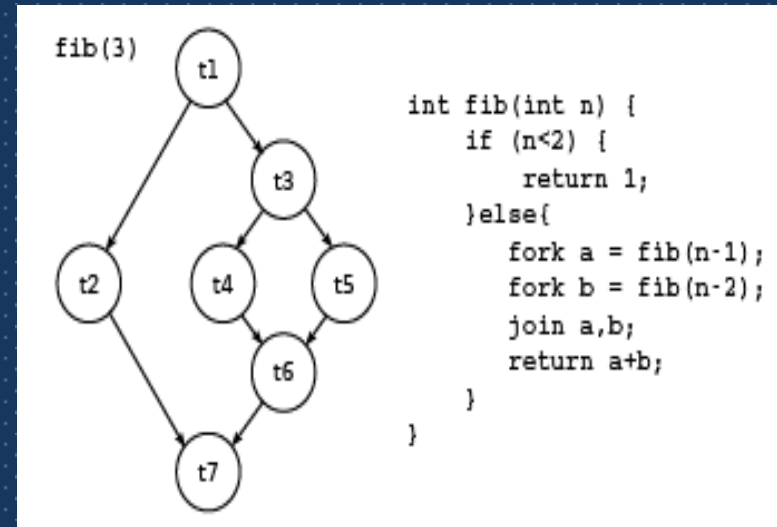
# Стратегије паралелног програмирања

- Потрага за конкурентностима

**Граф зависности** је усмерени граф у којем сваки чвор представља један задатак, док везе представљају зависности између њих. Стрелица из чвора  $U$  у чвор  $V$  значи да задатак  $U$  мора бити извршен пре него што задатак  $V$  почне. Ако нема путање из  $U$  у  $V$ , значи да су задаци независни и могу да се изврше истовремено. Дат је пример графа зависности који се користи приликом рекурзивног рачунања Фибоначијевог низа:

$$(F_n = F_{n-1} + F_{n-2}, \text{ где су } F_1 = F_2 = 1) \text{ или}$$

0,1,1,2,3,5,8,13,21, ..., N



# Стратегије паралелног програмирања

Паралелизам података је тип паралелизма у коме независни процеси примењују исту операцију на различите елементе скупа података. Сви таскови могу да се извршавају конкурентно. Уколико је са  $p$  означен број процесора, брзина може да се повећа и до  $p$  пута. На пример:

```
for (i=0; i<100; ++i)
    a[i]=b[i]+c[i];
```

Јасно је да свих 100 итерација наведене петље могу да се извршавају истовремено на 100 процесора.

Функционални паралелизам је тип паралелизма у коме независни задаци извршавају функције над истим или различитим елементима података. Степен конкурентности је лимитиран бројем конкурентних подзадатака. На пример:

```
a = 2;
b = 3;
m = (a+b)/2;
s = (a*2+b*2)/2;
v = s-m*2;
```

Овде трећи и четврти израз могу да се изврше паралелно.

# Приступи програмирању паралелних рачунара

У савременом паралелном рачунарству, присутно је, у мањој или већој мери, четири практична приступа, и то:

1. **Проширење компајлера** у смислу додавања могућности да секвенцијалне програме аутоматски преводи у паралелне. Задатак је модификација постојећег компајлера додавањем могућности да аутоматски детектује паралелизам у секвенцијалним програмима и да одатле произведе паралелни извршни фајл.
2. **Проширење постојећег језика**, додавање нових паралелних операција. Овај приступ се своди на додавање паралелних функција у постојећи секвенцијални језик. То су функције за креирање и терминирање паралелних процеса, њихову синхронизацију, међусобну комуницирају исл. Примери из праксе су MPI (Message Passing Interface), PVM (Parallel Virtual Machine), POSIX нити, OpenMP идр. Овај приступ је најлакши, најбржи, најјефтинији и најпопуларнији приступ паралелним програмирању. Захтева само развој библиотеке рутина.

# Пристапи програмирању паралелних рачунара

## 3. Додавање новог паралелног слоја на постојећи секвенцијални језик.

Паралелни програм можемо да посматрамо као да је двослојан. Доњи слој је компјутационо језгро, у коме процес манипулише сопственом порцијом података да би произвео своју порцију резултата. Овај слој може да се имплементира у постојећем секвенцијалном програмском језику. Горњи слој контролише креирање и синхронизацију процеса, као и партиционисање података међу процесима.

4. Увођење потпуно новог језика који природно подржава паралелизам. Пример је језик *Occam*, са потпуно новом синтаксом, који подржава и паралелно и секвенцијално извршавање процеса.

Други начин је додавање паралелних конструкција у већ постојећи програмски језик. Примери су *High Performance Fortran*, *C++ sa OpenMP*, *Smalltalk*, *Pony*, *Common Lisp*, *Ada*, *Java*, *Go*, *Scala*, *Cilk...*

Ту је и компајлерско решење CUDA компаније *nVidia*, које додаје специјалне инструкције за програмирање масивно паралелних графичких процесора. Предност овог приступа је што програмер предочава паралелизам самом компајлеру, што повећава вероватноћу да ће извршни програм достићи високе перформансе.

# Приступи програмирању паралелних рачунара

Efikasna paralelna implementacija serijskog programa ne može se postići pronalaženjem najefikasnijih paralelizacija svakog od njegovih koraka. Umesto toga, najbolja paralelizacija se može dobiti koraćanjem unazad i osmišljavanjem potpuno novog algoritma.

Naprimera, ako želimo da izračunamo sumu  $n$  vrednosti:

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

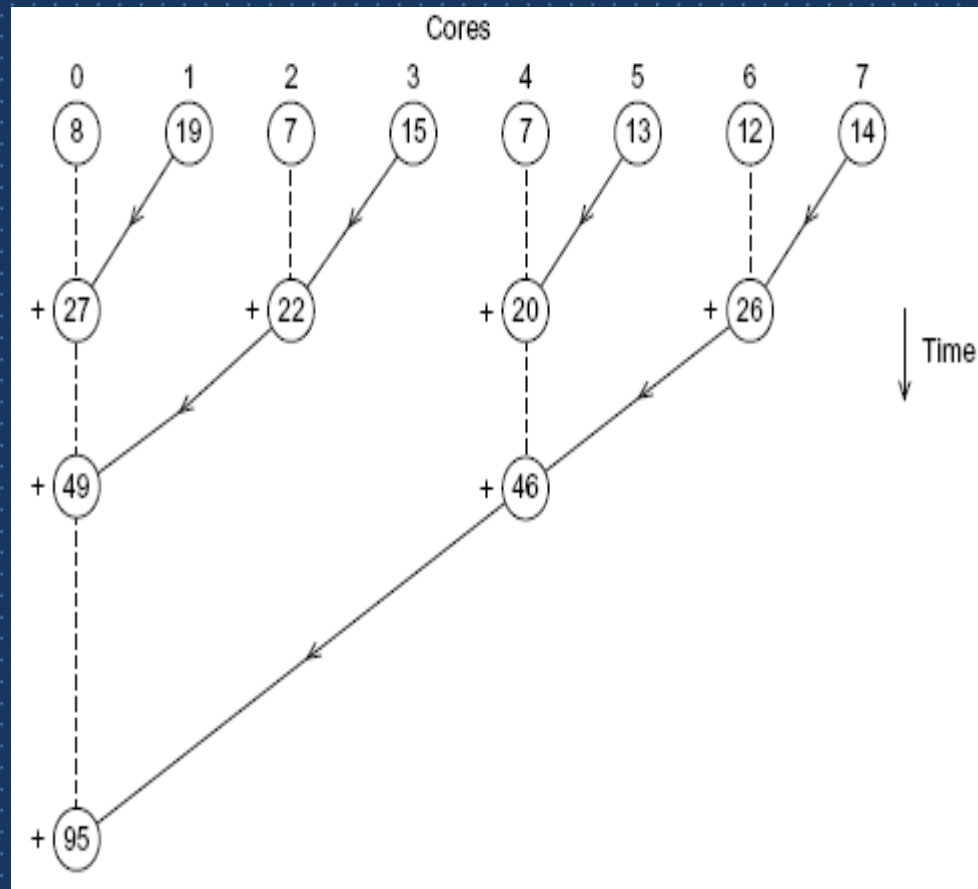
I ako imamo  $p$  jezgara pod uslovom da je  $p > n$ , tada svako jezgro može nezavisno formirati parcijalnu sumu od aproksimativno  $p/n$  vrednosti.

Za naprimera  $n=24$  sledi 1, 4, 3, 9, 2, 8, 5, 1, 1, 6, 2, 7, 2, 5, 0, 4, 1, 8, 6, 5, 1...

