Predavanja iz kolegija

Paralelno programiranje

Sadržaj:

Zadnja izmjena: 28. veljače 2023

Domagoj Jakobović, Fakultet elektrotehnike i računarstva, Zagreb

Osnovne informacije o predmetu

Predavač(i): Domagoj Jakobović, Ante Đerek
Web: http://www.fer.hr/predmet/parpro

1 Uvod

- **Ciljevi poglavlja**: upoznati osnovne modele paralelnih računala i programa, neka svojstva paralelnih programa i smjernice za stvaranje paralelnog algoritma
- Literatura:
  1. Introduction to Parallel Computing (https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial)
  2. "Designing and Building Parallel Programs", I. Foster, Addison-Wesley, 1995. (*online*) (http://www.mcs.anl.gov/~itf/dbpp/)
  3. "Introduction to Parallel Computing", A. Grama, A. Gupta, G. Karypis, V. Kumar, Addison-Wesley, 2003.
  4. "Introduction to Parallel Processing - Algorithms and Architectures", B. Parhami, Kluwer Academic Publishers, 2002
  5. Applications of Parallel Computers, U.C. Berkeley CS267 (https://inst.eecs.berkeley.edu/~cs267/archives.html)

1.1 Zašto paralelno računanje?

- riješiti problem u manje vremena nego što bi zahtijevalo slijedno računanje
- poboljšanje performansi, povećavanje propusnosti, bolja interaktivnost
- pitanje: "Hoće li računala ikada postati *dovoljno brza*?" - nastajanjem novih problema nastaju novi zahtjevi za većom računalnom moći (dakle, "Ne!")
- neki zahtjevni računalni problemi:
  - **modeliranje i simulacija** – rad na principu slijedne aproksimacije; više računanja, veća preciznost (modeliranje klimatskih utjecaja, simulacija seizmičkih pojava, turbulencija fluida, avio i auto industrija, farmacija i medicina, simulacija elektroničkih sustava, podrška upravljanju i odlučivanju …)
  - **obrada velikih količina podataka** (računalni vid, obrada multimedijalnih podataka, *real-time video servers*, pristup velikim bazama podataka, pretraživanje, *data mining*, duboko učenje, bioinformatika…)
  - razne primjene (dekodiranje DNA, utjecaj zagađenja okoliša, kvantna dinamika, komercijalne i zabavne aplikacije…)

1.1.1 Rast računalne moći

- 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 računarstvu: iskorištavanje potencijala "sveprisutne" računalne moći (*Grid, cloud*)
- povećanje brzine komunikacije među računalima (Gb/s i više za lokalne mreže)
- paralelizacija na razini sklopovlja:
  - cjevovodi (*pipelining*)
  - Hyperthreading
  - Dual/Quad/*n*/? Core procesori
  - GP/GPU (*General-Purpose computation on GPUs*)
  - asinkrono rukovanje memorijom (jezgra - procesor - radna memorija)
- što je potrebno za paralelno računanje:
  - arhitektura više procesora (ili računala)
  - veza između procesora (ili mreža)
  - okolina za paralelni rad (odgovarajući OS, alat paralelnog programiranja)
  - paralelni algoritam i program

1.2 Modeli paralelnih računala

- zašto je potreban model: zbog omogućavanja 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 (Flynnova taksonomija):
  - SISD (Single Instruction, Single Data Stream)
  - SIMD (Single Instruction, Multiple Data Stream)
  - MISD (Multiple Instruction, Single Data Stream)
  - MIMD (Multiple Instruction, Multiple Data Stream)

1.2.1 SISD model

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

1.2.2 SIMD model

- jedna instrukcija obrađuje više podataka istodobno
- polje procesora koje izvode *jednake* instrukcije na različitim podacima
- procesori rade sinkronizirano (*lock-step* način)
- primjeri: Cray, Fujitsu VP, NEC SX-2, Maspar MP-1, MP-2
- nedostatak: grananja unutar petlji moraju se primijeniti na sve procesore
- primjene ograničene na probleme jednolike obrade velikog skupa podataka (obrada slike, numeričke simulacije…)
- moderni procesori uključuju posebne instrukcije za vektorske operande
- danas: najbliža usporedba s GPU procesorima

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
  - teže za programirati
- primjeri: Cray 2, Intel Paragon, nCUBE (*hypercube* struktura), IBM SP2, …

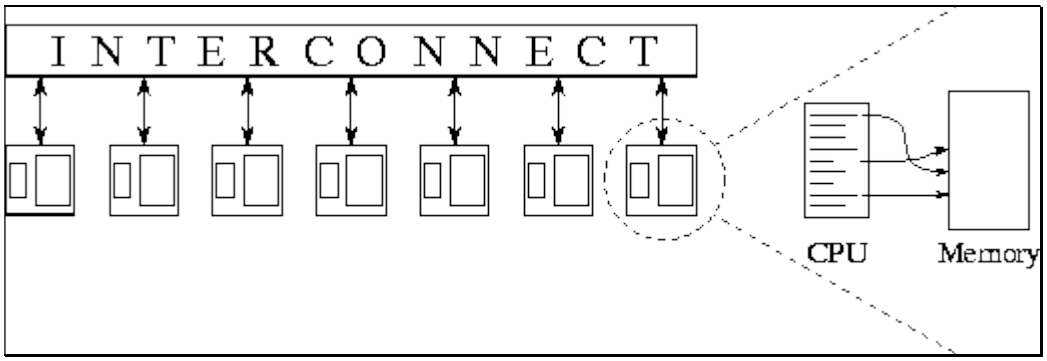
1.2.4 Podjela modela po memorijskoj strukturi

- **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 istovremeno pristupa podacima u memoriji (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
- primjeri: SMP (*symmetric multiprocessing*): višejezgreno (višeprocesorsko) računalo
- **Model raspodijeljene memorije** (*distributed memory*): više neovisnih procesora s vlastitim spremnicima
  - podjela podataka i sinkronizacija odvija se porukama (*message passing*)

- 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 dijeljenje podataka
- **NUMA model** (*non uniform memory access*) – funkcionalno se može promatrati kao skup procesora (ili računala) s više memorijskih jedinica, ali zajedničkim adresnim prostorom
- svaki procesor ima "vlastitu" memoriju, ali može pristupiti memoriji bilo kojeg procesora → različito trajanje pristupa različitim memorijskim lokacijama
  - može se primijeniti i na današnje procesore s više jezgri i višerazinskom priručnom memorijom (*cache*)

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 [2]

- u današnjoj primjeni često se susreću dva oblika višekorisničkih paralelnih računala:
- **Grozd računala** (*cluster*) – skup računala povezan lokalnom mrežom
  - značajke: manji broj računala (~10^3 procesora) i veća brzina komunikacije
  - najsličniji modelu multiračunala
- **Splet računala** (*grid*) – infrastruktura koja omogućuje pristup računalnim resursima na (u budućnosti) svakom mjestu
  - značajke: veći broj računala koji podržavaju splet, različita brzina komunikacije (u prosjeku manja)

1.3 Modeli (paradigme) paralelnih programa

- potreban je dogovor oko strukture paralelnih algoritama koji se razvijaju
- najčešće paradigme paralelnog programiranja (MIMD model)

1.3.1 Razmjena poruka

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

1.3.2 Podatkovni paralelizam

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

1.3.3 Zajednička memorija

- svi zadaci dijele isti memorijski spremnik; višeprocesni ili višedretveni model (primjeri: POSIX, OpenMP)
- čitanje i pisanje je asinkrono (razlika od računalnog modela!)
- moguća uporaba (pseudo-) nedeterminističkih algoritama
- potrebni eksplicitni mehanizmi zaštite memorije (mutex, 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

1.4 Svojstva paralelnih algoritama

- definiramo poželjna svojstva koja bi paralelni algoritmi trebali imati:
  - **istodobnost** (*concurrency*) - mogućnost izvođenja više radnji istovremeno – nužno za postojanje paralelnog 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 lokalne ili priručne memorije (*cache*)
  - **modularnost** (*modularity*) – mogućnost uporabe dijelova algoritma unutar različitih paralelnih programa
- u razvoju paralelnih algoritama dolaze do izražaja još neka područja razmatranja:

1.4.1 Amdahlov zakon

- jedan od načina opisivanja učinkovitosti paralelnog algoritma
- potencijalno ubrzanje definirano je onim udjelom (p, $p \in [0,1]$) 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:

$$ubrzanje = \frac{1}{s + \frac{p}{N_P}}$$

- s - slijedni udio programa, p - paralelni udio, N_P - broj procesora

ubrzanje	*p* = 50%	*p* = 90%	*p* = 99%
N_P = 10	1.82	5.26	9.17
N_P = 100	1.98	9.17	50.25
N_P = 1000	1.99	9.91	90.99
N_P = 10000	1.99	9.99	99.02

1.4.2 Gustafsonov zakon

- Amdahlov zakon pretpostavlja da je veličina problema (količina posla) konstantna te da je udio programa koji se može paralelizirati (p) neovisan o broju procesora
- imamo li promjenjivu veličinu problema, broj procesora mijenjamo u ovisnosti o veličini
- pretpostavke Gustafsonovog zakona:
  - broj procesora se skalira proporcionalno s veličinom problema
  - trajanje slijednog dijela programa se pri tome *ne povećava*
- neka je trajanje izvođenja na *paralelnom računalu* (uz N_p procesora) $T = 1$ (bez smanjenja općenitosti) i $T = T_s + T_p = s + p$
- tada će trajanje izvođenja na jednom procesoru biti $T_1 = T_s + N_p * T_p$ (uz idealnu paralelizaciju)
- ubrzanje je:

$$\frac{T_s + N_p * T_p}{T_s + T_p} = T_s + N_p * T_p = N_p + \left(1 - N_p\right) * T_s = N_p + \left(1 - N_p\right) * s$$

1.4.3 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 završi posao
- poseban problem u raznorodnim (*heterogeneus*) računalnim sustavima

1.4.4 Zrnatost (granularity)

- neformalna definicija zrnatosti: omjer između količine računanja (lokalnog rada) i količine komunikacije (nelokalnog rada)
- **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

1.4.5 Podatkovna ovisnost

- podatkovna ovisnost (ili slijedna 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

1.4.6 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

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*)

1.6.1 Računanje broja Pi (π)

- primjer algoritma za računanje broja *Pi*: računanje omjera 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 voditelj (*master*) sakuplja rezultate od svih ostalih (radnici, *workers*)
- paralelni algoritam:

```
b_tocaka = 100000; // bilo koji veliki broj - sto veci to preciznije
b_tocaka_krug = 0;
p = broj procesa;
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 voditelj
      primi b_tocaka_krug od svih radnika;
      izracunaj PI (pomocu vlastite i zbroja svih dobivenih vrijednosti);
inace ako sam radnik
      posalji voditelju b_tocaka_krug;
```

- potencijalni problem: računala na kojima izvodimo procese različitih su brzina; na kraju proces voditelj uvijek čeka najsporijeg radnika
- potrebno je ujednačiti opterećenje: jedna moguća strategija je *skup zadataka* (*pool of tasks*)
- posao se dijeli na više manjih dijelova (broj dijelova >> broja procesora); moguće uglavnom kod trivijalno paralelnih problema
- proces voditelj:
  - inicijalizira podatke o poslovima
  - šalje pojedinačni posao na zahtjev radnika
  - sakuplja rezultate
- proces radnik:
  - dohvaća poslove od voditelja i obavlja ih
  - šalje rezultate voditelju

```
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 voditelj
      ponovi dok ima poslova
            primi rezultat od radnika;      // prvi put samo kao prijava
            posalji radniku posao;
      kraj_ponovi
      posalji svim radnicima: nema poslova;
      izracunaj PI (pomocu vlastite i zbroja svih dobivenih vrijednosti);

inace ako sam radnik
      posalji voditelju prijavu;
      primi posao;
      ponovi dok ima poslova
            ponovi b_tocaka_p puta
                  R1, R2 = dva slucajna broja [0,2r];
                  ako (tocka (R1,R2) unutar kruga)
                        b_tocaka_krug++;
            kraj_ponovi
            posalji voditelju b_tocaka_krug;
            primi posao;
      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 radnicima: početni i krajnji indeksi podmatrice i vrijednosti elemenata
- primjer posla radnika:

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

```
...
```

2 MPI – Message Passing Interface

- **Ciljevi poglavlja**: upoznati MPI standard i osnovne MPI komunikacijske funkcije
- Literatura:
  1. MPICH implementacija (http://www.mpich.org/)
  2. OpenMPI implementacija (http://www.open-mpi.org/)
  3. MPI tutorijali (http://www.mcs.anl.gov/mpi/tutorial/, https://rookiehpc.org/mpi/index.html)
  4. MPI Standard (http://www.mcs.anl.gov/mpi/standard.html, http://www.mpi-forum.org/docs/)

2.1 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, 2012: MPI 3.0, 2015: MPI 3.1, 2021: MPI 4.0
- standard definira komunikaciju porukama, odnosno razmjenu podataka među procesima
- broj procesa u MPI programu je po definiciji konstantan tijekom izvođenja (u osnovnoj inačici standarda 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
- načini komunikacije u MPI:
  1. *point-to-point* komunikacija između dva određena procesa
  2. *collective* komunikacija unutar grupe procesa
  3. *probe* funkcije za asinkronu komunikaciju
  4. *communicator* mehanizam za razvoj modularnih paralelnih programa (mogućnost korištenja topologije mreže)
- opisano u ovom poglavlju: prva dva mehanizma i malo od trećega; četvrti naknadno
- MPI je samo standard – za korištenje je potrebna neka MPI implementacija
- korištena implementacija: MPICH (http://www.mpich.org)

2.2 Osnovne MPI funkcije

- bilo koja tehnika paralelnog programiranja razmjenom poruka mora za svaki proces omogućiti barem sljedeće mehanizme:
  - *otkriti ukupan broj procesa*
  - *identificirati vlastiti proces u grupi procesa*
  - *poslati poruku određenom procesu*
  - *primiti poruku (od određenog procesa)*
- MPI uključuje preko 150 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"`
- dvije funkcije koje se *moraju* naći u svakom MPI programu su:

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

```
int MPI_Finalize()
```

- prva funkcija mora biti prije svake MPI komunikacije, dok se druga nalazi na kraju
- funkciji `MPI_Init` se prosljeđuju odgovarajući parametri funkcije main – specifikacija u standardu za eventalnu uporabu 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)
```

- prva funkcija upisuje ukupan broj procesa u grupi u parametar *size*, dok druga funkcija upisuje indeks procesa pozivatelja u parametar *rank* (indeksi kreću od 0)
- **grupa** procesa se u MPI standardu naziva *communicator* – određuje skup procesa koji mogu komunicirati i na koje se odnosi trenutna funkcija (većina funkcija zahtjeva komunikator kao jedan od argumenata)
- globalna grupa koja uključuje **sve** uključene procese označava se sa MPI_COMM_WORLD
- jedan proces može biti član više grupa, no u svakoj grupi proces ima posebni indeks
- korisnik može sam definirati dodatne grupe procesa (sve takve grupe su podskup početne globalne grupe)
- određivanje na koliko i kojim računalima će se program izvoditi obavlja se izvan programa (koristeći opcije dotične MPI implementacije)
- 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();
...
```

2.3 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 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 pošiljatelj zove Send, primatelj Recv; parametri su:
  - *buf* je početna adresa podataka u memoriji koji se šalju (kod procesa pošiljatelja) odnosno primaju (kod primatelja)
  - *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 za globalnu grupu)

  - u strukturi *status* će nakon primitka biti zapisani podaci o poruci (oznaka kao `status.MPI_TAG`, proces pošiljatelj kao `status.MPI_SOURCE`)
- uvjeti uspjeha komunikacije:
  - indeksi pošiljatelja i primatelja moraju odgovarati
  - mora biti naveden isti komunikator (isti kontekst komunikacije!)
  - oznake poruke moraju biti iste
  - memorijski prostor primatelja mora biti dovoljno velik
- Poopćenje *primanja* poruke:
  - od bilo kojeg pošiljatelja – navodi se `MPI_ANY_SOURCE` kao *source* parametar
  - za bilo koju oznaku poruke – navodi se `MPI_ANY_TAG` kao *tag* parametar
- s 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);
}
```

2.4 Načini komunikacije

- Pitanje: kada će određena MPI 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 naknadno):
  - 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
- blokirajući način rada je uobičajen u MPI funkcijama (*standard comm. mode*)
- blokirajuća funkcija (kao što je MPI_Send):
  - *može* čekati dok poruka ne bude isporučena (odnosno dok proces primatelj ne pozove odgovarajući MPI_Recv - sinkroni način, *synchronous mode*), ili
  - *može* kopirati poruku u međuspremnik i odmah vratiti kontrolu pozivatelju – ovisno o veličini poruke i strategiji pojedine MPI implementacije (tzv. *buffered mode*)
- navedeno ponašanje nije definirano standardom! - ovisi o MPI implementaciji
- 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

2.4.1 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 (vrijedi samo za *point-to-point* komunikaciju)
- 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

2.5 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 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 definiranom 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)
```

- funkcija Reduce skuplja podatke od svih procesa, *uključujući i root proces*, (s adrese *sendbuf*, duljine *count*) ali ujedno nad njima provodi neku operaciju (definiranu s *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.
- korisnik može definirati i implementirati vlastitu operaciju (mora biti asocijativna!)
- primjer: zbrajanje svih vrijednosti varijable *x* sa svih procesa i spremanje rezultata u varijablu *rez* na procesu 0:

```
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

```
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* (uključujući i *root* proces!)

- podebljani parametri se koriste samo na *root* procesu!
- vrsta podataka i broj elemenata koje *root* šalje *svakom* procesu određeni su sa *sendtype* i *sendcnt*; *recvtype* i *recvcnt* određuju vrstu i količinu podataka koje prima pojedini proces (*sendcnt* i *recvcnt* moraju biti jednaki!)
- funkcija Gather prikuplja određeni dio niza od svih procesa (pohranjen u *sendbuf* na svim procesima) i sprema ih u *recvbuf* na *root* procesu (***)

2.5.1 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 poruka i konteksta (komunikatora), no ako je potrebno može se upotrijebiti i navedena funkcija

2.6 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
- 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
- **Primjer 3** (izvodi se na tri procesa, 0, 1 i 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, 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 (***)
- program koji računa samo na jedan od mogućih ishoda je neispravan

2.7 Neblokirajuća komunikacija

- sinkronizacija među procesima obično uključuje čekanje velikog broja procesa – ukoliko želimo maksimalno iskoristiti vrijeme, koristi se neblokirajuća komunikacija, a pozivaju se neblokirajuće (*non-blocking*) funkcije
- postupak neblokirajuće komunikacije:
  - pozivanje neblokirajuće funkcije
  - obavljanje posla koji *ne* uključuje podatke iz neblokirajućeg 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 (povratak iz neblokirajuće funkcije smatra se trenutnim, a uvjet završetka provjerava se naknadno)
- dvije obično korištene funkcije za provjeru završetka neblokirajuće operacije:

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

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

- funkcija Wait ne vraća kontrolu pozivatelju dok se ne završi radnja identificirana parametrom *request*
- funkcija Test vraća kontrolu odmah i upisuje vrijednost TRUE ili FALSE u parametar *flag*, ovisno o tome je li odgovarajuća neblokirajuća funkcija završila
- *završetak* neblokirajuće funkcije (ne *povratak* iz 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 uz pomoć 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,…)
ponavljaj
{     radi nesto sto ne ukljucuje x
      MPI_Test (request, flag, status)
} dok flag!=TRUE
radi nesto sto ukljucuje x
```

2.8 Asinkrona komunikacija u MPI modelu

- kako ostvariti da zadatak provjerava postoje li zahtjevi, tj. poruke upućene njemu?
- 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 informacije o poruci, izvor i oznaka poruke mogu se pročitati iz parametra *status* (status.MPI_SOURCE i status.MPI_TAG)
- poruka se tada može primiti pozivom funkcije MPI_Recv

2.9 Dodatne funkcije

- objašnjenja svih 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)
- detekcija pogrešaka:
  - MPI_Errhandler_set – definiranje ponašanja prilikom pojave greške
  - MPI_Errhandler_create – registriranje korisničke funkcije obrade pogreške

2.10 Primjer: računanje broja Pi

- različit algoritam od primjera iz poglavlja 1.6, ali također trivijalno paralelan
- 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 točnost rješenja)

```
#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();
}
```

- pitanje: kako realizirati Bcast i Reduce uz pomoć Send i Recv? koja je složenost tih operacija?

3 Sinkrono paralelno računalo sa zajedničkom memorijom

- **Ciljevi poglavlja**: upoznati PRAM model paralelnog računala
- definirati neke primjere algoritama i pokazati primjene PRAM modela
- Literatura:
  1. "Synthesis of Parallel Algorithms", J. Reif (ed.), Morgan Kaufmann, 1993.
  2. "Prefix Sums and Their Applications", Guy E. Blelloch (http://www.cs.cmu.edu/~blelloch/papers/Ble93.pdf)
  3. "Introduction to Parallel Computing", A. Grama, A. Gupta, G. Karypis, V. Kumar, Addison-Wesley, 2003.

