Sveučilište J.J. Strossmayera u Osijeku
Fakultet elektrotehnike, računarstva i informacijskih tehnologija
Osijek

RASPODIJELJENI RAČUNALNI SUSTAVI

(predavanja u ak. god. 2024/25.)

Diplomski studij računarstva

prof.dr.sc. Goran Martinović
goran.martinovic@ferit.hr
Tel: 031 495-401
Soba: K2-3

Osijek, 2025.

1

8
Paralelno i raspodijeljeno programiranje
MPI

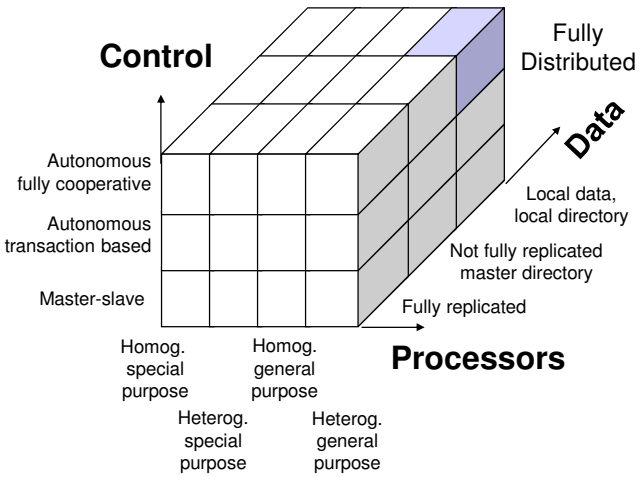
2

0.1. Sveprisutan problem

- Obilježja:
  - Ogromne količine podataka
  - Ogromni zahtjevi za računanjem
- Primjeri:
  - Problemi ugrađeni u web
  - Empirijska i istraživanja s velikim količinama podataka
  - "Post-genomička era" u istraživanju života (life science)
  - Visoko kvalitetne simulacije i animacije
  - Generativna umjetna inteligencija
  - Sve ostalo
- Podijeli i vladaj
- Uključiti što više strojeva u problem

3

0.1. Raspodijeljeni i paralelni sustavi



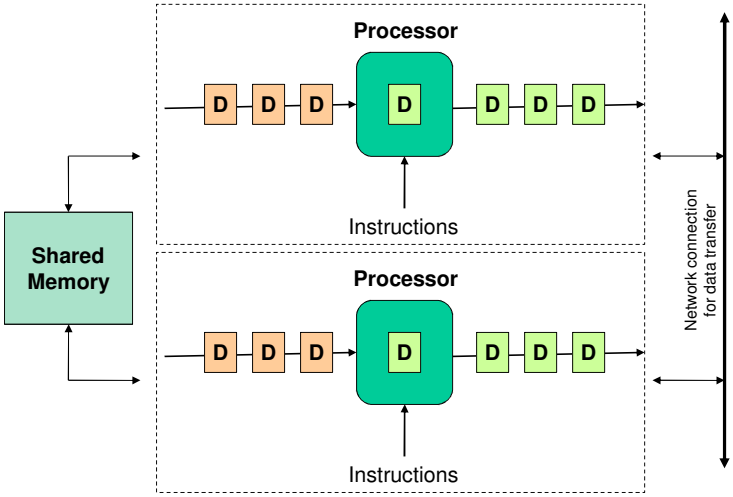
4

0.2. Paralelno nasuprot raspodijeljenog

- Paralelno računarstvo općenito znači:
  - Vektorska obrada podataka
  - Višestruke CPU u jednom računalu
- Raspodijeljeno računarstvo općenito znači:
  - Višestruke CPU na više računala

5

0.3. Paralelno nasuprot raspodijeljenog - nastavak

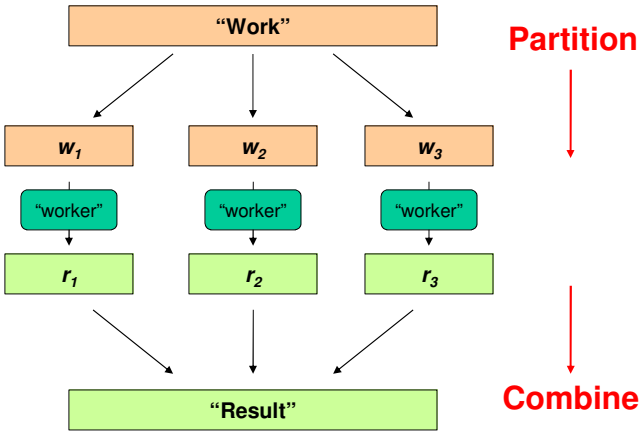


Parallel: Multiple CPUs within a shared memory machine

Distributed: Multiple machines with own memory connected over a network

6

0.3.1. Podijeli i vladaj



7

0.3.2. Različitost izvršitelja posla

- Različite niti na istoj jezgri
- Različite jezgre na istoj CPU
- Različite CPU u višeprocesorskom sustavu
- Različiti strojevi u raspodijeljenim sustavima

8

0.3.3. Problemi paralelizacije

- Kako dodijeliti jedinice poslova radnicima?
- Što kada imamo više jedinica posla nego radnika?
- Što kada radnici trebaju dijeliti parcijalne rezultate?
- Kako ćemo skupiti parcijalne rezultate?
- Kako ćemo znati kad su svi radnici obavili posao?
- Što ako radnik nestane?

9

0.3.4. Općenitiji problemi?

- Problemi paralelizacije pojavljuju se iz:
  - Komuniciranja između radnika
  - Pristupa dijeljenim sredstvima (npr. podaci)
- Također, potrebno je imati sustav sinkronizacije!
- Problemi:
  - Pronaći pogreške je teško
  - Riješiti pogreške još je teže

10

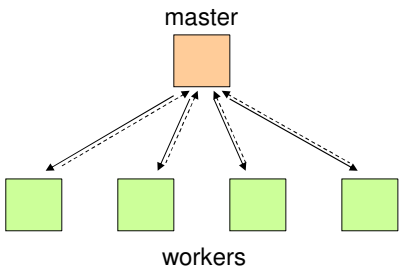
0.3.5. Višenitno programiranje

- Teško zbog
  - Ne znamo redoslijed pokretanja niti
  - Ne znamo kada nit prekida drugu nit
- Tada trebamo:
  - Semafore (lock, unlock)
  - Uvjetne varijable (wait, notify, broadcast)
  - Barijere
- Još uvijek dosta problema:
  - Zastoji
  - Uvjeti utrkivanja (Race conditions)
  - …

11

03.6. Master/Workers

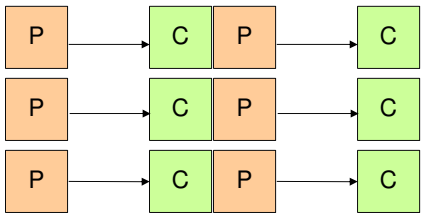
- Master početno posjeduje sve podatke
- Master kreira radnike i dodjeljuje im zadatke
- Master čeka da mu radnici vrate informaciju - izvješće



12

0.3.7. Tijek proizvođač/potrošač

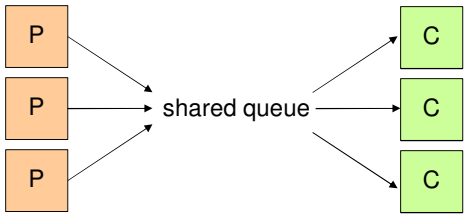
- Proizvođači stvaraju dijelove poslova
- Potrošači ih obrađuju
- Mogu biti ulančani



13

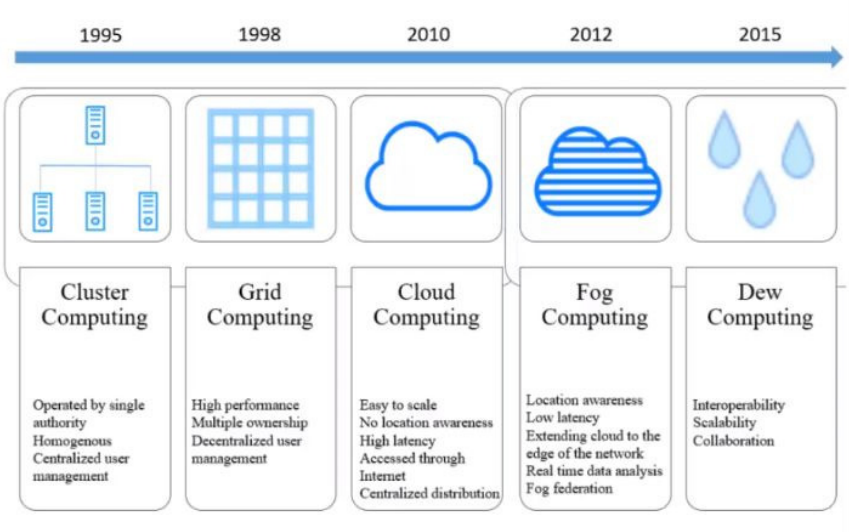
0.3.8. Redovi poslova

- Svi raspoloživi potrošači trebaju biti raspoloživi za obradbu podataka od bilo kojeg proizvođača
- Redovi poslova razdvajaju vezu 1:1 između proizvođača i potrošača



14

0.3.9. Raspodijeljene računalne okoline



15

1. Uvod

- **Ciljevi poglavlja**: upoznati osnovne modele paralelnih računala i programa, neka svojstva paralelnih programa i smjernice za stvaranje paralelnog algoritma

1.1 Zašto paralelno računanje?

- riješiti problem u manje vremena nego što bi zahtijevalo slijedno računanje
- pitanje: "Da li će računala ikada postati *dovoljno brza*?"
- nastajanjem novih problema nastaju novi zahtjevi za većom računalnom moći (dakle, "Ne!")
- zahtjevni računalni problemi:
  - modeliranje i simulacija - rad na principu slijedne aproksimacije; više računanja, veća preciznost (modeliranje klimatskih utjecaja, simulacija seizmičkih pojava, turbulencija fluida, kretanje oceanskih struja...)
  - obrada velikih količina podataka (računalni vid, obrada multimedijalnih podataka, *real-time video servers*, pristup velikim bazama podataka i *data mining*...)
  - razne druge primjere (dekodiranje DNA, utjecaj zagađenja okoliša, kvantna dinamika, komercijalne i zabavne aplikacije...)