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
    my_x = Compute_next_value(. . .);
    my_sum += my_x;
}
```

Primer:

| Core   | 0 | 1  | 2 | 3  | 4 | 5  | 6  | 7  |
|--------|---|----|---|----|---|----|----|----|
| my_sum | 8 | 19 | 7 | 15 | 7 | 13 | 12 | 14 |



Možemo upariti jezgra tako se jezgru 0 dodaje rezultat jezgra 1, jezgru 2 se može dodati rezultat jezgra 3, jezgru 4 se može dodati rezultat jezgra 5 i tako dalje. Tada možemo ponoviti postupak samo sa dalje rangiranim jezgrama: jezgru 0 se dodaje rezultat 2, jezgru 4 se dodaje rezultat 6, i tako dalje. Sad su jezgra deljive sa 4, ponavlja se postupak, i tako dalje.

# Архитектура паралелних рачунара

## Архитектура мреже:

Од 60-их до средине 90-их година прошлог века, испитане су многе архитектуре паралелних рачунара. Неке компаније су се ослониле на VLSI (Very Large Scale Integration) технологију и развиле сопствене процесоре за своје паралелне рачунаре.

Друге су се определиле за широко доступне процесоре опште намене. Основна дебата у то време би се могла формулисати као: **Да ли паралелни рачунар треба да има неколико десетина врло моћних процесора или на хиљаде комерцијалних процесора?**

Данас су рачунарски системи са хиљадама примитивних процесора реткост, док специјализовани процесори нису могли да прате брзи напредак процесора широке намене.

Зато се данас паралелни рачунари углавом конструишу од уобичајених, тржишно доступних процесора.

# Архитектура паралелних рачунара

Интерконекционе мреже пружају могућност процесорима у два различита рачунара да међусобно комуницирају. У оквиру једног рачунарског система могу се користити и механизми дељене меморије, али је у било ком типу кластера, услед непостојања заједничког адресног простора, размена порука једини могући начин интерпроцесне комуникације.

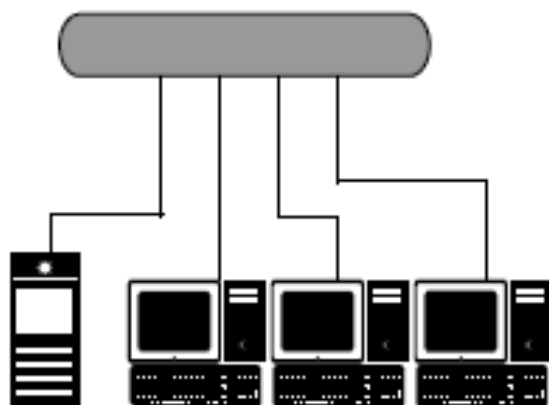
**Дељени медијум**, као што је, рецимо, *Ethernet* мрежа изведена коришћењем разводника (*hub*), дозвољава трансфер само једне поруке у једном временском тренутку. Како процесори шаљу поруке преко дељеног медијума, то значи да сви процесори слушају све поруке, а примају само оне намењене њима. Према CSMA (*Carrier Sense Multiple Access*) приступу, процесори пре слања ослушкују да ли је медијум слободан. Ако случајно два или више процесора пошаљу поруке истовремено, долази до колизије, па сваки од процесора чека неки насумични период времена пре поновног слања. Постоје и изведбе CSMA протокола у којима процесори уместо паузирања после колизије емитују не са вероватноћом један, већ са неком нижом вероватноћом.

**Свичовани медијум** подржава комуникацију типа од-тачке-до-тачке (P2P), и дозвољава конкурентну трансмисију више порука између различитих парова процеса и добро подржава скалирање, тј. додавање нових процесора.

Очигледно је да дељени медијум није погодан за интерпроцесорску комуникацију унутар паралелних рачунара.

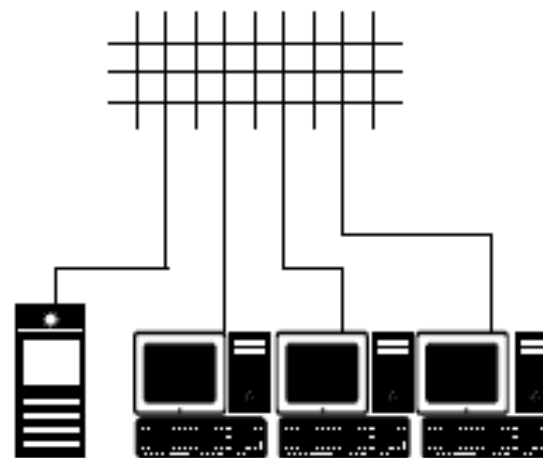
# Архитектура паралелних рачунара

Дељени медијум



Пропусни опсег: 10 - 40 Mbps

Свичовани медијум



Пропусни опсег: 100Mbps

# Топологије супер-рачунарских мрежа

Сваки процесор је повезан на сопствени свич. Свичеви су повезани са процесорима и/или другим свичевима.

Код **директне топологије** је однос броја свичева и броја процесора један према један.

Код **индиректне топологије** је однос броја свичева и процесора већи од 1:1. Наиме, неки свичеви само повезују друге свичеве.

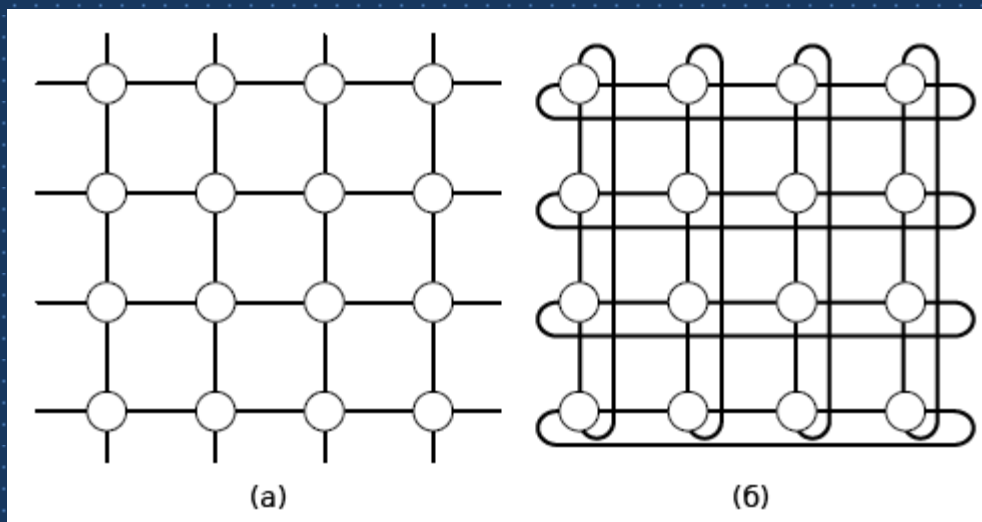
Перформансе дате конфигурације се могу квантификовати користећи неколико параметара. У супер-рачунарству се углавном користе следећи:

- **Дијаметар** је највећа удаљеност између два свича. Нижи дијаметар даје боље перформансе.
- **Бисекциона ширина** је минимални број веза између свичева које треба прекинути да би се мрежа поделила на два једнака дела. Виша бисекциона ширина је боља, јер омогућава већи проток података.
- **Број веза по свичу** је пожељно да буде константан и независан од величине мреже, јер се онда мрежа лакше скалира.
- **Дужина ивица** је пожељно да буде константна. Због лакшег скалирања је боље да се чворови и ивице организују у тродимензионом простору и да максимална дужина ивице буде константна и потпуно независна од величине саме мреже.

# Топологије супер-рачунарских мрежа

## Дводимензиона решетка:

Дводимензиона решетка је директна топологија у којој су свичеви поређани у решетку. Минимални дијаметар и максимална бисекциона ширина постижу се обликом квадрата. Тада су оба ова параметра реда  $\sqrt{n}$  за  $n$  чворова (процесора). **Обвојна варијанта** дводимензионе мреже има константан број ивица по чвору. Тада је могуће направити произвољно велику дводимензиону решетку са константном дужином ивице.

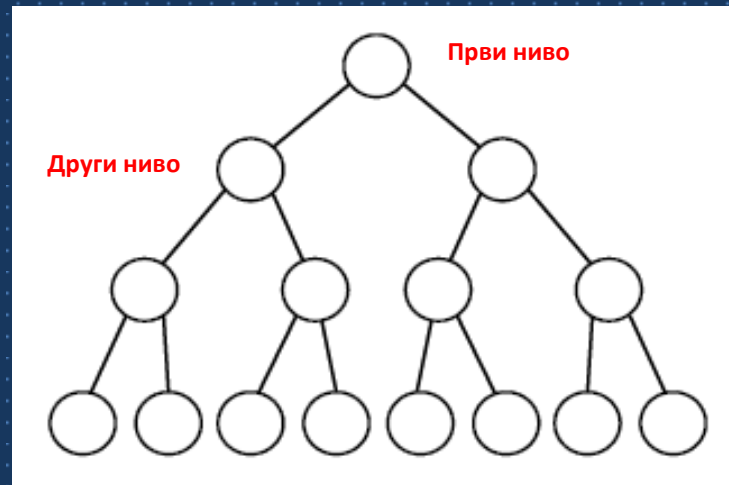


(a) Стандардна варијанта. (б) Обвојна варијанта

# Топологије супер-рачунарских мрежа

## Бинарно стабло:

Број процесора у топологији бинарног стабла је  $2^d$  где је  $d$  дубина стабла (0,1,2,3,...). Број свичева је  $2n - 1$  где је  $n$  број листова стабла који је једнак броју процесора, тј.  $S = 2 * 2^d - 1 = 2^{d+1} - 1$ . На првом нивоу се налази један свич, на другом два, на трећем четири итд.



$S = 2$

$d=4$ , Број проц.=8

Ако је са  $d$  означена дубина стабла, онда је број процесора један броју листова стабла, тј.  $n = 2^d$ . У обрнутом смеру, ако је дат број процесора, дубина стабла се може изразити као  $d = \log_2 n$ . Сваки процесор је повезан на лист бинарног стабла, па је ово индиректна топологија.

- Бисекциона ширина је минимална могућа - јединична.
- Унутрашњи свич чворови имају највише три везе, две за потомке и једну за родитеља.

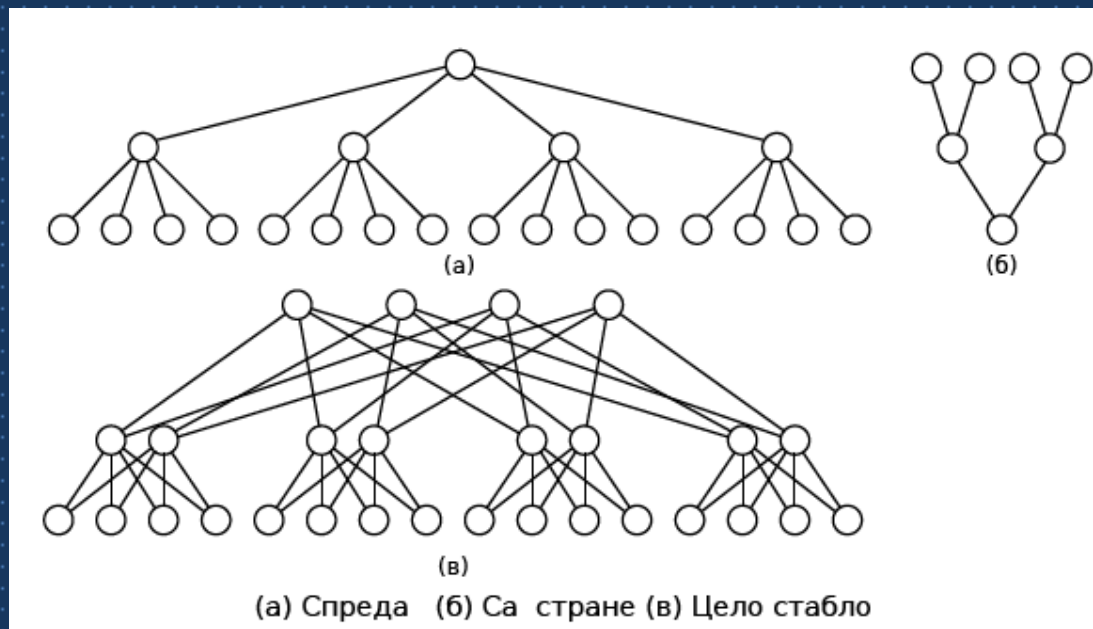
# Топологије супер-рачунарских мрежа

## Хиперстабло:

Хипертсабло је индиректна топологија која задржава мали дијаметар бинарног стабла, али има већу бисекциону ширину. Хиперстабло степена  $k$  и дубине  $d$  са предње стране изгледа као стабло  $k$ -тог реда дубине  $d$ , док из профила изгледа као обрнуто бинарно стабло дубине  $d$ .

Број процесора у стаблу је  $4^d$  односно  $k^d$ . Број свичева је  $S=2^{d+1} - 1$ . Сваки чвор има четворо деце (тј.  $k$  деце) и два родитеља. Значи, на сваком следећем нивоу има двоструко више чворова ( $k/2$  пута више). На првом нивоу имамо  $2^d$ , на другом  $2 * 2^d$ , на трећем  $4 * 2^d$  итд.

Број ивица по свич чвору никад није већи од шест (два родитеља и четворо потомака).



# Топологије супер-рачунарских мрежа

## Топологија лептира:

Топологија лептира је индиректна топологија. Број процесора у овој врсти мреже је  $2^d$ , где је  $d$  број тзв. рангова.

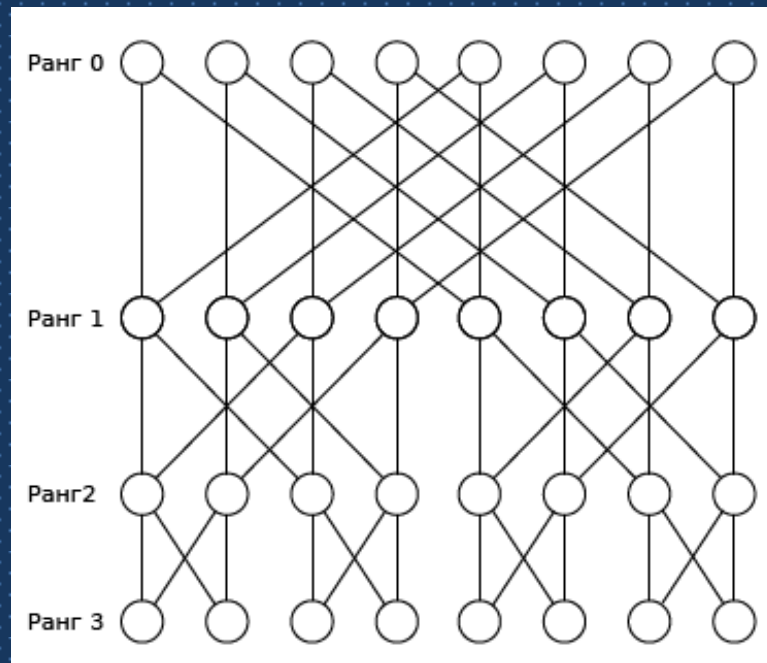
Број свич чворова у овој мрежи је  $n(\log_2 n + 1)$ . Свичеви су поређани у  $n$  колона (сваки за по један процесор) и  $\log_2 n + 1$  врста или рангова. Тада је  $\log_2 n$  дубина, зато што се нпр. осам чворова дели у четири, четири се деле у по два, два се деле у по један. Ако је  $\log_2 n$  дубина, то значи да од врха до дна има  $\log_2 n$  веза које повезују  $\log_2 n + 1$  чворова, одакле долази  $\log_2 n + 1$  рангова.

Дијаметар мреже је  $\log_2 n$ , док је бисекциона ширина једнака  $n$ .

## Алгоритам за рутирање порука:

Сваки свич чвор узима водећи бит из поруке.

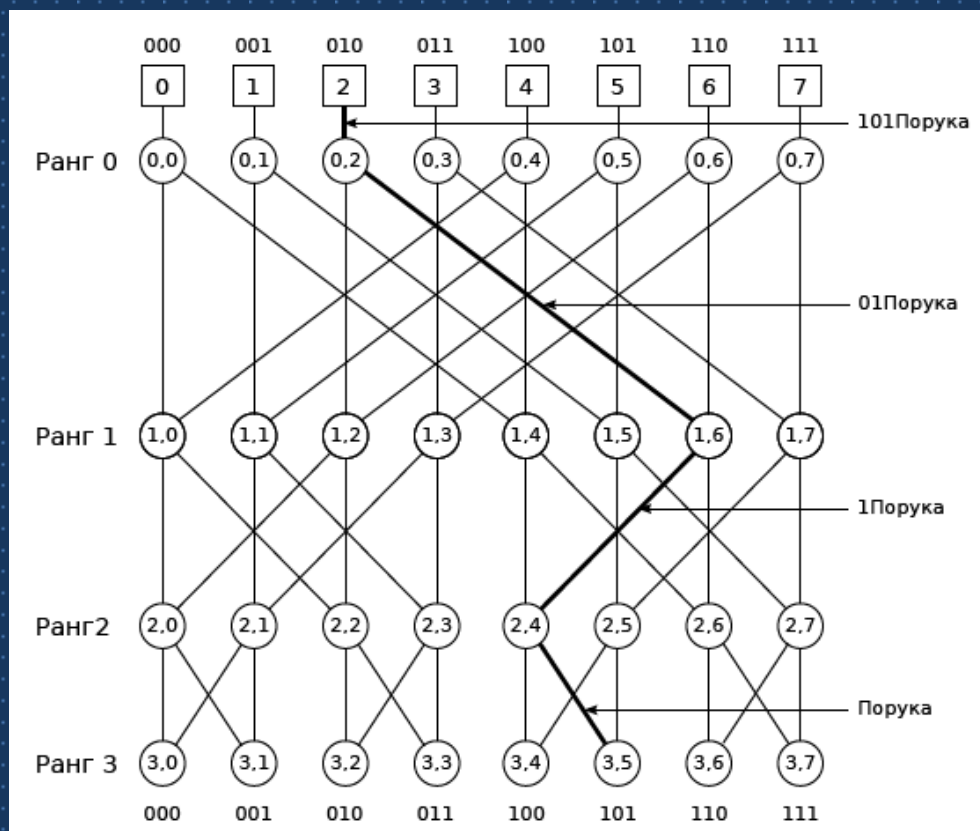
- Ако је тај бит нула, шаље остале битове доле лево,
- Ако је тај бит један, доле десно.



- Сваки свич чвор има константан број од четири везе.
- Порука је, у ствари, бинарна ознака редног броја одредишног процесора.

# Топологије супер-рачунарских мрежа

Рутирање лептир топологије иде на следећи начин: Рецимо, ознаком 101 из било ког процесора стижемо до процесора за ознаком 5 . На првом нивоу, све везе налево воде у колоне чији бинарни број почиње са 0, а све везе надесно у колоне које почињу са 1. На другом нивоу, лево воде у колоне чији је други бинарни број нула а, десно чији је други бит један итд.



Рутирање у топологији лептира

# Топологије супер-рачунарских мрежа

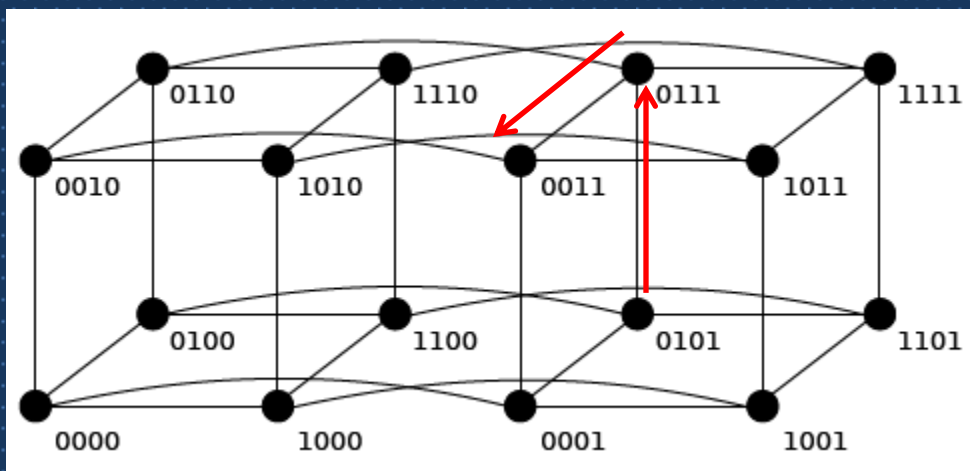
## Хиперкоцка:

Хиперкоцка, у ствари, представља топологију лептира у којој је свака колона свич чворова колабирана у један чвор. Тиме су избрисане везе које не мењају бит, а сваки чвор има везу која мења први бит, везу која мења други бит, и тако до  $\log_2 n$ . Пошто је код лептира свака колона била везана за по један процесор, овде је сваки чвор везан за по један процесор, па је топологија, у ствари, директна. Дакле, број процесора у хиперкоцки је  $2^d$ , као и број свичева.

Процесори су нумерисани од 0 до  $2^d - 1$ . У бинарном запису имају  $d$  цифара. Свич чворови су суседни ако се разликују за тачно један бит.

Дакле:

$$\begin{array}{ccccccc} 0001 & 0010 & 0100 & 1000 & 10000-1=1111 & & \\ 2^0 = 1 & 2^1 = 2 & 2^2 = 4 & 2^3 = 8 & 2^4 - 1 = 15 & d = 4 = \log_2 16. & \end{array}$$



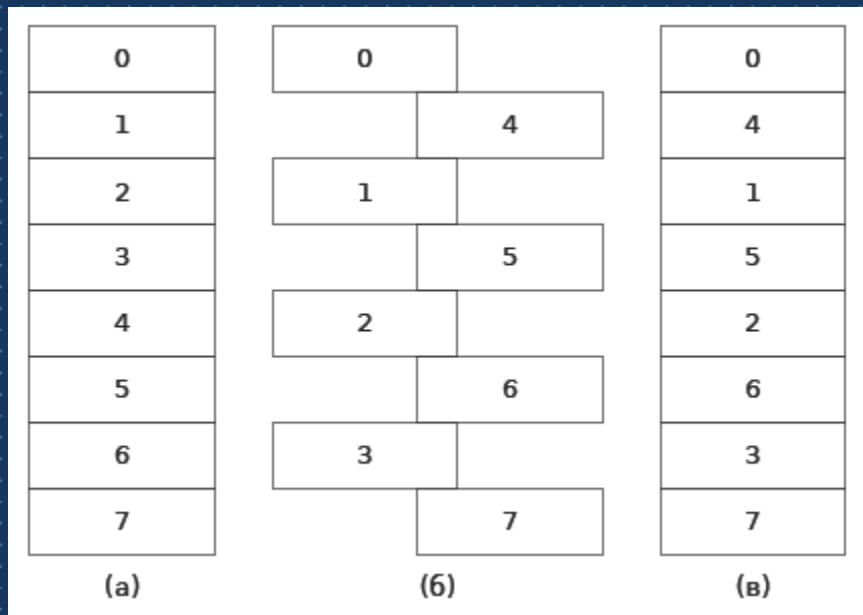
Рутирање порука у хиперкоцки одређује чињеница да број бита у којима се ознаке свичева разликују одређује њихово растојање. На пример, од процесора 0101 (5) до процесора 0011 (3) растојање је једнако 2:

# Топологије супер-рачунарских мрежа

## Shuffle exchange топологија:

Идеја за ову врсту топологије је савршено измешан шпил карата, као што је приказано на слици. Ово је пример директне топологије. Број процесора је  $n = 2^d$ , док је број свичева исти. Нумеришу се од 0 до  $n - 1$ . У *shuffle exchange* архитектури постоје две врсте веза:

- **Exchange везе** - повезују свичеве чије се бинарне ознаке разликују у најмање значајном биту. Ове везе су двосмерне.
  - **Shuffle везе** - повезују свич  $i$  са свичом  $j$ , где се  $j$  добија цикличном битовском ротацијом бинарног записа броја  $i$ . Ове везе су једносмерне.
- Ово је пример директне топологије.