3.1 Model računala PRAM

- najčešće korišteni model *slijednog računala* je najvjerojatnije RAM (*Random Access Machine*)
- svojstva RAM-a: konstantni broj registara i neograničen broj memorijskih lokacija (adrese su prirodni brojevi)
- u ocjeni 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 **sinkrono** računalo: svi procesori izvode jednu (bilo koju različitu) instrukciju u jednoj jedinici vremena (*lock step*)
- često se u razvoju algoritma pretpostavlja i inačica sa zadanim brojem procesora
- skupu osnovnih pseudokodnih instrukcija dodjemo ključnu riječ "**paralelno**" koja označava istovremeno odvijanje na potrebnom broju procesora (koliko? po potrebi - ovisi o konkretnom slučaju)
- primjer PRAM algoritma:

```
...
paralelno (za svaki element vektora, i)
      Vektor[i]++;
kraj_paralelno
...
```

- 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*)
- u slučaju CW (*concurrent write*) pristupa, samo jedan (slučajni) procesor 'uspijeva' u pisanju
- inačice: EREW PRAM, CRCW PRAM, CREW PRAM, ERCW PRAM (sve kombinacije)
- postoji velik skup algoritama za PRAM model koji služe kao komponente (*building blocks*) za izgradnju složenijih paralelnih algoritama
- izgradnja algoritama za PRAM: pretežno teorijska i akademska disciplina

3.1.1 Primjeri PRAM algoritama

- za PRAM model računala razvijeno je puno algoritama za razne namjene:
- **Algoritmi manipulacije grafovima:**
  - određivanje povezanosti grafa (najveći dio/dijelovi grafa koji su međusobno povezani)

  - *list ranking* problem (određivanje udaljenosti čvorova liste od nekog početnog čvora)
  - Eulerov put u grafu (za određivanje dubine čvorova, broj pod-čvorova, određivanje nadređenog čvora itd.)
  - obavljanje paralelnih operacija nad slabo popunjenim matricama, gdje je povezanost elemenata matrice opisana grafom (*sparsity graph*)
  - ispitivanje planarnosti grafa, bi- i tri-povezanosti, raspoređivanje zadataka predstavljenih grafom, pronalaženje najnižeg zajedničkog čvora roditelja u stablu, brza Fourierova transformacija (*FFT*) itd.
- **Algoritmi paralelnog sortiranja i računalne geometrije**
- **Paralelni algebarski algoritmi** (aritmetika polinoma, matrične operacije, rješavanje sustava jednadžbi...)
- **Paralelni algoritmi kombinatorne optimizacije** (problemi iz domene operacijskih istraživanja)
- veliki broj ovih algoritama postoji u nekoliko inačica, s obzirom na konkretni model PRAM računala: EREW, CREW, CRCW...
- za svaki od algoritama određuje se složenost i eventualni dokaz *optimalnosti* algoritma
- velika većina algoritama za složenije probleme temelji se na uporabi rješenja jednostavnijih problema (kao što je algoritam zbroja prefiksa)

3.2 Algoritam zbroja prefiksa

- jedan od najčešće korištenih algoritama-komponenata je zbroj prefiksa (*all-prefix-sums*)
- operacija zbroja prefiksa još se naziva i operacija *scan*
- iako u nazivu ima riječ "zbroj", u općenitom slučaju ne mora biti operator zbrajanja nego bilo koji asocijativni binarni operator
- **Definicija problema**: zadan je binarni asocijativni operator $\oplus$ i uređeni skup (niz) od *n* elemenata:

$$[a_0,\ a_1,\ \ldots\ ,\ a_{n-1}]$$

- operacija zbroja prefiksa treba vratiti uređeni skup:

$$[a_0,\ (a_0 \oplus a_1),\ \ldots\ ,\ (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

- npr. ako operator $\oplus$ predstavlja zbrajanje, tada bi operacija za ulazni vektor

$$[3 \quad 1 \quad 7 \quad 0 \quad 4 \quad 1 \quad 6 \quad 3]$$

  dala rezultat

$$[3 \quad 4 \quad 11 \quad 11 \quad 15 \quad 16 \quad 22 \quad 25]$$

- *napomena:* iako je zbrajanje i komutativan operator, definicija algoritma se odnosi samo na svojstvo asocijativnosti (operator je proizvoljan i ne mora biti komutativan!)
- primjeri uporabe ove operacije:
  - leksička usporedba nizova znakova (npr. određivanje da niz "strategija" dolazi ispred niza "strateški" u rječniku)
  - paralelno ostvarenje *radix-sort* i *quick-sort* algoritama
  - određivanje vidljivosti točaka u 3D krajoliku – uz operator *max*
  - zbrajanje s brojevima višestruke (proizvoljne) preciznosti
  - alokacija procesora/memorije
  - pretraživanje regularnih izraza
  - izvedba nekih operacija nad stablima (npr. dubina svakog čvora u stablu), itd.
- pretpostavka: elementi niza zapisani su u memoriji slijedno (vektor) ili kao povezana lista
- slijedno rješenje problema:

```
algoritam zbroj_prefiksa(Ulaz[], Izlaz[], n) // n je duljina nizova
i = 0;
zbroj = Ulaz[0];
```

```
Izlaz[0] = zbroj;
ponovi dok (i < n)
      i++;
      zbroj += Ulaz[i];
      Izlaz[i] = zbroj;
kraj_ponovi
```

- vremenska složenost slijednog algoritma je $O(n)$
- Definiramo pomoćni algoritam: *prescan*
- operacija *prescan* prima iste ulazne vrijednosti, a vraća niz:

$$[I,\ a_0,\ (a_0 \oplus a_1),\ \ldots,\ (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$$

  gdje je I neutralni element operatora $\oplus$ (npr. "0" za zbrajanje)
- *scan* izlaz se od *prescan* izlaza može dobiti tako da se napravi pomak za jedno mjesto u lijevo, a na kraj se doda zbroj zadnjeg elementa rezultata sa zadnjim elementom ulaznog niza
- Definiramo još jednu pomoćnu operaciju: **reduciranje** (*reduce*)
- reduciranje za iste ulazne vrijednosti vraća vrijednost ($a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$), odnosno samo zadnji element *scan* operacije (zbroj svih elemenata vektora ako je operator zbrajanje)
- postupak: prvo definiramo paralelni algoritam za reduciranje, a pomoću njega za *prescan* odnosno *scan* za EREW PRAM model

3.2.1 Algoritam +_reduciranje

- operacija reduciranja uz operator zbrajanja (EREW PRAM)
- postupak reduciranja možemo implementirati pomoću binarnog stabla: najdublji čvorovi stabla su elementi ulaznog vektora, a ostali čvorovi su zbrojevi dva čvora ispod njih (*nacrtati primjer kao stablo!*)
- postupak je moguć zbog pretpostavke asocijativnosti operatora!
- algoritam prima ulazni niz A[] duljine *n*

```
algoritam +_reduciranje(A[], n)
ponovi za d = 0 do (log n)-1
      paralelno (za svaki i od 0 do n-1 korak 2^(d+1) )
            A[i+2^(d+1)-1] += A[i+2^(d)-1];
      kraj_paralelno
kraj_ponovi
```

- dubina stabla je $\lceil \log n \rceil$, pa je složenost $O(\log n)$ uz najmanje $n/2$ procesora (u ocjeni složenosti za sve logaritme pretpostavljamo bazu 2 ako nije drugačije rečeno)

3.2.2 Algoritam +_prescan

- međuvrijednosti u ulaznom vektoru (čvorovi stabla) mogu se iskoristiti za dobijanje *prescan* rezultata
- postupak:
  1. krećemo od korijena stabla u koga upisujemo neutralni element (0 za zbrajanje)
  2. svaki čvor radi sljedeće: u desni podlist upisuje zbroj (ili zadanu binarnu operaciju) svoje vrijednosti i vrijednosti lijevog podlista, a zatim u lijevi podlist upisuje svoju vrijednost
- proceduru nazivamo *down_sweep* (nacrtati primjer na stablu!)

```
algoritam down_sweep(A[], n)
A[n-1] = 0;  // neutralni elm.
ponovi za d = (log n)-1 do 0 korak -1
      paralelno (za svaki i od 0 do n-1 korak 2^(d+1) )
```

```
            temp = A[i+2^(d)-1];      // spremi vrijednost lijevoga
            A[i+2^(d)-1] = A[i+2^(d+1)-1]; // postavi lijevog
            A[i+2^(d+1)-1] = A[i+2^(d+1)-1] + temp; // postavi desnog (isto mjesto u
memoriji!)
      kraj_paralelno
kraj_ponovi
```

- po završetku procedure u polju A[] nalazi se rješenje *prescan* operacije
- ukupni algoritam: +_prescan je +_reduciranje te down_sweep
- trajanje ukupnog postupka (*+_reduciranje* i *down_sweep*) je 2log *n*, pa je složenost također *O*(log *n*), uz *n*/2 raspoloživih procesora
- za CREW PRAM postupak je moguće obaviti u samo log *n* koraka, također *O*(log *n*)
- **MPI podrška**: funkcije MPI_Reduce, MPI_Allreduce, MPI_Scan, MPI_Exscan

3.2.3 Algoritam uz zadani broj procesora

- **Reduciranje:** imamo li fiksni broj procesora *p*, *p*<*n*/2, tada svaki procesor zbraja *n*/*p* elemenata, a zatim se rezultati (polje duljine *p*) prosljeđuju u postojeći algoritam
- algoritam *+_reduciranje_p* radi na ulaznom nizu A[] duljine *n* uz broj procesora *p*

```
algoritam +_reduciranje_p(A[], n, psuma[], p)
paralelno (za svaki procesor i, i = 0,...,p-1)
      elm = n/p;          // za koliko elemenata niza je procesor zaduzen
      poc = (n/p)*i;      // indeks pocetnog elementa procesora p
      if(n%p != 0)        // ako n nije djeljivo sa p
      {     if(i < n%p)  elm++;
            poc += min(i, n%p);
      }
      psuma[i] = A[poc];
      za j = 1 do elm - 1       // slijedno zbrajanje
            psuma[i] += A[poc + j];
kraj_paralelno
+_reduciranje(psuma[],p); // poziv potprograma
```

- rezultat algoritma je u polju psuma[] duljine *p*
- zbrajanje na jednom procesoru traje $\lceil n/p \rceil$, pa je ukupna složenost *O*(*n*/*p* + log *p*)
- **Prescan:** ukoliko imamo fiksni broj procesora *p*, *p*<*n*/2, tada svaki procesor prvo zbraja svojih *n*/*p* elemenata, potom se odvija *+_prescan* na polju duljine *p* elemenata, a potom svaki procesor izvodi slijedni *prescan* na svojem dijelu niza

```
algoritam +_prescan_p(A[], n, p)
+_reduciranje_p(A[], n, psuma[], p);  // dobivamo +_reducirano polje psuma[]
down_sweep(psuma[], p);               // dobivamo prescan polja psuma[]
paralelno (za svaki procesor i, i = 0,...,p-1)
      elm = n/p;          // za koliko elemenata niza je procesor zaduzen
      poc = (n/p)*i;      // indeks pocetnog elementa procesora p
      if(n%p != 0)        // ako n nije djeljivo sa p
      {     if(i < n%p)  elm++;
            poc += min(i, n%p);
      }
      prethodni = A[poc];
      A[poc] = psuma[i]; // posmak (offset) za sve ostale – pocetna vrijednost
      ponovi za j = 1 do elm - 1      // slijedni prescan
            temp = A[poc + j];
            A[poc + j] = prethodni + A[poc + j - 1];
            prethodni = temp;
      kraj_ponovi
```

```
kraj_paralelno
```

- broj operacija algoritma je $2(\lceil n/p \rceil + \lceil \log p \rceil)$, pa je vremenska složenost $O(n/p + \log p)$
- **Brentovo pravilo** (*Brent's principle*): u općenitom slučaju, prilagodba paralelnog algoritma složenosti $O(\log n)$ na proizvoljnom broju procesora ne može biti učinkovitija od $O(n/p + \log p)$ na p procesora!
- analiza: što bi bilo kad bismo koristili slijedni algoritam *zbroj_prefiksa* na polju psuma[] (zbog jednostavnosti)?
  - broj operacija toga algoritma je $2n/p + p$, pa je složenost $O(n/p + p)$
  - usporedba broja operacija u ovisnosti o veličini problema i broju procesora

n	p	+_prescan_p	pojednostavljeni
100	4	54	54
1000	4	504	504
10000	4	5004	5004
1000	50	51,28771	90
1000	100	33,28771	120
1000	200	25,28771	210

3.3 Primjene algortima

- uz primjenu odgovarajućeg asocijativnog operatora (ne samo zbrajanja) algoritam se može primijeniti na puno načina

3.3.1 Najveći element u nizu

- zadan je niz (brojeva) duljine n; kako pronaći najvećega (ili najmanjega)?
- slijedno algoritam mora proći sve elemente niza – složenost $O(n)$
- paralelna izvedba: na zadanom nizu izvedemo postupak reduciranja uz operator max() koji prima dva argumenta i vraća većega: *max-reduciranje*
- ukoliko svi trebaju 'znati' najveću vrijednost, potreban je i povratak po binarnom stablu
- složenost postupka je $O(\log n)$

3.3.2 Provjera uređenosti niza

- zadan je niz (brojeva) duljine n; kako provjeriti je li niz poredan (npr. uzlazno sortiran)?
- slijedni algoritam provjerava relaciju uzastopno dok ne naiđe na negativan rezultat – očekivana složenost je $\Theta(n/2)$
- paralelna izvedba:
  - pridijelimo procesor svakom elementu niza
  - svaki procesor provjerava je li njegov element manji ili jednak sljedećemu i rezultat zapisuje kao 1 ili 0
  - na dobivenom vektoru izvedemo *and-prescan* i provjerimo vrijednost zadnjeg elementa (zapravo je dovoljno i *and-reduciranje*)
- složenost je jednaka složenosti *prescan* (odnosno *reduce*) algoritma, $O(\log n)$

3.3.3 Alokacija procesora

- problem: za zadani skup zadataka, od kojih je svakome pridružen neki prirodni broj, svakom zadatku treba dodijeliti taj broj novih procesora
- primjeri:
  - dodjela memorijskih segmenata određene veličine (gdje počinje sljedeći segment?)
  - paralelno crtanje linija (koliko piksela za svaku liniju?)
  - grananje u *branch-and-bound* algoritmima pretraživanja (npr. program za igranje šaha s pretraživanjem u dubinu)
- npr. imamo zadan vektor zahtjeva za memorijom za tri procesa/zadatka (element vektora govori koliko memorije traži određeni proces/zadatak):

```
[4   1   3]
```

- kako odrediti početne adrese memorijskih segmenata? rješenje se dobiva *+_prescan* postupkom:

```
[0    4    5]
```

- Dodatak: pretpostavimo da želimo memoriju svakog procesa inicijalizirati sa njegovom oznakom, npr. A, B, C za tri procesa:

```
[A    A    A    A    B    C    C    C]
```

- paralelno rješenje se dobiva primjenom kopiranja zadanog elementa po binarnom stablu (logaritamska složenost)
- posebni postupak kopiranja se pokreće (paralelno) za svaki segment, tako da se kao argumenti proslijede početak i duljina segmenta

3.3.4 Ostvarenje algoritma radix-sort

- pretpostavimo da želimo sortirati *n* vrijednosti predstavljenih *jednakim brojem bitova* (ili znamenki u bilo kojoj bazi)
- radix sort uzastopno provodi operaciju *split* koja na osnovi jednog (promatranog) bita u podacima sve podatke s "0" za promatrani bit sprema ispred podataka koji imaju "1" u promatranom bitu
- kreće se od bita najmanje ka bitu najveće težine (postoji i obrnuta inačica) – na kraju postupka dobivamo sortirane podatke (*napisati primjer*)
- vremenska složenost slijedne operacije *split*: npr. uz brojeće sortiranje (*counting sort*) je $O(n)$ za binarni prikaz, $O(n + M)$ za vrijednosti u opsegu 0 do M
- vremenska složenost *slijednog* sortiranja je $O((n + k) \log_k M)$, za proizvoljnu bazu k i vrijednosti u opsegu 0 do M; za bazu 2 i za vrijednosti od 0 do n (tj. ako je raspon vrijednosti linearan s brojem elemenata niza), složenost je $O(n \log n)$
- problem je izvedba *split* operacije – izvodi se uporabom *scan* postupaka
- algoritam prima niz brojeva A[] duljine *n*, niz promatranih bitova Bitovi[] jednake duljine, a vraća novo polje Index[]-a elemenata (ne pomiču se elementi u memoriji)

```
algoritam split(A[], ,Bitovi[], Index[], n)
Dolje[] = +_prescan( not(Bitovi[]) ); // prescan na negiranim bitovima
Gore[] = (n-1) - +_prescan( unatrag(Bitovi[]) ); // prescan unatrag na polju bitova

paralelno (za svaki indeks i)
      ako ( Bitovi[i]==1 )
            Index[i] = Gore[i];
      inace
            Index[i] = Dolje[i];
kraj_paralelno
```

- algoritam se dalje izvodi za nove indekse, dok se ne pređu svi bitovi podataka
- želimo li proceduru *split* izvoditi uzastopno, polju Bitovi[] se treba pristupati neizravno, preko polja Index[] (*i*-tom elementu polja pristupa se kao "Bitovi[Index[i]]")
- složenost *paralelne* operacije *split* je jednaka složenosti *scan* algoritma $O(n/p + \log p)$
- složenost paralelnog sortiranja uz bazu 2 i vrijednosti do n je $O((n/p) \log n + \log p \log n)$

4 Asinkrono paralelno računalo sa zajedničkom memorijom

- **Ciljevi poglavlja**: upoznati asinkroni model PRAM računala i načine prilagodbe PRAM algoritama za asinkroni model
- Literatura:
    1. "Synthesis of Parallel Algorithms", J. Reif (ed.), Morgan Kaufmann, 1993.