16

1.1.1 Rast racunalne moci

- računala obavljaju sve više i više operacija u sekundi (npr. 1950. oko 100 FLOPS, danas oko 10^{12} FLOPS)
- brzina računala ograničena vremenom jednog ciklusa - *clock cycle*, koji se ne smanjuje tako brzo (danas <5 ns)
- vrijeme ciklusa polako se približava fizikalnim ograničenjima:
  - brzina svjetlosti: 30 cm/ns
  - brzina signala u bakrenom vodiču: 9 cm/ns
- trend u računalstvu: iskorištavanje potencijala "sveprisutne" računalne moći (*Grid*)
- povećanje brzine komunikacije među računalima (100 Mb/s i više za lokalne mreže)
- što je potrebno za paralelno računalstvo:
  - arhitektura više procesora (ili računala)
  - veza između procesora (ili mreža)
  - okolina za paralelni rad (odgovarajući OS, model paralelnog algoritma)
  - paralelni algoritam i program

1.2 Modeli paralelnih računala

- potreba za modelom: omogućavanje razvoja algoritama koji su primjenjivi na velikom broju različitih računala (a ne samo na određenom računalu)
- model bi trebao biti jednostavan (da omogući učinkovito programiranje) i realan (da se algoritmi napisani za model lako primjene na stvarna računala)
- **Paralelno računalo**: skup procesora koji mogu zajednički rješavati neki računalni problem
- osnovna podjela računala po odnosu programskih instrukcija i podataka:

  - SISD (Single Instruction, Single Data Stream) model
  - SIMD (Single Instruction, Multiple Data Stream)
  - MISD (Multiple Instruction, Single Data Stream) - bez stvarnih primjera (?)
  - MIMD (Multiple Instruction, Multiple Data Stream)

1.2.1 SISD model

- Von Neumannov model računala - jedan procesor i jedan memorijski spremnik
- jedna instrukcija - jedan (skalarni) podatak

1.2.2 SIMD model

- jedna instrukcija obrađuje više podataka (paralelno i sinkronizirano)
- dvije inačice
- **vektorski SIMD**: instrukcije za skalarne i vektorske operande
- primjeri: Cray 1, Fujitsu VP, NEC SX-2
- **Paralelni SIMD**: polje procesora koje izvode *jednake* instrukcije na različitim podacima
- procesori rade sinkronizirano (*lock-step* način)
- nedostatak: grananja unutar petlji moraju se primijeniti na sve procesore
- primjene ograničene na probleme jednolike obrade velikog skupa podataka (obrada slike, numeričke simulacije...)
- primjeri: Maspar MP-1, MP-2

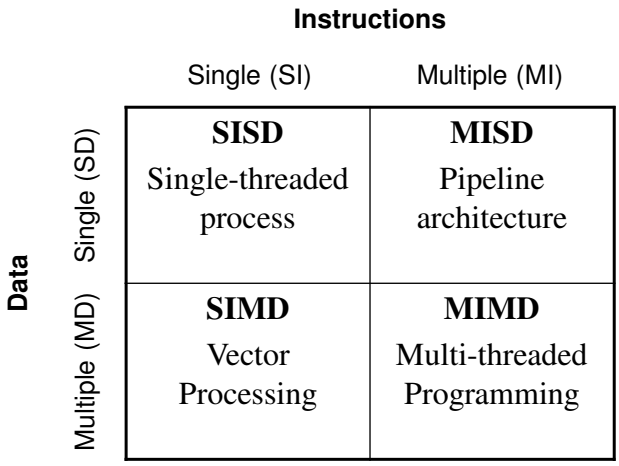
18

1.2.3 MIMD model

- više procesora izvode različite instrukcije na različitim podacima
- prednosti:
  - paralelno izvođenje više poslova
  - svaki procesor neovisan o drugima
- nedostaci:
  - ujednačavanje opterećenja (*load-balancing*) na kraju paralelne obrade zahtjeva sinkronizaciju - gubitak vremena
  - moguće teško za programirati
- primjeri: Cray 2, Intel Paragon, nCUBE, IBM SP2, ...

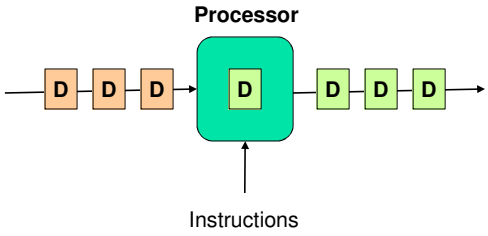
19

Flynnova podjela

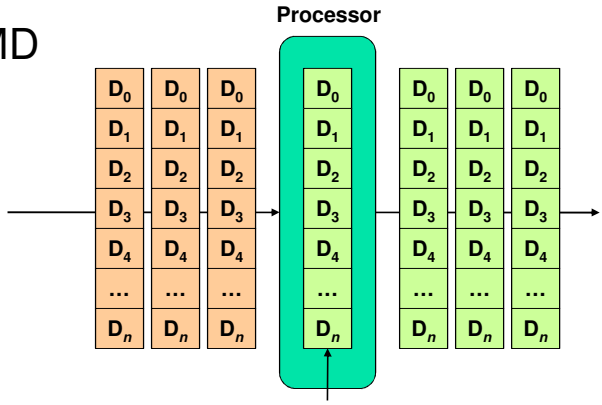


20

SISD

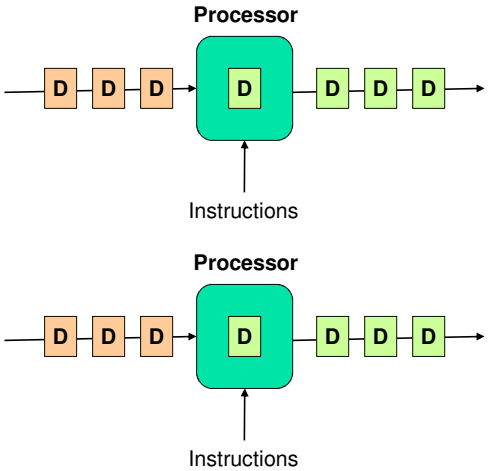


SIMD



21

MIMD



22

1.2.4 Modeli memorijske strukture

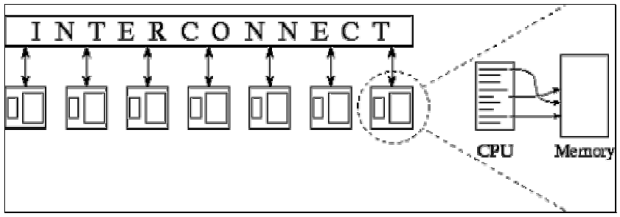
- **Model zajedničke memorije** (*shared memory*): više procesora rade neovisno jedan o drugome ali koriste isti memorijski spremnik
- čitanje i pisanje je *ekskluzivno*: samo jedan procesor pristupa istovremeno (jedna sabirnica)
- prednosti:
  - lakše programiranje
  - nije potrebno dijeliti podatke među zadacima
- nedostaci:
  - povećanje broja procesora uz jedanku količinu memorije može uzrokovati zagušenje zbog ograničene brzine pristupa memoriji (*bandwidth*)
  - korisnik odgovoran za sinkronizaciju

- **Model raspodijeljene memorije** (*distributed memory*): više neovisnih procesora sa vlastitim spremnicima
- podjela podataka i sinkronizacija odvija se porukama (*message passing*)
- prednosti:
  - količina raspoložive memorije promjenjiva (dodavanje novih procesora)
  - brz pristup lokalnoj memoriji
- nedostaci:
  - teško je postojeće podatkovne strukture prilagoditi ovom modelu
  - korisnik (programer) odgovoran za dijeljenje podataka

23

1.2.5 Koje modele ćemo koristiti?

- **Višeprocesorsko računalo** (*multiprocessor; shared memory MIMD*)
- idealizirana inačica: PRAM (*Parallel Random Access Machine*) - definicija kasnije
- **Multiračunalo** (*multicomputer, distributed memory MIMD*) - više neovisnih računala povezanih nekom vrstom komunikacije gdje brzina komunikacije ne ovisi o međusobnom položaju računala (Slika 1.1)
- pristup lokalnoj memoriji vremenski je *manje skup* od pristupa udaljenoj memoriji



Slika 1.1 Model raspodijeljenog računala

24

- u današnjoj primjeni često se susreću dva oblika paralelnog računala:
- **Grozd računala** (*cluster*) - skup računala povezan lokalnom mrežom
  - značajke: manji broj računala i veća brzina komunikacije
  - najsličniji modelu multiračunala
  - na tome ćemo izvoditi algoritme
- **Splet računala** (*grid*) - infrastruktura koja omogućuje pristup računalnim resursima na (u budućnosti) svakom mjestu
  - značajke: veći broj računala koji podržavaju splet, različita brzina komunikacije (manja)

1.3 Modeli (paradigme) paralelnih programa

- potreban je dogovor oko strukture paralelnih algoritama koji se razvijaju
- navedene su neke najčešće paradigme paralelnog programiranja

1.3.1 Komunikacija porukama

- komunikacija porukama (*message passing*) - vjerojatno najkorišteniji model paralalnog programiranja
- više (stalni broj) zadataka izvode se neovisno; podaci se razmjenjuju porukama
- ponekad nazivano i SPMD - *single program, multiple data*: jedan program se izvodi na više procesora
- unutar jedinstvenog programa implementiraju se različite uloge u sustavu (*master-slave* i sl.)