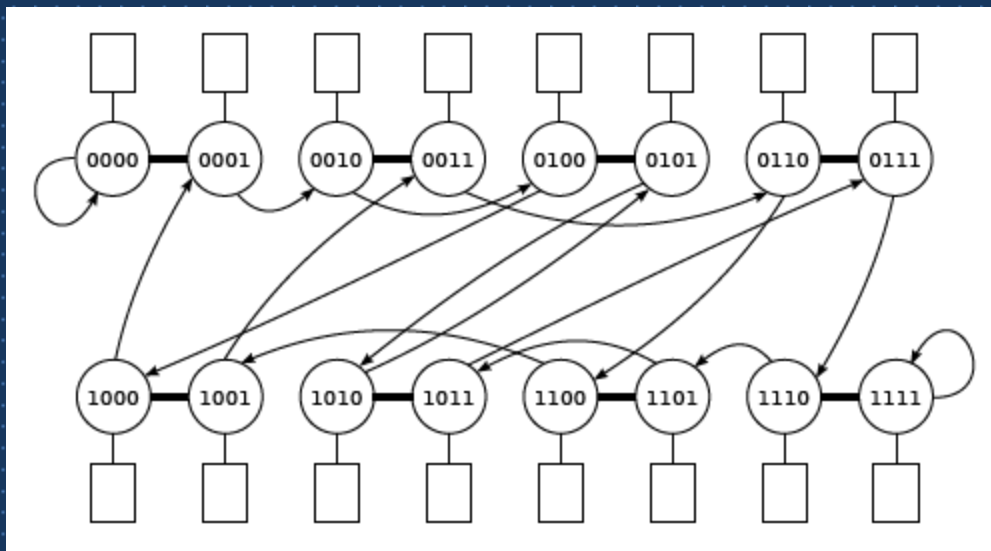
Дакле, као што се види са слике, да би се добила позиција карте у измешаном шпилу, врши се померање битова уз премештање првог бита слева на крај (ротација). На пример:

|           |           |           |               |
|-----------|-----------|-----------|---------------|
| 000 → 000 | 100 → 001 | 001 → 010 | 101 → 011 ... |
| 0 → 0     | 4 → 1     | 1 → 2     | 5 → 3         |

# Топологије супер-рачунарских мрежа

Дијаметар за мрежу од  $n$  процесора је  $2 \log_2 n - 1$ . Чворови са највећом удаљеношћу су 00...00 и 11...11. Да би се дошло од једног до другог потребно је  $\log_2 n$  инвертовања (*exchange*) и  $\log_2 n - 1$  ротирања (*shuffle*), што укупно даје  $2 \log_2 n - 1$  корака:

0000  $\xrightarrow{E}$  0001  $\xrightarrow{S}$  0010  $\xrightarrow{E}$  0011  $\xrightarrow{S}$  0110  $\xrightarrow{E}$  0111  $\xrightarrow{S}$  1110  $\xrightarrow{E}$  1111  
 $n = 16$   
 $E = 4 = \log_2 n$   
 $S = 3 = \log_2 n - 1$ .



Мрежа *shuffle exchange* са 16 чворова. Дебеле линије представљају двосмерне *exchange* везе, а стрелице једносмерне *shuffle* везе.

Бисекциона ширина за *shuffle exchange* топологију износи  $n/(\log_2 n)$ .

- Карактеристичан је и константан број ивица по чвору - две долазеће и две одлазеће.
- Неповољно утиче само чињеница да дужина најдуже ивице расте са величином мреже, што отежава скалирање.

# Векторски рачунари

Векторски процесори:

- Мотивација
- Основне векторске архитектуре
- Векторско време извршења
- Векторске load/store јединице и векторски меморијски системи
- Дужина вектора
- Корак вектора
- Технике за побољшање векторских перформанси
- Програмски језици за векторске рачунаре

# Векторски рачунари

Проточност и ILP значајно поправљају перформансе али постоје ограничења:

- Период тактног сигнала.

- Брзина добављања и декодирања инструкција.

Брзи проточни процесори су корисни за велике техничке апликације али велики скупови података са мало локалности изазивају слабе перформансе хијерархијског меморијског система.

# Векторски рачунари

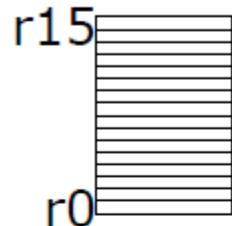
И поред техника које ублажавају овај проблем он је и даље велики за неке апликације.

Векторски процесори пружају могућност операција на нивоу виших програмских језика које раде над векторима.

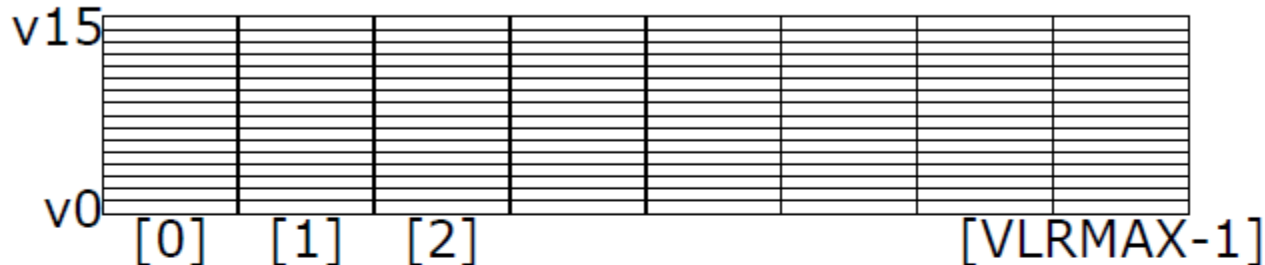
Типична векторска инструкција еквивалентна је читавој петљи где се у једној итерацији обрађује један елемент вектора.

# Vector Programming Model

Scalar Registers



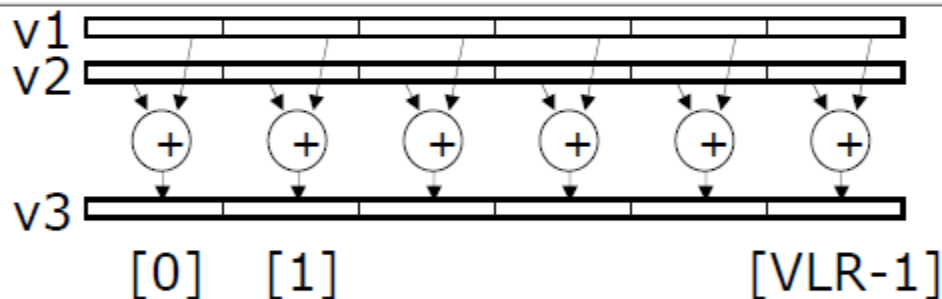
Vector Registers



Vector Length Register VLR

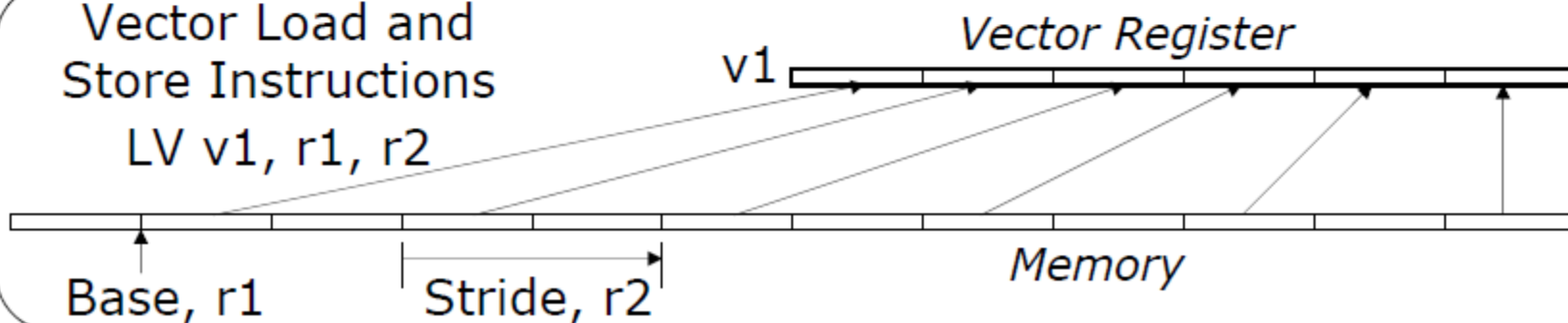
Vector Arithmetic  
Instructions

ADDV v3, v1, v2



Vector Load and  
Store Instructions

LV v1, r1, r2



# Векторски рачунари

Векторске инструкције имају неколико важних особина које решавају поменуте проблеме:

- Израчунавање сваког резултата независно од израчунавања претходног, што је могуће код проточног система велике дубине, али се овде не јављају хазарди по подацима.
- Једна векторска инструкција специфицира велики износ посла.
- Векторке инструкције које приступају меморији имају познат шаблон приступа.
- Како је цела петља замењена векторком инструкцијом чије је понашање унапред одређено, управљачки хазарди су непостојећи.

# Векторски рачунари

Из наведених разлога се векторске операције могу брже извршавати од секвенце скаларних операција над истим бројем података.

Код векторских процесора операције над појединим елементима вектора су проточне.

Проточни систем не обухвата само аритметичке операције већ и приступ меморији и одређивање ефективне адресе.

Осим овога, многи *high-end* процесори дозвољавају и да се више векторских операција извршава истовремено.

# Векторски рачунари

Векторски процесор се обично састоји од обичне проточне скаларне јединице и векторске јединице.

Све функционалне јединице у оквиру векторске јединице имају кашњење од неколико тактних периода.

Ово омогућава краћи период тактног сигнала и компатибилно је са векторским операцијама које се дуго извршавају и које се могу обрађивати проточним системом велике дубине без генерисања хазарда.

# Векторски рачунари

## Основне векторске архитектуре

Постоје два основна типа векторских архитектура:

- Векторско-регистарски процесори.

- Меморија-меморија векторски процесори.

У векторско-регистарским процесорима све операције, осим *load* и *store*, се обављају над регистрима.

# Векторски рачунари

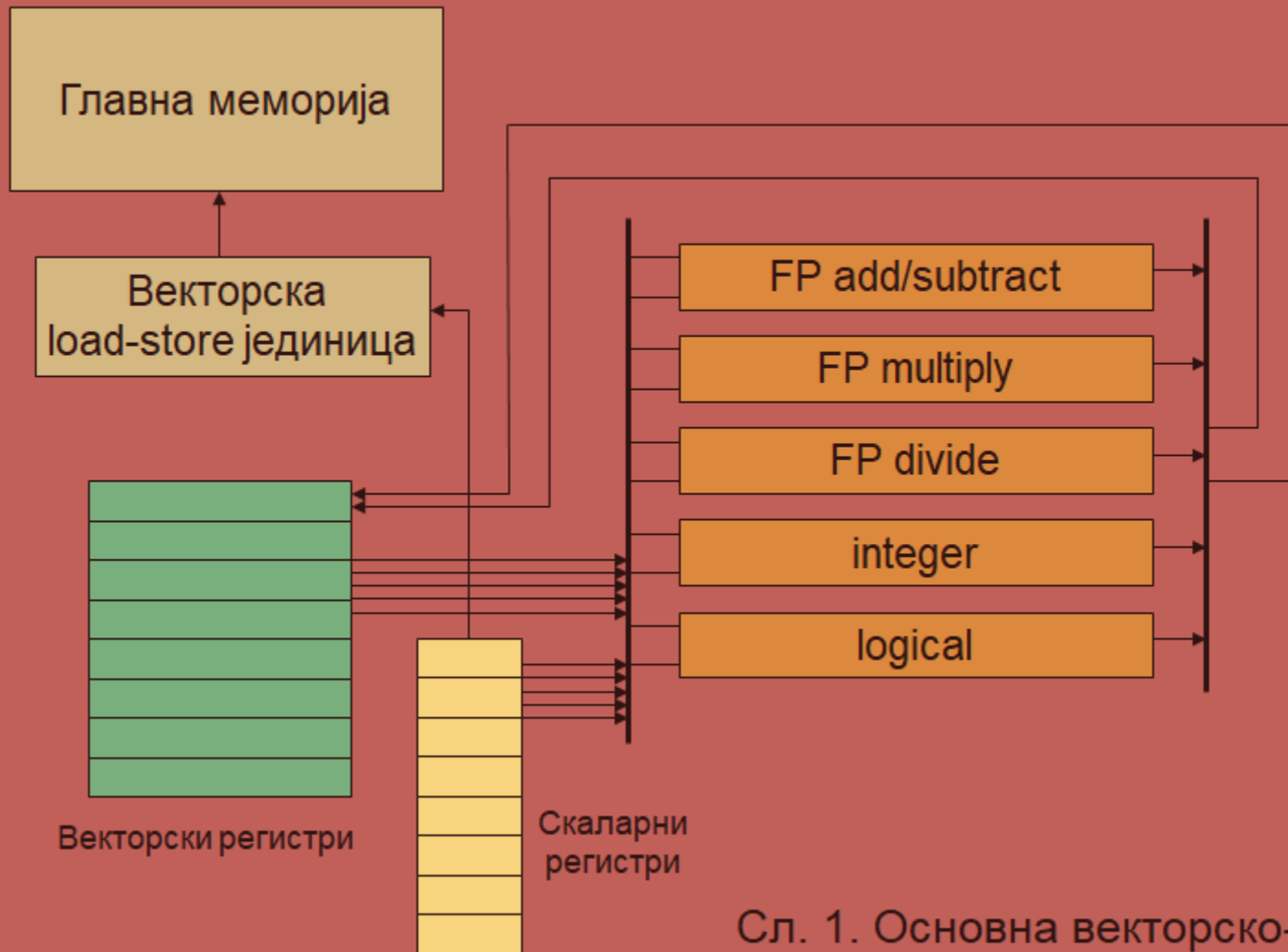
Сви велики векторски рачунари на тржишту од касних 80-тих су овог типа:

- Cray Research: CRAY-1, CRAY-2, X-MP, Y-MP, C-90;
- japanski NEC SX/2 и SX/3,
- Fujitsu VP200 i VP400, Hitachi S820) као и
- супер-мини рачунари (Convex C-1 и C-2).

У векторском процесору типа меморија-меморија све векторске операције су типа меморија-у-меморију. Први векторски рачунари били су овог типа (CDC фамилија).

# Векторски рачунари

## Основне векторске архитектуре



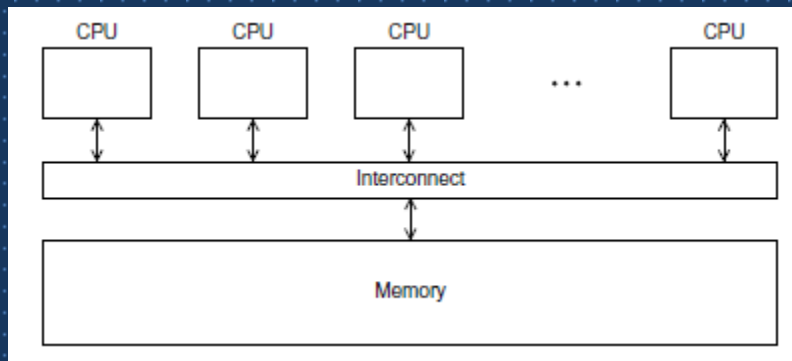
Сл. 1. Основна векторско-регистарска архитектура.

# Векторски рачунари

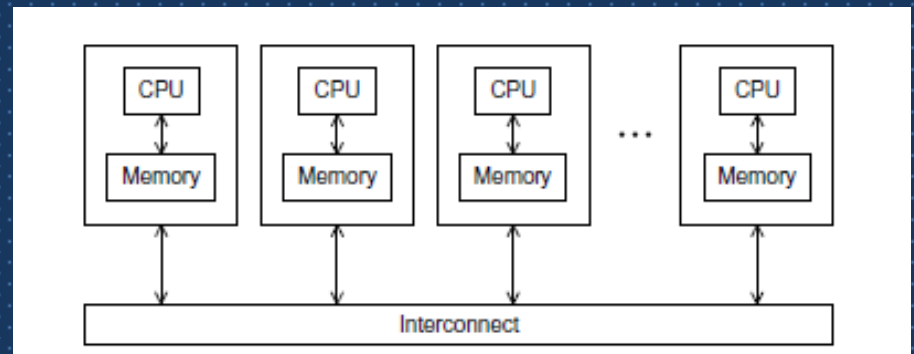
Основне компоненте ове архитектуре су:

- Векторски регистри. Сваки векторски регистар је фиксне дужине намењен да садржи један вектор. Сваки векторски регистар мора да има најмање два порта за читање и један порт за упис.
- Векторске функционалне јединице. Свака јединица је потпуно проточна и може да започне нову операцију после сваког тактног периода. Управљачка јединица мора да детектује хазарде.
- Векторска *load-store* јединица. То је векторска меморијска јединица која чита и уписује вектор у меморију.
- Скуп скаларних регистара. Њихови подаци могу бити улаз у векторске функционалне јединице а служе и за израчунавање адреса које се преносе векторској *load-store* јединици.

# Векторски рачунари - меморија



U sistemu zajedničke memorije zbirka autonomnih procesora povezana je na memorijski sistem putem mreže za povezivanje, a svaki procesor može pristupiti svakoj memorijskoj lokaciji. U sistemu zajedničke memorije, procesori obično implicitno komuniciraju pristupom deljenim strukturama podataka.



U sistemu distribuirane memorije svaki procesor je uparen sa vlastitom privatnom memorijom, a parovi procesora-memoriije komuniciraju preko interkonekcijske mreže. Tako u distribuiranim memorijskim sistemima procesori obično komuniciraju slanjem poruka ili korišćenjem posebnih funkcija koje omogućuju pristup memoriji drugog procesora.

# Векторски рачунари

Векторско време извршења

Време извршења секвенце векторских операција првенствено зависи од три фактора:

- Дужина вектора.

- Структурни хазарди међу операцијама.

- Зависности по подацима.

За дату дужину вектора и *брзину иницијације* (брзина којом векторска јединица користи нове операнде и производи нове резултате) можемо одредити време потребно за једну векторску инструкцију.

Брзина иницијације је обично једна инструкција по тактном периоду за индивидуалну операцију.

# Векторски рачунари

Неки суперрачунари имају векторске јединице које производе два или више резултата по тактном периоду, док неки имају јединице које се не могу у потпуности учинити протичним.

Претпоставићемо да је брзина иницијације 1 по тактном периоду, тако да је време потребно за једну векторску инструкцију приближно једнако дужини вектора.

Под *конвојом* ћемо подразумевати скуп векторских инструкција које се могу потенцијално извршавати заједно у једном тактном периоду.

Инструкције у конвоју не смеју садржати било који структурни хазард или хазард по подацима.

Појму конвоја придружићем појам *сагласја* који се може користити за процену перформанси векторске секвенце која се састоји од конвоја. Сагласје је приближна мера времена извршења за векторску секвенцу.

Мерење сагласја је независно од дужине вектора. Према томе векторска секвенца од  $t$  конвоја која се извршава у  $t$  сагласја, а за дужину вектора  $n$ , захтеваће приближно  $t \times n$  тактних периода.

# Векторски рачунари

Векторске load-store јединице и векторски меморијски системи

Понашање векторске load-store јединице је компликованије од понашања аритметичких функционалних јединица. Време стартовања за операцију load је време потребно да се прва реч смести у регистар. Ако се остатак вектора може додати без застоја, брзина иницијације једнака је брзини којом се једна реч може прочитати или уписати.

Насупрот једноставнијим функционалним јединицама, брзина иницијације није увек један по тактном периоду. Типично време стартовања load-store јединице је и до 50 тактова (CRAY-1 и CRAY X-MP имају ово време између 9 и 17 тактних периода).

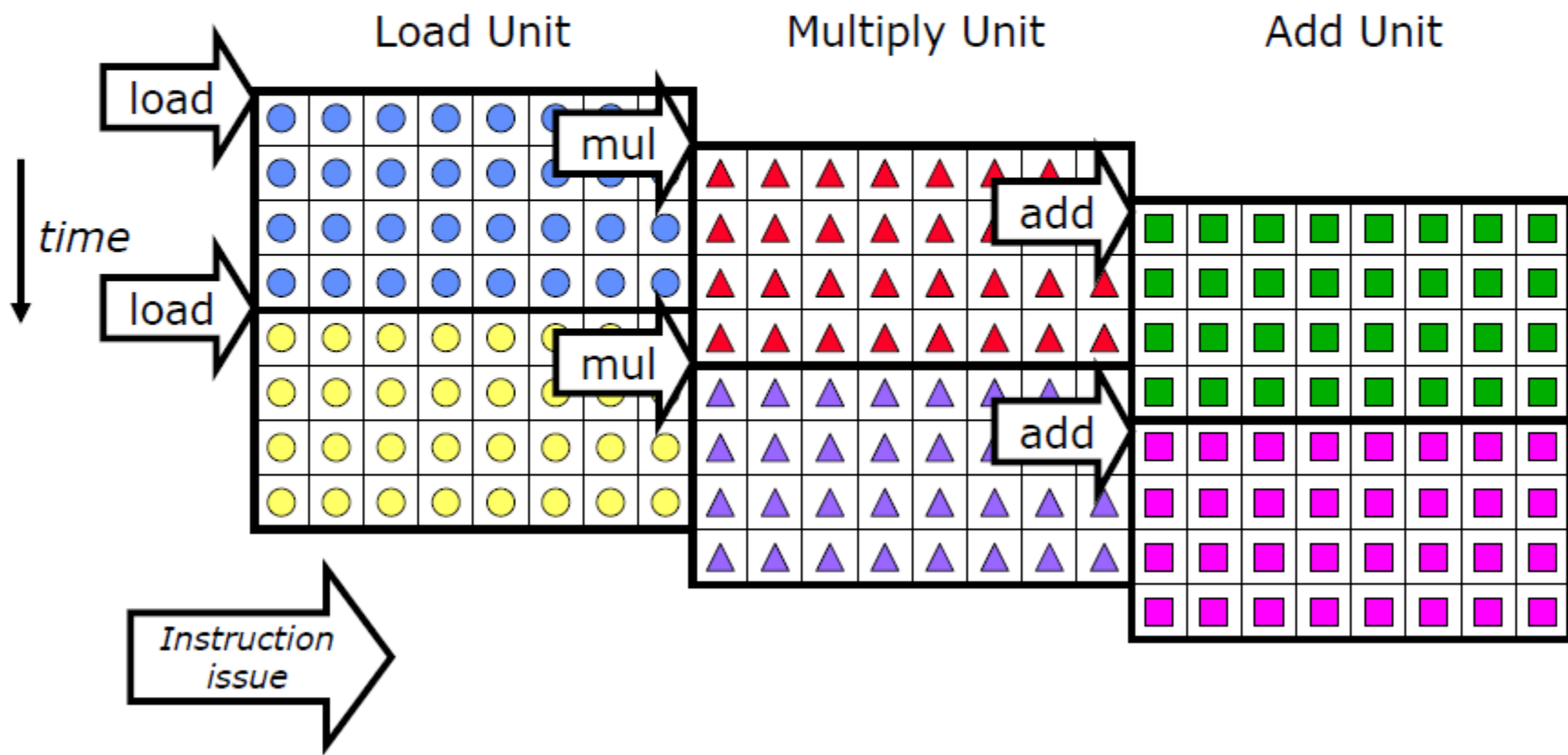
Већина векторских процесора користи меморијске банке уместо обичног преклапања из два разлога:

- Многи векторски рачунари подржавају вишеструка читања или уписе по истом тактном интервалу. Да би се подржао вишеструки симултани приступ меморијски систем мора да има вишеструке банке и да буде у стању да управља независним адресирањем банака.
- Многи векторски процесори подржавају могућност читања и уписа података који нису секвенцијални. У таквим случајевима независно адресирање банака је пожељније од преклапања.

# Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

- example machine has 32 elements per vector register and 8 lanes



Complete 24 operations/cycle while issuing 1 short instruction/cycle

# Векторски рачунари

## Дужина вектора

Векторско регистарски процесори имају природну дужину вектора одређену бројем елемената у сваком од векторских регистара. У реалним програмима дужина појединих векторских операција није позната у време компилације.

Шта више, различити делови кода могу да захтевају различите дужине вектора.