4.1 Izmjene u PRAM modelu

- svi algoritmi za sinkroni PRAM model instrukcije na različitim procesorima obavljaju u *lock step* načinu
- algoritmi također pretpostavljaju jednaki trošak pristupa memoriji za bilo koji procesor za bilo koju memorijsku lokaciju
- uvodimo neke izmjene PRAM modela koji ga čine sličnijim 'stvarnim' računalima
- **Prva pretpostavka**: procesori su međusobno neovisni i nesinkronizirani
- svaki procesor instrukcije obavlja brzinom neovisnom o drugim procesorima
- kako izvoditi postojeće PRAM algoritme? u teoriji bi se nakon *svake* instrukcije trebala obaviti sinkronizacija svih procesora
- **Druga pretpostavka**: uvodi se pojam lokalnog i globalnog pristupa
- vrijeme pristupa udaljenoj memorijskoj lokaciji može biti puno dulje od lokalnog pristupa
- uvodi se jedinstveni parametar ***d*** – označava odnos vremena potrebnog za globalni prema vremenu potrebnom za lokalni pristup
- uzastopni pristup memoriji može se ulančavati (*pipelining*) – pretpostavka je da procesor može u svakom koraku dati novi zahtjev za pristup udaljenoj memoriji bez čekanja na završetak prethodnoga pristupa (uz pretpostavku da pristup traje *d* koraka)

4.1.1 Primjer: otkrivanje prvog čvora jednostruko vezane liste

- pretpostavka: imamo vezanu listu; svaki čvor liste pridružen je jednom PRAM procesoru
- nemamo pokazivač na početak liste te trebamo otkriti koji je čvor početni
- Postupak:
    - svaki procesor *i* upisuje na svoju (globalnu) memorijsku lokaciju M_i vrijednost 0
    - ako čvor *i* ima sljedbenika *j* (pokazivač nije null), upisuje 1 na lokaciju M_j
    - čvor *i* je glava liste akko je $M_i = 0$
- složenost algoritma je O(1) na sinkronom PRAM modelu
- ako procesori nisu sinkronizirani, bilo koji procesor u koraku provjere ne može znati da li zaista ne postoji nijedan procesor koji bi mu upisao "1" u memoriju ili je jednostavno "zaostao" u izvođenju operacije
- kako bi se odluka donijela sa sigurnošću, potrebno je sinkronizirati sve procesore (odnosno uvjeriti se da su svi obavili drugi korak algoritma)

4.1.2 Primjer: raspodjela jedne vrijednosti svim procesorima

- jedna vrijednost treba se raspodijeliti na *n* memorijskih lokacija
- PRAM algoritam: vrijednost se raspodjeljuje na principu binarnog stabla
- u svakom koraku aktivni procesori prepisuju vrijednost na još jednu lokaciju (*nacrtati*)
- uz trošak pristupa globalnoj memoriji (*d*), algoritam se izvodi u $O(d \log n)$ vremenu
- algoritam se može ubrzati ako se vrijednosti kopiraju po d-arnom stablu (svaki aktivni procesor piše *d*-1 novu vrijednost) (*primjer: binarno i d-arno stablo uz d = 1, 3*)
- ako se *d* upisa u memoriju može obaviti u vremenu $O(d)$ (zbog ulančavanja), tada cijeli postupak zahtijeva $O(d \log_d n)$ vremena – ubrzanje za faktor $\log d$

4.2 Asinkroni model PRAM računala

- **Asinkrono PRAM računalo** (aPRAM) sastoji se od *p* procesora

- svaki procesor ima lokalnu memoriju i svima je na raspolaganju zajednička globalna memorija
- svaki procesor radi neovisno o drugima (nema globalnog sata)
- svaki procesor izvodi svoj program (vlastiti niz instrukcija)
- postoje 4 vrste instrukcija:
  - *globalno čitanje*: čita sadržaj ćelije globalne u lokalnu memoriju
  - *globalno pisanje*: piše sadržaj lokalne memorijske ćelije u globalnu
  - *lokalna operacija*: obavlja bilo koju RAM operaciju – operandi i rezultat su u lokalnoj memoriji
  - *sinkronizacija*: sinkronizacija na skupu procesora S je logički korak u programu na kojemu svaki procesor čeka da svi ostali iz S dođu do istog koraka
- zbog jednostavnosti, uzimamo da skup sinkronizacije S obuhvaća sve procesore (postoje i inačice gdje je to podskup svih procesora)
- ukoliko je riječ o sinkronizaciji svih procesora, to se naziva **ogradom** (*synchronization barrier*)
- lokalni programi aPRAM-a sastoje se od niza asinkronih odsječaka odvojenih sinkronizacijskim ogradama
  - zadnja instrukcija u svakom programu je uvijek ograda!
- **Ograničenje**: procesori mogu asinkrono pristupati globalnoj memoriji, ali samo *jedan procesor* može pristupiti *istoj globalnoj memorijskoj lokaciji* unutar *istog asinkronog odsječka*
- primjer aPRAM programa prikazan je na slici (znak " * " označava lokalne operacije, vodoravna crta označava sinkronizacijsku ogradu)

	procesor 1	*procesor 2*	*procesor p*
odsječak 1	čitaj A čitaj B * piši A	* piši C	čitaj D * piši D
odsječak 2	čitaj C * piši D	čitaj A * piši B	*

Slika 4.1 Primjer aPRAM programa

4.2.1 Ocjena složenosti aPRAM operacija

- **Lokalna operacija**: zbog jednostavnosti, definira se da svaka lokalna instrukcija ima trajanje 1 vremensku jedinicu
- **Globalni pristup**: jedna operacija globalnog pristupa traje d vremenskih jedinica
- u slučaju višestrukog pristupa globalnoj memoriji, pretpostavlja se mogućnost preklapanja instrukcija
- u slučaju k uzastopnih globalnih pristupa, za utrošeno vrijeme se uzima (d+k-1) vremenska jedinica
- **Sinkronizacija**: uvodimo parametar $B = B(p)$, vrijeme potrebno za sinkronizaciju svih p procesora
- $B(p)$ je nepadajuća funkcija broja procesora; zbog jednostavnosti možemo uzeti konstantnu vrijednost za konkretni model
- pretpostavlja se sljedeći odnos: $2 \leq d \leq B \leq p$
- Koliko je ovaj model realističan?
  - uzima se u obzir asinkroni rad – program mora raditi bez obzira na brzinu nekog od procesora
  - istovremeno, pretpostavlja se približno jednako vrijeme jedne lokalne instrukcije za sve procesore

4.3 Prilagodba PRAM algoritama za asinkroni model

- motivacija: kad god je moguće, iskoristiti postojeće PRAM algoritme za aPRAM model
- najjednostavnija prilagodba: transformacija pojedinih instrukcija algoritma
- jedna PRAM instrukcija se općenito može prikazati u tri koraka: globalno čitanje, računanje i globalno pisanje
- pretpostavimo da se zadani PRAM algoritam izvodi na p procesora
- PRAM algoritam možemo simulirati s jednakim ili manjim brojem aPRAM procesora

4.3.1 Prilagodba PRAM algoritma uz jednaki broj procesora (p)

- postupak: umetanje ograde nakon svakog čitanja i pisanja
- jedna EREW PRAM instrukcija na aPRAM računalu:
  1. d koraka za čitanje
  2. B koraka za ogradu
  3. 1 korak za lokalnu instrukciju
  4. d koraka za zapisivanje
  5. B koraka za ogradu
- jedna PRAM instrukcija izvodi se u $2B+2d+1$ koraka, što je $O(B)$, jer je $B>d$

4.3.2 Prilagodba PRAM algoritma uz manje procesora

- trošak sinkronizacije (B) obično znatno raste sa brojem procesora koje treba sinkronizirati - zbog toga se nastoji upotrijebiti manji broj procesora, odnosno ravnoteža između vremena sinkronizacije i vremena globalnog pristupa
- neka PRAM procesori imaju indekse $0,\ldots,p-1$
- svaki aPRAM procesor (s indeksom i) simulira rad n/p PRAM procesora: $(n/p)*i$, $(n/p)*i+1,\ldots, (n/p)*(i+1)-1$
- postupak za svaki aPRAM procesor:
  1. čitanje za (n/p) PRAM procesora ($d+(n/p)-1$ korak)
  2. ograda (B koraka)
  3. lokalne instrukcije za (n/p) PRAM procesora ((n/p) koraka)
  4. pisanje za (n/p) PRAM procesora ($d+(n/p)-1$ korak)
  5. ograda (B koraka)
- ako je PRAM algoritam složenosti $O(t)$, ovako izgrađeni aPRAM imat će složenost $O((B+n/p)t)$
- za veliki broj problema ipak su razvijeni aPRAM algoritmi koji koji imaju manju složenost od ovako dobivene granice

4.3.3 Algoritam reduciranja za aPRAM model

- reduciranje na PRAM računalu ima složenost $O(\log n)$, pa se uz opisanu transformaciju dobiva APRAM algoritam složenosti $O(B \log n)$
- algoritam reduciranja (a time i sume prefiksa) može se modificirati tako da se dobije manja vremenska složenost od one dobivene opisanim općenitim postupkom
- ideja: koristiti B-arno, a ne binarno stablo
- postavke: polje elemenata nalazi se u globalnoj memoriji
- **postupak za n procesora**:
  1. svaki procesor čita i-ti element polja iz globalne memorije te inicijalizira lokalnu varijablu *suma* na tu vrijednost
  2. u petlji koja traje $\lceil \log_B n \rceil$ (odnosno $\lceil \log n \,/\, \log B \rceil$) iteracija:
     - samo jedan procesor u svakoj skupini od njih B (jedan nivo B-arnog stabla) iz globalne memorije čita vrijednosti svih ostalih $B-1$ procesora ($d+B-2$ koraka) te zbraja svoju vrijednost polja sa vrijednošću svih ostalih ($B-1$ korak)
     - dobivenu sumu zapisuje u svoj element polja u globalnoj memoriji (d koraka)
     - na kraju iteracije postavlja se ograda (B koraka)
- trajanje *jedne iteracije* je (oko) $3B+2d$ koraka, složenost $O(B)$

- složenost postupka reduciranja je $O(B \log n / \log B)$
- **Primjer:** n = 1024, B je funkcija od broja procesora $B = f(p) = \sqrt{p}$, d = 3 (npr.)
- uz broj procesora $p = n$ = 1024, parametar B = 32
- broj koraka: (3 * B + 2 * d) * (log n / log B) = 102 * 2 = 204
- iz predloženog postupka može se dobiti 'optimalan' algoritam na sljedeći način:
  - uvedimo T = $B \log n / \log B$; algoritam radi sa ***n*/T procesora**
  - svaki procesor pročita i slijedno zbroji T elemenata, potom se sinkroniziraju - trajanje je (d+T-1) + (T-1) + B', složenost O(T)
  - dobivenih n/T suma reducira se opisanim postupkom (B'-arnim stablom); broj iteracija je $\lceil \log (n / T) / \log B' \rceil = O(\log n / \log B')$, složenost jedne iteracije je $O(B')$ pa je reduciranje n/T suma opet $O(B' \log n / \log B') = O$(T)
  - poradi uporabe manjeg broja procesora (n/T) novi trošak ograde B' imat će (pretpostavljamo) manju vrijednost!
- ukupna složenost (slijedno zbrajanje, reduciranje n/T elemenata) je također $O(B \log n / \log B)$, no uporaba manje procesora troši manje vremena za sinkronizaciju ako B raste s brojem procesora
- **Primjer:** n = 1024, B je funkcija od broja procesora $B = f(p) = \sqrt{p}$, d = 3 (npr.)
- uvodimo T = $B \log n / \log B$ = 64; novi broj procesora $p = n$/T = 16; *novi* parametar B' = 4
- broj koraka: (d+T-1) + (T-1) + B + (3*B' + 2*d) * (log (n/T) / log B') = 133 + 36 = 169

4.4 Završne primjedbe

- osim opisanih, postoje još neki načini prilagodbe PRAM algoritama za aPRAM model (*rescheduling*, *Brent's principle*, itd.)
- dodatnu složenost u razvoj algoritama unose izmjene u aPRAM inačicama:
  - dopuštanje sinkronizacije samo dijela od ukupnog broja procesora
  - različito vrijeme izvođenja lokalnih instrukcija za pojedine procesore (*Variable Speed aPRAM*)
  - nepredvidivi zastoji neograničenog trajanja u radu procesora
- predstavljeni su i stvarni računalni modeli koji mogu simulirati rad PRAM računala s određenom dodatnom vremenskom složenošću (npr. $O(\log p)$ za svaku instrukciju) te uz određenu vjerojatnost pridržavanja te ograde

5 Postupak oblikovanja paralelnih algoritama

- **Ciljevi poglavlja**: proučiti općenitu metodologiju razvoja paralelnih algoritama
- Literatura:
  1. "Designing and Building Parallel Programs", I. Foster, Addison-Wesley, 1995. (*online*) (http://www.mcs.anl.gov/dbpp/)
  2. "Introduction to Parallel Computing", A. Grama, A. Gupta, G. Karypis, V. Kumar, Addison-Wesley, 2003.

5.1 Računalni i programski model

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

5.1.1 Računalni model

- *Multiračunalo* - više neovisnih računala povezanih nekom vrstom mreže (poglavlje 1.2)
- 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

5.1.2 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 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 utječe na funkcionalnost programa (jedan ili više zadataka po procesoru)
- opisani programski model može se primijeniti na više modela paralelnih računala

5.2 Primjeri paralelnih programa

- u cilju ilustracije pojedinih faza razvoja paralelnog algoritma, opisani su neki jednostavni primjeri paralelnih programa
- za sada nam nije bitna realizacija navedenih primjera

5.2.1 Metoda konačnih razlika (*finite differences*)

- zadan je jednodimenzijski vektor *X* veličine *N*
- za svaki element vektora potrebno je izračunati *T* iteracija definiranih sa:

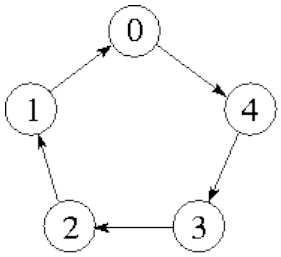
$$X_i^{(k+1)} = \frac{X_{i-1}^{(k)} + 2X_i^{(k)} + X_{i+1}^{(k)}}{4}$$

- svaki element mijenja vrijednost za iteraciju t+1 tek nakon što se njegovi susjedi promijene u iteraciji t
- paralelni algoritam: svakom elementu dodijeljen je jedan zadatak
- u svakoj iteraciji zadatak:
  - šalje svoju vrijednost (koja odgovara iteraciji t) lijevom i desnom susjedu

    - prima vrijednosti (koje odgovaraju iteraciji t) od lijevog i desnog susjeda
    - računa svoju novu vrijednost
- sinkronizacija je potrebna nakon računanja nove vrijednosti, odnosno prilikom primanja vrijednosti od susjeda
- budući da se operacija primanja vrijednosti definira kao sinkrona (blokirajuća), nije potrebno dodavati posebnu ogradu

5.2.2 Uparena međudjelovanja (*pairwise interactions*)

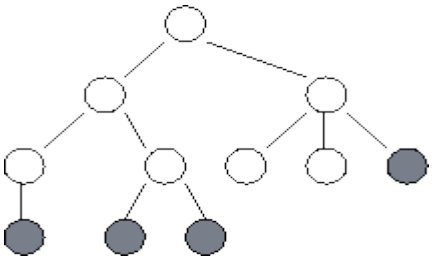
- također je zadan niz elemenata, ali se u svakoj iteraciji treba računati sa vrijednostima svih elemenata, a ne samo sa susjednima
- ukupno N(N-1) operacija u svakoj iteraciji (svaki sa svakim)
- operacije su često simetrične, tj. neovisne o redoslijedu dva elementa (komutativnost)
- paralelni algoritam: ponovno dodijeljen jedan zadatak svakom elementu
- Kako dobiti podatke od svih ostalih zadataka? – najjednostavniji način je pomoću N(N-1) kanala (svaki sa svakim)
- složenost jedne iteracije je N-1 (ukupan broj operacija primanja)
- uz asocijativnost, jednostavnija komunikacijska struktura - kanali uređeni u prsten:



- svaki zadatak ima svoju lokalnu vrijednost (npr. V) i pomoćnu varijablu (npr. B)
- na početku iteracije svaki zadatak postavlja pomoćnu varijablu na trenutnu vrijednost (B=V) a zatim u petlji:
    - šalje vrijednost B na izlaz (sljedećem elementu polja)
    - prima vrijednost sa ulaza (od prethodnog elementa) u B
    - računa međurezultat operacije sa B i sprema ga u V
- petlja se ponavlja N-1 put, čime završava jedna iteracija
- jednaka složenost ali jednostavnija komunikacija (manje kanala i manje globalnih pristupa)

5.2.3 Pretraživanje stabla

- pretraživanje stabla za određenim čvorom koji predstavlja rješenje (npr. u igrama; rješenje je zapravo put od vrha stabla do završnog čvora)
- paralelni algoritam: u početku jedan zadatak pridružen je vrhu stabla
- svaki zadatak radi sljedeće: ako čvor nije rješenje, **stvara** *n* novih zadataka za svaki čvor dijete i poziva ih rekurzivno



5.2.4 Trivijalno paralelni problemi

- u općenitom slučaju: zadaci se mogu izvoditi istovremeno i bez potrebne komunikacije
- primjer: rješavanje jednog te istog problema (slijedno) uz različite parametre (svaki zadatak rješava neki problem uz različit skup parametara) za određivanje 'optimalnih' parametara
- u svrhu ujednačavanja opterećenja moguća je sljedeća organizacija:

    - jedan zadatak izvor odgovara na zahtjeve 'radnika' šaljući im potrebne parametre
    - zadaci 'radnici' primaju vrijednosti parametara od izvora a rezultate šalju zadatku za izlaz
    - jedan izlazni zadatak prima rezultate od svih radnika
- izvođenje je u principu nedeterministički (ne znamo redoslijed izvođenja pojedinog skupa parametara), ali to ne utječe na rezultate izračunavanja

5.3 Faze oblikovanja paralelnog algoritma

- razvoj (paralelnog) algoritma nije moguće svesti na recept, ali je moguće koristiti metodički pristup za lakše otkrivanje nedostataka
- 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:
    1. **Podjela** (*partitioning*) – dekompozicija problema na manje cjeline (zanemaruje se broj procesora i memorijska struktura fizičkog računala)
    2. **Komunikacija** (*communication*) – određivanje potrebne komunikacije među zadacima
    3. **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)
    4. **Pridruživanje** (*mapping*) – svaki zadatak se dodjeljuje 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

5.4 Podjela

- **cilj**: definirati sitnozrnatu (*fine-grained*) podjelu posla na dijelove koji se mogu (mada u konačnoj inačici algoritma ne moraju) izvoditi istodobno
- podjela posla može se koncentrirati na podjelu računanja i/ili podjelu podataka uključenih u računanje
- ideja: postići podjelu koja ne zahtijeva dupliciranje podataka ili računanja (zasada)

5.4.1 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)

5.4.2 Podjela izračunavanja (*functional decomposition*)

- prvo se određuje 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.)

5.4.3 Pitanja za provjeru - podjela

- pitanja koja bi trebala upozoriti na neke moguće nedostatke podjele posla

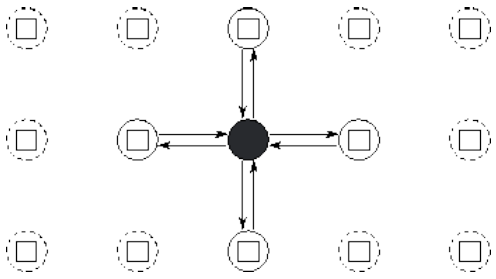
  - *Definira li podjela broj zadataka barem za red veličine veći od broja procesora na paralelnom računalu?* -- ako ne, mala je mogućnost prilagodbe u kasnijim fazama
  - *Izbjegava li podjela višestruko računanje i umnožavanje istih podataka?* -- ako ne, algoritam bi mogao biti osjetljiv na povećavanje opsega posla
  - *Jesu li zadaci podjednake veličine?* -- ako ne, moguća su neujednačena opterećenja procesora
  - *Povećava li se broj zadataka sa veličinom problema?* – postupak podjele bi trebao za veći problem definirati više zadataka, a ne jednak broj većih zadataka (slaba skalabilnost)
  - *Postoje li alternativne podjele?* -- za svaki slučaj, zbog veće fleksibilnosti u sljedećim fazama
- ako odgovori na neka od ovih pitanja nisu zadovoljavajući, možda je potrebno istražiti neke druge mogućnosti:
  - može li se problem opisati drukčijim (slijednim) algoritmom-modelom koji daje druge mogućnosti paralelizacije?
- 'dobar' slijedni algoritam može biti 'loš' u paralelnom izvođenju, a 'dobar' paralelni algoritam može biti 'loš' u slijednom izvođenju

5.5 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 nego kod podjele izračunavanja
- cjelokupnu komunikaciju dijelimo po nekoliko osnova:
  - lokalna/globalna: u lokalnoj komunikaciji svaki zadatak komunicija s 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
  - statička/dinamička: u statičkoj komunikaciji ne mijenjaju se procesi koji sudjeluju u istoj, dok se identitet zadataka u dinamičkoj komunikaciji mijenja tijekom izvođenja
  - sinkrona/asinkrona: u sinkronoj komunikaciji i pošiljatelj i primatelj zajednički (koordinirano) sudjeluju, dok u asinkronoj komunikaciji jedan zadatak može neplanski tražiti podatke od drugoga

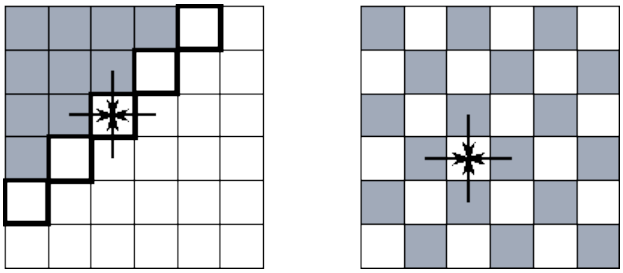
5.5.1 Lokalna komunikacija

- 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 razlika
- primjena: numeričko rješavanje slabo popunjenih (*sparse*) sustava linearnih jednadžbi
- skup susjeda koji su potrebni za računanje nove vrijednosti je *maska* (*stencil*)



- često postoji nekoliko načina izračunavanja novih vrijednosti - dvije skupine:

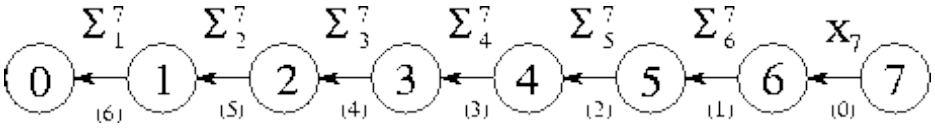
- **Jacobijeva metoda**: nova vrijednost računa se uz pomoć susjednih vrijednosti iz prethodne iteracije
  - jednostavno se paralelizira; u jednoj iteraciji svi elementi mogu se paralelno izračunati
- **Gauss-Seidel metode**: u računanju nove vrijednosti koriste se vrijednosti koje kod nekih susjeda odgovaraju *trenutnoj*, a kod nekih prethodnoj iteraciji
  - bolja konvergencija nego Jacobijeva metoda - u manje iteracija do veće preciznosti
  - upotrebljava se u slijednim algoritmima, ali teže je paralelizirati
  - obično možemo izabrati redoslijed izračunavanja novih vrijednosti po elementima
- dva primjera primjene Gauss-Seidel metode uz različito izračunavanje novih vrijednosti:
  1. elementi lijevo i gore su iz trenutne, a desno i dolje iz prethodne iteracije
  2. svi susjedni elementi moraju biti iz trenutne iteracije



- uz prvi primjer moguća su u prosjeku N/2 istodobna računanja, dok uz drugi primjer pola elemenata možemo izračunati istodobno
- izvedba lokalne komunikacije u MPI okruženju: *point-to-point* funkcije

5.5.2 Globalna komunikacija

- u globalnoj komunikaciji sudjeluje veća grupa zadataka
- **Primjer**: operacija reduciranja nad poljem duljine N s nekim binarnim operatorom
- *Prva izvedba*: svi zadaci (elementi polja) šalju svoje podatke jednom zadatku koji prima podatke i računa rezultat
- nedostaci izvedbe:
  - centralizirana – jedan zadatak mora sudjelovati u cjelokupnoj komunikaciji
  - slijedna – ne dopušta paralelno računanje ni komunikaciju
- *Druga izvedba*: raspodjeljivanje računanja i komunikacije [1]



- N-1 korak komunikacije i računanja raspodijeljeni su među svim zadacima (opaska: preslikani slijedni algoritam *+_scan*)
- nema (vremenskog) poboljšanja za jednu operaciju, no dozvoljava ulančano izvođenje!
- *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

```
Algoritam: komunikacijska struktura binarnog stabla
za(i = 0; (i < log n) && (proces_ID % 2^i == 0); i++)
{     dest_ID = proces_ID XOR 2^i;    // bitwise XOR
      ako(proces_ID % 2^(i+1) == 0)
      {     recv(drugi_podatak, dest_ID);
            operacija();              // ovisno o problemu
      }
      inače
            send(moj_podatak, dest_ID);
}
```

- MPI standard uključuje funkcije globalne komunikacije (Reduce, Bcast...)