1.3.2 Podatkovni paralelizam

- podatkovni paralelizam (*data parallelism*) - primjena iste operacije na više elemenata podatkovne strukture (npr. "pomnoži sve elemente polja sa 2")
- programski jezik *High Performance Fortran* (*HPC*)

1.3.3 Zajednička memorija

- svi procesori/zadaci dijele isti memorijski spremnik, gdje je čitanje i pisanje asinkrono (razlika od računalnog modela!)
- moguća uporaba (pseudo-) nedeterminističkih algoritama
- potrebni eksplicitni mehanizmi zaštite memorije (semafori i sl.)
- jednostavnije programiranje - manja razlika od slijednog modela algoritma

1.3.4 Sustav zadataka i kanala

- sustav zadataka i kanala (*tasks and channels*) prikazuje se usmjerenim grafom u kojemu su čvorovi zadaci (koji se mogu izvoditi paralelno i neovisno jedan o drugome) a veze su kanali kojima zadaci komuniciraju
- broj zadataka može se mijenjati tokom izvođenja
- poopćenje modela komunikacije porukama
- više detalja kasnije...

1.3.5 Koje modele ćemo koristiti?

- model zajedničke memorije uz višeprocesorsko računalo (odnosno PRAM)
- model zadataka i kanala (komunikacija porukama)

26

1.4 Svojstva paralelnih algoritama

- definiramo poželjna svojstva koja bi paralelni algoritmi trebali imati:
  - **istodobnost** (*concurrency*) - mogućnost izvođenja više radnji istovremeno - nužno za razvoj algoritma
  - **skalabilnost** (*scalability*) - mogućnost prilagođavanja proizvoljnom broju fizičkih procesora (odnosno mogućnost iskorištavanja dodatnog broja računala) - "algoritam koji radi samo na *x* procesora je loš algoritam"
  - **lokalnost** (*locality*) - veći omjer lokalnog u odnosu na udaljeni pristup memoriji - korištenje priručne memorije (*cache*)
  - **modularnost** (*modularity*) - mogućnost uporabe dijelova algoritma unutar različitih programa
- u razvoju paralelnih algoritama dolaze do izražaja još neka područja razmatranja:

1.4.1 Amdahl-ov zakon

- potencijalno ubrzanje definirano je onim udjelom (*P*, *P*=[0,1]) slijednog programa koji se može paralelizirati kao:

$$ubrzanje = \frac{1}{1-P}$$

- npr. ako se 50% programa može paralelizirati (*P* = 0.5), ***ubrzanje*** je 2; ako se cijeli program može paralelizirati, ubrzanje je beskonačno (teoretski)
- uključimo li i broj procesora koji izvode paralelni posao, izraz postaje:

$$ubrzanje = \frac{1}{S + \frac{P}{N}}$$

27

- *S* - slijedni udio programa, *P* - paralelni udio, *N* - broj procesora

ubrzanje	*P* = 50%	*P* = 90%	*P* = 99%
N = 10	1.82	5.26	9.17
N = 100	1.98	9.17	50.25
N = 1000	1.99	9.91	90.99
N = 10000	1.99	9.91	99.02

1.4.2 Ujednačavanje opterećenja (load balancing)

- raspodjela poslova i zadataka u cilju osiguravanja najveće vremenske učinkovitosti algoritma
- česta pojava kod neujednačenih izvođenja: svi zadaci čekaju da jedan zadatak završi
- poseban problem u raznorodnim (***heterogeneus***) računalnim sustavima

1.4.3 Zrnatost (granularity)

- u cilju koordinacije zadataka potrebno je osmisliti međusobnu komunikaciju
- neformalna definicija: omjer između količine (lokalnog) računanja i količine (nelokalne) komunikacije je zrnatost
- **Sitnozrnatost** (*fine-grained*) - mala količina računanja između uzastopnih udaljenih komunikacija
- lakše ujednačavanje opterećenja, ali veći trošak komunikacije (*overhead*)
- **Krupnozrnatost** (*coarse-grained*) - velika količina računanja u odnosu na komunikaciju
- više mogućnosti ubrzavanja ali teža kontrola ujednačavanja

28

1.4.4 Podatkovna ovisnost

- podatkovna ovisnost postoji kod višestruke uporabe iste memorijske lokacije (iste podatkovne strukture u algoritmu)
- čest uzrok nemogućnosti paralelizacije
- načini razrješavanja podatkovne ovisnosti:
  - raspodijeljena memorija: slanje potrebnih podataka u trenutku sinkronizacije
  - zajednička memorija: sinkronizacija čitanja i pisanja memorije među procesorima

1.4.5 Potpuni zastoj (deadlock)

- stanje u kojemu dva ili više procesa čekaju na događaj ili sredstvo od jednog od drugih procesa
- izbjegavanje potpunog zastoja: uklanjanje barem jednoga uvjeta nastajanja istoga

29

1.5 Pretvorba slijednog u paralelni algoritam

- najčešće imamo na raspolaganju slijedni algoritam kojega želimo izvoditi paralelno
- neki od koraka u razvoju paralelnog algoritma (***detaljnije u kasnijim poglavljima***):
  1. pronaći dijelove slijednog programa koji se mogu izvoditi istodobno
     - zahtijeva detaljno poznavanja rada algoritma
     - može zahtijevati i potpuno novi algoritam
  2. rastaviti algoritam
     - funkcionalna dekompozcija - podjela problema na manje dijelove (koji se mogu rješavati istodobno)
     - podatkovna dekompozicija - podjela podataka s kojima algoritam radi na manje dijelove; obično jednostavnije izvesti
     - kombinacija gornja dva načina
  3. ostvarenje programa
     - odabir programske paradigme, sklopovskog okruženja
     - usklađivanje komunikacije (način, učestalost, sinkronizacija...)
     - vanjska kontrola izvođenja
  4. ispravljanje grešaka, optimiranje izvođenja...

1.6 Primjeri

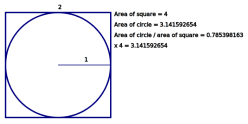
- ilustracija razvoja paralelnog algoritma uz korištenje paradigme komunikacije porukama (*message passing*)

30

1.6.1 Računanje broja Pi (π)

- primjer algoritma za računanje broja *Pi*: ideja je u odnosu površine kruga upisanog kvadratu (*skicirati*):

$$\pi = 4 \cdot \frac{P_{KRUG}}{P_{KVADRAT}}$$



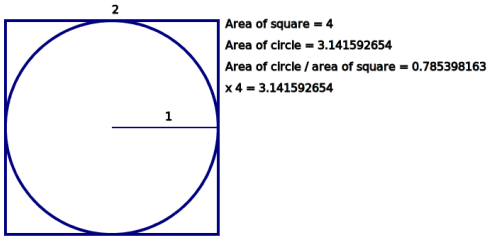
- slijedni algoritam:

```
b_tocaka = 100000; // bilo koji veliki broj - sto veci to preciznije
b_tocaka_krug = 0;

ponovi b_tocaka puta
      R1, R2 = dva slucajna broja [0,2r];
      ako (tocka (R1,R2) unutar kruga)
            b_tocaka_krug++;
kraj_ponovi

PI = 4*b_tocaka_krug/b_tocaka;
```

- *Dijelovi algoritma koji se mogu izvoditi* istodobno: većina vremena troši se u petlji - glavni predmet paralelizacije
- *Paralelizacija algoritma*:
  - svaki procesor izvodi svoj dio petlje - funkcionalna dekompozicija
  - svaki procesor tijekom rada ne treba nikakvu informaciju od drugih - situacija koja se naziva **trivijalno paralelni** algoritam (*embarassinlgy parallel*)
- *Ostvarenje algoritma*: korištenje SPMD modela (komunikacija porukama) - jedan proces gospodar (*master*) sakuplja rezultate od svih ostalih (sluge, *workers*)
- paralelni algoritam:

1.6.1. Računanje broja Pi (π)

https://www.codedrome.com/a-stochastic-estimation-of-pi-in-python/



http://www.johndcook.com/blog/2011/03/14/algorithm-record-pi-calculation/

32

```
b_tocaka = 100000; // bilo koji veliki broj - sto veci to preciznije
b_tocaka_krug = 0;
p = broj procesora;
b_tocaka_p = b_tocaka / p;

ponovi b_tocaka_p puta
      R1, R2 = dva slucajna broja [0,2r];
      ako (tocka (R1,R2) unutar kruga)
            b_tocaka_krug++;
kraj_ponovi

ako sam gospodar
      primi b_tocaka_krug od svih slugu;
      izracunaj PI (pomocu vlastite i zbroja svih dobivenih vrijednosti);
inace ako sam sluga
      posalji gospodaru b_tocaka_krug;
```

- potencijalni problem: računala na kojima izvodimo procese različitih su brzina; na kraju proces gospodar uvijek čeka najsporijeg slugu
- potrebno je ujednačiti opterećenje: jedna moguća strategija je *skup zadataka* (*pool of tasks*)
- posao se podijeli na više manjih dijelova (broj dijelova > broja procesora); moguće uglavnom kod trivijalno paralelnih problema
- proces gospodar:
  - inicijalizira podatke o poslovima
  - šalje pojedinačni posao na zahtjev sluge

33

  - sakuplja rezultate
- proces sluga:
  - dohvaća poslove od gospodara i obavlja ih
  - šalje rezultate gospodaru

```
b_tocaka = 100000; // bilo koji veliki broj - sto veci to preciznije
b_tocaka_krug = 0;
jobs = broj poslova;
b_tocaka_p = b_tocaka / jobs;

ako sam gospodar
      ponovi dok ima poslova
            posalji slugi posao;
            primi b_tocaka_krug od sluge;
      kraj_ponovi
      posalji svim slugama: nema poslova;
      izracunaj PI (pomocu vlastite i zbroja svih dobivenih vrijednosti);

inace ako sam sluga
      ponovi dok ima poslova
            primi posao;
            ponovi b_tocaka_p puta
                  R1, R2 = dva slucajna broja [0,2r];
                  ako (tocka (R1,R2) unutar kruga)
                        b_tocaka_krug++;
            kraj_ponovi
            posalji gospodaru b_tocaka_krug;
      kraj_ponovi
```