```
do 10 i = 1,n  
10 Y(i) = a*X(i) + Y(i)
```

Решење овог проблема је у креирању регистра дужине вектора (VLR – *vector length register*).

Овај регистар би садржао дужину сваког вектора, а та дужина не би могла да буде већа од максималне дужине вектора (MVL) дефинисане од стране процесора.

# Векторски рачунари

Шта ако је стварна дужина вектора већа од MVL?

Овај проблем може се решити техником извлачења трака (*strip mining*).

Ова техника претпоставља генерисање таквог кода да се свака векторска операција обавља за онај део вектора који није већи од MVL.

```
low = 1
    VL = (n mod MVL)
    do 1 j = 0, (n / MVL)
        do 10 i = low, low+VL-1
            Y(i) = a*X(i) + Y(i)
10    continue
    low = low + VL
    VL = MVL
1    continue
```

Осим губитака при стартовању, морамо да урачунамо и губитке извршавања петље код које су извучене траке.

# Векторски рачунари

## Корак вектора

Други проблем односи се на то да позиција у меморији суседних елемената вектора не мора бити секвенцијална.

```
do 10 i = 1,100
  do 10 j = 1,100
    A(i,j) = 0.0
    do 10 k = 1,100
10 A(i,j) = A(i,j) + B(i,k)*C(k,j)
```

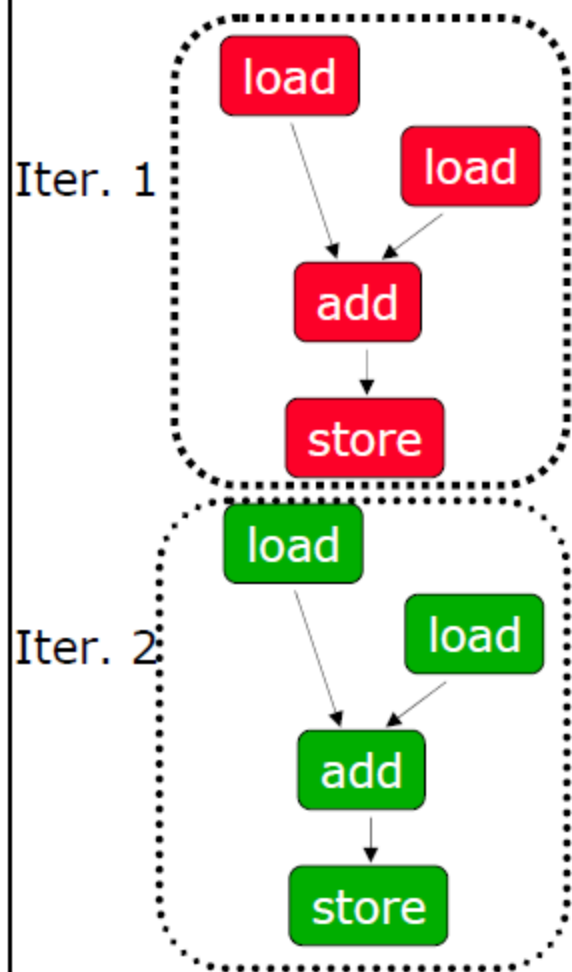
У изразу са лабелом 10 може се векторисати множење сваке врсте матрице  $B$  са сваком колоном матрице  $C$  и вршити извлачење трака унутрашње петље којој је  $k$  бројач петље.

Да би ово урадили морамо да размотримо начин смештања елемената матрице у меморију.

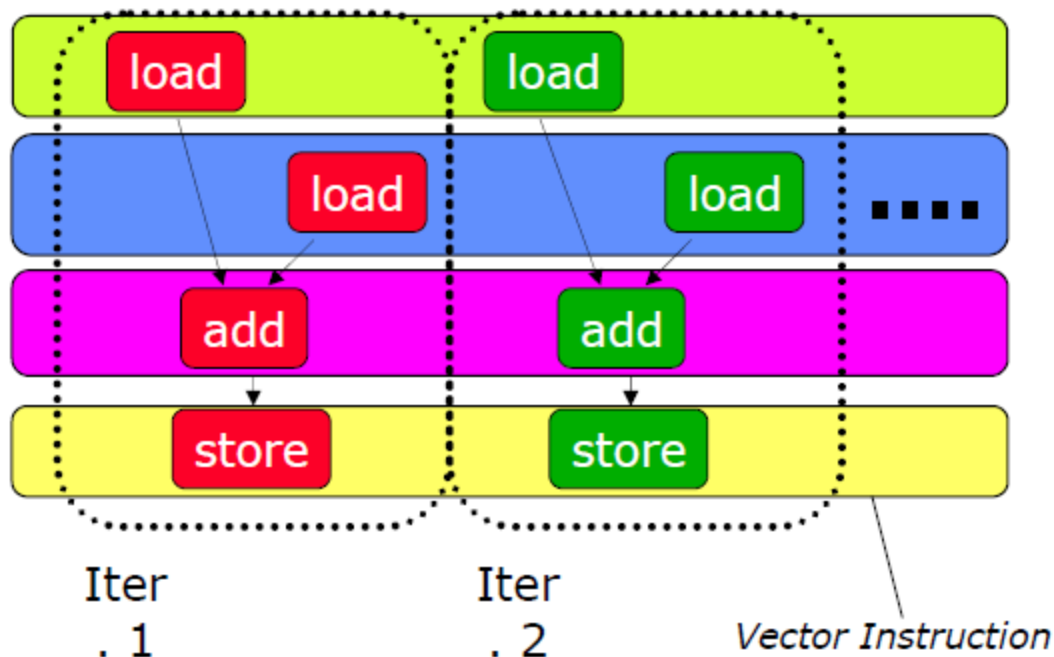
# Automatic Code Vectorization

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```

Scalar Sequential Code



Vectorized Code



Vectorization is a massive compile-time  
reordering of operation sequencing  
⇒ requires extensive loop dependence analysis

# Векторски рачунари

## Корак вектора

Растојање које одваја елементе који треба да буду у сакупљени у један векторски регистар назива се корак вектора (*vector stride*).

Када се вектор напуни у векторски регистар онда се са њим може радити као да су у питању логички суседни елементи.

Према томе, векторско-регистарски процесори могу да обрађују нејединичне кораке користећи само векторске *load* и *store* операције које имају могућност подешавања корака.

Треба имати у виду да до меморијских конфликта не може да дође уколико су корак вектора и број меморијских банака међусобно прости бројеви; у супротном конфликти су могући.

# Векторски рачунари

Технике за побољшање векторских перформанси

Ланчање. Размотримо једноставну векторску секвенцу

```
MULTV  V1, V2, V3  
ADDV   V4, V1, V5
```

Према досадашњим разматрањима ове две инструкције морају да буду у различитим конвојима.

Међутим, ако би се векторски регистар, V1 у овом случају посматрао као група индивидуалних регистара, онда можемо да применимо технику ланчања.

Ланчање дозвољава да се векторска операција започне што је пре могуће, тј. чим поједини елементи вектора буду доступни. Резултат се из прве функционалне јединице у ланцу прослеђује наредној.

У пракси се ланчање често имплементира тако што се омогући процесору да врши истовремено упис и читање регистра, али над различитим елементима вектора.

# Векторски рачунари

Прве имплементације ланчања радиле су по систему прослеђивања уз временска ограничења за изворне и одредишне инструкције у ланцу.

Новије имплементације користе *флексибилно ланчање* које дозвољава векторској инструкцији да се уланчава са било којом другом активном векторском инструкцијом, под условом да то не изазива структурне хазарде.

Овде је неопходно постојање више улазних и излазних портова за векторске регистре.

Условно извршење исказа. Условна гранања ограничавају степен векторизације петљи.

```
do 100 i = 1, 64
    if (A(i) .ne. 0) then
        A(i) = A(i) - B(i)
    endif
100 continue
```

Ова петља се не може векторисати због условног извршења тела петље.

# Векторски рачунари

## Технике за побољшање векторских перформанси

За решење проблема користи се управљање маском вектора.

Ради се о вектору логичких вредности дужине MVL који управља извршењем векторских инструкција баш као што условне инструкције користе логичке изразе.

```
LV      V1, Ra ; Ra адреса вектора A
LV      V2, Rb ; Rb адреса вектора B
LD      F0, #0 ; puni FP nulu u F0
SNESV 0, V1          ; postavlja VM(i) na 1
          ; ako je V(i) <> F0
SUBV    V1, V1, V2    ; oduzimanje prema masci
          ; вектора
CVM          ; postavlja masku na sve 1
SV      Ra, V1 ; smesta rez. u A
```

# Векторски рачунари

Многи новији процесори подржавају управљање маском вектора.

Постоје и недостаци. Условне инструкције захтевају одређено време и када услов није испуњен. Ипак елиминација гранања и управљачких зависности може ове инструкције да учини бржим чак иако не раде користан посао.

Ретко посегнуте матрице. Постоје технике које омогућавају да се програми који раде са ретко посегнутим матрицама извршавају у векторском начину рада.

```
do 100 i = 1,n
100  A(K(i)) = A(K(i)) + C(M(i))
```

# Векторски рачунари

Основни механизам подршке раду са РПМ су распрши-скупни операције које користи индексне векторе. Циљ таквих операција је прелазак са згуснуте репрезентације матрице на обичну репрезентацију.

Операција *скуп* користи индексни вектор и добавља векторе чији елементи су на адресама које се добијају сабирањем базне адресе са офсетом датим у индексном вектору. Резултат је *густ* вектор у векторском регистру.

Пошто се ти елементи обраде у густом облику, ретко посегнути вектор се може сместити у развијеном облику операцијом типа *распрши*.

|      |             |                  |
|------|-------------|------------------|
| LV   | Vk, Rk      | ; napuni K       |
| LVI  | Va, (Ra+Vk) | ; napuni A(K(I)) |
| LV   | Vm, Rm      | ; napuni M       |
| LVI  | Vc, (Rc+Vm) | ; napuni C(M(I)) |
| ADDV | Va, Va, Vc  | ; saberi vektore |
| SVI  | (Ra+Vk), Va | ; smesti A(K(I)) |

# Vector Code Example

# C code

```
for (i=0; i<64; i++)
    C[i] = A[i] + B[i];
```

# Scalar Code

```
LI R4, 64
loop:
    L.D F0, 0(R1)
    L.D F2, 0(R2)
    ADD.D F4, F2, F0
    S.D F4, 0(R3)
    DADDIU R1, 8
    DADDIU R2, 8
    DADDIU R3, 8
    DSUBIU R4, 1
    BNEZ R4, loop
```

# Vector Code