5.5.3 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 (*poll*) 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
- asinkrona komunikacija uz MPI: *probe* funkcije (`MPI_Probe`, `MPI_Iprobe`)

5.5.4 Pitanja za provjeru - komunikacija

- neka od mogućih pitanja:
    - *Izvode li svi zadaci podjednaku količinu komunikacije?* -- ako ne, paralelni program je loše skalabilan (uz povećanje problema sve veći komunikacijski zahjevi na jednom dijelu zadataka)
    - *Komunicira li svaki zadatak s 'manjim' brojem susjeda?* -- ako ne, možda je moguće potrebe lokalne komunikacije zamijeniti globalnom komunikacijom (vidi primjer 5.2.2: Uparena međudjelovanja)
    - *Može li se komunikacija odvijati paralelno?* -- ako ne, program je neučinkovit i slabo skalabilan
    - *Može li se računanje u više različitih zadataka odvijati neovisno?* -- ako ne, možda je moguće izmijeniti redoslijed komunikacije i računanja ili promijeniti način računanja

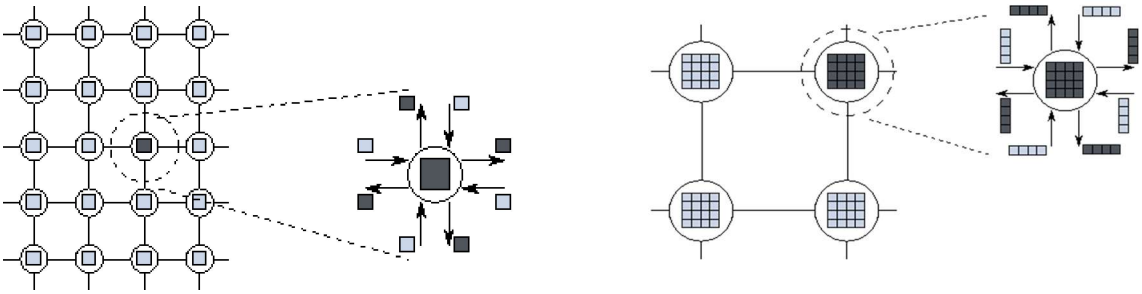
5.6 Aglomeracija

- dosadašnje faze ne uzimaju u obzir paralelno okruženje za odvijanje programa
- ciljevi aglomeracije:
    - ujednačiti troškove komunikacije i/ili računanja povećavanjem zrnatosti
    - održati prilagodljivost programa s obzirom na veličinu problema
    - smanjenje troškova implementacije s obzirom na iskoristivost postojećih algoritama
- navedeni ciljevi su često međusobno proturječni
- ukoliko je broj zadataka nakon aglomeracije i dalje veći od broja procesora, potrebno je provesti i pridruživanje (4. faza)

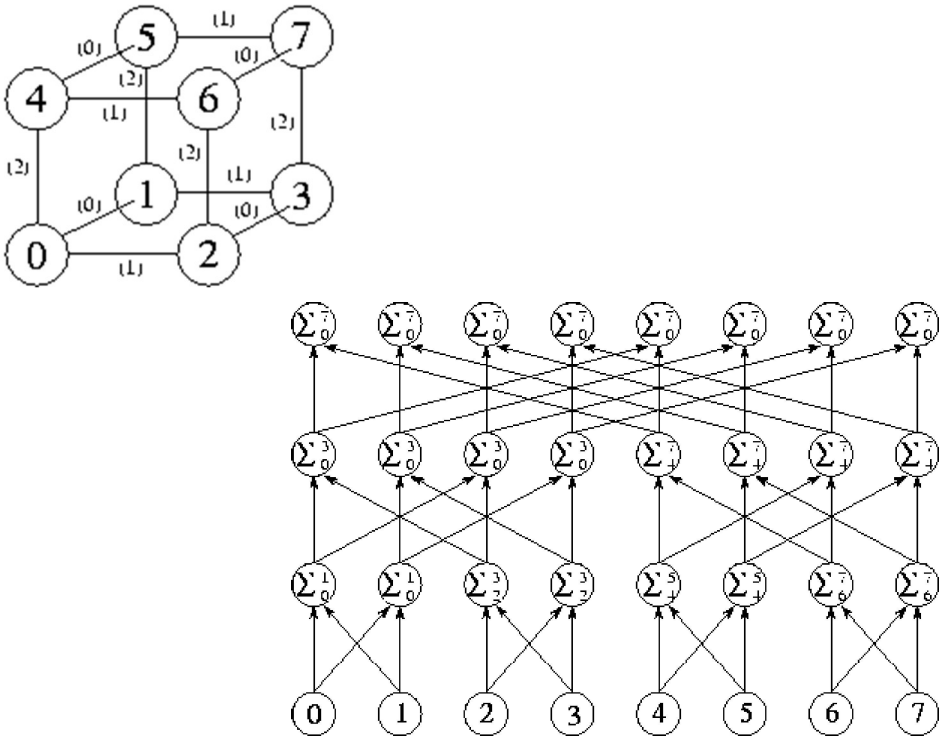
5.6.1 Povećavanje zrnatosti

- 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 poruka*

- povećavanjem zrnatosti moguć je i gubitak dijela istodobnosti, no on je isplativ ako dovoljno smanjuje komunikacijske troškove
- **Povećavanje zadataka** najbolje 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 - na slici lijevo je originalna struktura, dok su na slici desno zadaci grupirani u oblik 4x4 [1]



- **Uvišestručavanje računanja** je postupak kojime možemo *povećati* potrebe za računanjem u cilju smanjenja ukupnog vremena ili komunikacije
- **Primjer**: zbrajanje svih elemenata niza s tim da svi elementi moraju imati pohranjen rezultat (*allreduce*)
- algoritam binarnog stabla: jedan prolaz za zbrajanje i jedan prolaz za raspodjelu rezultata svim elementima (npr. 3 + 3 = 6 koraka za $n = 8$)
- prilagodba: postupak se može obaviti 'u jednom prolazu' u (log n) koraka pomoću tzv. komunikacijske strukture hiperkocke (*hypercube*), slika dolje lijevo [1] (brojevi u zagradama označavaju u kojem se koraku koristi dotični komunikacijski kanal)



- rezultat: ukupna količina računanja je povećana, ali uz smanjenje broja koraka

- ukoliko je potrebno uzastopno obaviti više takvih operacija, strutura se može koristiti ulančano (*pipelining*) u obliku 'leptira' (*butterfly*), slika gore desno (svaki redak na slici prikazuje trenutno stanje skupa zadataka, 3 koraka za $n = 8$)
- leptir struktura se u sustavu zadataka i kanala može opisati hiperkockom koja se primjenjuje u velikom broju paralelnih algoritama
- struktura hiperkocke omogućuje izvedbu algoritama koji koriste i jednostavnije komunikacijske strukture! (stablo, prsten, polje)

```
Algoritam: komunikacijska struktura hiperkocke
za(i = 0; i < log n; i++)
{     dest_ID = proces_ID XOR 2^i;     // bitwise XOR
      send(moj_podatak, dest_ID);
      recv(drugi_podatak, dest_ID);
      operacija();                     // ovisno o problemu
}
```

5.6.2 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 raspoloživih procesora omogućuje ujednačenije 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)

5.6.3 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 (kao u 5.6.1) ako to omogućuje korištenje postojećih algoritama (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 modul, ili dodati poseban modul za pretvorbu međurezultata

5.6.4 Pitanja za provjeru - aglomeracija

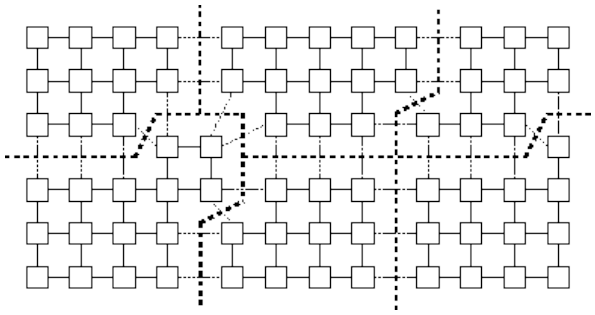
- neka od ovih pitanja podrazumijevaju kvantitativnu analizu programa, o čemu više u idućem poglavlju
    - *Ako algomeracija uvišestručuje računanje, je li to isplativo za različite veličine problema i broja procesora?*
    - *Jesu li komunikacijski i računalni troškovi dobivenih zadataka usporedivi?* -- što su zadaci veći, time je važnije da troškovi rada budu približno jednaki
    - *Da li se broj zadataka i dalje prilagođava veličini problema?* -- ako ne, program neće biti dobro prilagodljiv
    - *Može li se broj zadataka još smanjiti, bez štete po prilagodljivost ili ujednačenost?* -- program sa manjim brojem zadataka je u općenitom slučaju jednostavniji i učinkovitiji
    - *Ukoliko se mijenja postojeći slijedni algoritam, je li trošak prilagodbe postojećeg algoritma sumjerljiv dobiti od boljeg paralelnog algoritma?* – možda je drugom paralelnom izvedbom moguće iskoristiti neki postojeći slijedni algoritam (ili dio)

5.7 Pridruživanje

- cilj: odrediti na kojem računalu/procesoru će se pojedini zadatak izvoditi
- *napomena:* odnos procesor – proces – zadatak za MPI model
- 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 kompleksan
- najjednostavniji pristup: zadaci **jednoliko raspodijeljeni** po procesorima (npr. 3x3=9 zadatka iz 2D mreže zadataka po jednom procesoru)
- često je nužna uporaba algoritama **ujednačavanja opterećenja** i/ili **raspoređivanja zadataka** (*task scheduling*)

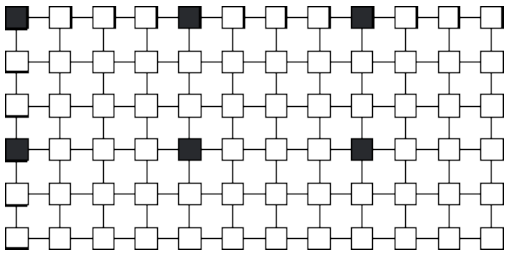
5.7.1 Ujednačavanje opterećenja

- uporaba ujednačavanja opterećenja podrazumijeva manje-više stalan broj zadataka duljeg životnog vijeka (npr. jedakog trajanju cjelokupnog paralelnog programa)
- ujednačavanje opterećenja može biti
  - statičko: zadaci se raspoređuju samo na početku rada;
  - dinamičko: ako se mijenjaju uvjeti izvođenja, ujednačavanje se pokreće nekoliko puta tijekom rada paralelnog programa
- u dinamičkom slučaju poželjno je imati *lokalne* algoritme ujednačavanja (bez potrebe dohvaćanja globalnih informacija o opterećenju)
- postoji puno različitih algoritama ujednačavanja opterećenja – uporaba bilo kojega mora se ocijeniti s obzirom na troškove (trajanje) i učinkovitost
- **Algoritmi rekurzivne bisekcije** koriste pristup "podijeli pa vladaj" pri čemu se prostor zadataka dijeli na manje dijelove po računalnim i komunikacijskim zahtjevima
- za svaki dobiveni dio podjela se nastavlja rekurzivno
- podjele se rade na nekoliko načina:
  - podjele po koordinatama (bez informacije o komunikaciji) tako da se područje uvijek dijeli po duljoj dimenziji
  - podjele po koordinatama, uzimajući u obzir računalne i/ili komunikacijske troškove među zadacima (broj kanala i predviđena količina prometa), itd. (*vidi sliku: http://www.mcs.anl.gov/dbpp/text/pictures/part.gif*)
- navedeni algoritmi zahtjevaju globalnu informaciju o sustavu – veći troškovi rada
- **Lokalni algoritmi** ujednačavaju opterećenje na osnovu malog broja informacija



- svaki procesor povremeno uspoređuje svoje opterećenje sa opterećenjem svojih susjeda te, ako je omjer opterećenja veći od definiranoga, neki zadaci se premještaju
- ovi algoritmi zahtjevaju manje računanja od globalnih, no relativno se sporo prilagođavaju brzim promjenama opterećenja (ako se opterećenje naglo poveća na jednom procesoru, potrebno je nekoliko primjena algoritma za raspodjelu opterećenja)
- **Vjerojatnosne metode** (*probabilistic methods*) raspodjeljuju zadatke po procesorima po nekom slučajnom kriteriju
- ukoliko je broj zadataka dovoljno velik (red veličine više nego broj procesora), može se postići ravnomjerna raspodjela opterećenja
- prednosti: mali troškovi postupka i velika prilagodljivost
- nedostaci: mogu prouzročiti velike komunikacijske troškove

- najučinkovitiji u slučaju male komunikacije među zadacima i kod većeg broja zadataka
- **Cikličko pridruživanje** (*cyclic mapping*) svakom procesoru (kojih ima P) dodjeljuje svaki P-ti zadatak dok se svi zadaci ne dodijele – *opaska: MPICH*



- inačica postupka može dodjeljivati cikličke *grupe* zadataka pojedinim procesorima (tj. ne samo svaki P-ti zadatak nego i neke njegove susjede, npr. grupu 2x2)

5.7.2 Raspoređivanje zadataka

- koristi se najčešće u slučajevima funkcionalne dekompozicije kada imamo više zadataka kraćeg životnog vijeka - algoritmi raspoređivanja dodjeljuju nove zadatke slobodnim procesorima
- obično se održava centralizirani ili raspodijeljeni 'bazen' zadataka
- problem: odabir načina raspodjele zadataka procesorima
- **Voditelj/radnik** model (*manager/worker*) se sastoji od jednog procesa voditelja koji na zahtjev raspodjeljuje zadatke radnicima
- učinkovitost pristupa ovisi o broju radnika i troškovima dohvata zadataka (opasnost: zagušenje voditelja)
- poboljšanja: radnici dohvaćaju sljedeći problem prije završetka rada na trenutnom zadatku (*prefetching*) kako bi se iskoristilo vrijeme potrebno za komunikaciju
- **Hijerarhijski voditelj/radnik** model koristi, osim zajedničkog voditelja, dodatnu podjelu na podgrupe zadataka s 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 procesora (svi kontaktiraju 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

5.7.3 Pitanja za provjeru - pridruživanje

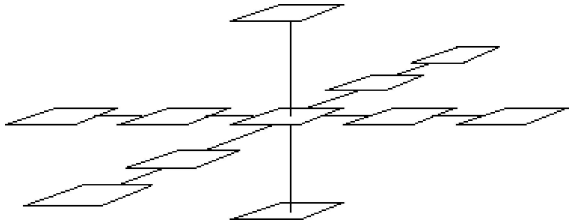
- zadatak pridruživanja je ujednačiti opterećenje procesora uz održavanje malih komunikacijskih troškova
    - *Koristimo li SPMD model (primjer: MPI), možda bi model s promjenjivim brojem zadataka bio bolji?* -- drugi pristup možda daje jednostavnije rješenje
    - *Koristimo li centralizirano raspoređivanje, jesmo li sigurni da voditelj neće biti usko grlo sustava?* -- opterećenje voditelja može se u nekim slučajevima smanjiti smanjenjem informacije o zadacima koju je potrebno slati radnicima
    - *Koristimo li dinamičko ujednačavanje opterećenja, isplati li se njegov trošak?* -- u većini slučajeva poželjno je zbog jednostavnosti isprobati vjerojatnosno ili cikličko pridruživanje
    - *Koristimo li vjerojatnosno pridruživanje, imamo li dovoljno velik broj zadataka?* -- preporuča se za red veličine veći broj zadataka od broja procesora

5.8 Primjer: atmosferski model

- primjer paralelnog programa pogodnog za domensku dekompoziciju

5.8.1 Opis problema

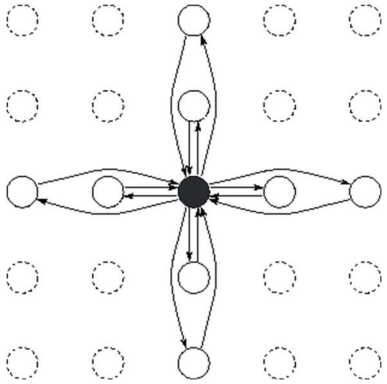
- program za simulaciju atmosferskih uvjeta (vjetar, oblaci, padaline itd.)
- koristi se uzastopno numeričko rješavanje sustava parcijalnih diferencijalnih jednadžbi (analiza prijelaznih pojava)
- prostor analize je kvadar podijeljen na diskretne dijelove (razlučivosti reda veličine 30 točaka po uspravnoj i 500 točaka po vodoravnim osima)
- pretpostavka: stanje svake točke računa se u ovisnosti o stanjima susjednih točaka
- u računanju vodoravnih dimenzija koristi se maska s 9 točaka, a za uspravnu s 3 točke:



- rad algoritma uključuje računanje dinamike fluida, osvjetljenosti, padalina itd. (*vidi sliku*: *http://www.mcs.anl.gov/dbpp/text/pictures/anderson.gif*)
- ovaj model je predstavnik skupine paralelnih aplikacija u znanstvenim istraživanjima

5.8.2 Modeliranje algoritma

- **Podjela** - domenska dekompozicija (podjela podataka)
- 3D područje promatranja se rastavlja tako da svaka točka predstavlja jedan zadatak (koristimo najveću moguću podjelu)
- broj zadataka $N_X * N_Y * N_Z$
- **Komunikacija** - određujemo vrste potrebne komunikacije
- *lokalna* komunikacija obuhvaća maske za računanje vrijednosti pojedinačnog elementa u vodoravnim (slika desno) i uspravnoj dimenziji
- *globalna* komunikacija je potrebna jer atmosferski model povremeno računa ukupnu masu u sustavu zbog kontrole pogreške (primjena zakona o očuvanju mase) - zbroj se može provesti algoritmima reduciranja, *butterfly* strukturom itd.
- računanje izlaznih varijabli modela zahtjeva dodatnu komunikaciju

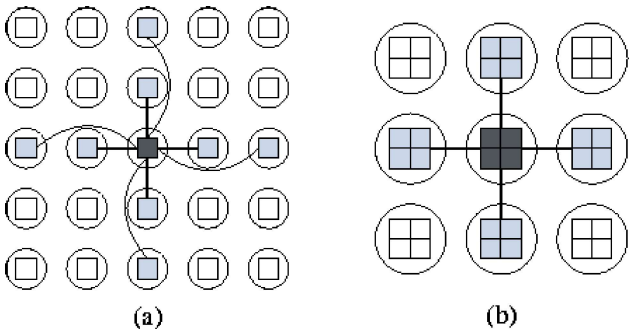


- primjer izlazne vrijednosti: stupanj vedrine neba (*total clear sky level, TCS*) definira se rekurzivno kao

$$TCS_k = \prod_{i=1}^{k}(1 - cld_i) TCS_1 = TCS_{k-1}(1 - cld_k)$$

gdje je cld_i oblačnost na visini *i*

- vrijednost se može izračunati **_prescan* algoritmom nad svim elementima u jednoj vertikali
- izlazne vrijednosti obično se računaju u jednoj vertikali (gledamo situaciju na tlu)
- najveći problem u komunikaciji moglo bi biti upravo računanje izlaznih vrijednosti
- **Aglomeracija**: broj zadataka je u početku prevelik (oko 10^6)
- nekoliko mogućnosti primjene aglomeracije
- definiranje manje grupe zadataka (npr. 2x2) smanjuje ukupan broj poruka za jednu točku u lokalnoj komunikaciji [1]:



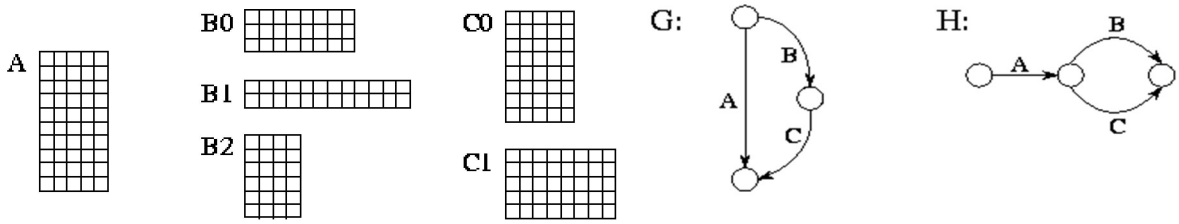
- komunikacija po uspravnoj osi je relativno velika zbog lokalnih (2 poruke po točki) i globalnih zahtjeva (računanje izlaznih vrijednosti, oko 50 poruka) - isplativo je grupirati sve zadatke *u jednoj vertikali*
- uz grupiranje po vertikali također je moguće iskoristiti postojeći slijedni algoritam za računanje izlaznih vrijednosti
- nakon primjene oba načina algomeracije, ostajemo sa $(N_X * N_Y)/4$ zadataka (oko 10^4)
- *napomena*: kvantitativnu analizu troškova provest ćemo u sljedećem poglavlju!
- **Pridruživanje:** najjednostavnije je koristiti jednoliku raspodjelu po procesorima - pretpostavka vrijedi za puno problema ove vrste
- u konkretnom slučaju, uobičajena je pojava neravnomjernog opterećenja zbog izmjene dana i noći (određene vrijednosti računaju se samo po danu) - gubici u vremenu oko 20%
- pojava se može izbjeći cikličkim pridruživanjem, u kojem slučaju treba provjeriti eventualno povećanje troškova komunikacije (http://www.mcs.anl.gov/~itf/dbpp/text/node20.html)

5.9 Primjer: optimiranje zauzeća površine

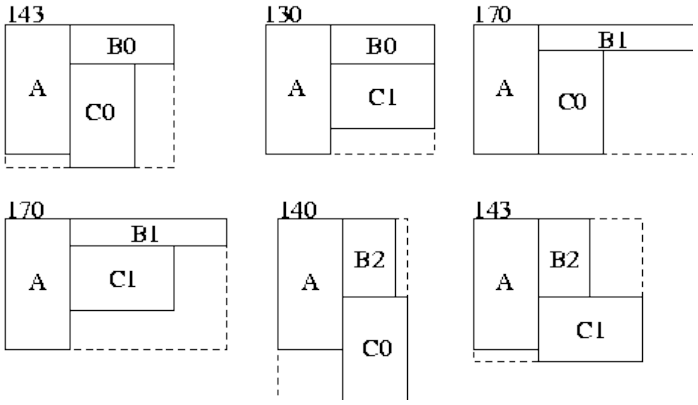
- predstavnik problema VLSI dizajna ili optimiranja zauzeća površine (*floorplan optimization*)

5.9.1 Opis problema

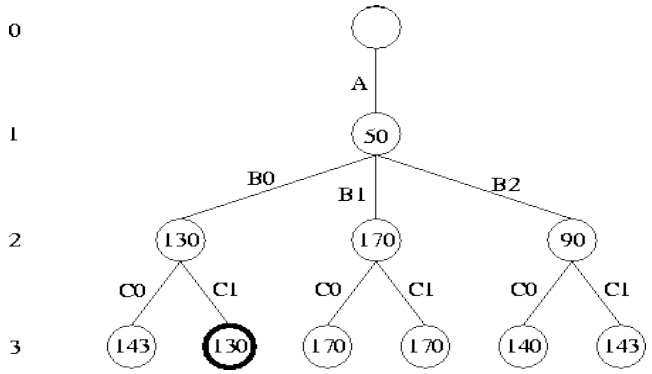
- optimiranje zauzeća površine javlja se i kao dio procesa oblikovanja mikroprocesora ili memorijskih čipova
- obično je zadan određen broj *nedjeljivih* blokova (ćelija) sa dodatnim zahtjevima za međusobni odnos u konačnom rasporedu (npr. dva određena bloka moraju biti jedan ispod drugoga ili jedan pored drugoga i sl.)
- cilj je naći najmanju moguću (pravokutnu) površinu na koju se blokovi mogu smjestiti uz zadovoljenje svih ograničenja (analogija: problem smještaja kuhinjskih elemenata)
- za svaki blok obično postoji nekoliko mogućih **izvedbi**: jedan blok može se izgraditi u nekoliko različitih (pravokutnih) oblika
- ograničenja relativnog položaja blokova predstavljena su dvama grafovima (jedan za vodoravnu, jedan za uspravnu dimenziju): lukovi grafa su blokovi, a čvorovi u grafu označavaju da dva bloka (lukovi) moraju biti spojeni



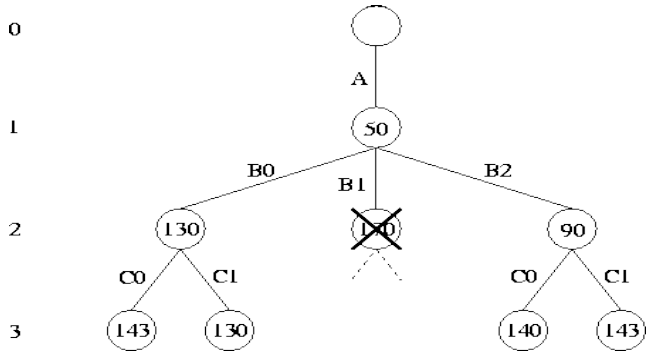
- uz dana ograničenja, ukupan mogući broj rasporeda je umnožak broja oblika svih blokova
- površina koju zauzima raspored definirana je kao umnožak najveće veličine u vodoravnoj i uspravnoj dimenziji (najmanji pravokutnik u kojega se blokovi mogu smjestiti)



- primjer: na slici je prikazano svih 6 (1*3*2) kombinacija sa pripadajućom površinom
- slijedni algoritam: rješenje se nalazi rekurzivnim algoritmom pretraživanja stabla
- u svakoj razini stabla odabiru se sve moguće izvedbe jednoga bloka; vrijednost čvora stabla je trenutno zauzeta površina (konačna vrijednost dobiva se tek u listovima stabla)



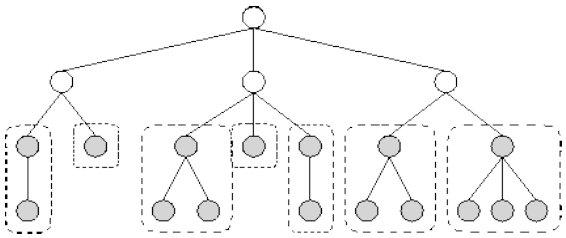
- nedostatak: vrlo veliki broj čvorova u imalo složenijim uvjetima
- prilagodba: slijedni algoritam *branch and bound* koji ne uzima u obzir podstablo ako je roditeljski čvor podstabla već lošiji od najboljeg rješenja nađenog do toga trenutka (obilazak stabla u slijednom algoritmu je u dubinu, s lijeva na desno)



5.9.2 Modeliranje algoritma

- **Podjela** algoritma je funkcionalna - dodjeljujemo zadatak svakom čvoru stabla
- problemi:
  1. tako definirani algoritam pretražuje stablo u širinu, što ne omogućava učinkovitu primjenu 'rezanja' podstabla
  2. trenutna najbolja vrijednost (npr. A_{MIN}) mora biti poznata svim zadacima
- **Komunikacija**: prijenos parametara zadacima je trivijalan - svaki čvor roditelj šalje potrebne vrijednosti čvorovima djeci
- dodatno: kako raspodijeliti A_{MIN} svim zadacima?
- jednostavni pristup: definirati jedan zadatak odgovoran za primanje i slanje najbolje vrijednosti svim ostalim zadacima

- nedostaci: slabo prilagodljivo broju zadataka - učinkovito samo ako su troškovi komunikacije mali u usporedbi sa troškovima obrade čvora i ako nema previše zadataka
- prilagodbe: provjeravati najbolju vrijednost samo u nekim trenucima, npr. samo u nekim nivoima dubine grafa (svakih 3 i sl.); raspodijeliti posao slanja A_{MIN} na nekoliko zadataka (svaki zadatak se brine o svom podstablu)
- **Aglomeracija**: potrebno je uskladiti troškove obrade čvora i troškove stvaranja novih čvorova i međusobne komunikacije
- primjer: stvaramo nove zadatke do neke zadane dubine stabla (D), a potom svako podstablo pretražujemo unutar istog zadatka, odnosno prelazimo na pretraživanje u dubinu
- promjena je također moguća nakon što ukupan broj zadataka prijeđe određenu vrijednost



- **Pridruživanje**: stablo se po *defaultu* pretražuje u širinu, što nije pogodno za 'rezanje'
- pod pretpostavkom da ćemo generirati više zadataka nego fizičkih procesora (red veličine više), raspoređivanje zadataka se može obaviti na nekoliko načina:
    - nakon dolaska do dubine D, svaki procesor preuzima jedan zadatak (podstablo) – pretpostavljamo da imamo više zadataka nego procesora – dok ga ne obavi do kraja, a nakon toga od voditelja (*manager*) zahtijeva novi
    - nakon dubine D, zadaci se podijele među procesorima po slučajno generiranom rasporedu (npr. po principu cikličkog pridruživanja). Ako neki procesor završi ranije, traži zadatke od drugih procesora (nema potrebe za voditeljem)
    - prvi čvor se dodijeli jednome radniku/procesoru; slobodni radnici zahtijevaju nove zadatke od ostalih. Procesor će unutar svoga okruženja koristiti pretraživanje u dubinu, a radnicima koji traže zadatke slat će one koji su bliže vrhu njegovog podstabla
- ovi načini raspoređivanja mogu se iskoristiti i za komunikaciju trenutno najbolje vrijednosti ostalim zadacima (za uporabu *branch and bound* algoritma)

6 Kvantitativna analiza paralelnog algoritma

- **Ciljevi poglavlja**: opisati metode kvantitativne analize paralelnog programa
- Literatura:
  1. "Designing and Building Parallel Programs", I. Foster, Addison-Wesley, 1995. (*online*) (http://www.mcs.anl.gov/dbpp/)

6.1 Definiranje performansi

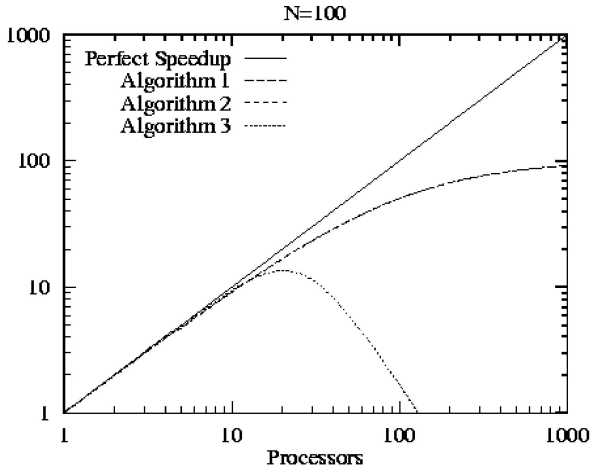
- cilj razvoja paralelnog programa najčešće nije optimiranje samo jednog kriterija (brzina)
- performanse programa mogu obuhvaćati i učinkovitost, memorijske zahtjeve, količinu ili tok podataka kroz mrežu, troškove implementacije i održavanja, prenosivost itd.
- primjeri:
  - u paralelnoj obradi multimedijalnih podataka može biti važan kapacitet programa, odnosno koju se količina podataka može obraditi u jedinici vremena - **propusnost**
  - u obradi podataka sa senzora u sustavu stvarnog vremena važno je vrijeme obrade jedne informacije u programu (koji može biti izgrađen kao cjevovod) - **odziv**
  - u slučaju plaćanja vremena paralelnog računala, važan je omjer ukupnog vremena i računalnih zahtjeva aplikacije, itd.
- u ovom poglavlju obradit će se uglavnom **ukupno trajanje izvođenja** programa i **skalabilnost** (prilagodba većem problemu i/ili većem broju procesora)
- za početak: tri uobičajena načina opisa performansi programa

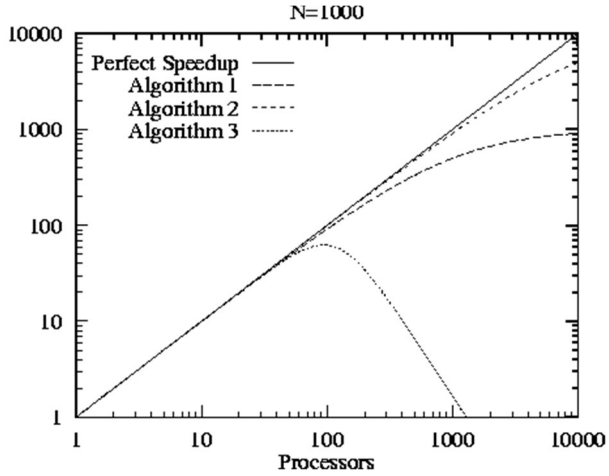
6.1.1 Amdahlov zakon

- opisuje moguće 'ubrzanje' paralelnog u odnosu na slijedni program, u ovisnosti o broju procesora te 'slijednom' i 'paralelnom' udjelu algoritma (poglavlje 1.4.1)
- primjer: ako je slijedni dio algoritma (onaj koji se ne može paralelizirati) 5%, tada je najveće ubrzanje 20 (bez obzira na broj procesora)
- u stvarnosti, gotovo svaki problem može se riješiti na način da je proizvoljno odabrani dio posla paralelan, pa navedeni zakon ne opisuje dobro potencijale programa
- uvijek postoji ograničenje na skalabilnost programa, no ono je uzrokovano komunikacijom, međusobnim čekanjem zadataka ili uvišestručenim računanjem
- zakon se može primijeniti u slučajevima kada se (relativno veliki) program paralelizira dio po dio, te neki dijelovi ostaju slijedni

6.1.2 Opažanje iz primjera

- često se u literaturi mogu naći opaske kao sljedeća: "Algoritam postiže ubrzanje od 10.8 na 12 procesora uz veličinu problema N=100"
- ovakva opažanja nisu pokazatelj za promjenjive uvjete rada
- primjer: tri različita algoritma imaju sljedeće ukupno trajanje (T) u ovisnosti o veličini problema i broju procesora:
  1. $T = N + N^2 / P$
  2. $T = (N + N^2)/P + 100$
  3. $T = (N + N^2)/P + 0.6\ P^2$
- svi navedeni algoritmi imaju otprilike isto ubrzanje u slučaju P=12 i N=100, no ponašanje im je vrlo različito [1]:





6.1.3 Asimptotska analiza

- u znanstvenoj literaturi algoritmi se također često ocjenjuju asimptotskom složenošću, kao *O*() notacijom (npr. vremenska složenost algoritma je *O*(N log N))
- nedostaci navedenog opisa:
  - *procjena trajanja*: ocjena ne uzima u obzir veličinu problema i broj procesora za praktične primjene (npr. točna složenost može biti (10N + N log N); pribrojnik 10N je veći od drugoga za N<1024 i mora se uzeti u obzir ako je problem koga rješavamo u tome opsegu)
  - *usporedba algoritama*: algoritam čije je trajanje (1000N log N) je asimptotski bolji od ($10N^2$), no drugi je bolji za N<996, što nam u praksi može biti važnije
  - ocjene složenosti često u obzir uzimaju idealizirane modele (PRAM) koji ne moraju imati slične parametre konkretnom paralelnom računalu

6.2 Model ocjene performansi

- potrebno je definirati model koji će željenom preciznošću opisivati ponašanje programa, bez ulaska u nepotrebno puno detalja
- najvažnija osobina programa je upravo trajanje: funkcija veličine problema, broja procesora, broja zadataka itd.
- **Trajanje izvođenja** (*execution time*) definiramo kao zbroj trajanja računanja, komuniciranja i čekanja (*idle time*) procesora koji izvodi paralelni program
- neformalna definicija: proteklo vrijeme između početka rada bilo kojeg (od svih koji izvode program) procesora i završetka rada bilo kojeg procesora
- trajanje izvođenja može biti definirano kao količina vremena utrošena na jednom procesoru:

$$T = T_R^i + T_K^i + T_Č^i$$

ili kao zbroj trajanja na svim procesorima podijeljen s brojem procesora:

$$T = \frac{1}{P}\left(T_R + T_K + T_Č\right) = \frac{1}{P}\left(\sum_{i=1}^{P} T_R^i + \sum_{i=1}^{P} T_K^i + \sum_{i=1}^{P} T_Č^i\right)$$

- zadatak: definirati matematičke izraze za navedene elemente
- pretpostavke:
  - model računala je multiračunalo - u najjednostavnijoj izvedbi ne uzimamo u obzir topologiju mreže, priručnu memoriju i slično
  - koristit će se *analiza skalabilnosti* u cilju ocjenjivanja utjecaja manje značajnih elemenata (npr. vrijeme potrebno za pokretanje algoritma)
  - za dodatno podešavanje modela koristit će se *empirijski podaci* (iz pokusa)

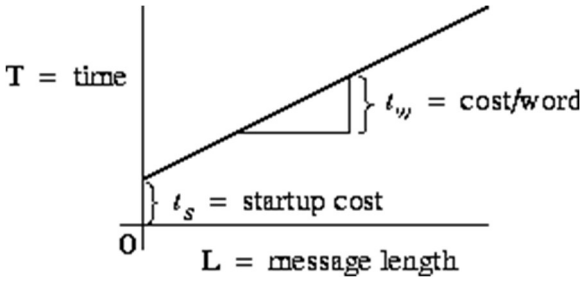
6.2.1 Trajanje izvođenja

- potrebno je odrediti trajanje računanja, komunikacije i čekanja cjelokupnog algoritma ili svakog procesora
- **Trajanje računanja** (T_R) je ukupno vrijeme koje *svi* procesori utroše na obradu podataka tijekom izvođenja cjelokupnog algoritma
- trajanje računanja paralelnog algoritma može se odrediti količinom vremena koja je slijednom algoritmu potrebna za obavljanje posla (ukoliko je slijedni program istovjetan paralelnom uz izvođenje na jednom procesoru)
- ako nemamo takav slijedni algoritam, trajanje se određuje analitički kao funkcija veličine problema (uz jedan ili više ulaznih parametara)
- ako paralelni algoritam uvišestručuje računanje (tj. određena količina posla ponavlja se u različitim zadacima), tada ukupno trajanje računanja ovisi i o broju zadataka
- u raznorodnom (*heterogeneous*) sustavu, trajanje računanja ovisi i o raspodjeli zadataka po procesorima
- **Trajanje komunikacije** je ukupno vrijeme koje svi zadaci utroše na komunikaciju
- dvije vrste komunikacija: unutarprocesna i međuprocesna (na jednom ili između više računala)
- zbog jednostavnosti, pretpostavljamo da su obje vrste usporedive
- trajanje slanja jedne poruke opisujemo dvama parametrima: trajanjem postavljanja poruke (*message startup time*) t_S i trajanjem prijenosa jedne riječi (npr. 4 bajta) t_W
- vrijeme potrebno za slanje jedne poruke duljine *L* riječi je:

$$T_{msg} = t_s + t_w L$$

- vrlo jednostavan izraz, no dovoljan za većinu primjera

Machine	t_s	t_w
IBM SP2	40	0.11
Intel DELTA	77	0.54
Intel Paragon	121	0.07
Meiko CS-2	87	0.08
nCUBE-2	154	2.4
Thinking Machines CM-5	82	0.44
Workstations on Ethernet	1500	5.0
Workstations on FDDI	1150	1.1