1.6.2 Računanje elemenata matrice

- problem: obrada svih elemenata neke podatkovne strukture (npr. matrice) na način koji ne zahtijeva informaciju o drugim elementima (drugim poljima matrice)
- npr: Matrica[i,j] = Funkcija(i,j);
- također trivijalno paralelni problem
- paralelizacija algoritma: svaki procesor računa samo dio matrice (podatkovna dekompozicija), npr. samo redak ili stupac ili podmatricu
- informacije koje se šalju slugama: početni i krajnji indeksi podmatrice i vrijednosti elemenata
- primjer posla sluge:

```
...
ponovi za moje indekse retka
      ponovi za moje indekse stupca
            Matrica[i,j] = Funkcija(i,j);
      kraj_ponovi
kraj_ponovi
...
```

35

2. MPI – Message Passing Interface

1 Nastanak i svojstva standarda

- u razvoju paralelnih aplikacija javlja se potreba za mehanizmom razmjene poruka (*message passing*) za uporabu na računalima sa raspodijeljenom memorijom (nCUBE, iPSC itd, danas nisu u uporabi)
- neki od ranih razvijenih sustava 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 standard
- standard definira komunikaciju porukama, odnosno razmjenu podataka među procesima
- broj procesa u MPI programu je po definiciji konstantan tijekom izvođenja (nisu predviđeni mehanizmi stvaranja odnosno gašenja procesa)
- svi procesi obično izvode isti program (SPMD model), međutim mogu se pokrenuti i različiti programi za različite procese (MPMD model)
- MPMD model se uvijek može simulirati uvrštenjem više funkcija u jedan SPMD program
- **2022: MPI 4.0 i 4.1, a trenutno nastojanja prema 4.2 i 5.0**

https://computing.llnl.gov/tutorials/mpi/
https://www.open-mpi.org/

36

- načini komunikacije u MPI:
  - *point-to-point* komunikacija između dva određena procesa
  - *collective* komunikacija između grupe procesa
  - *probe* funkcije za asinkronu komunikaciju
  - *communicator* mehanizam za razvoj modularnih paralelnih programa (definicija topologije mreže)
- opisano u ovom poglavlju: prva dva mehanizma i malo od trećega, četvrti u kasnijim poglavljima
- MPI je samo standard - za korištenje je potrebna neka MPI implementacija
- korištena implementacija: MPICH http://www.mcs.anl.gov/research/projects/mpi/

2 Osnovne MPI funkcije

- 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)
- svaki C/C++ program koji koristi MPI mora imati `#include"mpi.h"` preprocesorsku naredbu
- dvije funkcije koje se *moraju* naći u svakom MPI programu:

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

- prva funkcija mora biti prije svake MPI komunikacije, dok se druga nalazi na kraju

37

- funkciji `MPI_Init` se prosljeđuju odgovarajući parametri funkcije main
- 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 )
```

- prva funkcija upisuje ukupan broj procesa u grupi u parametar *size*, dok druga funkcija upisuje index procesa pozivatelja u parametar *rank* (indeksi kreću od 0)
- **grupa** procesa se u MPI standardu naziva *communicator* - određuje skup procesa na koje se odnosi konkretna akcija (većina funkcija zathjeva komunikator kao jedan od argumenata)
- korisnik može sam definirati grupe procesa
- globalna grupa koja uključuje **sve** uključene procese označava se sa MPI_COMM_WORLD
- primjer programa:

```
...
int rank, size;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
printf( "Ja sam %d. od %d procesa\n", rank, size );
MPI_Finalize();
...
```

38

3 Razmjena 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
- 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 pošiljatelj zove Send, primatelj Recv; parametri su:
  - *buf* je početna adresa podataka u memoriji koji se šalju odnosno primaju
  - *count* je broj jedinica podataka (duljina niza)
  - *datatype* je odgovarajući MPI tip podatka
  - *dest* i *source* određuju indeks (*rank*) procesa pošiljatelja i primatelja
  - *tag* je oznaka vrste poruke, *comm* je oznaka komunikatora unutar koga se komunikacija odvija (npr. MPI_COMM_WORLD)
  - u parametru *status* će nakon primitka biti zapisani podaci o poruci (oznaka kao `status.MPI_TAG`, proces pošiljatelj kao `status.MPI_SOURCE`)

39

- uvjeti uspjeha komunikacije:
  - indeksi pošiljatelja i primatelja moraju odgovarati
  - mora biti naveden isti komunikator (!)
  - oznake poruke moraju biti iste
  - memorijski prostor primatelja mora biti dovoljno velik
- Poopćenje primanja poruke:
  - od bilo kojeg pošiljatelja - `MPI_ANY_SOURCE` kao *source* parametar
  - bilo koju oznaku - `MPI_ANY_TAG` kao *tag* parametar
- sa navedenih 6 funkcija može se ostvariti velik broj paralelnih programa
- primjer slanja poruke između 2 procesa:

```
if (myrank == 0)    // proces 0
{  strcpy(message,"Poruka!");
   MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
}
else  // proces 1
{  MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
   printf("primljeno :%s:\n", message);
}
```

40

4 Načini komunikacije

- Pitanje: kada će određena funkcija 'završiti', odnosno kada se nastavlja izvođenje procesa pozivatelja?
- Definicija završetka: 'završetak' funkcije označava trenutak kada se može sigurno pristupiti memorijskim lokacijama korištenim u komunikaciji:
  - slanje: poslana varijabla se ponovno može upotrijebiti (npr. pisati)
  - primanje: primljena varijabla se može upotrijebiti (čitati)
- MPI nudi dva osnovna načina *point-to-point* komunikacije:
  - **blokirajući** (*blocking*) - povratak iz funkcije znači da je ista završila u navedenom smislu
  - **neblokirajući** (*non-blocking*) - povratak iz funkcije je trenutan; korisnik mora naknadno provjeriti uvjet završetka
- navedene funkcije MPI_Send i MPI_Recv su *blokirajuće* funkcije (neblokirajuće će biti opisane kasnije):
  - povratak iz MPI_Recv znači da je poruka primljena
  - povratak iz MPI_Send *ne znači* da je poruka primljena, nego da se korištena memorija može ponovno upotrijebiti - završetak može a i ne mora implicirati da je poruka primljena, ovisno o implementaciji i uvjetima
- MPI_Send *može* čekati dok poruka ne bude isporučena (odnosno dok proces primatelj ne pozove odgovarajući MPI_Recv) ili može kopirati poruku u međuspremnik i odmah vratiti kontrolu pozivatelju - ovisno o veličini poruke i strategiji pojedine MPI implementacije
- postoje još neke (blokirajuće) inačice Send funkcije, npr. MPI_SSend (sinkroni send, povratak kada je poruka primljena), MPI_RSend (vraća odmah, ali uspjeva samo ako je odgovarajući Recv već pozvan), itd.
- najčešće korištena metoda je upravo MPI_Send

41

Determinizam u MPI programima

- MPI programski model je u osnovi nedeterministički: ukoliko dva ili više procesa šalju poruke jednom procesu, redoslijed primitka poruka nije definiran
- s druge strane, redoslijed poruka od *jednog* procesa prema drugom je uvijek očuvan
- programer je odgovoran za postizanje determinizma - uporabom komunikatora te parametara *source* i *tag* za svaku poruku
- preporuča se korištenje parametara *source* i *tag* kadgod je to moguće, osim ako eksplicitno ne želimo postići nedeterminizam

5 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 moraju pozvati određenu funkciju
  - sve funkcije su blokirajuće
  - nema oznake poruke (parametar *tag*)
  - memorijski prostori za primanje moraju biti jednake veličine
- operacija slanja svima od jednog procesa:

```
int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype, in
                    root, MPI_Comm comm )
```

- funkcija šalje isti podatak svim procesima u definiranom komunikatoru; izvorišni proces označen je parametrom *root* (indeks procesa) (*nacrtati*)
- parametar *buffer* se kod procesa *root* čita, dok se kod ostalih u njega upisuju podaci

```
int MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm cm)

int MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm cm)
```

- funkcija Scatter raspodjeljuje po jedan dio niza (pohranjen u *sendbuf* na procesu *root*) svakom od procesa (na adresu *recvbuf*)
- podebljani parametri bitni su samo na *root* procesu
- vrsta podataka i broj elemenata određeni su sa *sendtype* i *sendcnt*; *recvtype* i *recvcnt* su obično jednaki
- funkcija Gather prikuplja određeni dio niza od svih procesa (pohranjen u *sendbuf* na svim procesima) i sprema ih u *recvbuf* na *root* procesu (*nacrtati*)
- često je potrebno izračunati neki rezultat na temelju podataka od svih procesa - u tu svrhu se koristi funkcija:

```
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )
```

- funkcija Reduce skuplja podatke od svih procesora (sa adrese *sendbuf*, duljine *count*) ali ujedno nad njima provodi neku operaciju (definiranu sa *op* parametrom) i rezultat sprema na adresu *recvbuf* na procesu *root*
- neke predefinirane operacije: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD (umnožak), MPI_LAND, MPI_BAND (logički i bitwise AND), itd.
- primjer: zbrajanje svih vrijednosti varijable x sa svh procesa i spremanje rezultata u rez na procesu 0:

43

```
MPI_Reduce(&x, &rez, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

- svi procesi moraju pozvati funkciju, no drugi parametar (rez) bitan je samo na procesu 0, dok se na ostalima ne koristi

5.1 Globalna sinkronizacija

- globalna sinkronizacija za sve procese unutar grupe postiže se pozivom funkcije:

```
int MPI_Barrier ( MPI_Comm comm )
```

- za sve procese koji pozivaju funkciju vrijedi da niti jedan neće nastaviti sa radom dok je svi procesi ne pozovu
- obično se sinkronizacija među porukama postiže uporabom oznaka i komunikatora, no ako je potrebno može se upotrijebiti i navedena funkcija