```
LI VLR, 64
LV V1, R1
LV V2, R2
ADDV.D V3, V1, V2
SV V3, R3
```

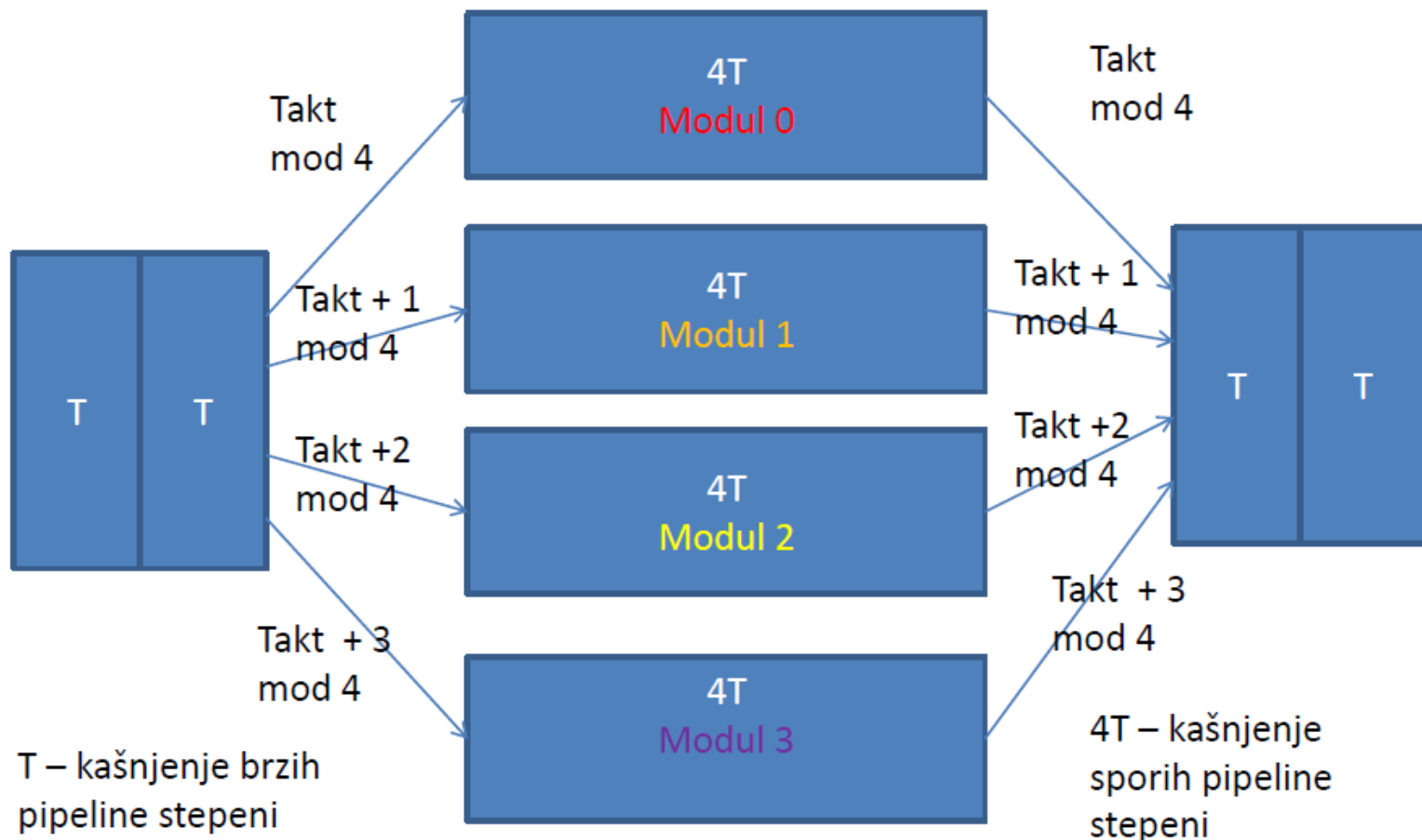
# Векторски рачунари

## Paralelizam je u DoAll petljama?

- Odvojimo deo mašine za skalarni deo koda sa neregularnim paralelizmom i uradimo neku paralelizaciju (skalarni procesor)
- Deo mašine odvojimo samo za programske petlje, realno za DoAll i napravimo da postoji hardverska podrška za nizove (vektorski procesor)
- Svaki deo radi ono za šta je bolji, a kompajler ili čak programer odvaja kôd za skalarni ili vektorski procesor
- Kako napraviti vektorski procesor?

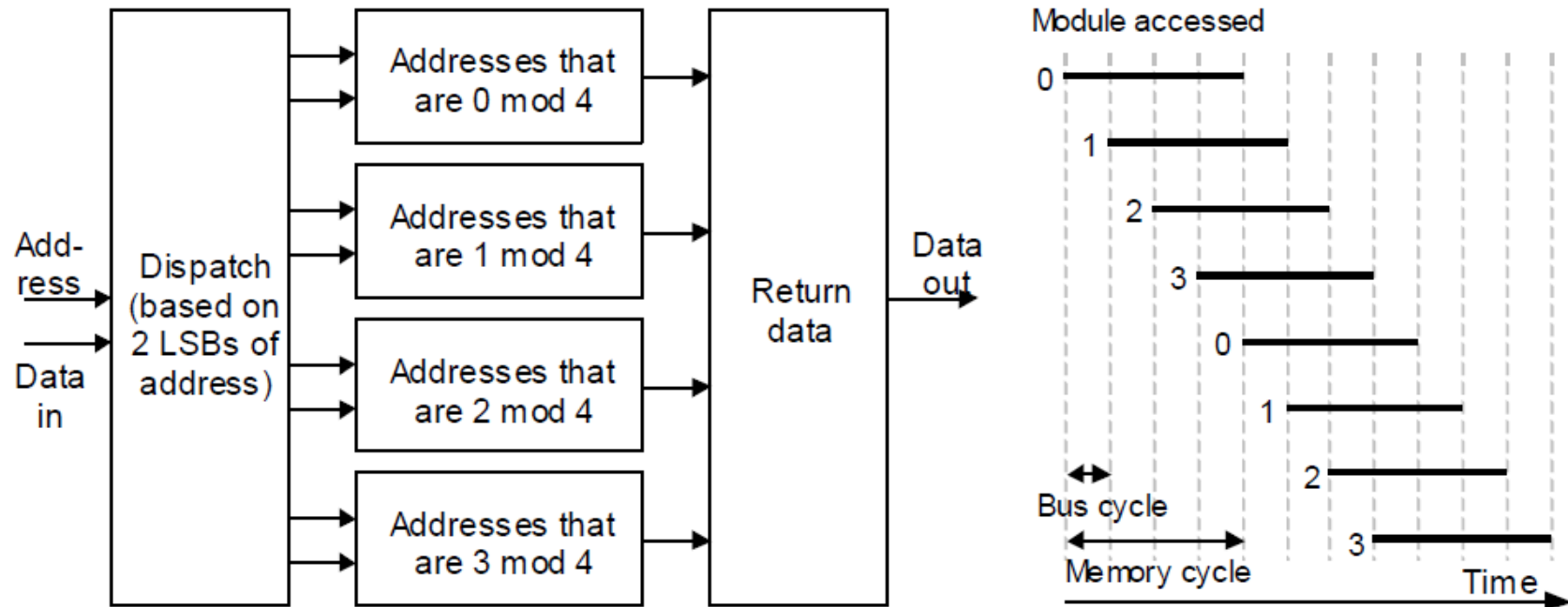
# Векторски рачунари

Protočni stepeni sa velikim kašnjenjem  
kombinacione logike u bržem pipeline-u



# Векторски рачунари

## Memory Interleaving – memorija je spora!!!



Ako se sekvencijalno ide po adresama u memorijskim operacijama, iluzija je sledeća:



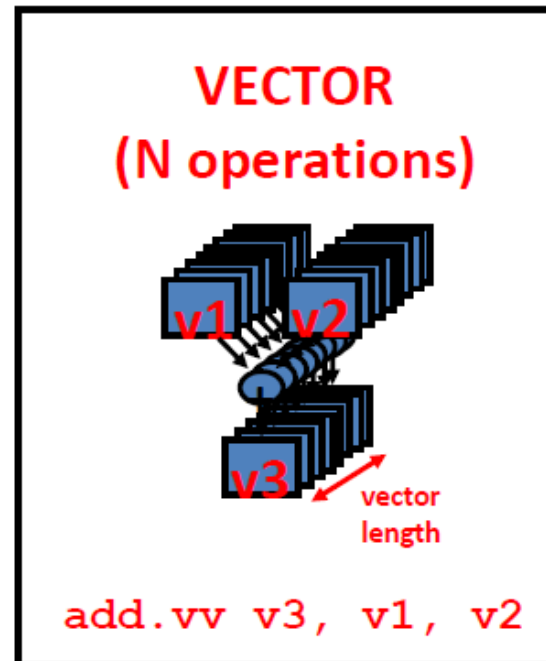
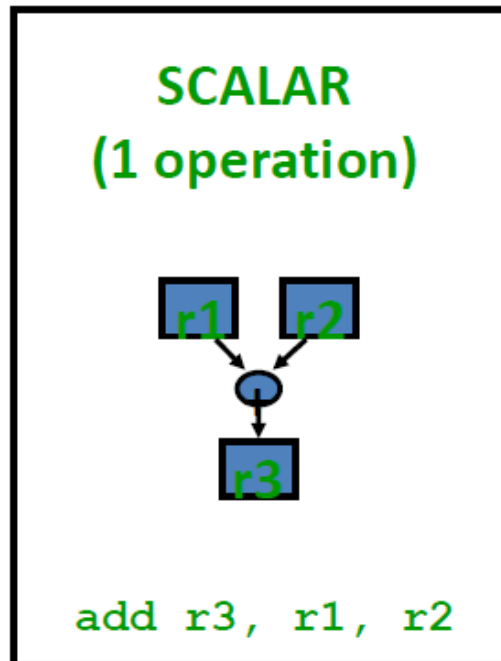
“Memorijski  
pipeline”

4\* sporija memorija radi efektivno  
punom brzinom u pipeline-u ako čitamo  
ili upisujemo u blokove podataka!!!  
CRAY 16\* sporija memorija (mod 16)!!

# Векторски рачунари

## Model za petlje koje obrađuju nizove podataka – naročito DoAll

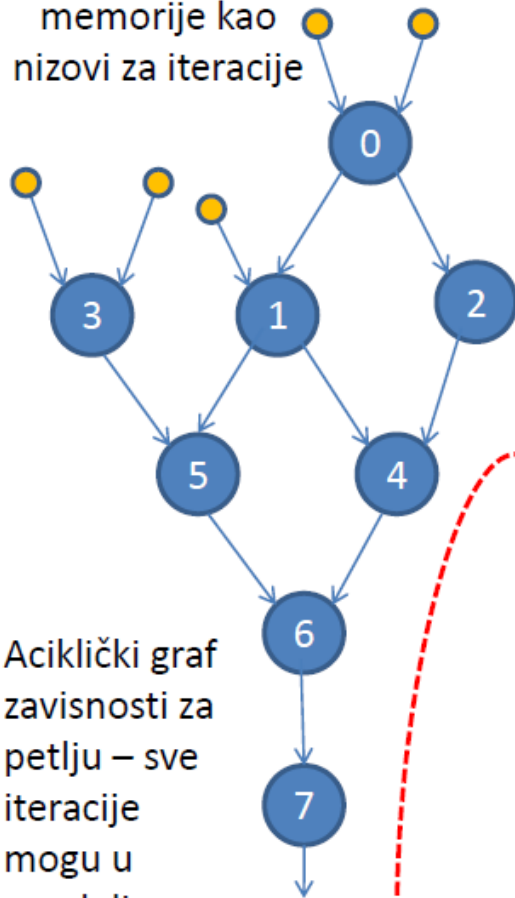
- Iskoristiti Memory interleaving da se napune vektorski registri (sadrže cele nizove za sve ili deo operacija data ready na vrhu grafa)
- Međurezultati se takođe čuvaju u vektorskim registrima



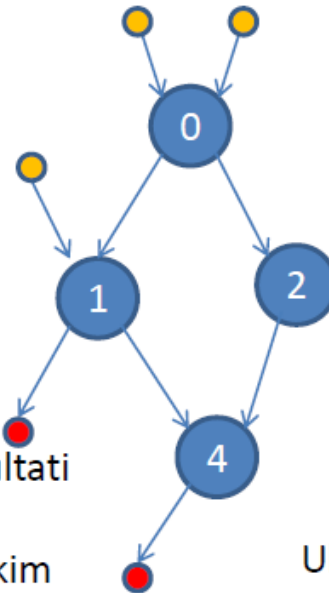
# Векторски рачунари

## DoAll i vektorski registri

● Argumenti iz memorije kao nizovi za iteracije

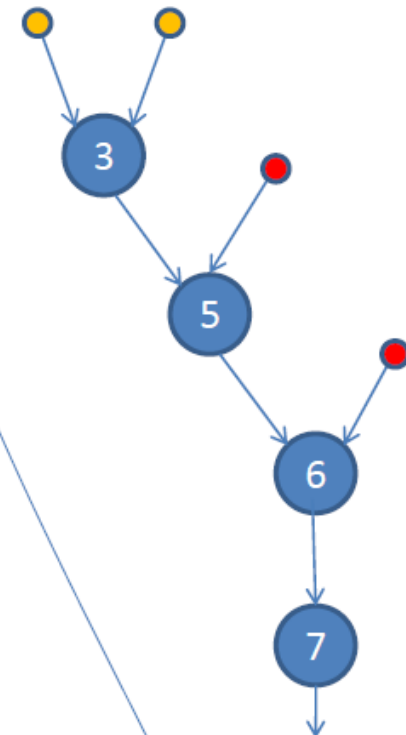


Ideja: prvo uraditi za sve iteracije deo grafa (~4 operacije se započinje po ciklusu)



Ukupno 9 vektorskih registara

Zatim uraditi sledeći deo (~4 operacije po ciklusu) ...



# Векторски рачунари

## Vektorska jedinica i ulančavanje instrukcija

- Iz acikličkog grafa otkidamo čitave podgrafove slobodne na vrhu i povezujemo ih u složene pipeline-e prema grafu zavisnosti po podacima.
- Data ready nizove ubacujemo u vektorske registre – **Interleaving. Nema cache za nizove!!!** ●
- Veličina podgrafa zavisi od broja raspoloživih ALU i broja vektorskih registara
- Veza rezultat-argument ostvaruje se preko istog vektorskog registra
- Vektorski registri mogu da ubace kašnjenja (pointer(i) za read i write pomereni) da se sinhronizuje stizanje argumenata za ulančani pipeline

# Векторски рачунари

## Vektorska jedinica i ulančavanje instrukcija (2)

- Deo argumenata za nove podgrafove koji postaju slobodni na vrhu su već u vektorskim registrima nakon završetka prethodnog podgrafa!! ●
- Nema adresiranja memorije ili cache-a i povećava se dubina pipeline-a
- Broj lokacija vektorskih registara za elemente nizova je isti
- Ako je broj lokacija svakog vektorskog registra manja od ukupnog broja iteracija, mora se raditi manji broj iteracija - malo manje iteracija od broja lokacija vektorskih registara

# Векторски рачунари

## Komponente Vektorskih Procesora

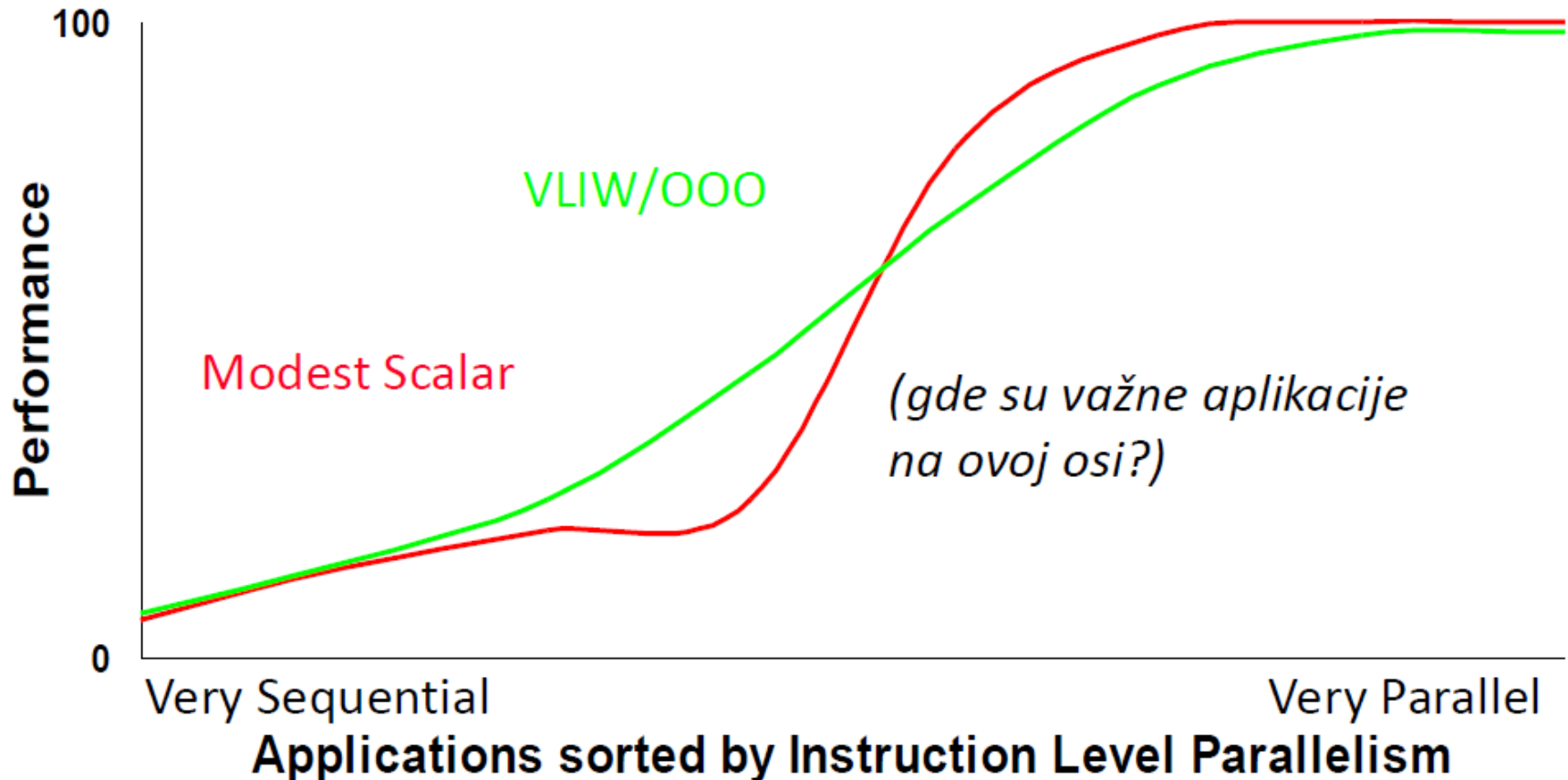
- *Vektorski Registri*: Fiksne veličine za jedan vektor (niz)
  - ima najmanje 2 read i 1 write port
  - tipično je 8-32 vektorska registra, svaki sa 64-128 64-bit elemenata
- *Vektorske Funkcionalne Jedinice (FJ)*: pipeline koji započinje novu operaciju svakog ciklusa
  - tipično 4 do 8 FJ: npr. 2x FP add, 2x FP mult, FP reciprocal ( $1/X$ ), integer add, logical, shift;
- *Vector Load-Store jedinice (LSUs)*: memorijski pipeline interleaved memorije
- *Skalarni registri*: po jedan element za FP skalar ili adresu
- Cross-bar da poveže FJ , LSU, registre

# Векторски рачунари

VLIW/Out-of-Order prema

Modest Scalar+Vector

Vector



# Векторски рачунари

## Vektorski procesori - mane

- Posvećeno suviše pažnje DoAll, ignorišući pripremu za vektorske operacije i podrazumevajući veliki broj iteracija (pipeline je sa velikim brojem pipeline stepeni pa traži puno iteracija da se isplati !)
- Povećanje vektorskih performansi bez posvećivanja dovoljno pažnje skalarnim performansama (Amdahl-ov zakon)
- Nedovoljna je propusnost memorije u odnosu na performanse vektorskog procesora

# Векторски рачунари

## Prednosti vektorskih procesora

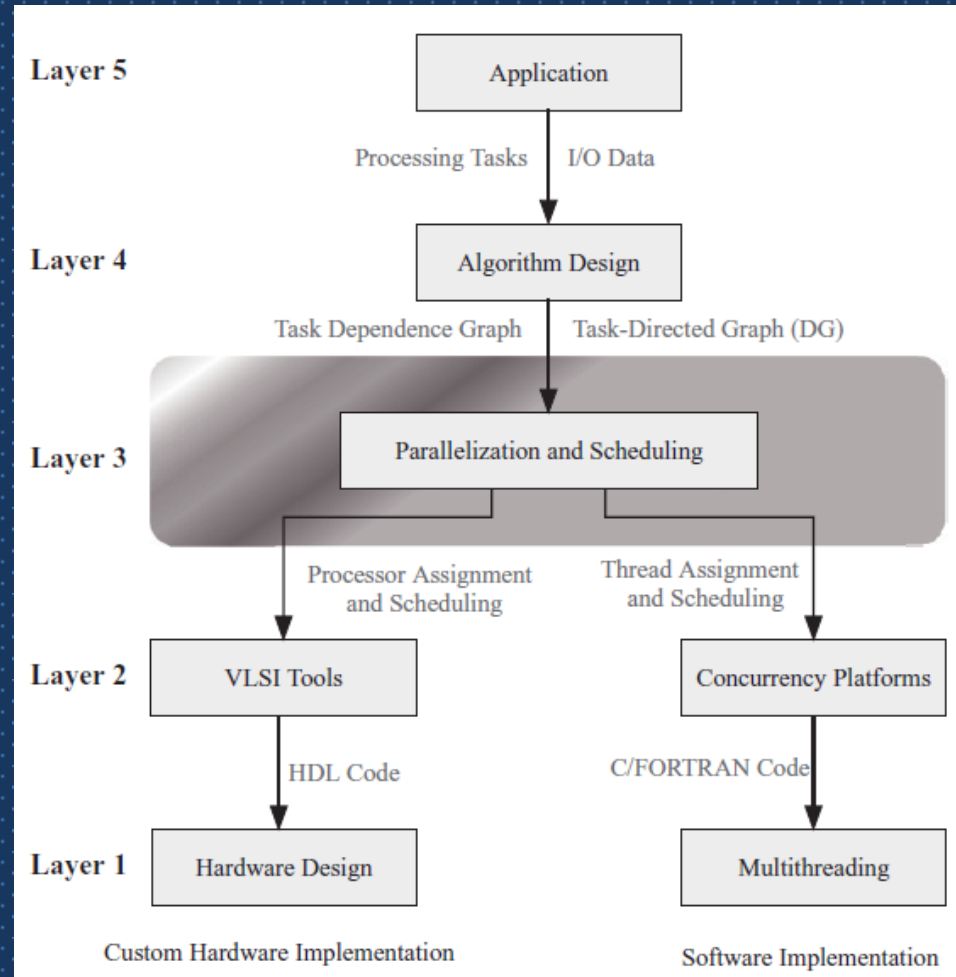
- Lako se dobijaju visoke [performanse](#); N operacija se simultano (u dubokom pipeline-u) izvršava za DoAll petlje:
- [Skalabilno](#) (dobijaju se bolje performanse sa porastom HW resursa)
- [Kompaktno](#): N operacija se prikazuje jednom jednostavnom instrukcijom (nasuprot VLIW)
- [Prediktabilne](#) (real-time) performanse prema statističkim performansama (cache)
- [Multimedia](#), spremno i može se birati  $N * 64b$ ,  $2N * 32b$ ,  $4N * 16b$ ,  $8N * 8b$
- Zrela i razvijena tehnologija [kompajlera](#)

# Paralelno programiranje

Svi smo upoznati sa procesom implementacije algoritama u softveru. Kad pišemo kod, ne trebamo znati detalje ciljnog računarskog sistema jer će se prevodilac brinuti o detaljima.

Međutim, sada razmišljamo u smislu jedne središnje procesne jedinice (CPU) i sekvencijalne obrade kad započnemo pisati kod ili ispravljamo pogrešku u izlazu. Sa druge strane, procesi implementacije hardverskih ili softverskih algoritama za paralelne sisteme više su sofisticirani nego što možemo misliti.

Na slici su prikazane glavne faze ili slojevi implementacije softverske ili hardverske aplikacije pomoću paralelnog računarstva.



# Modeli paralelnog programiranja

## ► Eksplicitni

Programer je odgovoran za paralelizaciju posla:

Dekompoziciju podataka i poslova

Mapiranje poslova i resursa podataka

Upravljanje komunikacijom i sinhronizacijom

## ► Pojedinačne klase:

Kontrolne naredbe (petlje i paralelni regioni)  
(Fortran-S, KSR-Fortran, OpenMP)

Distribucija podataka: HPF (historical)

Distribuirani modeli: PVM , MPI, ProActive

# Modeli paralelnog programiranja

## Podela modela po memorijskoj strukturi:

- **Model zajedničke memorije** (*shared memory*): više procesora rade nezavisno jedan od drugog ali koriste isto memorijsko skladište
  - ✓ čitanje i pisanje je *ekskluzivno*: samo jedan procesor pristupa istovremeno (jedinstvena sabirnica)
- prednosti:
  - ✓ lakše programiranje
  - ✓ nije potrebno dijeliti podatke među zadacima
- nedostaci:
  - ✓ povećanje broja procesora uz jednaku količinu memorije može uzrokovati
  - ✓ zagušenje zbog ograničene brzine pristupa memoriji (*bandwidth*)
  - ✓ korisnik odgovoran za sinkronizaciju
- **Model raspodeljene memorije** (*distributed memory*): više neovisnih procesora sa vlastitim spremnicima
  - ✓ podela podataka i sinhronizacija odvija se porukama (*message passing*)

# Modeli paralelnog programiranja

- prednosti:

- ✓ količina raspoložive memorije promjenjiva (uz dodavanje novih procesora s
- ✓ lokalnom memorijom)
- ✓ brz pristup lokalnoj memoriji

- nedostaci:

- ✓ teško je postojeće podatkovne strukture prilagoditi ovom modelu
- ✓ korisnik (programer) odgovoran za deljenje podataka

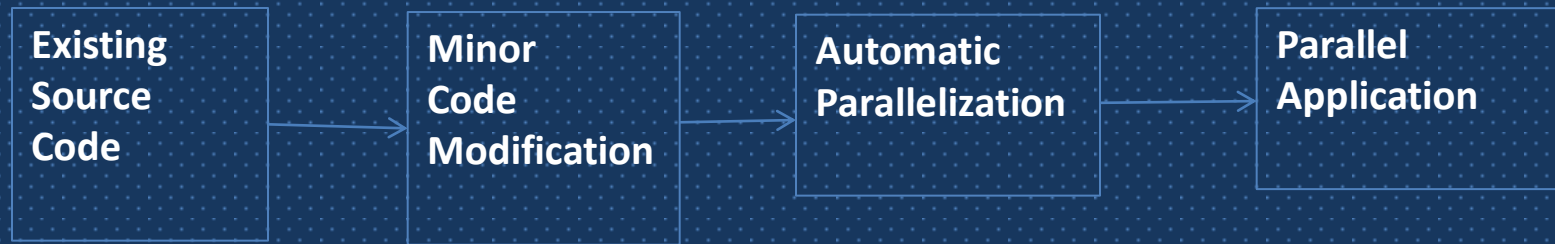
➤ **NUMA model** (*non uniform memory access*) – funkcionalno se može promatrati kao skup paralelnih računara sa zajedničkom memorijom, uz različito trajanje pristupa različitim memorijskim lokacijama

- ✓ može se primeniti i na današnje procesore sa više jezgri i višenivoskom
- ✓ priručnom memorijom (*cache*)
- ✓ takođe približno odgovara organizaciji memorije na GPU

# Modeli paralelnog programiranja

## Strategija 1: Automatska paralelizacija

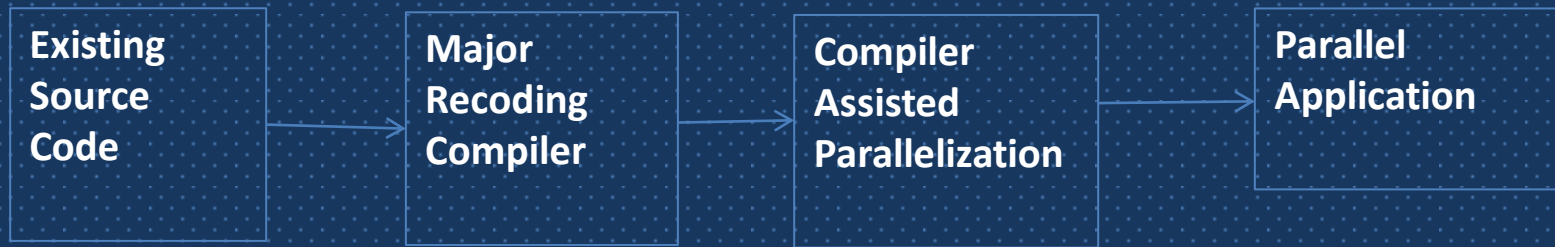
- ▶ Paralelizacija preko računara
- ▶ Bez kontrole programera
- ▶ Teško za ostvarivanje



# Modeli paralelnog programiranja

## Strategija 2: Glavno rekodiranje

- Pisanje paralelne aplikacije od nule
- Mala iskorišćenost koda
- Limitirana distribucijom podataka i smeštajem

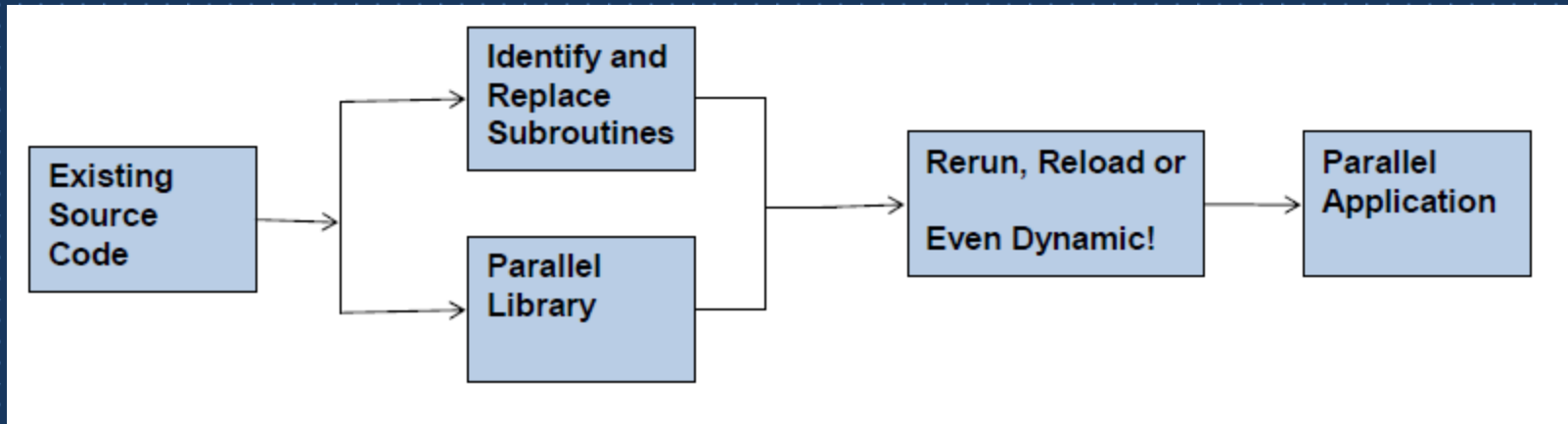


MPI, OpenMP, PVM

# Modeli paralelnog programiranja

## Strategy 3: Paralelne biblioteke (ProActive)

- Uspešno implementiranje biblioteka kodova koje pomažu upravljanjem paralelizma.



ProActive, GridGain, Hadoop

# Amdalov zakon za multiprocesorske sisteme

Pretpostavimo da se algoritam ili zadatak sastoji od paralelnog dela  $f$  i serijskog dela  $1 - f$ .  
Pretpostavimo da je vreme potrebno za obradu ovog zadatka na jednom procesoru

$$T_p(1) = N(1-f)\tau_p + Nf\tau_p = N\tau_p,$$

pri čemu je prvi izraz na desnoj strani (RHS) vreme za koje procesor treba da obradi serijski deo. Drugi termin na RHS-u je vreme koje procesor treba da obradi za paralelni deo. Kada se ovaj zadatak izvrši na  $N$  paralelnih procesora, dobiće se vreme koje je potrebno

$$T_p(N) = N(1-f)\tau_p + f\tau_p,$$

gde je jedino ubrzanje to, da je paralelni deo sada distribuiran na  $N$  procesora. Amdalov zakon za ubrzanje  $S(N)$ , postignut upotrebom  $N$  procesora, dat je kao

$$S(N) = \frac{T_p(1)}{T_p(N)} = \frac{N}{(1-f)N + f} = \frac{1}{(1-f) + f/N}.$$

Da bi postigli bilo kakvo ubrzanje, mora biti zadovoljeno:

$$1-f \ll f/N.$$

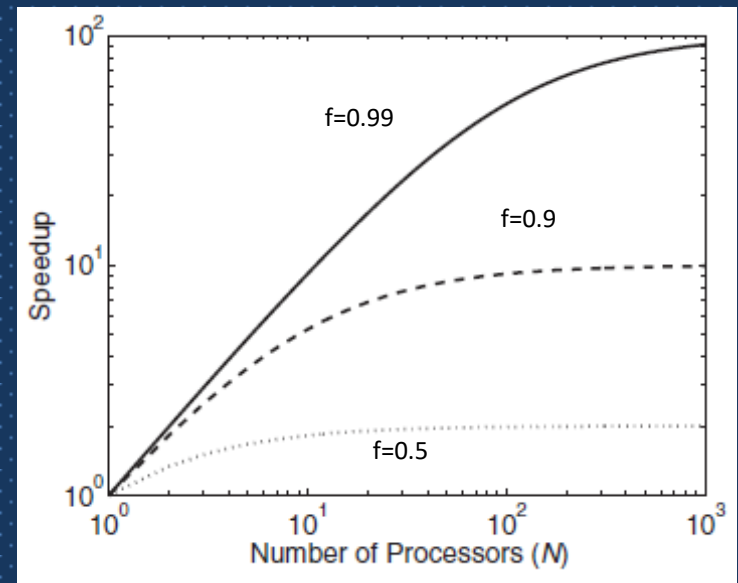
Za velike vrednosti  $N \gg 1$  ubrzanje procesa je

$$S(N) \approx \frac{1}{1-f}$$

Za ekstremne vrednosti  $f$ , dobijamo:

$S(N) = 1$  kada je  $f = 0$  kompletno serijski kod

$S(N) = N$  kada je  $f = 1$  kompletno paralelni kod



# Algoritmi paralelnog programiranja

## *Pretvaranje serijskog u paralelni algoritam:*

Najčešće imamo na raspolaganju serijski algoritam koga želimo izvoditi paralelno.

Neki od koraka u razvoju paralelnog algoritma :

1. pronaći delove slednog programa koji se mogu izvoditi istovremeno
  - ✓ zahteva detaljno poznavanja rada algoritma
  - ✓ može zahtevati i potpuno novi algoritam
2. rastaviti algoritam
  - ✓ funkcionalna dekompozicija – podjela problema na manje dijelove (koji se mogu rešavati istovremeno)
  - ✓ podatkovna dekompozicija – podela podataka sa kojima algoritam radi na manje delove; obično jednostavnije izvesti
  - ✓ kombinacija gornja dva načina
3. ostvarenje programa
  - ✓ odabir programske paradigme, sklopovskog okruženja
  - ✓ usklađivanje komunikacije (način, učestalost, sinhronizacija...)
  - ✓ spoljna kontrola izvođenja
4. ispravljanje grešaka, optimiranje izvođenja...

# MPI – Message Passing Interface

U razvoju paralelnih aplikacija javlja se potreba za mehanizmom razmjene poruka (*message passing*) za uporabu na računarima sa raspodeljenom memorijom (nCUBE, iPSC itd, danas nisu u upotrebi)

- neki od ranih razvijenih sistema su Express, p4, PICL, PARMACS, PVM
- zbog mnogih manjih sintaktičkih i funkcionalnih razlika, izgrađeni programi nisu bili jednostavno prenosivi
- 1993: osnovan Message Passing Interface Forum – 40-tak industrijskih i istraživačkih organizacija
- 1994: razvijen MPI standard (1.1)
- 1997: MPI 2.0, 2012: MPI 3.0, 2015: MPI 3.1
- standard definira komunikaciju porukama, odnosno razmjenu podataka među procesima
- broj procesa u MPI programu je po definiciji konstantan tokom izvođenja
- svi procesi obično izvode isti program (SPMD model), međutim mogu se pokrenuti i različiti programi za različite procese (MPMD model)
- MPMD model se uvek može simulirati uvrštenjem više funkcija u jedan SPMD program

Načini komunikacije u MPI:

1. *point-to-point* komunikacija između dva određena procesa
2. *collective* komunikacija unutar grupe procesa
3. *probe* funkcije za asinhronu komunikaciju
4. *communicator* mehanizam za razvoj modularnih paralelnih programa (mogućnost korišćenja topologije mreže)

# Osnovne MPI funkcije

Bilo koja tehnika paralelnog programiranja mora za svaki proces omogućiti barem sledeće mehanizme:

- ✓ *otkriti ukupan broj procesa*
- ✓ *identifikovati vlastiti proces u grupi procesa*
- ✓ *poslati poruku određenom procesu*
- ✓ *primiti poruku (od određenog procesa)*

- MPI uključuje preko 120 funkcija, no većina funkcionalnosti može se postići sa mnogo manjim skupom
- sve funkcije prikazane u ovom poglavlju opisane su sa C sintaksom (postoji i Fortran sintaksa, te implementacije za druge jezike)
- svaki C/C++ program koji koristi MPI mora imati stavku `#include "mpi.h"`
- dve funkcije koje se *moraju* naći u svakom MPI programu su:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- prva funkcija mora biti pre svake MPI komunikacije, dok se druga nalazi na kraju
- funkciji `MPI_Init` se prosleđuju odgovarajući parametri funkcije `main` – specifikacija u standardu za eventualnu upotrebu u različitim implementacijama
- identifikacija procesa i grupe radi se funkcijama:

```
int MPI_Comm_size (MPI_Comm comm, int *size)
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

# Razmena poruka

Poruka se sastoji od *oznaka* poruke i *podataka*:

- podaci su niz jednoga od MPI podatkovnih tipova: osnovni MPI tipovi podataka odgovaraju osnovnim tipovima u C-u (MPI\_CHAR, MPI\_SHORT, MPI\_INT, MPI\_LONG, MPI\_UNSIGNED – sve varijante, MPI\_FLOAT, MPI\_DOUBLE...)
- korisnik kao podatke prosljeđuje 'obične' C tipove, ali u pozivu funkcija navodi odgovarajuću MPI oznaku
- korisnik također može definirati vlastite tipove podataka u porukama ili koristiti generički MPI\_BYTE (za proizvoljne podatke)

Osnovne funkcije slanja i primanja poruka:

int MPI\_Send (void \***buf**, int **count**, MPI\_Datatype **datatype**, int **dest**, int **tag**, MPI\_Comm **comm**)

int MPI\_Recv (void \***buf**, int **count**, MPI\_Datatype **datatype**, int **source**, int **tag**, MPI\_Comm **comm**, MPI\_Status \***status**)

Proces pošiljalac zove Send, primalac Recv; parametri su:

- ✓ *buf* je početna adresa podataka u memoriji koji se šalju (kod procesa pošiljalca) odnosno primaju (kod primatelja)
- ✓ *count* je broj jedinica podataka (dužina niza)
- ✓ *datatype* je odgovarajući MPI tip podatka
- ✓ *dest* i *source* određuju indeks (*rank*) procesa pošiljalca i primatelja
- ✓ *tag* je oznaka vrste poruke, *comm* je oznaka komunikatora unutar koga se komunikacija odvija (npr. MPI\_COMM\_WORLD za globalnu grupu)

# Globalna komunikacija

Često se u praksi javlja potreba slanja podataka više (svim) procesima ili skupljanja podataka od više procesa – MPI nudi funkcije globalne komunikacije (*collective communication*)

Svojstva globalnih operacija:

- ✓ *svi procesi u grupi moraju pozvati određenu funkciju* (globalna operacija odnosi se na sve procese u nekoj grupi)
- ✓ sve funkcije su *blokirajuće*
- ✓ nema oznake poruke (parametar *tag*)
- ✓ količina poslanih podataka mora odgovarati količini primljenih podataka! (MPI standard: *the amount of data sent must exactly match the amount of data specified by the receiver*)

Operacija slanja svima od jednog procesa:

int MPI\_Bcast (void \***buffer**, int **count**, MPI\_Datatype **datatype**, int **root**, MPI\_Comm **comm**)

- funkcija šalje isti podatak svim procesima u definisanom komunikatoru; izvorišni proces označen je parametrom *root* (indeks procesa) (\*\*\*)
- parametar *buffer* se kod procesa *root* čita, dok se kod ostalih u njega upisuju podaci
- 'suprotna' operacija od Bcast: MPI\_Reduce; za računanje jednog rezultata na temelju podataka od svih procesa:

int MPI\_Reduce (void \***sendbuf**, void \***recvbuf**, int **count**, MPI\_Datatype **datatype**, MPI\_Op **op**, int **root**, MPI\_Comm **comm**)

## Primer: računanje broja Pi

· broj Pi računa se kao suma

$$\frac{4}{n} \sum_{i=1}^n \frac{1}{1 + \left( \frac{i-0.5}{n} \right)^2}$$

(veći  $n$  daje veću tačnost rešenja)

```
#include "mpi.h"
#include <math.h>

int main(int argc, char *argv[])
{ int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  while (!done)
  { if (myid == 0)
    { printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;

    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs)
    { x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",pi,fabs(pi - PI25DT));
  } // while
  MPI_Finalize();
}
```

# Model računara PRAM

Najčešće korišćeni model *serijskog računara* je najvjerovatnije RAM (*Random Access Machine*)

Svojstva RAM-a: konstantni broj registara i neograničen broj memorijskih lokacija (adrese su prirodni brojevi)

U oceni složenosti:

- ✓ svaka operacija obavlja se u jednoj jedinici vremena
- ✓ svaka memorijska lokacija je jedna jedinica prostora

Model paralelnog računala: PRAM (*Parallel RAM*)

Svojstva:

- ✓ PRAM je MIMD model sa zajedničkom memorijom
- ✓ PRAM ima proizvoljni broj procesora
- ✓ PRAM je **sinhroni** računar: svi procesori izvode jednu (bilo koju različitu) instrukciju u jednoj jedinici vremena (*lock step*)

Često se, u razvoju algoritma, pretpostavlja i način sa zadanim brojem procesora  
Skupu osnovnih pseudokodnih instrukcija dodjemo ključnu reč "**paralelno**" koja označava istovremeno odvijanje na potrebnom broju procesora (koliko? po potrebi - zavisi od konkretnog slučaja)

Pristup memoriji može biti dopušten ili samo jednom procesoru odjednom (*exclusive*) ili za više njih odjednom (*concurrent*), kako za čitanje (*read*) tako i za pisanje (*write*).

# Računarski i programski model

Za opis razvoja paralelnih algoritama, potrebno je definisati model paralelnog računara i strukturu paralelnog programa

## Računarski model

*Multiračunar* - više nezavisnih računara povezanih nekom vrstom mreže

Pretpostavke:

- ✓ brzina komunikacije ne zavisi od međusobnog položaja računara u topologiji mreže
- ✓ pristup lokalnoj memoriji vremenski je *manje skup* od pristupa udaljenoj memoriji (na drugom računaru)

Po osobinama, multiračunar je MIMD računar raspodeljene memorije

## Programski model

- ✓ koristi se model zadataka i komunikacijskih kanala
- ✓ opis programa: usmereni graf u kojem su čvorovi zadaci, a veze su komunikacijski kanali

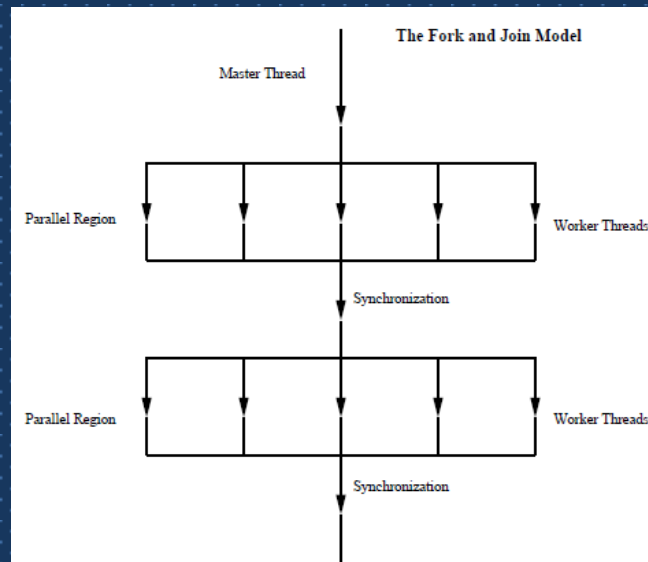
Osobine sistema:

- ✓ paralelni program sastoji se od jednog ili više zadataka; zadaci se mogu izvoditi istovremeno
- ✓ broj zadataka može se menjati tokom izvođenja
- ✓ zadatak obuhvata program i lokalnu memoriju
- ✓ zadatak može, osim rada sa lokalnom memorijom, izvesti četiri operacije: slati poruku, primiti poruku, stvoriti nove zadatke i završiti s radom
- ✓ operacija slanja je po pretpostavci neblokirajuća (odmah završava), dok je operacija primanja blokirajuća (završava po primitku poruke)
- ✓ broj i položaj komunikacijskih kanala može se mijenjati tijekom izvođenja
- ✓ zadaci mogu biti pridruženi procesorima na više načina, što ne utiče na funkcionalnost programa (jedan ili više zadataka po procesoru)

# OpenMP

OpenMP (Open Multi-Processing) prvi je put objavljen 1997. godine i standardni je aplikacijski programski interfejs (API) za pisanje paralelnih aplikacija zajedničke memorije na C, C++ i Fortranu. Prednosti OpenMP-a su što ih je vrlo lako implementirati na postojeće serijske kodove i omogućiti inkrementalnu paralelizaciju. Takođe ima prednosti široke upotrebe, visoko prenosivog i idealno prilagođenog za arhitekture više jezgara (koje postaju sve popularnije u svakidašnjim stolnim računarima).

OpenMP deluje po modelu paralelne Fork i Join egzekucije što je prikazano na slici. Svi OpenMP programi počinju kao jedan proces koji se naziva glavna nit. Ova se glavna nit izvršava uzastopno dok se ne nađe paralelna regija. U ovom trenutku glavna nit se "stapa" u niz paralelnih radničkih niti. Tada skup radničkih niti izvršava naredbe u paralelnom području. Na kraju paralelnog područja niti se sinhroniziraju i spajaju kako bi ponovno postale jedinstvena glavna nit. Obično bi pokrenuli jednu nit po procesoru, ali moguće je pokrenuti i više. Paralelizacija sa OpenMP-om određuje se putem naredbi kompilatora koje su ugrađene u izvorni kod.



# Kako se kompiluje kod i izvršava na OpenMP?

Jedna od korisnih stvari kod OpenMP je da ona dopušta da korisnici mogu koristiti isti izvorni kod OpenMP kompajlera i kod normalnih kompajlera. Ovo se izvodi tako da su OpenMP naredbe i direktive sakrivene za normalne kompajlere.