- u tablici lijevo su parametri komunikacije za neka računala (u mikrosekundama, 1995.[1])
- asimptotski gledano jedino je t_W bitan, no za manje poruke t_S ima puno veći utjecaj
- u većini slučajeva moguće je eksperimentalno odrediti navedene parametre
- **Trajanje čekanja** procesora je obično teže odrediti
- čekanje može biti uzrokovano:
  - nedostatkom posla – zadatak je obavio svoj posao i eventualno čeka na neki uvjetni događaj (npr. da svi ostali završe) ili još nije dobio parametre za početak posla
  - nedostatkom podataka – zadatku su za nastavak rada potrebni podaci koji se nalaze na drugom procesoru
- prva vrsta čekanja može se izbjeći raspoređivanjem zadataka (vidi 5.7.2)
- druga vrsta može se izbjeći lokalnom preraspodjelom zadataka: npr. 2 zadatka se nalaze na procesoru; dok jedan čeka na potrebne podatke, drugi se aktivira
- složeniji pristup: ponekad je moguće eksplicitno projektirati zadatke da obavljaju neki drugi posao dok čekaju na podatke (neblokirajuća komunikacija u MPI)
- **Primjer: trajanje izvođenja atmosferskog modela**
- pretpostavljamo 3D mrežu čvorova dimenzije *N* * *N* * *Z*, gdje je, uz P zadataka, svakom zadatku pridruženo *N*/*P* uspravnih ravnina (odnosno *N* * *Z* * *N*/*P* čvorova mreže)
- budući da je količina računanja jednaka za sve čvorove, ukupno trajanje računanja u jednoj iteraciji postupka je:

$$T_R = t_c N^2 Z$$

gdje je t_C trajanje računanja za jednu točku u mreži

- uzimamo u obzir da jedan čvor koristi podatke od 8 drugih točaka (u vodoravnoj ravnini, uspravnu smo aglomerirali)
- ako pretpostavimo da je $P <= N/2$, tada će svaki zadatak morati komunicirati samo sa dva svoja susjeda (lijevi i desni – jer će svaki zadatak obuhvaćati barem 2 uspravne ravnine!
- u tom slučaju, u jednoj iteraciji izmjenjuju se dvije poruke sa po 2 * *N* * *Z* podataka (zadatak može biti i 'deblji', no šalje se informacija o samo 2 rubna sloja)
- ukupno vrijeme komunikacije za sve procesore u jednoj iteraciji (ne vrijedi za $P > N/2$):

$$T_K = 2P\left(t_s + t_w 2NZ\right)$$

- ako je trajanje računanja po zadatku jednako i ako su procesori jednakih brzina, trajanje čekanja se može zanemariti
- ukupno trajanje izvođenja je:

$$T_{1D,atm.} = \frac{T_R + T_K}{P} = t_c \frac{N^2 Z}{P} + 2t_s + t_w 4NZ$$

6.2.2 Učinkovitost i ubrzanje algoritma

- budući da je trajanje izvođenja ovisno o veličini problema, definiramo relativnu učinkovitost paralelnog algoritma kao:

$$E_{rel} = \frac{T_1}{PT_P}$$

gdje je T_1 trajanje algoritma na jednom procesoru, a T_P trajanje na P procesora

- relativno ubrzanje algoritma definirano je kao:

$$S_{rel} = PE$$

- veličine su označene kao *relativne* jer različiti paralelni programi mogu imati različito trajanje na jednom procesoru
- primjer: jedan paralelni algoritam traje 10000 na jednom i 20 sekundi na 1000 procesora, dok drugi traje 1000 na jednom i 5 sekundi na 1000 procesora
- drugi algoritam je brži od prvoga (barem za broj procesora <=1000), ali prvi algoritam ima veće relativno ubrzanje
- *apsolutno* ubrzanje (i učinkovitost) može se definirati u odnosu na trajanje algoritma koji najbrže obavlja posao na jednom procesoru (tj. najbolji slijedni algoritam)
- **Primjer:** za atmosferski model, relativna učinkovitost je:

$$E_{1D,atm.} = \frac{t_c N^2 Z}{t_c N^2 Z + 2t_s P + t_w 4NZP}$$

- budući da je za ovaj slučaj najbolji slijedni program jednak paralelnom na jednom procesoru, relativna učinkovitost jednaka je apsolutnoj

6.3 Analiza skalabilnosti

- izvedene mjere trajanja izvođenja i učinkovitosti algoritma dovoljne su za kvalitativnu analizu sve dok se ne odrede specifične vrijednosti parametara za konkretno paralelno okruženje (npr. parametri komunikacije, trajanje jednog zadatka i sl.)
- nakon određivanja vrijednosti parametara, model performansi može (uz određene pretpostavke) poslužiti za dobivanje odgovora na neka od pitanja:
    - odgovara li program zahtjevima za trajanje izvođenja i drugim kriterijima na odredišnom računalu?
    - kako se program prilagođava povećanjima količine računanja ili broja procesora?

- koliko je program osjetljiv na računalne parametre (npr. t_S ili t_W)?
- koji je odnos performansi programa s postojećim algoritmima? i sl.

6.3.1 Ponašanje uz stalnu veličinu problema

- Problem: odrediti kako se program ponaša uz povećan broj procesora, a konstantnu količinu računanja (veličinu problema)
- matematički: kako se mijenjaju trajanje izvođenja (*T*) i učinkovitost algoritma (*E*) uz povećanje broja procesora (*P*)?
- učinkovitost obično monotono opada s povećanjem procesora (vidi izraz za atmosferski model!)
- kako bi učinkovitost programa ostala *jednaka* uz povećan broj procesora i jednaku veličinu problema, vrijeme utrošeno na troškove paralelnog rada (komunikaciju, sinkronizaciju, čekanje...) bi moralo biti nepromjenjivo, što je u praksi u pravilu nemoguće postići
- trajanje izvođenja može i rasti ako formula uključuje član proporcionalan sa brojem procesora na neku pozitivnu potenciju (navedeni član može biti zanemariv uz manji broj procesora)
- motivacija: ako za izvođenje aplikacije plaćamo i *vrijeme* i *broj* korištenih procesora, može nas zanimati koliko najviše procesora program može koristiti a da učinkovitost bude veća od neke minimalne granične vrijednosti

6.3.2 Ponašanje uz promjenjivu veličinu problema

- Pitanje: kako se mora mijenjati *veličina posla* uz povećanje broja procesora kako bi učinkovitost ostala jednaka? (ili obrnuto: koliko trebamo povećati broj procesora uz veću količinu posla kako bi učinkovitost ostala jednaka)
- navedena ovisnost naziva se funkcija **izoučinkovitosti** algoritma (*isoeffeiciency function*) i izražava se *O()* notacijom ovisnosti o broju procesora
- npr: funkcija izoučinkovitosti složenosti O(*P*) znači da bi se količina posla morala linearno povećavati uz linearno povećanje broja procesora i uz zadržavanje jednake učinkovitosti, što se smatra oznakom dobrog algoritma (kvadratna ili eksponencijalna izoučinkovitost smatra se lošom)
- analiza ponašanja uz promjenjivu veličinu problema ima smisla samo za određene paralelne algoritme (npr. nema smisla ukoliko postoje čvrsta vremenska ograničenja za završetak posla (*hard real-time constraints*) ili je veličina problema inherentno stalna, kao broj piksela u obradi slike i slično)
- **Primjer: izoučinkovitost atmosferskog modela**
- izoučinkovitost se može odrediti promatrajući izraz za učinkovitost određenog modela – npr. za atmosferski model dobiven dekompozicijom u jednoj dimenziji
- formulu za učinkovitost možemo napisati u sljedećem obliku:

$$t_c N^2 Z \approx E\left(t_c N^2 Z + t_s 2P + t_w 4NZP\right)$$

- koja funkcija za *N* ovisna o *P* zadovoljava gornji uvjet?
- kako bi učinkovitost ostala jednaka, potrebno je odrediti kako se mora povećati *N* uz povećanje *P*; za zadani izraz, ovisnost *N* o *P* bi trebala biti (približno) linearna, odnosno vrijedilo bi $N \sim P$
- stoga uz uvrštenje $N = P$ dobivamo:

$$t_c Z \approx E\left(t_c Z + t_s \frac{2}{P} + t_w 4Z\right)$$

  što približno vrijedi za slučajeve kada *P* nije premali
- budući da je količina posla ovisna o N^2 (gledamo ukupno vrijeme računanja, koje ovisi o kvadratu *N*), izoučinkovitost algoritma je O(P^2), jer bi količina posla trebala rasti sa *kvadratom* broja procesora kako bi učinkovitost ostala jednaka
- **Primjer: izoučinkovitost atmosferskog modela uz dekompoziciju u dvije dimenzije**

- dekompozicijom u dvije dimenzije dijelimo mrežu $N * N * Z$ na dijelove koji su 'izrezani' u dvije vodoravne dimenzije, dok je uspravna ostala očuvana (svi dijelovi imaju visinu Z)
- skup čvorova pojedinog zadatka je kvadar dimenzija $(N / \sqrt{P}) * (N / \sqrt{P}) * Z$ (*skicirati podjelu!*) – izraz za broj čvorova po zadatku jednak je kao i u 1D dekompoziciji
- u ovoj podjeli, svaki zadatak mora u jednoj iteraciji razmijeniti 4 poruke (umjesto dvije kod 1D podjele) u kojima šalje informaciju o $2(N / \sqrt{P})Z$ čvorova u svakoj poruci (i dalje vrijedi pretpostavka da je svaki zadatak 'širok' barem dvije ravnine čvorova)
- učinkovitost modela s 2D podjelom je stoga:

$$E = \frac{t_c N^2 Z}{t_c N^2 Z + t_s 4P + t_w 8NZ\sqrt{P}}$$

  što se može napisati kao

$$t_c N^2 Z \approx E\left(t_c N^2 Z + t_s 4P + t_w 8NZ\sqrt{P}\right)$$

- kako bi učinkovitost ostala jednaka uz povećanje broja procesora, N mora biti proporcionalan sa kvadratom broja procesora, pa uz uvrštenje $N = \sqrt{P}$ imamo:

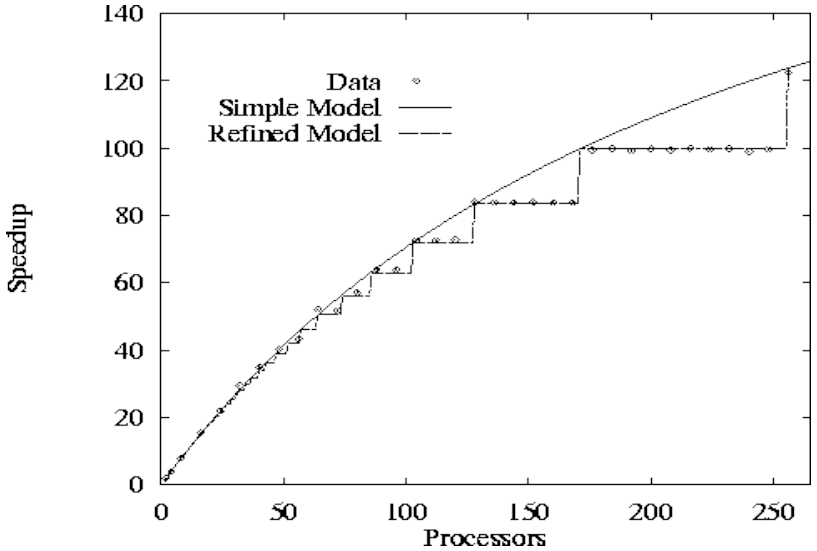
$$t_c Z = E\left(t_c Z + t_s 4 + t_w 8Z\right)$$

  što daje izraz u kojemu učinkovitost ne ovisi o broju procesora
- budući da je količina posla proporcionalna s N^2, izoučinkovitost algoritma s 2D podjelom je $O(P)$, tj. drugi algoritam ima *bolje svojstvo skalabilnosti* od prve inačice
- iako koristi više poruka u jednoj iteraciji, komunikacijski troškovi algoritma su manji uz promjenjive uvjete rada (velike N i P)
- **Općenito vrijedi**: u velikom broju slučajeva podjele po *više* dimenzija imaju bolju skalabilnost od podjela u *manje* dimenzija

6.4 Provjera modela i implementacije

- nakon izrade paralelnog programa često je korisno (nužno) odrediti ponašanje algoritma u stvarnim uvjetima rada, odnosno usporediti predviđene i eksperimentalno utvrđene parametre rada (npr. ukupno trajanje)
- ukoliko se rezultati mjerenja u velikoj mjeri razlikuju od vrijednosti dobivenih modelom ocjene performansi, razlozi mogu biti:
  - model ocjene je nedovoljno precizan ili netočan
  - aplikacija nije napisana u skladu sa modelom
  - rezultati ili metodologija mjerenja su nepouzdani
- često se događa da je trajanje programa veće od predviđenoga, čemu je razlog najčešće nepotpunost modela – model ne uzima u obzir neke pojave pri radu programa
- neki od mogućih uzroka neprikladnosti modela:
  - *nejednako opterećenje* – tijekom rada mogu se pojaviti neujednačena opterećenja u računanju ili komunikaciji koje model nije uzeo u obzir
  - *uvišestručeno računanje* – jedan dio posla ponavlja se na svakom procesoru (nije paraleliziran), što može biti zanemarivo kod malog broja procesora, ali doći do izražaja kod većeg broja
  - *nesklad algoritma i programskog alata* – odabrani programski alat možda je neučinkovit u implementaciji određenog programskog modela (npr. možda koristimo više zadataka na jednom procesoru, a programski alat zadužen za održavanje zadataka ima velike računalne zahtjeve)
  - *ograničen kapacitet komunikacije* – opisani jednostavni model komunikacije ne uzima u obzir ograničenja komunikacijskog kanala (*bandwidth*), što može doći do izražaja pri većem opterećenju
- **Primjer: provjera atmosferskog modela uz 1D podjelu**
- program je predstavnik općenitog modela računanja konačnih razlika (primjer 5.2.1)
- parametri komunikacije su t_S = 200 μs, t_W = 2 μs (višeprocesorsko računalo)

- na slici je prikazano ubrzanje u ovisnosti o broju procesora po prije definiranom modelu (puna linija) i po rezultatima mjerenja (kružići) za *N* = 512 (vodoravne dimenzije mreže) [1]



- zbog čega nastaje razlika, odnosno zašto su rezultati mjerenja diskontinuirani?
- po izvedenoj formuli, svaki procesor dobiva jednaki broj čvorova u mreži: $N * Z * N/P$
- u stvarnosti, budući da *N* nije uvijek djeljivo sa *P*, neki procesori dobivaju $N * Z * \lceil N/P \rceil$, a neki $N * Z * \lfloor N/P \rfloor$ čvorova
- nejednako opterećenje uzrokuje čekanje nekih procesora jer su iteracije postupka sinkronizirane (svi čekaju na onoga koji ima najviše posla)
- točniji izraz koji opisuje trajanje jedne itracije je stoga:

$$T = t_c NZ \left\lceil \frac{N}{P} \right\rceil + 2t_s + t_w 4NZ$$

6.4.1 Nepravilnosti ubrzanja algoritma

- ponekad se paralelni program odvija brže nego što je modelom predviđeno; ukoliko se ta razlika povećava s povećanjem broja procesora, govorimo o anomaliji ubrzanja (*speedup anomaly*)
- ukoliko je ubrzanje veće od linearnog (npr. trostruko ubrzanje uz 2 procesora), govorimo o *superlinearnom* ubrzanju
- mogući uzroci nepravilnosti ubrzanja:
  - *uporaba priručne memorije (cache effects)* – kod nekih problema može se dogoditi da se dodavanjem procesora više potrebnih podataka može spremiti u priručnu memoriju, te da se zbog toga smanjuje trajanje računanja. Ako je smanjenje u trajanju računanja veće od povećanja trajanja komunikacije (zbog više procesora), nastaje napravilnost
  - *anomalije pretraživanja (serach anomalies)* – ova pojava je česta u nekim algoritmima koji obavljaju paralelno pretraživanje prostora stanja (pretraživanje struktura podataka, evolucijski algoritmi, *branch-and-bound*); u ovom slučaju, paralelna inačina algoritma nema jednaka svojstva kao i slijedna, pa je zapravo riječ o drugačijem algoritmu (opaska: najbolji slijedni algoritam bi mogao biti onaj koji simulira paralelni model)

7 Paralelno programiranje grafičkih procesora

- **Ciljevi poglavlja**: upoznati osnove rada grafičkih kartica i mogućnosti paralelizacije
- Literatura:
  1. CUDA C Programming Guide (http://docs.nvidia.com/cuda/cuda-c-programming-guide/)
  2. The OpenCL Specification (https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html)
  3. OpenCL C Specification (https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html)
  4. https://github.com/KhronosGroup/OpenCL-Guide
  5. AMD OpenCL User Guide (http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf)
  6. OpenCL Programming Guide for the CUDA Architecture (https://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf)

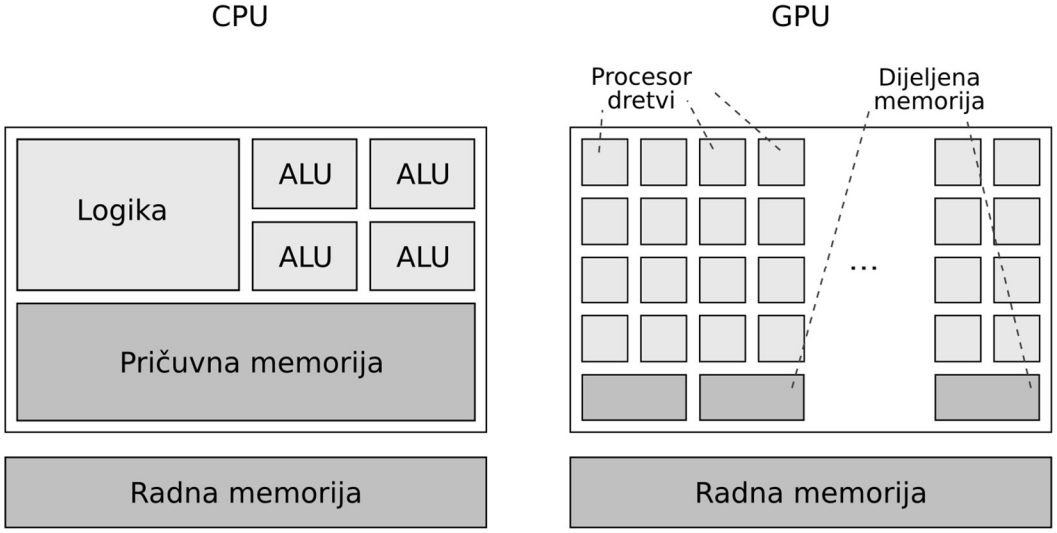
7.1 Računanje na grafičkim karticama

- grafičke kartice (GPU) uključuju velik broj procesora specifične namjene (obrada grafičkih primitiva)
- početkom stoljeća :), u grafičke kartice uvode se programabilni procesori (*shader*) i podrška za aritmetiku s pomičnim zarezom – GPU postaje skalabilni paralelni procesor opće namjene
- struktura grafičkih podataka (najčešće 2D) omogućava jednostavnu obradu višedimenzijskih struktura podataka (vektori, matrice, tenzori - višedimenzijski vektori)
- posebno učiknovita arhitektura u slučaju podatkovne dekompozicije ili trivijalno paralelnih algoritama
- primjena grafičkih procesora u obradi podataka nazvana je *general purpose computing on GPU* (GP GPU)
- prve primjene uključivale su *matrične operacije* i rješavanje sustava linearnih jednadžbi (*LU faktorizacija*; vidi APR)
- postupak računanja morao je biti opisan koristeći grafičke biblioteke (npr. OpenGL, DirectX), što je bilo nespretno
- podrška za prenosive tipove podataka nije posve ujednačena:
  - cjelobrojni tipovi podataka nisu uvijek dostupni na GPU
  - računanje s pomičnom točkom nije uvijek bilo usklađeno s IEEE standardom (manjak prikaza dvostruke preciznosti, nekompatibilnost nekih operacija – vidi APR)
- pojavljuju se (i nestaju) razne biblioteke za razvoj općenitih programa na GPU: RapidMind, C++ AMP, OpenVIDIA, …
- poznatije biblioteke/standardi za GPU: OpenCL, Nvidia CUDA, OpenACC, OpenMP
- uz ove, postoje i drugi alati koji imaju izvedenice priladođene izvođenju na GPU: Magma, Thrust, cuBLAS, cuFFT, cuSPARSE, cuRAND, TensorFlow,…
- primjene: duboko učenje, *data mining*, bioinformatika, FFT, obrada signala, fizikalne simulacije, kriptografija i kriptoanaliza, kriptovalute :) …
- u ovom dokumentu koristi se OpenCL (pretpostavljeno) odnosno CUDA nomenklatura

7.2 Računalni model

- usporedba CPU/GPU:
  - CPU: manji (relativno govoreći) broj brzih jezgri; relativno velika priručna memorija u više razina; velik skup instrukcija; relativno mali broj registara; manja brzina komunikacije s radnom memorijom

- GPU: veći broj sporijih jezgri; manja priručna memorija u jednostavnijoj hijerarhiji; manji skup instrukcija (manja podrška za razlilite operande); velik prostor za kontekst (sadržaji registara); veća propusnost glavne memorije