44

6 Ispravnost programa

- Pitanje: uključuju li funkcije globalne komunikacije međusobnu sinkronizaciju procesa?
- Odgovor: Ne! standardom nije propisano hoće li se procesi prilikom globalne kom. sinkronizirati (kao sporedni učinak) ili ne, već to ovisi o izvedbi navedenih funkcija
- u većini implementacija globalne komunikacije izvedene su kao niz *point-to-point* funkcija, pa sinkronizacija ovisi i o trenutnim uvjetima rada (veličina poruke i sl.)
- *Ispravan* program:
  - ne smije se oslanjati na sinkronizaciju prilikom globalne komunikacije,
  - mora pretpostavljati da globalna komunikacija *može* biti sinkonizirajuća.
- Primjer 1 (izvodi se na dva procesa, 0 i 1):

```
switch(rank)
{   case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf2, count, type, 1, comm);
        break;
    case 1:
        MPI_Bcast(buf2, count, type, 1, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

- primjer je neispravan jer ukoliko je operacija sinkronizirajuća, nastaje potpuni zastoj
- rješenje: globalni pozivi moraju imati jednaki redoslijed za sve procese u grupi

45

- Primjer 2:

```
switch(rank)
{   case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

- primjer je neispravan jer dolazi do potpunog zastoja ako proces 0 čeka na Bcast poziv procesa 1
- rješenje: relativni poredak globalnih i *point-to-point* komunikacija mora biti takav da ne smije doći do potpunog zastoja u i slučaju kada su obje vrste operacija sinkronizirajuće

46

- Primjer 3:

```
switch(rank)
{   case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

- primjer je ispravan ali nedeterministički, ovisno o tome da li će pozivi Bcast biti sinkronizirajući ili ne - dva moguća ishoda programa (*nacrtati!*)
- program koji računa samo na jedan od mogućih ishoda je neispravan

47

7 Asinkrona komunikacija

- sinkronizacija među procesima obično uključuje čekanje velikog broja procesa - ukoliko želimo maksimalno iskoristiti vrijeme, koristi se asinkrona komunikacija
- u asinkronoj komunikaciji pozivaju se neblokirajuće (*non-blocking*) funkcije
- postupak asinkrone komunikacije:
  - pozivanje neblokirajuće funkcije
  - obavljanje posla koji *ne* uključuje podatke iz asinkronog poziva
  - čekanje ili provjeravanje (u petlji) završetka funkcije
- funkcije za slanje i primanje poruke:

```
int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int
   dest, int tag, MPI_Comm comm, MPI_Request *request )

int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int
  source, int tag, MPI_Comm comm, MPI_Request *request )
```

- umjesto status varijable, u ove pozive dodan je parametar *request* pomoću kojega se ispituje završetak funkcije
- dvije obično korištene funkcije:

```
int MPI_Wait ( MPI_Request *request, MPI_Status *status)

int MPI_Test ( MPI_Request *request, int *flag, MPI_Status *status)
```

48

- funkcija Wait ne završava dok se ne završi radnja identificirana parametrom *request*
- funkcija Test završava odmah i upisuje vrijednost TRUE ili FALSE u parametar *flag* ovisno o tome je li odgovarajuća asinkrona funkcija završila
- *završetak* neblokirajuće funkcije definiran je jednako kao i završetak blokirajuće inačice:
  - MPI_Isend završava kada se izlazni međuspremnik može ponovno iskoristiti
  - MPI_Issend završava kada proces primatelj pozove odgovarajuću Recv funkciju (sinkrona inačica neblokirajuće funkcije)
  - MPI_Irecv završava kada se ulazni međuspremnik može koristiti (podaci upisani)
- moguće je kombinirati blokirajuće i neblokirajuće pozive (npr. neblokirajući Send i blokirajući Recv i obrnuto)
  - načini korištenja neblokirajućeg primanja pomoću Test i Wait:

```
// rad sa Wait
MPI_Irecv( x, …, request,…)
   radi nesto sto ne ukljucuje x
MPI_Wait( request, status )
   radi nesto sto ukljucuje x
. . .
// rad sa Test
MPI_Irecv( x, …, request,…)
MPI_Test( request, flag, status )
ponavljaj
{      radi nesto sto ne ukljucuje x
       MPI_Test( request, flag, status )
} dok flag!=TRUE
radi nesto sto ukljucuje x
```

8 Ostale funkcije

- objašnjenja svih ostalih funkcija, primjeri korištenja, upute za pojedine platforme i slično mogu se naći na stranici MPICH implementacije
- neke korisne funkcije:
  - MPI_Wtime - mjerenje utrošenog vremena
  - MPI_Get_processor_name - dohvat imena računala (ako je definirano)

9 Primjer: računanje broja Pi

- primjer paralelnog programa koji računa Pi (različit algoritam od onoga iz poglavlja 1.6, ali također trivijalno paralelan)

50

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

int main(int argc, char *argv[])
{ int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.14159265358979323846264 3;
  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();
}
```

51

3. Postupak oblikovanja paralelnih programa

1 Računalni i programski model

- za opis razvoja paralelnih algoritama, potrebno je definirati model paralelnog računala i strukturu paralelnog programa

1.1 Računalni model

- Multiračunalo - više neovisnih računala povezanih nekom vrstom mreže
- pretpostavke:
  - brzina komunikacije ne ovisi o međusobnom položaju računala u topologiji mreže
  - pristup lokalnoj memoriji vremenski je *manje skup* od pristupa udaljenoj memoriji (na drugom računalu)
- po osobinama, multiračunalo je MIMD računalo raspodijeljene memorije

52

1.2 Programski model

- koristit ćemo model zadataka i komunikacijskih kanala
- opis programa: usmjereni graf u kojmu su čvorovi zadaci, a veze su komunikacijski kanali
- osobine sustava:
  - paralelni program sastoji se od jednog ili više zadataka; zadaci se mogu izvoditi istovremeno
  - broj zadataka može se mijenjati tijekom izvođenja
  - zadatak obuhvaća slijedni program i lokalnu memoriju
  - zadatak može, osim rada s lokalnom memorijom, izvesti četiri operacije: slati poruku, primiti poruku, stvoriti nove zadatke i završiti s radom
  - operacija slanja je po pretpostavci asinkrona (odmah završava), dok je operacija primanja sinkrona (završava po primitku poruke)
  - broj i položaj komunikacijskih kanala može se mijenjati tijekom izvođenja
  - zadaci mogu biti pridruženi procesorima na više načina, što ne utječe na funkcionalnost programa (jedan ili više zadataka po procesoru)
- opisani programski model može se primijeniti na više modela paralelnih računala

53

3 Faze oblikovanja paralelnog algoritma

- razvoj (paralelnog) algoritma nije moguće svesti na recept, ali je moguće koristiti metodički pristup za lakše otkrivanje nedostataka u strukturi
- svojstva koja želimo postići kod paralelnog algoritma (poglavlje 1.4):
  - **istodobnost** (*concurrency*) - mogućnost izvođenja više radnji istovremeno
  - **skalabilnost** (*scalability*) - mogućnost prilagođavanja proizvoljnom broju fizičkih procesora (odnosno mogućnost iskorištavanja dodatnog broja računala) -
  - **lokalnost** (*locality*) - veći omjer lokalnog u odnosu na udaljeni pristup memoriji
  - **modularnost** (*modularity*) - mogućnost uporabe dijelova algoritma unutar različitih paralelnih programa
- definiramo četiri faze razvoja algoritma:
  - **Podjela** (*partitioning*) - dekompozicija problema na manje cjeline (zanemaruje se broj procesora i memorijska struktura fizičkog računala)
  - **Komunikacija** (*communication*) - određivanje potrebne komunikacije među zadacima
  - **Aglomeracija** (*agglomeration*) - skupovi zadataka i komunikacijskih kanala iz prve dvije faze se, ako je to isplativo, grupiraju u odgovarajuće logičke cjeline (u cilju povećavanja performansi i smanjenja potrebne komunikacije)
  - **Pridruživanje** (*mapping*) - svaki zadatak se dodijeljuje konkretnom procesoru; može biti određeno *a priori* ili se dinamički mijenjati tijekom izvođenja
- u prve dvije faze želimo postići istodobnost i skalabilnost, dok se lokalnost istražuje u druge dvije (modularnost u posebnom poglavlju)
- proces razvoja ne mora se odvijati slijedno nego i uz preklapanje i ponavljanje faza

54

4.1 Podjela podataka (*domain decomposition*)

- podaci nad kojima algoritam radi dijele se u manje cjeline (obično podjednake veličine) - domenska dekompozicija
- definiramo zadatke koji su 'zaduženi' za odgovarajući dio podataka
- u općenitom slučaju među zadacima je potrebno definirati neki oblik komunikacije - u ovoj fazi ne razmatramo kakav
- primjeri: podjela višedimenzijske podatkovne strukture na elemente smanjenih dimenzija (npr. volumen, površina, niz, točka)

4.2 Podjela izračunavanja (*functional decomposition*)

- prvo se izvodi podjela računanja, a zatim eventualna raspodjela podataka - funkcionalna dekompozicija
- u općenitom slučaju teže za izvesti i manje intuitivno
- primjeri: algoritam pretraživanja stabla; simulacija klimatskog modela (rastav na fizikalne komponente: atmosferu, površinu, ocean itd.)