Evo vrlo jednostavnog malog višeslojnog paralelnog programa HelloWorld.c, napisanog na C jeziku koji će ispisati 'Hello World', prikazujući broj niti koji obrađuje svaku izjavu za pisanje.

```
#include 'omp.h'
void main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num()
        printf('Hello(%d) ', ID);
        printf('World(%d) \n', ID);
    }
}
```

Prva linija OpenMP je include hederskog fajla *omp.h*.

Paralelni region je smešten izmedju direktiva *#pragma omp parallel {...}*.

Funkcija runtime biblioteke *omp\_get\_thread\_num()* vraća ID niti u program.

Ako želimo da kompilujemo program sa komandne linije pisaćemo:

> *icc -openmp HelloWorld.c -o Hello*

Za Intel C kompajler ili

> *gcc -fopenmp HelloWorld.c -o Hello*

Za GNU C kompajler.

# Kako se kompiluje kod i izvršava na OpenMP?

Takođe trebamo postaviti broj niti po kojima će se program pokretati. Postoji nekoliko načina za to. Prvo, u naredbeni red možemo upisati

```
> setenv OMP NUM THREADS 4  
za 4 niti (t) csh shell ili  
> export OMP NUM THREADS=4  
za 4 niti u bash shell.
```

Alternativno, možemo specificirati broj niti unutar programa uključujući liniju:  
*omp set num threads(4);*  
pre naredbe paralelnog regiona  
*#pragma omp parallel.*

Izlaz iz programa bi izgledao:

```
Hello(5) Hello(0) Hello(3) Hello(2) World(5)  
Hello(1) Hello(4) World(0)  
World(3)  
World(2)  
World(1)  
World(4)
```



Iz ovog izlaza možemo videti da se program izvodio na 6 niti (od nula do pet) i da se niti ne moraju nužno izvoditi u određenom redosledu.

# Kako se kompiluje kod i izvršava na OpenMP?

## Primer 1:

Ovaj primer je vrlo jednostavna petlja koja ilustruje zajedničke i privatne varijable i sintaksu pisanja programa OpenMP. Program inicijalizira dva niza i ispisuje sadržaj tih nizova na ekran.

```
#include <omp.h>
#define N 10

int main(void) {
    float a[N], b[N];
    int i;

#pragma omp parallel default(none), private(i), shared(a,b)
    {
#pragma omp for
        for (i = 0; i < N; i++) {
            a[i] = (i+1) * 1.0;
            b[i] = (i+1) * 2.0;
            printf("%d, %f, %f \n",i+1,a[i],b[i]);
        }
    }
}
```

Izlaz iz programa:

|    |          |          |
|----|----------|----------|
| 1  | 1.000000 | 2.000000 |
| 2  | 2.000000 | 4.000000 |
| 5  | 5.000000 | 10.00000 |
| 6  | 6.000000 | 12.00000 |
| 9  | 9.000000 | 18.00000 |
| 10 | 10.00000 | 20.00000 |
| 3  | 3.000000 | 6.000000 |
| 4  | 4.000000 | 8.000000 |
| 7  | 7.000000 | 14.00000 |
| 8  | 8.000000 | 16.00000 |

# Kombinacija CPU i GPU paralelizma

Korišćenje grafičke procesorske jedinice za obavljanje zadataka koji nisu grafičke prirode se naziva programiranje grafičke procesorske jedinice u opšte svrhe („general-purpose GPU“ ). Pre 2010. godine, GPGPU programiranje je smatrano za novinu u svetu programiranja sa visokim performansama i nije mu se poklanjalo puno pažnje.

To se promenilo u oktobru te godine kada je superračunar Tianhe-1A, konstruisan od strane kineskog Nacionalnog Centra za Superračunare u Tiandinu, postao najbrži superračunar na svijetu. Ono što je specifično za Tianhe-1A je da je za njegovu konstrukciju korišćena kombinacija centralnih procesorskih jedinica ( CPU – Central Processing Unit ) i grafičkih procesorskih jedinica ( GPU – Graphical Processing Unit ). Tianhe-1A se sastojao od 14 336 Xeon X5670 centralnih procesorskih jedinica, 2 048 FeiTeng 1000 procesora baziranih na SPARC tehnologiji i sa 7 168 Tesla M2050 grafičkih procesorskih jedinica kompanije Nvidia. Ova hardverska kombinacija je omogućila da se dostigne maksimalna brzina od 2,566 PFLOPS-a što je skoro 50% brže od tada najbržeg Cray-ovog superračunara Jaguar, koji je imao brzinu od 1,759 PFLOPS-a.

U periodu nakon ovog događaja drastično je porasla zainteresovanost za GPGPU programiranje, jer je primećeno da je GPU zbog svoje specifične konstrukcije pogodan za paralelno programiranje. Kod paralelnog programiranja zadaci za obradu podataka se dodeljuju višestrukim elementima za obradu i ona se vrši istovremeno, što GPU čini idealnim za tu namenu jer se sastoji od velikog broja malih procesora.

# Kombinacija CPU i GPU paralelizma

## OpenCL

OpenCL je tehnologija koja omogućava pisanje programa koji se izvršavaju na heterogenim platformama sastavljenim od centralnih procesorskih jedinica ( CPU ), grafičkih procesorskih jedinica ( GPU ), digitalnih signalnih procesora ( DSP ), FPGA čipova, ...

Primena ove tehnologije omogućava paralelnu obradu u odnosu na podatke ( eng: data-based parallelism ) kao i paralelnu obradu u odnosu na zadatke ( eng: taskbased parallelism ). OpenCL je standard otvorenog tipa održavan od strane neprofitnog tehnološkog konzorcijuma zvanog Khronos Group. Ovaj konzorcijum konstantno unapređuje OpenCL izdavanjem novih verzija OpenCL specifikacije.

Nakon što su mnogi inženjeri i akademici došli do zaključka da su sistemi koji kombinuju CPU i GPU predstavljaju budućnost programiranja sa visokim performansama, u startu su bili suočeni sa značajnom preprekom. Veliki problem je što nije bilo načina da se iskoristi potencijal ovih sistema jer programi pisani u tradicionalnim programskim jezicima mogu da koriste samo CPU. Čak i programi koji koriste tehnologiju CUDA ( Compute Unified Device Architecture ) kompanije Nvidia, ne mogu da se koriste na gore pomenuti način jer za izvršavanje koriste isključivo GPU. Pored toga, CUDA programi mogu da rade samo na grafičkim procesorima koje proizvodi samo Nvidia.

# OpenCL

Rešenje ovoga problema se pojavilo u vidu tehnologije OpenCL ( Open Computing Language ), čije rutine se mogu izvršavati na uređajima svih velikih proizvođača hardvera.

OpenCL je nastao na inicijativu kompanije Apple koja je željela da se stvori zajednički interfejs za sve uređaje, jer Apple ne proizvodi svoje procesore već za svaku sledeću liniju proizvoda bira čiji će hardver koristiti. Sa obzirom na dominantnu poziciju ove kompanije, proizvođači hardvera su odlučili da ispune taj zahtev.

OpenCL standard definiše skup tipova podataka, funkcija i struktura podataka koje proširuju programske jezike C i C++. Sa vremenom su kreirani portovi za druge programske jezike ( C#, Java, Python, ... ) ali standard zahteva da se obezbede samo biblioteke za C i C++. Pitanje koje se nameće je šta OpenCL može a da to ne mogu jezici C i C++? Iako je odgovor na to pitanje prilično komplikovan, tri glavne prednosti koje se izdvajaju su:

1. Portabilnost
2. Standardizovana obrada vektorskih struktura podataka
3. Paralelno programiranje

# OpenCL

Filozofija OpenCL-a je slična filozofiji programskog jezika Java, čiji kod se bez izmena može izvršavati na svakom operativnom sistemu koji podržava Java virtualnu mašinu ( JVM ). Stoga se za Java kod kaže „Napiši jednom, pokreni svuda“.

Razlika sa OpenCL-om je ta što se OpenCL kod može bez izmena izvršiti na različitim uređajima pa bi se za OpenCL moglo reći da mu je filozofija „Napiši jednom, pokreni na bilo čemu“.

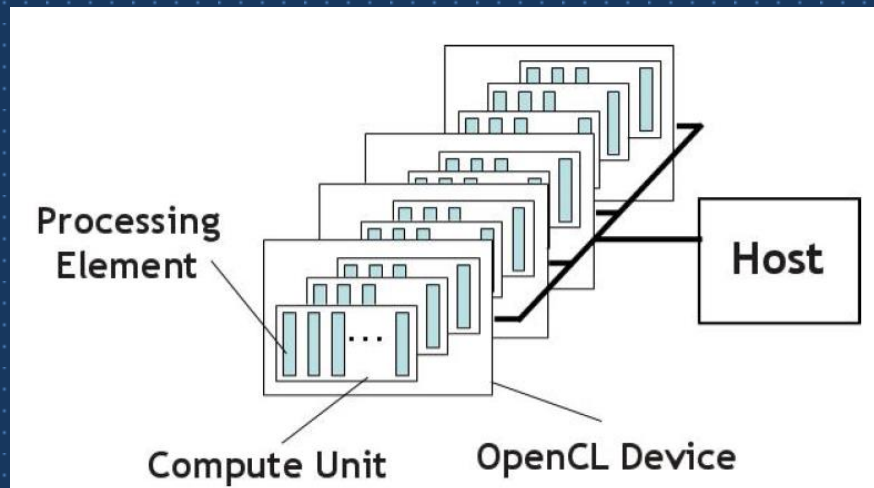
Svaki proizvođač koji proizvodi uređaje kompatibilne sa OpenCL-om takođe obezbeđuje alatke koje kompajliraju OpenCL kod za izvršavanje na tim uređajima. Ovo znači da se da se OpenCL kod može napisati jednom i kasnije kompilovati za svaki kompatibilan uređaj, nezavisno od toga da li je to grafička precesorska jedinica ili centralna procesorska jedinica sa više jezgara.

Ovo je velika prednost u odnosu na standardno programiranje sa visokim performansama, kod kojega se mora naučiti programski jezik specifičan za proizvođača da bi se mogao koristiti uređaj tog proizvođača.

# OpenCL

## Paralelno programiranje

Kada se koristi paralelno programiranje, tada se zadaci za obradu podataka dodeljuju višestrukim elementima za obradu i ona se vrši istovremeno. U OpenCL terminologiji, ovi zadaci se nazivaju jezgra ( eng: kernel ). Jezgro je funkcija napisana na određen način i ona je predviđena da se izvrši na jednom ili više OpenCL kompatibilnih uređaja. Jezgra se šalju na uređaje na kojima će se izvršavati putem kontrolne aplikacije ( eng: host application ). Kontrolna aplikacija je standardna C/C++ aplikacija koja se izvršava na korisnikovoj centralnoj procesorskoj jedinici. Kontrolne aplikacije upravljaju OpenCL uređajima koristeći poseban kontejner zvan kontekst. Da bi stvorila jezgro, kontrolna aplikacija prvo bira jednu od funkcija koje se nalaze u kontejneru zvanom program. Nakon toga, ona stvara asocijaciju između jezgra i podataka koji će biti korišteni kao argumenti, a potom se jezgro prosleđuje na komandni red. Komandni red je mehanizam putem kojeg kontrolna aplikacija šalje uređajima jezgra na izvršavanje. Uređaj tada uzima jezgro iz reda i izvršava ga koristeći odgovarajuće argumente.



# OpenCL

## Programiranje kontrolne aplikacije

Kao što smo već rekli, kontrolna aplikacija je standardna aplikacija napisana u programskom jeziku C ili u programskom jeziku C++. Da bi se napisala kontrolna aplikacija, potrebno je poznavanje šest osnovnih struktura podataka pomoću kojih kontrolna aplikacija kreira jezgra i memorijske spremnike, kompajlira programe za odgovarajuće uređaje u kontekstu, šalje jezgra uređajima preko komandnih redova, ... Nije lako razumjeti ove strukture pri prvom susretu sa njima ali nakon što se prevaziđe ta prepreka, pisanje kontrolnih aplikacija postaje relativno jednostavno. Iz tog razloga, ove strukture će koristiti delove izvornog koda ( koji predstavljaju ove strukture ) u programskoj jeziku C++.

Da bi se izvršila kompilacija OpenCL aplikacije mora se pristupiti alatima u radnom okviru za OpenCL. Po specifikaciji, radni okvir za OpenCL se sastoji od tri dela:

1. Platformski sloj – omogućava pristup uređajima i formiranje konteksta.
2. Izvršni sloj – omogućava kontrolnim aplikacijama da kreiraju komande redove za uređaje unutar konteksta i da šalju jezgra na te redove.
3. Kompilator – kompilira programe koji sadrže jezgra za izvršavanje.

# OpenCL

## Paralelno izvršavanje jezgara

Kada je u pitanju paralelna obrada podataka, veoma je važno da se podaci koji se obrađuju pravilno rasporede u skladu sa tipom i količinom resursa za njihovu obradu. Pored već pomenute mogućnosti da u OpenCL aplikaciji podelimo podatke za obradu na delove i svaki dio posaljemo drugom uređaju na obradu, postoji i dodatna mogućnost raspodele podataka unutar samog uređaja što omogućava dodatni nivo paralelizma.

Razlog za postojanje ove mogućnosti je činjenica da skoro svi OpenCL kompatibilni uređaji poseduju više jedinica za obradu podataka.

Pomenuta podela se vrši direktno pri izdavanju naredbe za izvršavanje jezgra korišćenjem metode `enqueueNDRangeKernel` klase `cl::CommandQueue`, čiji je prototip slijedeći:

```
cl_int cl::CommandQueue::enqueueNDRangeKernel(
    const Kernel &kernel,
    const NDRange &offset, const NDRange &global_size,
    const NDRange &local_size,
    const VECTOR_CLASS<cl::Event> *events = NULL,
    Event *event = NULL);
```

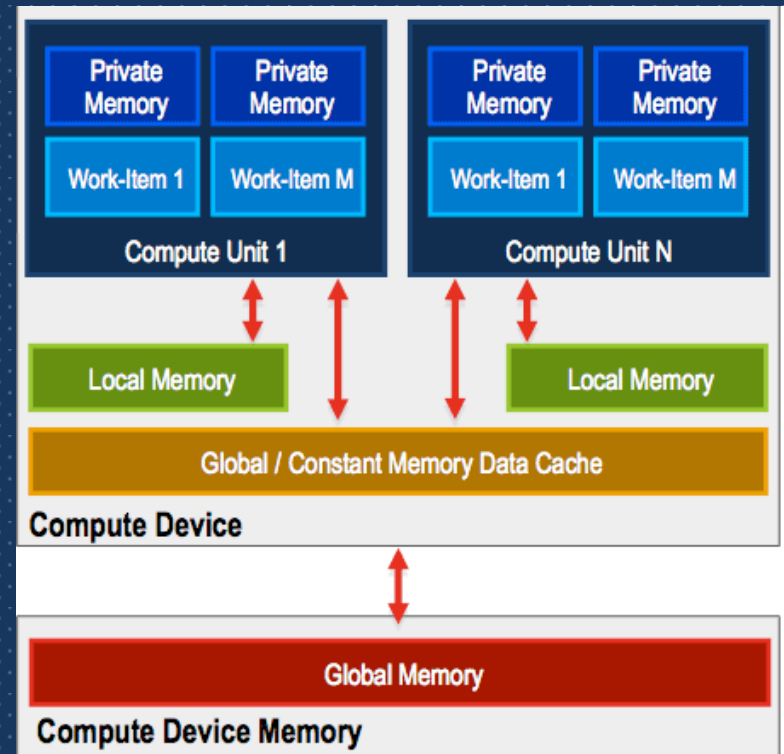
# OpenCL

## Memorijski model OpenCL uređaja

Cilj ovog poglavlja je da se razjasne već ranije pominjane različite vrste memorijskih regiona u OpenCL uređajima. Pored toga, želimo da predstavimo odnose između memorijskih regiona i sa radnim predmetima i radnim grupama. Ovi odnosi nisu jednostavni ali njihovo razumevanje je apsolutno neophodno da bi se razumelo kako se jezgro izvršava na uređaju, što dovodi do pisanja efikasnijih algoritama.

OpenCL uređaj ima četiri adresna prostora:

- ✓ Globalna memorija - Koristi se za skladištenje podataka dostupnih čitavom uređaju.
- ✓ Konstantna memorija - Isto kao i globalna memorija, s tim da dozvoljava samo čitanje.
- ✓ Lokalna memorija - Koristi se za skladištenje podataka dostupnih radnim grupama i radnim predmatima.
- ✓ Privatna memorija – Koristi se za skladištenje podataka dostupnih samo određenom radnom predmetu.



# OpenCL

## Obrada grafičkih podataka

Kako se obrada vrši unutar jezgra? Pre toga, sa obzirom da se za obradu grafičkih podataka mogu koristiti i opšti memorijski spremnici, navešćemo prednosti korišćenja memorijskih objekata namenjenih grafičkim podacima:

- Ako se za obradu koristi grafička procesorska jedinica, grafički podaci su smešteni unutar posebnog dela globalne memorije koji se zove teksturna memorija. Ova memorija je keširana radi bržeg pristupa, što ubrzava proces obrade podataka.
- \* Funkcije koje se koriste za čitanje i upis grafičkih podataka se mogu koristiti bez obzira na način formatiranja podataka koji predstavljaju piksele, dokle god je način formatiranja jedan on mnogih koje OpenCL podržava.
- \* OpenCL poseduje posebne strukture podataka zvane uzornici, pomoću kojih se lako konfiguriše metoda čitanja grafičkih podataka.
- \* U OpenCL jezgrima se mogu koristiti ugrađene funkcije za pristupanje podacima specifičnim za grafičke podatke ( dimenzije, bitna dubina, format predstavljanja piksela ), što olakšava programiranje algoritama za obradu grafičkih podataka.

Kada su u pitanju strukture podataka, najbitnije su `image2d_t` i `image3d_t` ( koje predstavljaju dvodimenzionalne i trodimenzionalne grafičke podatke ) i `sampler_t` koja predstavlja gore pomenute uzornike.

# Aplikacije za paralelno računanje

## Klimatsko modeliranje

Klimatske simulacije koriste se za predviđanje vremena kao i za predviđanje globalnih klimatskih promena na temelju različitih pojava ili ljudskih aktivnosti. Kao što literatura ističe, rezolucija današnjih klimatskih modela iznosi 200 km. Ovo se smatra niskom rezolucijom sa obzirom na činjenicu da neki klimatski sistemi u potpunosti postoje unutar takvih rezolucija.

Pretpostavimo model visoke rezolucije za klimatske simulacije particija globusa koristeći 3 - D ćelije veličine 1 km u svakom smeru. Uzmimo i da je ukupna površina zemlje  $510 \times 10^6 \text{ km}^2$ , a debljina atmosferskog sloja približno 1000 km. Zatim trebamo simulirati približno  $5 \times 10^{11}$  ćelija vremenske prognoze. Uzmimo na dalje da svaka ćelija treba obaviti 200 operacija sa pokretnim zarezom za svaku ponovnu iteraciju simulacije. Dakle, moramo izvršiti ukupno  $10^{14}$  operacija sa pokretnim zarezom po iteraciji.