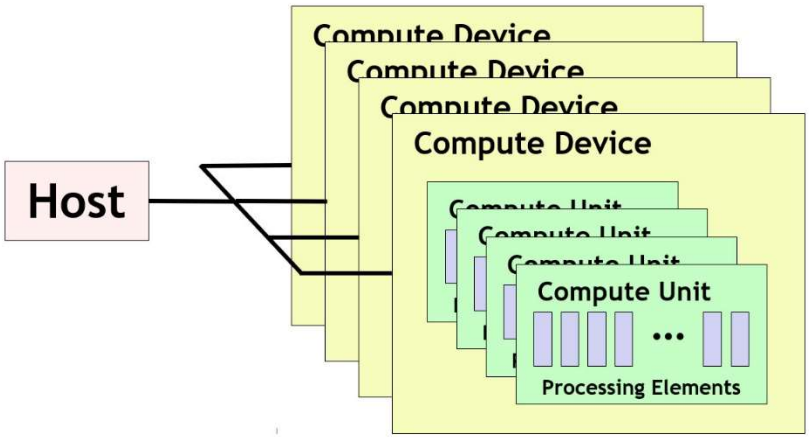


- model procesiranja: SIMT (*single instruction, multiple threads*) – jedan niz instrukcija izvodi se na velikom broju procesora
- računalni model GPU moguće je predočiti na više načina:
  - sličnost s modelom SIMD, uz razliku da se ne koriste vektorske operacije (svaka dretva obično pristupa jednom elementu)

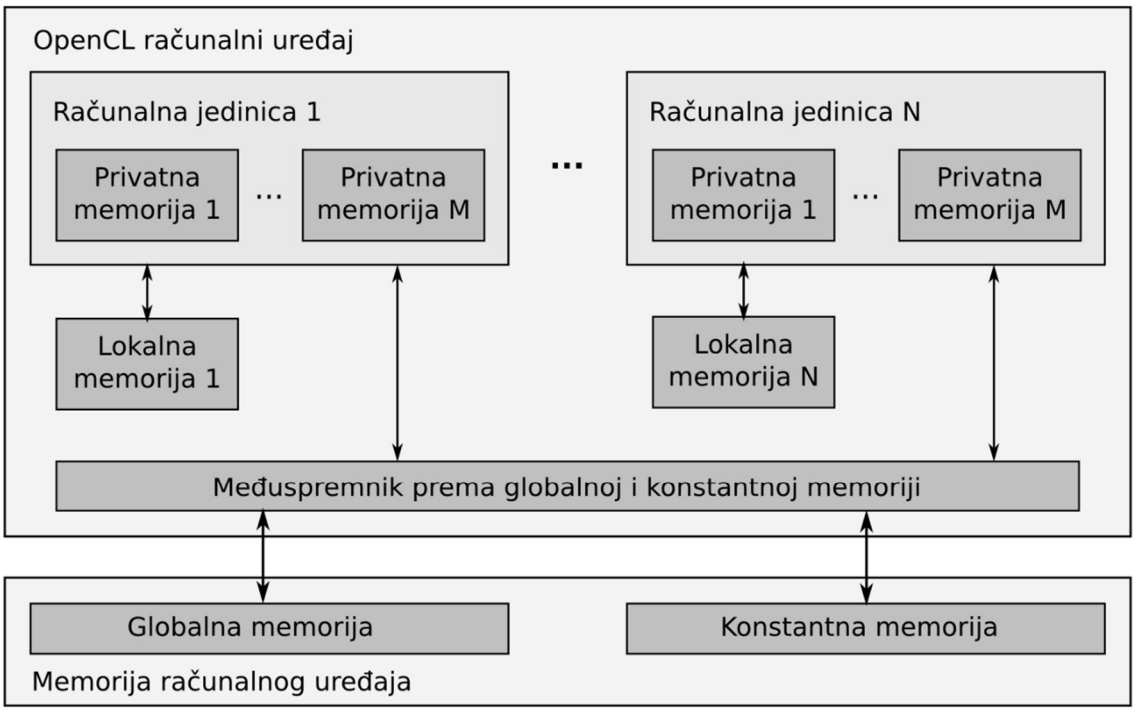
OpenCL naziv	**CUDA naziv**	
host	host	
compute device	device	(računski) uređaj
compute unit	multiprocessor	računska jedinica, multiprocesor, GPU procesor
processing element	scalar processor	procesni element
global memory	global memory	globalna memorija
local memory	shared memory	lokalna memorija
private memory	private memory	privatna memorija, memorija dretve
wavefront	warp	val
index range	grid	skup dretvi
work group	thread block	radna grupa
work item	thread	dretva, radna jedinica

- struktura: jedan GPU sadrži više (nekoliko desetaka) računskih jedinica (GPU procesor, multiprocesor, *streaming multiprocessor*)

- jedna računska jedinica sadrži nekoliko desetaka procesnih elemenata (jezgra, procesor dretve, skalarni procesor) – ukupno nekoliko tisuća jezgara
- *primjer*: Tesla V100 sadrži 84 računske jedinice; svaka sadrži 64 jezgre (točnije, 64 jezgre sposobne za 32-bitne cjelobrojne ili 32-bitne FP operande, ali 32 jezgre za 64-bitne FP operande dvostruke preciznosti)



7.2.1 Memorijska struktura



- u GPU arhitekturi ralikuje se nekoliko vrsta memorije
- **globalna memorija** (*global memory*): radna memorija jednog uređaja
  - najveća, najsporija (u odnosu na druge memorijske elemente)
  - sve dretve mogu pristupiti sadržaju
  - koristi se za komunikaciju s uređajem domaćinom (*host*) – jedina memorija kojoj je moguće pristupiti izvan uređaja
  - tipično 8-16 GB
- unutar globalne memorije definiraju se dva područja:
  - *konstantna memorija*: namijenjena samo za čitanje (*read only*) od strane radnih dretvi
  - *teksturna memorija*: također samo za čitanje, ali optimirana za pristup *prostorno bliskim* podacima od strane više dretvi (uz pomoć priručne pohrane)

- **lokalna memorija** (*local memory, shared memory*): radna memorija jednog multiprocesora
  - puno manja ali brža od globalne memorije
  - mogu joj pristupiti samo dretve koje se izvode na dotičnom (istom) multiprocesoru
  - tipično reda veličine 48-64 kB
- **privatna memorija** (*private memory*): radna memorija jedne dretve (registri)
  - najmanja veličina, najbrži pristup

7.2.2 Paralelno izvođenje dretvi

- u najjednostavnijoj inačici, svi procesori na GPU izvode jednaku dretvu (isti niz instrukcija), iako je danas moguće izvođenje različitih dretvi unutar jednog uređaja
- dretvu je potrebno izvesti u zadanom broju instanci, koji je neovisan o broju procesnih elemenata (može biti veći): GPU raspoređuje dretve na multiprocesore
- skup svih dretvi (svih instanci koje treba izvesti) se dijeli na **radne grupe** (*work group*, *thread block)*
- svaka radna grupa dodjeljuje se nekom multiprocesoru
- jednom multiprocesoru može biti dodijeljeno *više* radnih grupa (uz ograničenje…)
- svaki procesni element unutar multiprocesora izvodi jednu dretvu
- kada multiprocesor izvede sve dodijeljene radne grupe, dodjeljuju mu se nove grupe
- broj dretvi koje se *istovremeno* izvode na jednom multiprocesoru naziva se **val** (*warp, wavefront*) i ovisi o arhitekturi:
  - CUDA (NVIDIA): val se sastoji od 32 dretve
  - OpenCL (AMD): val se sastoji od 64 dretve (neki uređaji koriste samo 16 jezgri ali uz protočno izvođenje svake instrukcije u 4 ciklusa)
- dretve unutar istog vala dijele programsko brojilo: instrukcije dretvi unutar jednog vala su *sinkronizirane*!
  - pruža implicitnu sinkronizaciju za dretve unutar istog vala! (kao PRAM)
  - nažalost, Nvidia je proširila funkcionalnost te za novije arhitekture (Volta i dalje) to više ne mora biti istina… (ne možemo računati na implicitnu sinkronizaciju!)
- unutar jednog vala, dretve se izvode u SIMD načinu rada: svaka dretva u istom ciklusu izvodi istu instrukciju
- ako postoji grananje, sve grane se izvode slijedno!
  - bitovna maska definira dretve unutr vala koje su trenutno *aktivne*
  - prvo se izvodi grana A
  - maska se invertira, izvodi se grana B
- ukupno trajanje ovisi o uvjetima u trenutku izvođenja (maksimalno jednako zbroju svih grana)
- u slučaju programske petlje s dinamičkim uvjetom ponavljanja: trajanje izvođenja petlje (ukupan broj iteracija) je trajanje dretve s *najvećim brojem iteracija*

```
if(x)
{
// grana A
...
}
else
{
// grana B
...
}
```

7.2.3 Pristup memoriji

- početno se svi ulazni podaci nalaze u globalnoj memoriji uređaja
- pristup globalnoj memoriji je puno sporiji od jednog ciklusa! → sve dretve koje čekaju na pristup memoriji postaju *blokirane*
- *što ako su sve dretve u valu blokirane?* → moguće je postojanje više valova na jednom multiprocesoru!
- ako su sve dretve nekog vala blokirane, pokreće se val koji ima barem jednu *aktivnu* dretvu – na taj način skriva se kašnjenje memorije:
  - "*compute device uses number of wavefronts to hide memory access latencies having the scheduler switch the active wavefront whenever the current wavefront is waiting for a memory access to complete*" [2]
- *što ako više dretvi u valu pristupa istoj memorijskoj lokaciji?*

  - ako je riječ o „običnoj“ instrukciji: pisanje je slijedno, a rezultat je nedefiniran (“*which thread performs the final write is undefined*“)
- programski model definira i *nedjeljive* operacije pristupa memoriji (*atomic*), koje izvode čitanje, promjenu i pisanje u istu memorijsku lokaciju (*read-modify-write*)
- ukoliko više dretvi izvodi nedjeljivu operaciju, sve operacije će se izvesti slijedno, ali u nedefiniranom redoslijedu
- *Koliko košta izmjena aktivnog vala?* (*conext switch*)
  - ništa! jer se *sve* lokalne varijable *svih* dretvi dodijeljenih multiprocesoru čuvaju unutar lokalne memorije multiprocesora
  - “switching from one execution context to another has no cost” [1]
- *posljedica* 1: sustav mora *unaprijed* znati koliko dretvi može pokrenuti na jednom GPU procesoru
  - “*The number of blocks and warps that can reside and be processed together on the multiprocessor for a given kernel depends on the amount of registers and shared memory used by the kernel and the amount of registers and shared memory available on the multiprocessor*”
- ako nema dovoljno memorije za izvođenje barem jedne grupe dretvi, pokretanje neće uspjeti
- *posljedica* 2: u dretvama koje se izvode na GPU nije moguće koristiti dinamičko zauzimanje memorije
  - ukoliko dretva ima prevelik zahtjev za memorijom, dolazi do greške prilikom pokretanja (nije moguće izvesti na GPU procesoru)
- *posljedica* 3: nema rekurzije (barem ne u općenitom obliku)

7.3 Programski model

- dvije najpopularnije platforme za programiranje: CUDA, OpenCL
- u ovom dokumentu: pretpostavlja se *OpenCL*
- OpenCL aplikacija sastoji se od dva dijela:
  - dio programa koji se izvodi na CPU – *host*
  - dio programa koji se izvodi na računskom uređaju (*compute device*) i obično se naziva *kernel*
- prednosti: prenosivost (Nvidia, AMD, Intel…); učinkovitost jednaka ostalim alatima
- isti programi mogu se pokrenuti na različitim uređajima: GPU, CPU, FPGA, DSP… → omogućava izvođenje u heterogenim okolinama
- nedostaci:
  - paralelni dio programa (kerneli) piše se u OpenCL C jeziku
  - kerneli se prevode tijekom izvođenja programa (*runtime*) – zbog prenosivosti (ali za istu platformu moguće je iskoristiti prethodno prevedeni kernel)
- osnovna pravila pisanja OpenCL kernela:
  - kernel se označava ključnom riječi `kernel`
  - povratna vrijednost je uvijek `void`
- primjer kernela pisanog za OpenCL: množenje elemenata niza zadanom vrijednošću

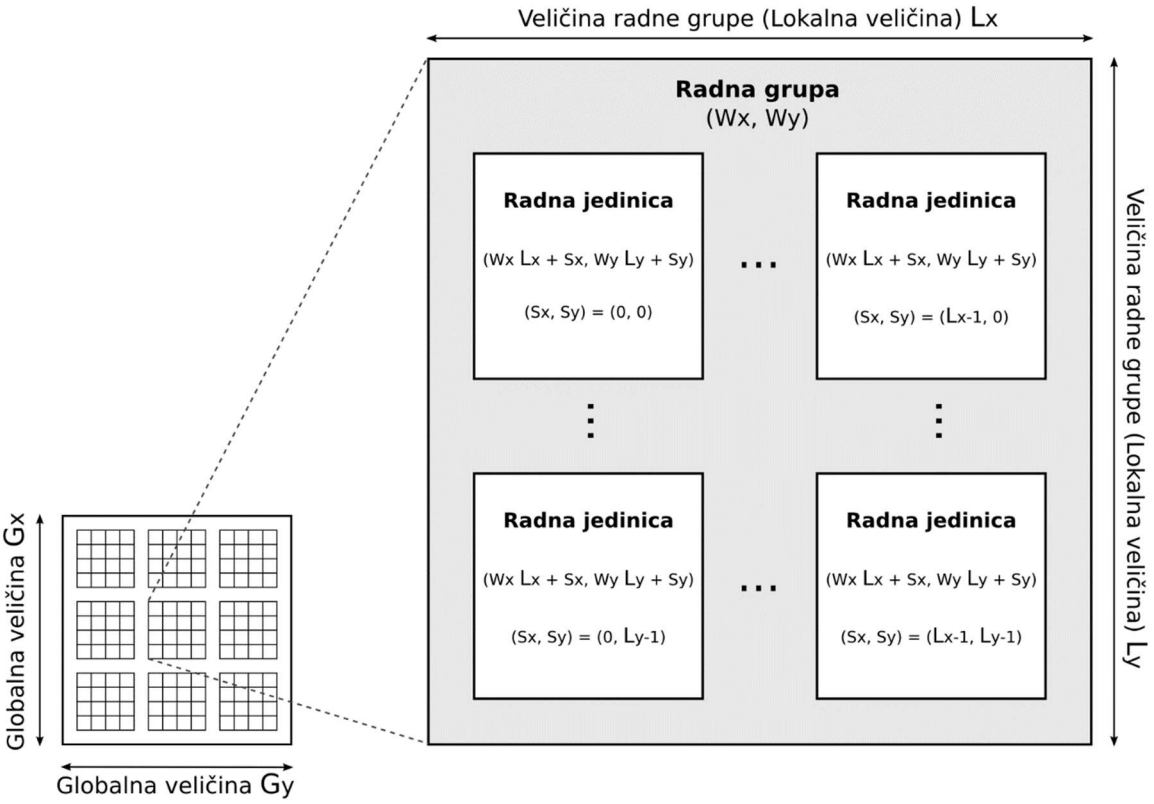
```
kernel void vektor(global const int *A, global int *B, const int n)
{
  // dohvati indeks
  int i = get_global_id(0);
  if (i > n) return;
  B[i] = A[i] * i;       // npr.
}
```

7.3.1 Definiranje skupa dretvi

- host definira dio programa (kernel) koji se na računskom uređaju treba izvesti u zadanom broju instanci - *dretvi*
- skup svih dretvi koje se trebaju izvesti naziva se prostor indeksa (*indeks range*), *NDRange*

  - veličina globalnog skupa je G
  - ako je skup jednodimenzijski, svaka dretva ima jedinstveni globalni ID (0..G-1)
- globalni skup dretvi podijeljen je na *grupe dretvi* (radna grupa, *work group*)
  - sve radne grupe imaju istu veličinu (L), a unutar jednodimenzijske grupe svaka dretva ima jedinstveni lokalni ID (0..L-1)
  - jedna radna grupa dodjeljuje se jednom GPU procesoru
- pojedina dretva unutar radne grupe naziva se *radna jedinica* (*work item*)
  - svaku dretvu izvodi jedan procesni element (procesor dretve)
- npr: skup od 100 dretvi moguće je definirati kao 100 grupa veličine 1 dretve, 10 grupa veličine 10 dretvi, 1 grupu veličine 100 dretvi
- svaka grupa će se prilikom izvođenja podijeliti na valove (redom po 32/64 dretve)
- optimalna podjela: broj dretvi u radnoj grupi treba biti **višekratnik veličine vala**!
- dretve unutar radne grupe dijele se u valove po lokalnom rednom broju: prvih 32/64 dretvi su prvi val, sljedećih 32/64 su drugi val itd.
- globalni skup dretvi definiran je kao vektor u jednoj, dvije ili tri dimenzije: česta je podjela dretvi u obliku matrice
- u slučaju višedimenzijske podjele, redni broj dretve se definira u svakoj dimenziji
- dretve mogu saznati svoj identifikator uporabom sljedećih funkcija:

`uint get_work_dim ();`	`oznake (za dim = 2)`
`size_t get_global_size (dim);`	`Gx, Gy`
`size_t get_global_id (dim);`	`gx, gy`
`size_t get_global_offset (dim);`	
`size_t get_num_groups (dim);`	
`size_t get_group_id (dim);`	`Wx, Wy`
`size_t get_local_id (dim);`	`Sx, Sy`
`size_t get_local_size (dim);`	`Lx, Ly`



7.3.2 Izvođenje dretvi

- izvođenje zadanog kernela obično uključuje sljedeće korake:

1. pripremu ulaznih podataka na *hostu*
2. kopiranje podataka na uređaj
3. izvođenje svih dretvi
4. kopiranje rezultata na *host*

- kada je to isplativo? trajanje izvođenja dretvi trebalo bi biti dulje od trajanja prijenosa podataka
- osim toga, broj lokalnih instrukcija (računanje koje koristi privatnu memoriju) trebao bi biti veći od broja globalnih pristupa (čitanja i pisanja u globalnu memoriju uređaja)
- svaki skup dretvi dodaje se u *red izvođenja* (*command queue*)
- host aplikacija može definirati više različitih kernela za izvođenje na uređaju
- različiti skupovi dretvi (od različitih kernela) mogu se izvoditi slijedno (*in order*) ili paralelno (*out of order*)
  - *slijedno izvođenje*: sve dretve iz jednog skupa moraju se izvesti prije bilo koje dretve iz sljedećeg skupa u redu
  - *paralelno izvođenje*: dretve iz različitih skupova mogu se izvoditi istodobno na uređaju
- paralelno izvođenje omogućava maksimalnu iskoristivost računskog uređaja!
- pretpostavljeni način izvođenja koristi slijedni red skupova dretvi
- unutar jednog skupa, redoslijed izvođenja različitih grupa dretvi je *proizvoljan* (sve grupe su ekvivalentne)
  - moguće je emulirati sinkronizacijske primitive (npr. semafore) uz pomoć atomičkih operacija
- za dretve unutar iste grupe, postoji mogućnost eksplicitne sinkronizacije instrukcijom ograde (*barrier*)
- unutar istog vala dretvi, sinkronizacija je implicitna – dretve dijele programsko brojilo
  - uz iznimku novijih primjera arhitekture Nvidia!