55

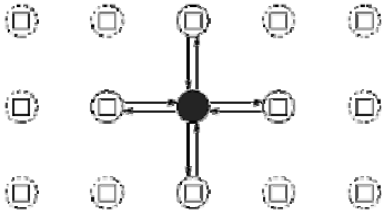
5 Komunikacija

- nakon podjele posla na zadatke, definira se sva komunikacija potrebna za rad algoritma
- definiraju se kanali za razmjenu podataka i količina podataka koja prolazi tim kanalima
- ciljevi: smanjiti ukupnu količinu komunikacije i raspodijeliti komunikaciju tako da se može odvijati paralelno
- podjela posla podjelom podataka obično zahtjeva složeniju i manje intuitivnu komunikaciju od podjele izračunavanja
- cjelokupnu komunikaciju dijelimo na cjeline po nekoliko osnova:
  - lokalna/globalna: u lokalnoj komunikaciji svaki zadatak komunicija sa manjim skupom zadataka koji čine njegovu okoliku, dok globalna komunikacija uključuje sve ili velik dio zadataka
  - strukturirana/nestrukturirana: strukturirana komunikacija obuhvaća grupu zadataka koji tvore pravilnu strukturu (stablo, prsten i sl.), dok nestrukturirana može obuhvaćati bilo koji skup zadataka
  - statična/dinamična: u statičnoj komunikaciji ne mijenjaju se procesi koji sudjeluju u istoj, dok se identitet zadataka u dinamičnoj komunikaciji mijenja tijekom izvođenja
  - sinkrona/asinkrona: u sinkronoj komunikaciji i pošiljatelj i primatelj zajednički (koordinirano) sudjeluju, dok u asinkronoj komunikaciji jedan zadatak može tražiti podatke od drugoga, bez njegove aktivne suradnje

56

5.1 Lokalna komunikacija

- često je isplativo optimirati lokalnu komunikaciju jer predstavlja najviše troškova
- primjer: iterativno računanje elemenata dvodimenzijskog polja - u svakoj iteraciji nove vrijednosti računaju se pomoću vrijednosti susjeda (4 za 2D polje) - metoda konačnih elemenata
- skup susjeda koji su potrebni za računanje nove vrijednosti je *maska* (*stencil*)

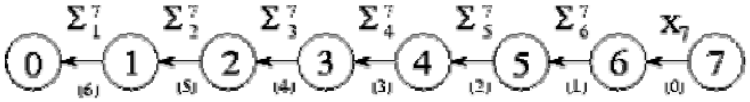


5.2 Globalna komunikacija

- u globalnoj komunikaciji sudjeluje veća grupa zadataka
- potrebno je razviti globalne komunikacijske algoritme
- primjer: operacija reduciranja nad poljem duljine N sa nekim binarnim operatorom
- *Prva izvedba*: svi zadaci (elementi polja) šalju svoje podatke jednom zadatku koji prima podatke i računa rezultat

57

- nedostaci izvedbe:
  - centralizirana – jedan zadatak mora sudjelovati u cjelokupnoj komunikaciji
  - slijedna – ne dopušta paralelno računanje ni komunikaciju
- *Druga izvedba*: raspodjeljivanje računanja i komunikacije



- N-1 korak komunikacije i računanja raspodijeljeni su među svim zadacima
- opaska: preslikani slijedni algoritam *+_scan*
- Treća izvedba: korištenje binarnog (b-arnog) stabla - *+_reduce* algoritam
- korištenje binarnog stabla primjer je principa "podijeli pa vladaj" (*divide and conquer*) koji se može primijeniti na puno problema u paralelnom okruženju
- MPI standard uključuje funkcije globalne komunikacije (Reduce, Bcast...)

58

5.3 Asinkrona komunikacija

- u siknronoj komunikaciji svi zadaci sudjeluju aktivno (planski se i predviđeno provode operacije slanja i primanja)
- u asinkronoj komunikaciji, zadatak primatelj mora zatražiti određeni podatak od zadatka pošiljatelja
- primjer: skup zadataka koji povremeno moraju pristupiti zajedničkoj podatkovnoj strukturi, podaci su ili preveliki da bi se replicirali na svim zadacima ili se prečesto javljaju zahtjevi za pristupom (što izaziva probleme pri sinkronizaciji) ili oboje
- moguća rješenja:
  - podaci su raspodijeljeni po zadacima; svaki zadatak obrađuje svoj dio podataka i traži dodatne podatke od ostalih zadataka. Također povremeno provjerava (prekidom) postoje li zahtjevi za njegovim podacima od strane drugih zadataka
    - nedostaci: nemodularnost programa koji povremeno mora provjeravati ima li zahtjeva za njegovim podacima, uz neizbježno trošenje vremena za provjeru
  - podaci su raspodijeljeni po zadacima odgovornim isključivo za čuvanje podataka; ovi zadaci ne sudjeluju u računanju nego ispunjavaju zahtjeve za dohvatom i pisanjem podataka od strane drugih zadataka (koji obavljaju računanje)
    - nedostaci: mala lokalnost - svi zahtjevi za podacima su udaljeni
  - na računalu sa zajedničkom memorijom: svi zadaci jedoliko pristupaju podacima, no pristupanje podacima (čitanje i pisanje) se mora odvijati po predefiniranom rasporedu

59

5.4 Izvedba asinkrone komunikacije u MPI modelu

- definicija asinkrone komunikacija u okviru ovog poglavlja ne odgovara uporabi neblokirajućih funkcija iz pretodnog poglavlja
- neblokirajuće funkcije MPI_Isend i MPI_Irecv koriste se kada zadaci komuniciraju planski (i jedan i drugi zadatak 'svjesni' su procesa komunikacije)
- funkcije za provjeravanje pristiglih poruka:

```
int MPI_Iprobe( int source, int tag, MPI_Comm com, int *flag,
                MPI_Status *status )

int MPI_Probe( int source, int tag, MPI_Comm com,
               MPI_Status *status )

int MPI_Get_count( MPI_Status *status, MPI_Datatype datatype,
                   int *count )
```

- MPI_Iprobe je neblokirajuća funkcija koja provjerava postoji li (dolazna) poruka od izvora *source* sa oznakom *tag*, a rezultat sprema u parametar *flag*
- MPI_Probe je blokirajuća funkcija koja završava kada proces pozivatelj dobije poruku s odgovarajućim parametrima
- MPI_Get_count je funkcija koja u parametar *count* sprema broj podataka tipa *datatype* u pristigloj poruci čija je oznaka *status* (dobiven od funkcije MPI_Iprobe ili MPI_Probe.)
- nakon dospijeća poruke, izvor i oznaka poruke mogu se pročitati iz parametra *status* (status.MPI_SOURCE i status.MPI_TAG)
- poruka se tada može primiti pozivom MPI_Recv funkcije

60

6.1 Povećavanje zrnatosti

- mijenjanjem zrnatosti moguće je uskladiti omjer komunikacije i računanja po zadatku
- ukoliko je početna zrnatost mala, komunikacijski troškovi su najčešće veći od računanja
- smanjiti komunikacijske troškove možemo izvesti komuniciranjem manje količine podataka ili uporabom *manjeg broja komunikacija*
- povećavanjem zrnatosti moguć je i gubitak dijela istodobnosti, no on je isplativ ako dovoljno smanjuje komunikacijske troškove
- **Povećavanje zadataka** najboje je obaviti grupiranjem po svim dimenzijama podatkovne strukture
- primjer: grupiranje zadataka u 2D polju gdje zadaci komuniciraju sa susjedima
- grupiranjem po obje dimenzije (npr. 4 zadatka tvore novi zadatak) ne uvodimo nove troškove u računanju ali dobijamo manji *ukupan* broj poruka sa većom količinom podataka po poruci

6.2 Očuvanje prilagodljivosti

- dobra osobina algoritma je mogućnost stvaranja odnosno prilagođavanja različitom broju zadataka ovisno o veličini problema ili broju procesora paralelnog računala
- ukupan broj zadataka ne bi smio biti ograničen nekim konstantnom vrijednošću
- veći broj zadataka od broja procesora omogućuje bolje pridruživanje zadataka procesorima
- ukoliko je operacija primanja podataka blokirajuća (ili je algoritam tako osmišljen), poželjno je grupirati više zadataka tako da u grupi uvijek postoje zadaci koji mogu izvoditi računanje (ako određeni broj zadataka čeka na poruku)

61

6.3 Smanjenje troškova implementacije

- ukoliko za razvoj paralelnog programa koristimo postojeći slijedni algoritam, poželjno je odabrati izvedbu u kojoj je potrebno što manje promjena originalnog algoritma
- primjer: višedimenzijska podatkovna struktura ne mora biti podijeljena po svim dimenzijama ako to omogućuje korištenje postojećih algoritmama (npr. koji rade nad nizom elemenata)
- ukoliko se paralelni program namjerava koristiti unutar većeg paralelnog sustava, poželjno je odabrati izvedbu koja se sa što manje troškova može prilagoditi postojećim modulima
- primjer: 'najbolja' izvedba algoritma podrazumijeva 3D dekompoziciju podataka, no prethodni stupanj obrade podataka kao rezultat daje 2D dekompoziciju
- možemo prilagoditi ili jedan ili drugi ili oba modula, ili dodati poseban modul za pretvorbu međurezultata

62

7 *Pridruživanje*

- cilj: odrediti na kojem računalu/procesoru će se pojedini zadatak izvoditi
- dvije jednostavne strategije (često proturječne):
- zadaci koji se izvode neovisno/istodobno dodjeljuju se različitim procesorima
- zadaci koji često komuniciraju dodjeljuju se istom procesoru
- u općenitom slučaju, problem pridruživanja je NP kompletan
- najjednostavniji pristup: zadaci **jednoliko raspodijeljeni** po procesorima (npr. 3x3=9 zadatka iz 2D mreže po procesoru)
- često je nužna uporaba algoritama **ujednačavanja opterećenja** i/ili **raspoređivanja zadataka** (*task scheduling*)
- ujednačavanje opterećenja može biti i *dinamičko* (ako se mijenjaju uvjeti izvođenja), u kojem slučaju se ujednačavanje pokreće nekoliko puta tijekom rada paralelnog programa
- poželjno je u tom slučaju imati *lokalne* algoritme ujednačavanja (bez potrebe dohvaćanja globalnog opterećenja)
- u slučaju funkcionalne dekompozicije (ako je životni vijek zadataka kraći), obično se upotrebljavaju algoritmi raspoređivanja koji dodjeljuju zadatke slobodnim procesorima