Dakle, imamo sledeće zahteve za naš paralelni višejezgralni računarski sistem korišćenjem dva tipa mikroprocesora koji mogu izvršiti  $2.8 \times 10^{15}$  FLOPS :

| Processor            | Clock speed | GFLOPS/core | Cores needed      | Power (MW) |
|----------------------|-------------|-------------|-------------------|------------|
| AMD Opteron          | 2.8GHz      | 5.6         | $4.9 \times 10^5$ | 52.0       |
| Tensilica XTensa LX2 | 500.0MHz    | 1.0         | $2.8 \times 10^6$ | 0.8        |

Ukupni broj operacija za ovo je  $10^{14}$  operacije/iteraciji  $\times 10^6$  iteracija =  $10^{20}$  operacija sa pokretnim zarezom.

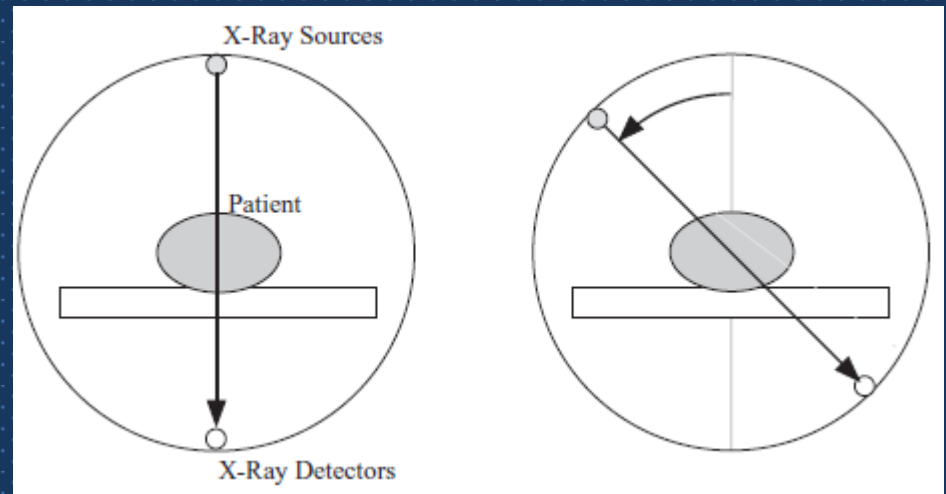
Računar koji radi brzinom od  $10^9$  operacija sa pokretnim zarezom u sekundi (FLOPS) dovršilo bi operacije u  $10^{11}$  sekundi, što iznosi oko 31 stoleće ???? Opisane simulacije bi mogle biti završene za jedan dan kada bi naš paralelni sistem operisao brzinom od  $2.8 \times 10^{15}$  FLOPS.

# Aplikacije za paralelno računanje

## CT

CT i tehnike magnetske rezonance (MRI) su za dobivanje mape visoke rezolucije unutrašnjosti tela za postavljanje medicinske dijagnoze. Na slici prikazan je pojednostavljeni prikaz CT sistema.

Pretpostavimo da se slika koju pokušavamo generisati sastoji od  $N \times N$  piksela, pri čemu bi  $N$  mogao biti približno jednak 4.000. Dakle, imamo približno  $10^7$  piksela koje možemo generisati po slici ili presek skeniranog tela. Kako se uređaj kreće, trebalo bi generisati više preseka. To omogućava trodimenzionalni prikaz područja tela koje se posmatra.



Za sistem koji generiše  $S = 1.000$  uzastopnih preseka,  $SN^2 = 10^{10}$  piksela moraće se obraditi. Za odrezak će biti potrebno izračunati približno  $N^2 (\log_2 N)^3$ . Za naš slučaj, potrebno nam je otprilike totalni broj operacija = operacija/presek  $\times 10^{10}$  presek =  $10^{13}$  operacija sa pokretnim zarezom. Pretpostavimo da ove slike moramo generisati u 1 sekundi kako bismo omogućili stvarni pregled pacijenta. U tom slučaju sistem bi trebao raditi brzinom od otprilike  $10^{13}$  FLOPS. Za još precizniju medicinsku dijagnozu, potrebna je kompjutorizirana tomografija visoke razlučivosti (HRCT), čak i na nivou nanoskala.

# Aplikacije za paralelno računanje

## Računarska dinamika fluida (CFD)

CFD je polje koje je usko povezano sa paralelnim računarima i paralelnim algoritmima. Na to se gleda kao na isplativ način istraživanja i dizajniranja sistema koji uključuju protok gasa ili fluida. Neki primjeri CFD-a su:

- Okeanske struje
- Atmosfersko i globalno vreme
- Tok krvi u arterijama
- Ispitivanje deformacija kroz visoke G manevre letelica
- Tok vazduha u plućima
- Dizajniranje krila aviona
- Izbacivanje sedišta kod borbenih aviona
- Sagorevanja gasova u cilindrima motora
- Kontrola avionskih motora u opitnim laboratorijumima
- Oblik karoserije automobila za redukciju otpora vazduha

Ciljani region za CFD aplikacije se kreće od  $10^{12}$  do  $10^{18}$  FLOPS . Ako računarski sistem deluje brzinom od  $10^9$  (giga) FLOPS-a, tada će CFD aplikacije dovršiti simulaciju u vremenskom razdoblju koje se kreće između 15 minuta i 30 godina. Sa druge strane, paralelni računarski sistem koji radi na  $10^{12}$  (tera) FLOPS-u obradio bi aplikaciju u vremenskom razdoblju između 1 sekunde i 12 dana.

# Paralelno Programiranje na Web\_u

Cilj projekta Intel Labs - River Trail, poznatog i kao Paralelni JavaScript, treba da omogući paralelizam podataka u web aplikacijama. U svetu u kojem je web brauzer korisnikov prozor u informacione tehnologije, aplikacije brauzera moraju iskoristiti sve dostupne računarske resurse kako bi pružile najbolje moguće korisničko iskustvo.

Danas web aplikacije ne koriste u potpunosti prednosti paralelnog klijentskog hardvera zbog nedostatka odgovarajućih programskih modela.

River Trail stavlja paralelnu računarsku snagu klijentovog hardvera u ruke web programera dok ostaje u sigurnim granicama poznate paradigme JavaScript programiranja. River Trail proširuje JavaScript jednostavnim determinističkim paralelnim konstrukcijama podataka koje se tokom izvođenja prevode u sloj apstrakcije hardvera na niskom nivou. Korišćenjem više CPU jezgri i vektorskih instrukcija, programi River Trail mogu postići značajno ubrzanje u odnosu na sekvencijalni JavaScript.

Prototip otvorenog koda implementacije River Traila za Firefox dostupan je na GitHub-u. Ovaj prototip implementira verziju API-ja River Trail koji se kompajlira u OpenCL i može se izvršiti na CPU-ima i GPU-ima.

# Paralelno Programiranje na Web\_u

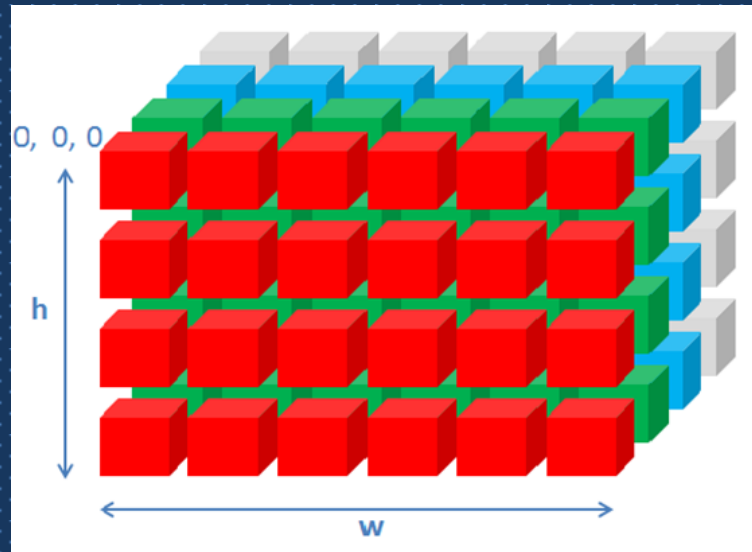
Primer: Podsetimo da bufer `frame.data` sadrži vrednosti boja kao linearni niz RBGA vrednosti. Petlja `for` u `sepia (tamno siva boja)_sequential` prevlači se preko ovog bafera i za svaki piksel očitava crvenu, zelenu i plavu vrednost (koje su u `pix [i]`, `pix [i + 1]`, odnosno `pix [i + 2]`). Izračunava se ponderisani prosek ovih boja kako bi proizveo nove vrednosti crvene, zelene i plave za taj piksel. Zatim uzima nove vrednosti crvene, zelene i plave u raspon `[0, 255]` i zapisuje ih natrag u bafer `frame.data`. Kada je petlja završena, zamenili smo RGB vrednosti za sve piksele njihovim vrednostima u tonu sepije i sada možemo zapisati sliku natrag u izlazni kontekst `ctx` metodom `putImageData`. Rezultat bi trebao izgledati ovako (slika s lijeve strane izvorni je okvir, slika sa desne strane je izlaz):



# Paralelno Programiranje na Web\_u

Ako pažljivo pogledamo gornju funkciju `sepia_sequential`, primetićemo da se svaki piksel se mora obraditi nezavisno od svih ostalih piksela, jer njegove nove RGB vrednosti zavise samo od trenutnih RGB vrednosti. Nadalje, svaka iteracija `for` petlje ne proizvodi i ne troši efekat strane. To olakšava paralelizovanje ove operacije sa River Trailom. Prisetimo se da tip `ParallelArray` ima konstruktor koji uzima objekt `canvas-slike` kao argument i vraća sveže modifikovan objekat `ParallelArray` koji sadrži podatke piksela sa te slike.

```
var pa = new ParallelArray(canvas);/*Kod kreira trodimenzionalno ParallelArray pa sa oblikom [h, w, 4] koje je prikazano na slici:*/
```



# Paralelno Programiranje na Web\_u

Dakle, za piksel na slici u koordinatama (x, y), pa.get (x, y, 0) sadrži crvenu vrednost, pa.get (x, y, 1) će sadržavati zelenu vrednost, a pa .get (x, y, 2) sadrži plavu vrednost. U computeFrameu stvaramo novi ParallelArray od ove slike:

```
else if (execution_mode === "parallel") {  
    frame = output_context.getImageData(0, 0, input_canvas.width, input_canvas.height);  
    stage_output = stage_input = new ParallelArray(input_canvas);  
    w = input_canvas.width; h = input_canvas.height;
```

Ovde su *stage\_input* i *stage\_output* objekti ParallelArray koji sadrže ulazne i izlazne podatke piksela za svaku "fazu" filtriranja. Pogledajmo sada kod koji uzrokuje primenu filtera:

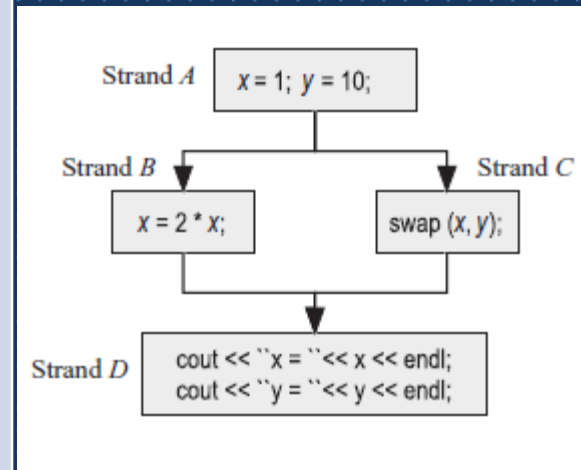
```
if(execution_mode === "parallel") {  
    switch(filterName) {    ...  
        case "sepia":  
            /* Add your code here... */  
            break;    ...    }  
            // Make this filter's output the input to the next filter.  
            stage_input = stage_output;
```

Da bi smo implementirali određeni filter, dodajemo kôd za stvaranje novog ParallelArray objekta koji sadrži transformisane podatke piksela i dodjeljujemo ga stage\_output.

```
case "sepia":  
    stage_output = /* new ParallelArray containing transformed pixel data */;    break;
```

# Paralelno Programiranje u C++\_u

| Serijski kod - deterministički                                                                                                                                                                                                                                                                                                                                                                                    | Paralelni kod - direktiva cilk_spawn                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 1: #include &lt; iostream &gt; 2: using namespace std; 3: void swap (int &amp; x, int &amp; y); 4: int main() 5: { 6: int x = 1, y = 10; 7: swap ( x , y ); 8: x = 2 * x ; 9: cout &lt;&lt; " x = " &lt;&lt; x &lt;&lt; endl; 10: cout &lt;&lt; " y = " &lt;&lt; y &lt;&lt; endl; 11: } 12: void swap (int &amp; x, int &amp; y) 13: { 14: int temp ; 15: temp = x ; 16: x = y ; 17: y = temp ; 18: }</pre> | <pre> 1: #include &lt; iostream &gt; 2: using namespace std; 3: void swap (int &amp; x , int &amp; y ); 4: int main() 5: { 6: int x = 1, y = 10; 7: cilk_spawn swap ( x , y ); 8: x = 2 * x ; 9: cilk_sync; 10: cout &lt;&lt; " x = " &lt;&lt; x &lt;&lt; endl; 11: cout &lt;&lt; " y = " &lt;&lt; y &lt;&lt; endl; 12: } 13: void swap (int &amp; x , int &amp; y ) 14: { 15: int temp ; 16: temp = x ; 17: x = y ; 18: y = temp ; 19: }</pre> |
| Rezultat x = 10 i y = 2                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                 |



# Paralelno Programiranje u MATLAB\_u

## \*sekvencijalni problem kvadrature\*

Kvadratura je idealna aplikacija za paralelno programiranje, jer ona uključuje veliki broj jednostavnih, nezavisnih zadataka čija svojstva mogu biti unapred definisana. Pretpostavimo da imamo funkciju  $f(x)$ , i interval  $[a; b]$ . Integral je dat kao  $I(f; [a; b]) = \int_a^b f(x) dx$ , pri čemu je vrednost integrala data kompozitnim pravilom srednje tačke:

$$Q_{CM}(f, [a, b], n) = \sum_{i=1}^n (b_i - a_i) \cdot f\left(\frac{a_i + b_i}{2}\right) = \frac{b-a}{n} \cdot \sum_{i=0}^n f\left(\frac{(n-i)a + ib}{n}\right)$$

Jednostavna MATLAB procedura primenom kvadraturnog problema za rešavanje može biti:

```
1 function q = quadseque n t i a l ( a , b , n , f )
2 %% q u a d s e q u e n t i a l . m
3 q = 0 . 0 ;
4 for i = 1 : n
5 a i = ( ( n - i + 1 ) a + ( i - 1 ) b ) / n ;
6 b i = ( ( n - i ) a + i b ) / n ;
7 x i = ( a i + b i ) / 2 . 0 ;
8 q = q + f ( x i ) ;
9 end
10 q = q ( b - a ) / n ;
11 return
12 end
```

```
1 %% q u a d s e q u e n t i a l c a l l . m
2 a = 0 . 0 ;
3 b = 2 . 0 ;
4 n = 2 1 ;
5 f = @ ( x ) ( 4 x + 1 ) ( 1 1 - 4 x ) / 8 ;
6 q = quadseque n t i a l ( a , b , n , f ) ;
7 fpr int f ( 1 , ' Vrednost integrala je%f nn ' , q ) ;
```

Ovde je funkcija za integraljenje  $f(x)=(4x+1)(11-4x)/8$

# Paralelno Programiranje u MATLAB\_u

## \*kvadratura korišćenjem paralel PARFOR\*

```
function q = quad_parfor ( a , b , n , f )
2 %% quad_parfor .m
3 q = 0.0;
4 parfor i = 1 : n
5 a i = ( ( n - i + 1 ) a + ( i - 1 ) b ) / n ;
6 b i = ( ( n - i ) a + i b ) / n ;
7 x i = ( a i + b i ) / 2.0 ;
8 q = q + f ( x i ) ;
9 end
10 q = q ( b - a ) / n ;
11 return
12 end
```

```
1 %% q u a d _ p a r f o r _ c a l l . m
2 a = 0.0 ;
3 b = 2.0 ;
4 n = 21 ;
5 worker s = 4 ;
6 f = @( x ) ( 4 x + 1 ) ( 11 - 4 x ) / 8 ;
7 matlabpool ( ' open ' , ' l o c a l ' , worker s ) ;
8 q = quad_parfor ( a , b , n , f ) ;
9 matlabpool close
10 fprintf ( 1 , ' Vrednost integrala je %f ' , q ) ;
```

Program radi sa 4 procesora (workers)

Srećom, MATLAB je automatski u stanju prepoznati što se događa sa promenljivom  $q$ , i u osnovi, kako započinje petlja, stvara za svakog radnika nuliranu privatnu promenljivu zvanu  $q$ , koju je taj radnik može slobodno modifikovati. Po završetku petlje, MATLAB prikuplja višestruke kopije  $q$  i dodaje ih vrednosti  $q$  koja je bila zapamćena u klijentu pre početka paralelne petlje. Promenljiva  $q$  je primer onoga što se naziva redukcijaska promenljiva.

Klijent inicijalizuje  $q$  na nulu, ali ne izvršava izraze unutar petlje koji ažuriraju njegovu vrednost. Dakle, radnici su ti koji moraju ažurirati vrednost  $q$  sa novim informacijama.

# Paralelno Programiranje u MATLAB\_u

## \*poredjenje sekvencijalnog i paralelnog pristupa\*

Kada isprobamo paralelni program na našem uzorku problema, za neke male veličine problema rezultati ne izgledaju baš dobro! Pokrećemo sekvencijalni kôd (" 0 radnika ") naspram paralelnog koda sa 1, 2 ili 4 radnika, i za  $n$  u rasponu od 1 do  $2^{10}$ . Paralelna vremena izvršavanja su stalno gora od izvršavanja sekvencijalnog koda, a čini se da se čak pogoršavaju kako mi dodajemo još radnika!

Moramo imati na umu da blagodati paralelizma dolaze kod problema sa značajnim brojem iteracija. Svakako je našu kvadraturnu funkciju  $f$  trivijalno proceniti, i možda nekoliko stotina evaluacijskih tačaka za datu funkciju ne daje povoljan radni ishod. U komparaciji koristimo  $n = 1, 2, 4, \dots, 2^{23}$  tačaka, za svaku veličinu problema, brzina sekvencijalnog koda prikazuje se kao vodoravna crta 1. Ako je paralelni kod brži za tu veličinu problema, pojavit će se iznad te crte.

| N/W  | 0      | 1      | 2      | 4      |
|------|--------|--------|--------|--------|
| 1    | 0.2923 | 0.4164 | 0.2306 | 0.2436 |
| 2    | 0.0073 | 0.0294 | 0.0270 | 0.2131 |
| 4    | 0.0008 | 0.0234 | 0.0223 | 0.2150 |
| 8    | 0.0008 | 0.0205 | 0.0235 | 0.0293 |
| 16   | 0.0008 | 0.0263 | 0.0263 | 0.0391 |
| 32   | 0.0008 | 0.0306 | 0.0395 | 0.0435 |
| 64   | 0.0008 | 0.0207 | 0.0326 | 0.0489 |
| 128  | 0.0008 | 0.0201 | 0.0303 | 0.0410 |
| 256  | 0.0009 | 0.0201 | 0.0332 | 0.0362 |
| 512  | 0.0010 | 0.0213 | 0.0258 | 0.0320 |
| 1024 | 0.0012 | 0.0204 | 0.0250 | 0.0342 |