Načini sinkronizacije OpenCL dretvi	
unutar jednog vala	implicitna (*lock step*)
unutar grupe dretvi	instrukcija ograde (*barrier*)
između grupa dretvi	nije podržano (nepotrebno)
između različitih kernela	redovi izvođenja (definira se na hostu)

7.3.3 Primjer koji svi koriste: zbrajanje vektora

- pretpostavimo kernel koji zbraja dva vektora:

```
kernel void zbroj(global const double *A, global const double *B, global double *C)
{
   int i = get_global_id(0);
   C[i] = A[i] + B[i];
}
```

- veličina niza (N) zadana je u vanjskom programu (kernel pretpostavlja da će broj dretvi biti jednak broju elemenata)
- priprema i izvođenje obavlja se u nizu koraka (ovdje je navedena C sintaksa)
- **Otkrivanje platformi**

```
// Dohvati prvu raspolozivu platformu:
cl_platform_id platform;
clGetPlatformIDs(1, // koliko platformi zelimo
   &platform,       // lista pronadenih platformi
   NULL);           // koliko platformi postoji
```

- **Detektiranje uređaja na platformi**

```
// Dohvati prvi GPU
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU,
   1,               // dodaj 1 uredaj
   &device,         // lista svih uredaja
```

```
   NULL);            // koliko je ukupno uredaja dostupno
```

- Stvaranje konteksta i reda izvođenja

```
cl_context context = clCreateContext(
   0,                // opcionalan parametar
   1,                // broj uredaja
   &device,          // lista uredaja
   NULL, NULL,       // opcionalna callback funkcija za dohvat informacija o greskama
   NULL);            // ako ne zelimo kod greske

cl_command_queue queue = clCreateCommandQueue(
   context,          // struktura s kontekstom
   device,           // uredaj na koji se kontekst odnosi
   0,                // opcionalni parametri
   NULL);            // ako ne zelimo kod greske
```

- **Učitavanje i prevođenje kernela**
- uputno je dodati provjeru i ispis grešaka prilikom prevođenja!

```
// Stvori objekt programa i ucitaj program kernela u taj objekt
cl_program program = clCreateProgramWithSource(context,
   1,                // broj stringova (kernel funkcija)
   &source,          // niz stringova
   NULL,             // duljina stringa ili string NULL terminiran
   NULL);            // ne vracaj kod greske

// Stvori izvrsni program
clBuildProgram(program,
   1,                // broj uredaja
   &device,          // pokazivac na listu uredaja
   NULL,             // dodatne postavke (build options)
   NULL, NULL);      // dodatno (callback funkcija i argumenti)

// Stvori objekt zadanog kernel programa
cl_kernel kernel = clCreateKernel(
   program,          // objekt koji sadrzi program
   "zbroj",          // ime kernel funkcije
   NULL);            // ne vracaj kod greske
```

- **Stvaranje i priprema struktura podataka**
- pretpostavimo da smo definirali nizove A[], B[] i C[] veličine N realnih brojeva

```
cl_mem a_buffer = clCreateBuffer(context,
   CL_MEM_READ_ONLY |     // spremnik je read only
   CL_MEM_COPY_HOST_PTR,  // automatski se kopira sadrzaj niza A u memoriju uredaja!
   N*sizeof(double),      // velicina spremnika u bajtovima
   A,                     // pokazivac na host memoriju
   NULL);                 // ne vracaj kod greske
cl_mem b_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
N*sizeof(double), B, NULL);
cl_mem c_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, N*sizeof(double), NULL,
NULL);
```

- oznake vrsta međuspremnika su CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, CL_MEM_READ_ONLY
- ako se navede i zastavica CL_MEM_COPY_HOST_PTR, podaci se automatski kopiraju na uređaj
- inače, podatke je moguće kopirati eksplicitno uz pomoć funkcije clEnqueueWriteBuffer
- **Zadavanje argumenata kernel funkcije**

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*) &a_buffer);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*) &b_buffer);
```

```
clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*) &c_buffer);
```

- redoslijed pozivanja nije bitan, redni broj argumenta navodi se u pozivu
- **Izvođenje kernela**
- kerneli se stavljaju u red izvođenja, koji je pretpostavljeno slijedan – ako se definira više kernela u redu, izvode se jedan za drugim

```
size_t global_work_size = N;
clEnqueueNDRangeKernel(queue, kernel,
   1, NULL,              // broj dimenzija i posmak indeksa
   &global_work_size,    // ukupan broj dretvi
   NULL,                 // veličina radne grupe (1 ako se ne navede)
   NULL, NULL,           // definiranje opcionalnih događaja
   NULL);                // bez povratne vrijednosti
// blokirajuce cekanje na sve objekte u redu izvođenja
clFinish(queue);
```

- primjer: definiranje dvodimenzijske strukutre dretvi:

```
cl_uint dim = 2;
size_t global_offset[] = { 0, 0 };
size_t global_size[] = { 32, 32 };
size_t local_size[] = { 4, 4 };
clEnqueueNDRangeKernel(queue, kernel, dim, global_offset, global_size, local_size, 0,
NULL, NULL);
```

- **Čitanje izlaznih podataka**

```
clEnqueueReadBuffer(
   queue,
   c_buffer,             // objekt za citanje
   CL_TRUE,              // blokirajuce citanje
   0,                    // offset
   N * sizeof(double),   // velicina
   C,                    // pokazivac u host memoriji
   NULL, NULL,           // opcionalno
   NULL);
```

- ukoliko je čitanje blokirajuće (CL_TRUE), poziv čeka na završetak svih prethodnih naredbi u redu izvođenja i na čitanje zadanih podataka
- nakon čitanja izlaznih podataka, potrebno je osloboditi memoriju svih objekata (kontekst, red, program, kernel, argumenti)
- osim navedene C sintakse, postoji i C++ sintaksa te velik broj pomoćnih biblioteka za pripremu OpenCL kernela (vidi upute za DZ!)

7.3.4 Pristup memorijskim lokacijama

- unutar kernela koriste se različite oznake za različite vrste memorijskih lokacija:
- **global**: podatak se nalazi u globalnoj memoriji uređaja
  - svi ulazni podaci početno se nalaze u globalnoj memoriji
  - dohvatljivo kao argument kernel funkcije
  - svi *izlazni* podaci također ove vrste
- constant: također globalna memorija, ali označena samo za čitanje; ograničene veličine
- **local**: podaci se nalaze u lokalnoj memoriji jednog multiprocesora
  - isključivo statička alokacija (veličina mora biti poznata prilikom prevođenja)
  - podaci zajednički svim dretvama iste radne grupe
- **private**: podatak pripada samo jednoj dretvi
  - samo statička alokacija
  - obično se koristi za lokalne varijable u kernel funkcijama
- *default* je private: ako se ne navede druga oznaka, lokalne varijable su ove vrste
- **Vektorski tipovi podataka**
- OpenCL podržava i vektorske tipove podataka fiksne veličine (2, 3, 4, 8 i 16); podrška određenoj veličini ovisi o sklopovlju!

- u kernelima se koriste oznake char2, int4, uint8, float16 itd.
- u host aplikaciji dodaje se prekiks cl_: cl_char2, cl_int4, cl_uint8, cl_float16 itd.

7.3.5 Atomičke (nedjeljive) operacije

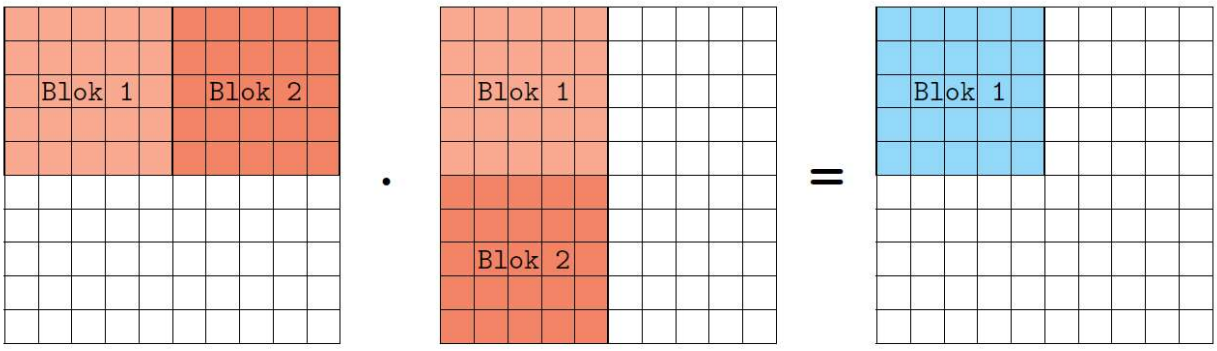
- prilikom izvođenja više dretvi, moguć je pokušaj pristupa do iste memorijske lokacije (u tom slučaju pisanje je slijedno, a rezultat je nedefiniran)
- ukoliko je potrebno, može se omogućiti izvođenje neprekidivih operacija nad zajedničkim podacima: atomičke instrukcije
- većina operacija definirana je samo za *cjelobrojne* podatke
- primjeri: `atomic_inc(int *x)` povećava vrijednost za jedan, `atomic_add(int *x, int val)` povećava za val; `atomic_max(int *x, int val)` zapisuje veću vrijednost, `atomic_xor(int *x, int val)` itd.
- atomičke instrukcije primjenjive su samo na globalnu ili lokalnu memoriju

7.3.6 Primjer: množenje matrica (lokalna memorija)

- pretpostavljamo množenje dvije kvadratne matrice A i B dimenzija NxN
- jednostavna inačica: svaka dretva računa jedan element rezultantne matrice:

```
kernel void matrix(global const double *A, global const double *B, global double *C,
const int N)
{
   int gx = get_global_id(0); // stupci (x smjer)
   int gy = get_global_id(1); // retci (y smjer)
   double sum = 0;
   for (int i = 0; i < N; i++) {
      double tempA = A[gy * N + i];
      double tempB = B[i * N + gx];
      sum += tempA * tempB;
   }
   C[gy * N + gx] = sum;
}
```

- učinkovitija izvedba dijeli matrice na blokove te računa međurezultate korištenjem kopija elemenata matrica u *lokalnoj* memoriji
- svaki blok matrice C računa se u broju iteracija koji je jednak broju blokova u jednoj dimenziji:



$A_{n\times n}$ $B_{n\times n}$ $O_{n\times n}$

- veličina bloka mora biti konstanta (zbog zauzimanja lokalne memorije!), a veličina matrica treba biti višekratnik veličine bloka (ili nadopuniti nulama)
- prepisivanje bloka u lokalnu memoriju sinkronizira se među svim dretvama radne grupe pozivom `barrier(CLK_LOCAL_MEM_FENCE)`

```
#define BLOCKSIZE 32 // poznato u trenutku prevodenja kernela
kernel void matrix2(global const double *A, global const double *B, global double *C,
const int N)
{
   uint lx = get_local_id(0);   // stupac unutar bloka
```

```
    uint ly = get_local_id(1);   // redak unutar bloka
    uint gx = get_group_id(0);   // koordinate bloka
    uint gy = get_group_id(1);
    uint n = get_num_groups(0);  // broj grupa (blokova) u jednom retku/stupcu
    // lokalna memorija za pohranjivanje blokova!
    local double tA[BLOCKSIZE][BLOCKSIZE];
    local double tB[BLOCKSIZE][BLOCKSIZE];

    // posmak za pocetak bloka u izvornim matricama
    uint iSubA = BLOCKSIZE * gy * N;
    uint iSubB = BLOCKSIZE * gx;

    double sum = 0;
    for (int i = 0; i < n; i++) { // za sve blokove u jednom retku/stupcu
       // kopiraju se blokovi matrica iz globalne memorije u lokalnu memoriju
       tA[ly][lx] = A[ly*N + lx + (iSubA + i* BLOCKSIZE)];
       tB[ly][lx] = B[ly*N + lx + (iSubB + i* BLOCKSIZE * N)];
       // sinkroniziraju se sve dretve u grupi!
       barrier(CLK_LOCAL_MEM_FENCE);

       // mnozenje dva bloka
       for (int k = 0; k < BLOCKSIZE; k++)
             sum += tA[ly][k] * tB[k][lx];
    }

    // pohrana u globalnu mem
    int globalIdx = get_global_id(0);
    int globalIdy = get_global_id(1);
    C[globalIdy * N + globalIdx] = sum;
}
```

- moguća su još učinkovitija dodatna poboljšanja…

8 Razvoj modularnih paralelnih programa

- **Ciljevi poglavlja**: opisati načine razvoja i kombiniranja modularnih paralelnih programa
- Literatura:
  1. "Designing and Building Parallel Programs", I. Foster, Addison-Wesley, 1995. (*online*) (http://www.mcs.anl.gov/dbpp/)
  2. MPI Standard 1.1 (http://www.mcs.anl.gov/mpi/standard.html)
  3. MPI Standard 4.0 (https://www.mpi-forum.org/)

8.1 Modularni razvoj

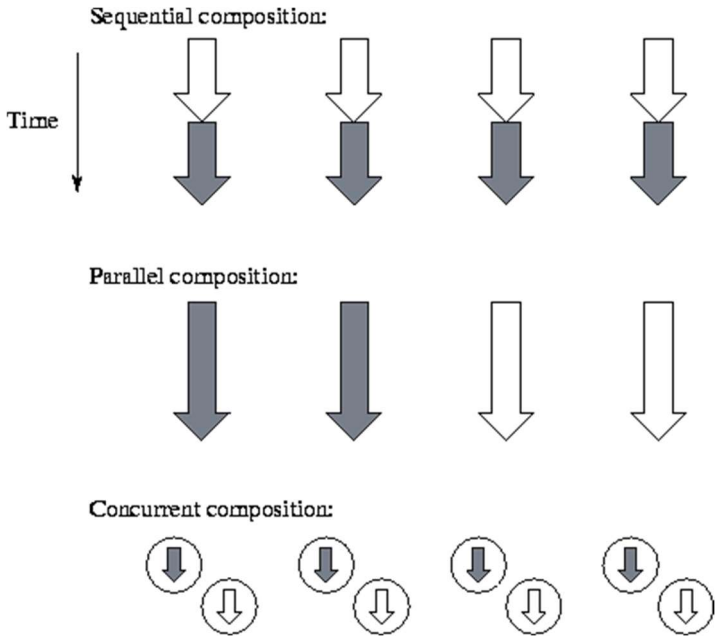
- paralelni programi se mogu oblikovati u svrhu korištenja unutar većeg paralelnog sustava
- svaki paralelni program je jedan *modul*; korištenje više modula unutar jednog sustava je *kompozicija* modula

8.1.1 Raspodjela podataka

- različiti paralelni moduli mogu koristiti različite raspodjele pri radu s istim podacima (npr. jedan modul koristi matrice podijeljene na stupce dok drugi koristi 2D podjelu)
- prilikom kompozicije nekompatibilnih modula, jedan od modula mora se promijeniti ili se mora uvesti eksplicitna pretvorba podataka između njih
- ponekad je module moguće definirati tako da budu *neutralni* po pitanju raspodjele podataka (npr. da omogućuju nekoliko načina podjele, ili se podjela može definirati tijekom izvođenja)
- u općenitom slučaju neutralnost po pitanju raspodjele teže je izvesti

8.2 Načini kompozicije modula

- prilikom razvoja paralelnog sustava, moduli se mogu kombinirati na više načina [1]:



8.2.1 Slijedna kompozicija

- u slijednoj kompoziciji, paralelni moduli se izvode slijedno jedan za drugim
- pogodno za SPMD paradigmu (jedan program slijedno poziva paralelne module)
- primjer: za simulaciju atmosferskog modela postoji modul koji obavlja jednu (ili više) iteracija računanja i paralelni modul koji provjerava kriterij završetka rada
- oba modula se zovu slijedno u petlji sve dok se ne dostigne uvjet završetka
- prednosti:
  - jednostavniji i čitljiviji programi
  - nema potrebe za dodatnom komunikacijom među modulima

8.2.2 Paralelna kompozicija

- u paralelnoj kompoziciji, različiti moduli izvode se istovremeno na različitim grupama računala (MPMD model)
- uporaba paralelne kompozicije može poboljšati svojstva skalabilnosti i lokalnosti: uz istovremeno izvođenje imamo više slobode u odabiru količine procesora za svaki modul
- moguće je i smanjenje potrebne količine memorije jer će se na pojedinom računalu izvoditi samo jedna vrsta modula

8.2.3 Zajednička kompozicija

- zajednička (*concurrent*) kompozicija je najopćenitiji način: različiti moduli odvijaju se istovremeno na istim procesorima
- odabir trenutnog modula na pojedinom računalu radi se na temelju dostupnih podataka od drugih modula (*data-driven computation*):
    - programski alat mora omogućavati promjenu aktivnog modula unutar procesora
    - svaki od modula se oblikuje uz pretpostavku da su drugi moduli u svakom trenutku dostupni (isprepleteni rad)
- prednosti zajedničke kompozicije:
    - unutarnje funkcioniranje modula je međusobno potpuno skriveno (postoje samo veze u obliku komunikacijskih kanala), pa se moduli mogu razvijati neovisno jedan o drugome
    - dodjeljivanje zadataka procesima može se definirati neovisno o pojedinim modulima (jer se mogu izvoditi na istim procesorima)
- nedostatak ovog modela mogu biti povećani troškovi zamjene konteksta na pojedinom računalu

8.3 MPI i modularno programiranje

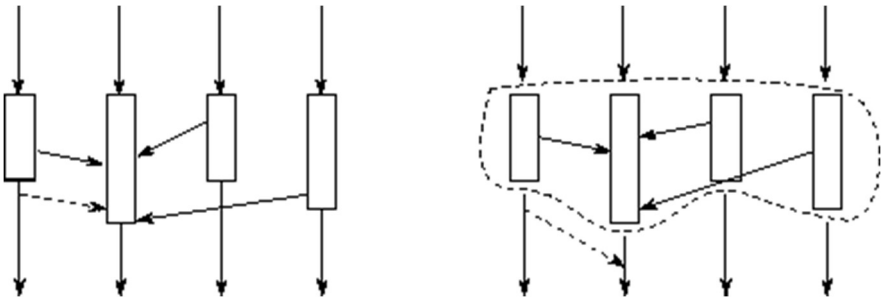
- u MPI standardu, modularnost i skrivanje podataka (enkapsulacija) postiže se uporabom **komunikatora**, koji definira grupu procesa – kontekst komunikacije – unutar kojega se razmjenjuju poruke (komunikator koji definira *sve* procese je MPI_COMM_WORLD)
- svaka MPI funkcija uključuje oznaku komunikatora, što omogućuje npr. da poruke između dva procesa ne dolaze u konflikt ako koriste različite komunikatore
- oznake poruka (*message tags*) ne mogu uvijek biti dovoljne za razlikovanje konteksta
- primjer: MPI program može koristiti postojeću biblioteku funkcija (npr. za paralelni rad s matricama); oznake poruka u našem programu morale bi biti različite od onih u biblioteci, što nije praktično za ostvarenje a ponekad i nije moguće ostvariti (npr. jer se oznake mogu izračunavati tijekom rada programa, ako se koristi promjenjiva količina poruka)

8.3.1 Kopiranje komunikatora

- stvaranje istovjetne kopije komunikatora radi se funkcijom:

```
int MPI_Comm_dup (MPI_Comm comm, MPI_Comm *comm_out)
```

- prvi argument je postojeći komunikator, a drugi je kopija prvoga
- stvaranjem novog komunikatora nastaje novi *kontekst* u kojemu se poruke razmjenjuju - u oba komunikatora nalaze se isti procesi, no poruke se ne mogu zamijeniti
- primjer uporabe: novi komunikator može se proslijediti kao argument nekom paralelnom modulu (funkciji iz bilblioteke) unutar kojega će biti korišten, bez bojazni od konflikata s porukama unutar postojećeg komunikatora



- mehanizam kopiranja komunikatora omogućuje *slijednu kompoziciju*

8.3.2 Dijeljenje komunikatora

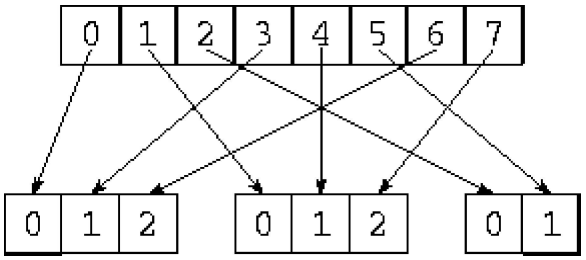
- jedna grupa procesa može se podijeliti na više podgrupa, od kojih svaka dobija vlastiti komunikator

```
int MPI_Comm_split (MPI_Comm comm, int color, int key,
                    MPI_Comm *comm_out)
```

- prvi argument je postojeći komunikator, drugi argument služi za oznaku određene podgrupe, parametar `key` služi za određivanje indeksa procesa unutar novog komunikatora, a posljednji argument je dobiveni komunikator
- navedena funkcija je *globalna*, što znači da je moraju pozvati svi procesi unutar postojećeg komunikatora!
- svi procesi koji su pozvali funkciju sa jednakim parametrom `color`, dobivaju isti novi komunikator (postaju članovi jedne podgrupe)
- ako proces ne želi dobiti novi komunikator (odnosno ne želi biti dodijeljen u novu grupu), tada za parametar color navodi konstantu MPI_UNDEFINED (a za novi komunikator može staviti NULL)
- primjer: stvaranje tri podgrupe procesa

```
MPI_Comm stari_comm, novi_comm;
int moj_id, color;
MPI_Comm_rank(stari_comm, &moj_id);
if(moj_id < 8)
   color = moj_id % 3;
else
   color = MPI_UNDEFINED;
MPI_Comm_split(stari_comm, color, moj_id, &novi_comm);
```

- svaki proces čiji je indeks manji od 8 ulazi u jednu od 3 nove grupe, dok procesi s većim indeksima ne ulaze u novi komunikator



- mehanizmom dijeljenja komunikatora podržana je paralelna kompozicija modula
- zajednička kompozicija nije podržana u MPI modelu (inačica standarda 1.1)

8.3.3 Stvaranje međukomunikatora

- **međukomunikator** omogućuje komunikaciju među procesima iz različitih grupa (načinjenih dijeljenjem)
- u komunikaciji među grupama podržana je samo individualna komunikacija (nema globalnih operacija)
- funkcija za stvaranje međukomunikatora je `MPI_Intercomm_create`

8.3.4 Ostale funkcije

- komunikatori stvoreni tijekom rada programa dealociraju se pozivom:

```
int MPI_Comm_free (MPI_Comm *comm)
```

- ukoliko postoji još neka nedovršena komunikacija koja koristi navedeni komunikator, ona se završava normalno
- osim navedenih, postoji još nekoliko funkcija za rad s komunikatorima (obično imaju oblik MPI_Comm_* te MPI_Group_*)