63

7.2 Raspoređivanje zadataka

- koriste se najčešće u slučajevima kada imamo više zadataka kraćeg životnog vijeka
- obično se održava centralizirani ili raspodijeljeni 'bazen' zadataka koji se dodjeljuju slobodnim procesorima
- problem: odabir načina raspodjele zadataka procesorima
- **Voditelj/radnik** model (*manager/worker*) se sastoji od jednog procesora voditelja koji na zahtjev raspodjeljuje zadatke radnicima
- učinkovitost pristupa ovisi o broju radnika i troškovima dohvata zadataka (zagušenje voditelja)
- poboljšanja: radnici dohvaćaju sljedeći problem prije završetka rada na trenutnom (*prefetching*) kako bi se iskoristilo vrijeme potrebno za komunikaciju
- **Hijerarhijski voditelj/radnik** model koristi, osim zajedničkog voditelja, dodatnu podjelu na podgrupe zadataka sa vlastitim voditeljem za svaku grupu
- **Decentralizirana metoda** nema jedinstveni bazen zadataka, već su zadaci raspodijeljeni po svim procesorima
- slobodni procesori zahtjevaju zadatke ili od predefiniranog skupa 'susjeda' ili od slučajno odabranih procesora
- moguće je definirati i procesor voditelj koji prosljeđuje zahtjeve slobodnih proceosra (svi kontakritaju njega) drugim procesorima po nekom algoritmu (npr. *round-robin*)
- za sve postupke raspoređivanja zadataka potrebno je definirati i mehanizam **otkrivanja završetka** (*termination detection*) - svim radnicima je potrebno javiti da više nema zadataka, za što se u decentraliziranom sustavu mora koristiti posebni algoritam

64

8. Nakupina računala (cluster)

- Postrojenje umreženih, samostalnih, common-off-the-shelf računala korištenih zajedno za rješavanje danog problema.
- Različite vrste nakupina računala

1. Nakupine visokih performansi (High Performance Computing Cluster)

- npr. Beowulf Cluster korišten 90-ih za rudarenje nad podacima, simulacije, paralelnu obradbu, modeliranje klimatskih promjena, itd.
- najpoznatiji ROCKS NPACI, HPCC

2. S raspodjelom opterećenja (Load Balancing)

- performanse u obliku raspodjele opterećenja
- s ftp i web poslužiteljima
- potreban veliki broj računala kako bi se dijelilo opterećenje

3. Visokoraspoložive nakupine (High Availability)

- sprječavaju ispade usluga, sa zalihošću, uglavnom s LB
- za 2. i 3. RedHat HA cluster, Turbolinux Cluster Server, Linux Virtual Server Project

65

8.1. HPCC nakupine i paralelno računarstvo u primjenama

- Message Passing Interface
  - MPICH (https://www.mpich.org/)
  - LAM/MPI (http://www.dcs.ed.ac.uk/home/trollius/www.osc.edu/lam.html)
- Mathematical
  - fftw (fast fourier transform)
  - pblas (parallel basic linear algebra software)
  - atlas (a collections of mathematical library)
  - sprng (scalable parallel random number generator)
- Quantum Chemistry software
  - gaussian, qchem
- Molecular Dynamic solver
- Weather modelling

66

8.1. Primjeri nakupina računala

Donedavni čvor FERIT Osijek

- 1 front-end podatkovni poslužitelj
  - DELL PowerEdge R805
  - 8 x86_64, Quad Core Opteron 2354 2.2 GHz
  - 8 GB RAM
- 16 radnih čvorova
  - 8 x86_64, Quad-core Intel Xeon Processor E5430 2.66 GHz
  - 16 GB RAM
  - 146 GB SCSI HD
- 1 podatkovni element
  - DELL PowerVault MD3000i
  - 5.5. TB

67

8.1. Primjeri nakupina računala - nastavak

68

8.1. Primjeri nakupina računala - nastavak

SRCE Zagreb

- 12 radnih čvorova
  - IBM NeXtScale nx360 M5
  - 8 x86_64 Intel Xeon E5-2683 v3 2GHz
  - 128 GB RAM
  - 1 TB HD
- 11 radnih čvorova
  - Sun Fire x4600
  - 32 x86_64 Quad-Core AMD Opteron 2.7 GHz
  - 64 GB RAM
  - 580 GB HD

  ....
- 1 GPU
  - Tesla M2075
  - 5 GB RAM

69

8.1. Primjeri nakupina računala - nastavak

- Interconnect configuration
- Extreme BlackDiamond Gigabit ethernet switch

70

8.1. HR-ZOO i Isabella

CRO NGI (ne koristi se više kao sustav)

Računalni klaster Isabella

HR-ZOO

Predavanje, 21.11.2023., 11:30, K2-1
https://www.hpc-cc.hr/Pristup_i_koristenje_hrvatskih_i_europskih_superracunala

71

8.2. Osvrt na kvantno računarstvo

Uvod u kvantno računarstvo
https://www.geeksforgeeks.org/introduction-quantum-computing/
https://quantum-computing.ibm.com/composer/docs/iqx/guide/

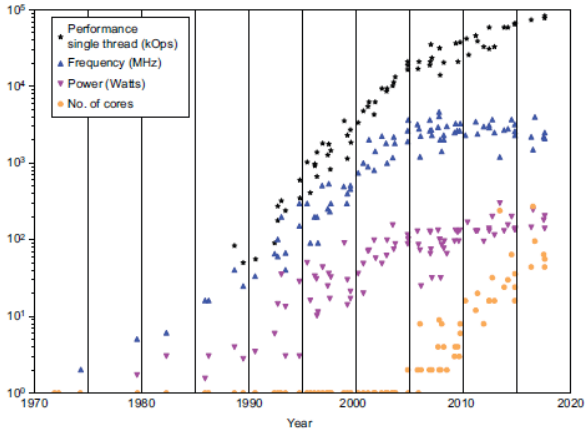
Projekti i alati

1. Microsoft Quantum Development Kit
2. IBM Quantum Experience
3. Rigetti Forest and Cloud Computing Services (QCS)
4. CAS-Alibaba Quantum Computing Laboratory – Superconducting Quantum Computer
5. ProjectQ
6. Cirq
7. CirqProjectQ
8. PennyLane and Strawberry Fields from Xanadu
9. Quantum Programming Studio
10. Atos/SFTC Hartree Centre – Quantum Learning as a Service (QLaaS)
11. QuEST
12. TensorFlow Quantum
13. Microsoft LIQUi|>
14. Quantum in the Cloud

72

Paralelni sustavi

Zašto koristiti paralelno računarstvo?

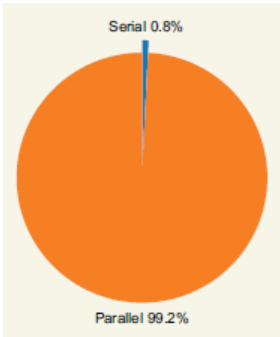


Jednonitno izvođenje, frekvencija takta CPU (MHz), potrošnja CPU (W) i broj procesorskih jezgri od 1970. do 2018.
(https://github.com/karlrupp/microprocessor-trend-data).

73

Paralelni sustavi

Zašto koristiti paralelno računarstvo?

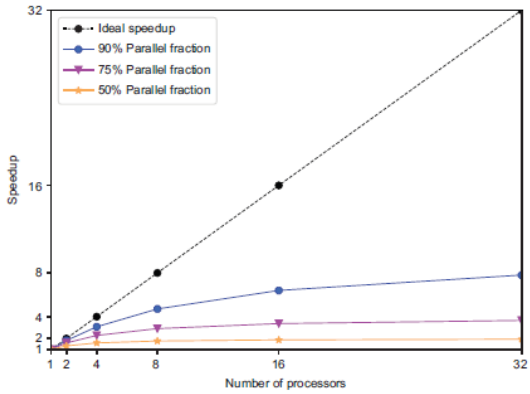


Slijedno izvođene aplikacije koriste samo 0.8% obradbene snage 16-jezgrenog procesora.

74

Paralelni sustavi

Zašto koristiti paralelno računarstvo?

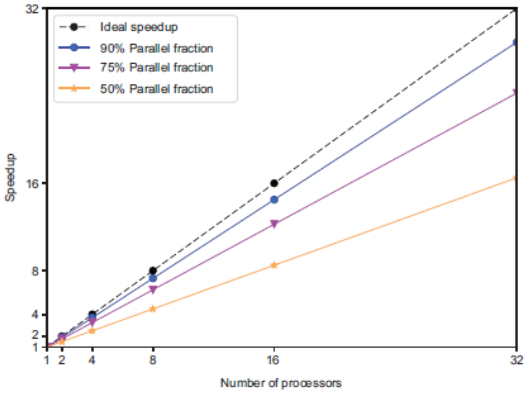


Ubrzanje za problem stalnog raspona prema Amdahlovom zakonu funkcija je broja procesora i udjela koji se može paralelizirati.

75

Paralelni sustavi

Zašto koristiti paralelno računarstvo?

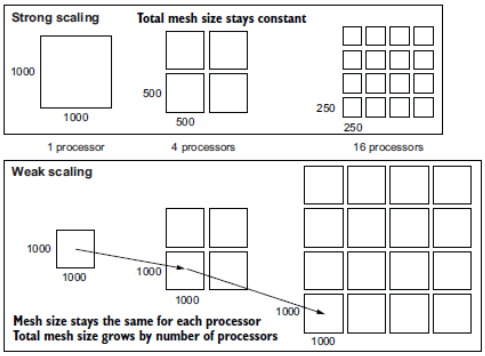


Ubrzanje kada raspon problema raste s brojem procesora prema Speedup for when the size of a problem grows with the number of Gustafson-Barsi zakonu (za udio paralelizma od 100, 90, 75 i 50%)

76

Paralelni sustavi

Jako i slabo skaliranje problema

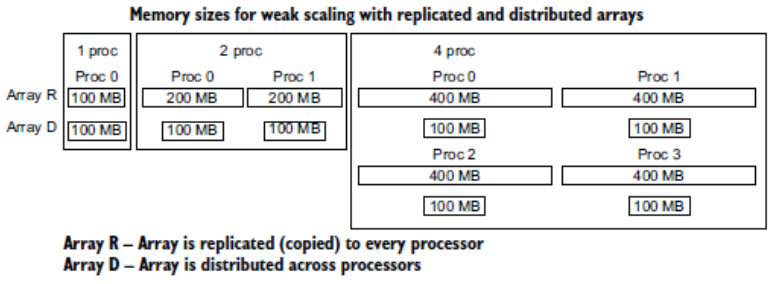


Jako skaliranje problema ostaje jednako za raspon problema i dijeli ga na dodatne procesore. Kod slabog skaliranja, duljina mreže ostaje jednaka za svaki procesor i ukupna duljina se povećava.

77

Paralelni sustavi

Jako i slabo skaliranje problema



Raspodijeljene okoline ostaju jednake duljine kada se raspon problema i broj procesora povećava (slabo skaliranje). Međutim, replicirane (kopirane) okoline trebaju sve podatke na svakom procesoru i memorija se rapidno povećava povećanjem broja procesora. Čak i ako se vrijeme pokretanja slabo povećava ili ostaje konstantno, zahtjevi za memorijom ograničavaju skalabilnost.

78

Paralelni sustavi

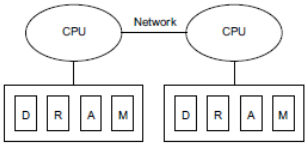
Paralelni pristupi u rješavanju problema

- Paralellizacija na razini procesa
- Paralelizacija na razini niti
- Vektiriziranje
- Obrada tokova

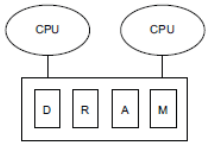
79

Paralelni sustavi

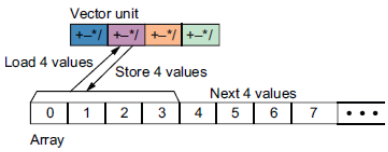
Sklopovski model raznorodnih paralelnih sustava



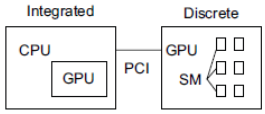
Raspodijeljena memorija na čvorovima



Dijeljena memorija omogućuje paralelizaciju na čvorovima



Vektorska obrada za pr. četiri elementa simultano

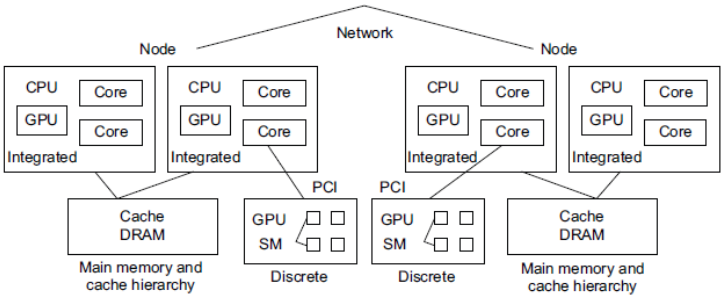


GPU može biti integrirana i diskretna. Diskretna ili predodređena obično ima veliki broj „streaming“ procesora i vlastiti DRAM. Pristup podacima kod diskretne izvedbe zahtijeva komuniciranje preko PCI sabirnice.

80

Paralelni sustavi

Sklopovski model raznorodnih paralelnih sustava

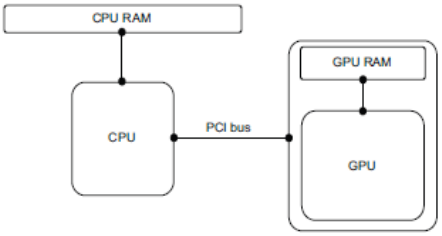


Opći heterogeni model paralelne arhitekture koji se sastoji od dva čvora povezani mrežom. Svaki čvor ima višejezgreni CPU s integriranim i diskretnim GPU i nešto memorije (DRAM). Moderna sklopovska arhitektura iam definiran raspored ovih komponenti.

81

GPU

Predodređena GPU arhitektura



GPU-akcelerirani sustav korištenjem predodređenih GPU. CPU i GPU svaka imaju svoju vlastitu memoriju, a komuniciraju preko PCI sabirnice.

82

GPU

Sklopovska terminologija

Host	OpenCL	AMD GPU	NVIDIA/CUDA	Intel Gen11
CPU	Compute device	GPU	GPU	GPU
Multiprocessor	Compute unit (CU)	Compute unit (CU)	Streaming multi-processor (SM)	Subslice
Processing core (Core for short)	Processing element (PE)	Processing element (PE)	Compute cores or CUDA cores	Execution units (EU)
Thread	Work Item	Work Item	Thread	
Vector or SIMD	Vector	Vector	Emulated with SIMT warp	SIMD

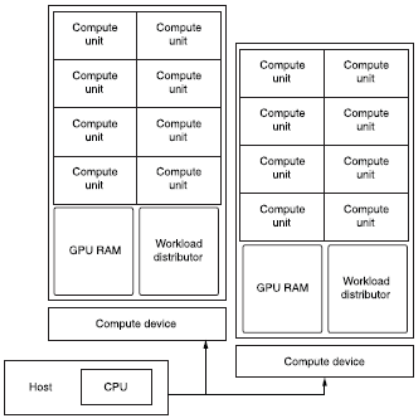
GPU jedinice repliciranja prema proizvođaču

AMD	NVIDIA/CUDA	Intel Gen11
Shader Engine (SE)	Graphics processing cluster	Slice

83

GPU

Pojednostavljeni blok dijagram GPU sustava

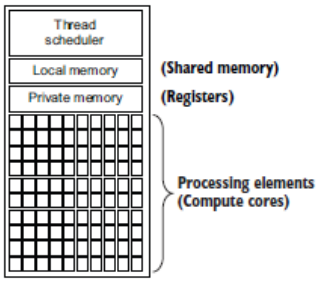


Dva obradbena uređaja od kojih svaki ima odvojene GPU, GPU memoriju i višestruke obradbe jedinice (CU). NVIDIA CUDA CU naziva *streaming multiprocessors* (SM).

84

GPU

Obradbene jedinice kao samostalni procesori



Pojednostavljeni blok dijagram compute jedinice (CU) s velikim brojem processing elementata (PEs).

85

GPU

Specifikacije nedavnih diskretnih GPU od tvrtki NVIDIA, AMD i integrirani Intel GPU

GPU	NVIDIA V100 (Volta)	NVIDIA A100 (Ampere)	AMD Vega 20 (MI50)	AMD Arcturus (MI100)	Intel Gen11 Integrated
Compute units (CUs)	80	108	60	120	8
FP32 cores/CU	64	64	64	64	64
FP64 cores/CU	32	32	32	32	
GPU clock nominal/boost	1290/1530 MHz	1410 MHz	1200/1746 MHz	1000/1502 MHz	400/1000 MHz
Subgroup or warp size	32	32	64	64	
Memory clock	876 MHz	1215 MHz	1000 MHz	1200 MHz	Shared memory
Memory type	HBM2 (32 GB)	HBM2(40 GB)	HBM2	HBM2 (32 GB)	LPDDR4X-3733
Memory data width	4096 bits	5120 bits	4096 bits	4096 bits	384 bits
Memory bus type	NVLink or PCIe 3.0x16	NVLink or PCIe Gen 4	Infinity Fabric or PCIe 4.0x16	Infinity Fabric or PCIe 4.0x16	Shared memory
Design Power	300 watts	400 watts	300 watts	300 watts	28 watts

86

GPU

Primjeri vršnih performansi

Peak Theoretical Flops (GFlops/s)
= Clock rate MHZ × Compute Units × Processing units
× Flops/cycle

Example: Peak theoretical flop for some leading GPUs

Theoretical Peak Flops for NVIDIA V100:

- 2 x 1530 x 80 x 64 /10^6 = 15.6 TFlops (single precision)
- 2 x 1530 x 80 x 32 /10^6 = 7.8 TFlops (double precision)

Theoretical Peak Flops for NVIDIA Ampere:

- 2 x 1410 x 108 x 64 /10^6 = 19.5 TFlops (single precision)
- 2 x 1410 x 108 xx 32 /10^6 = 9.7 TFlops (double precision)

Theoretical Peak Flops for AMD Vega 20 (MI50):

- 2 x 1746 x 60 x 64 /10^6 = 13.4 TFlops (single precision)
- 2 x 1746 x 60 x 32 /10^6 = 6.7 TFlops (double precision)

Theoretical Peak Flops for AMD Arcturus (MI100):

- 2 x 1502 x 120 x 64 /10^6 = 23.1 TFlops (single precision)
- 2 x 1502 x 120 x 32 /10^6 = 11.5 TFlops (double precision)

Theoretical Peak Flops for Intel Integrated Gen 11 on Ice Lake:

- 2 x 1000 x 64 x 8 /10^6 = 1.0 TFlops (single precision)

87

GPU

Primjeri:

- FluidsGL
- Nbody

OpenACC/mass_sum/mass_sum.c

```
 1 #include "mass_sum.h"
 2 #define REAL_CELL 1
 3
 4 double mass_sum(int ncells, int* restrict celltype,
 5                 double* restrict H, double* restrict dx,
                   double* restrict dy){
 6    double summer = 0.0;
 7 #pragma acc parallel loop reduction(+:summer)
 8    for (int ic=0; ic<ncells ; ic++) {
 9       if (celltype[ic] == REAL_CELL) {
10          summer += H[ic]*dx[ic]*dy[ic];
11       }
12    }
13    return(summer);
14 }
```

Adds a reduction clause to a parallel loop construct

88