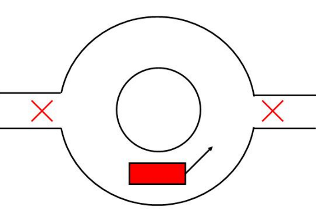
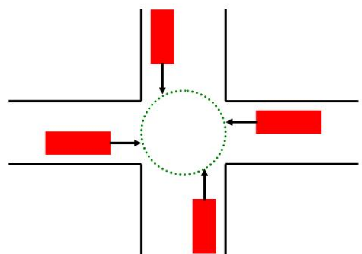
LYGIAGRETAUS PROGRAMAVIMO KONSPEKTUKAS

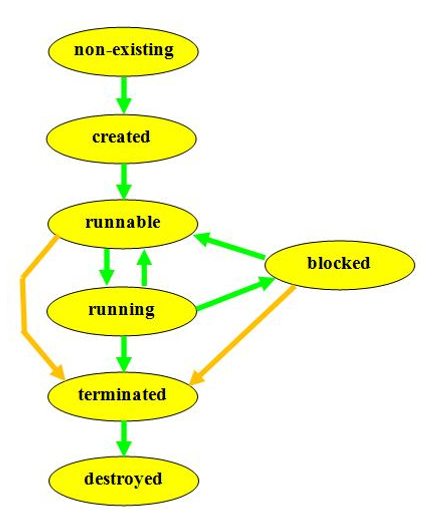
1. LP pagrindinės sąvokos

* **Lygiagrečiųjų programų poreikis ir nauda.**
  + programinis modelis geriau atitinka realųjį pasaulį;
  + daugiaprocesorinėje sistemoje galima padidinti vykdymo greitį; vienprocesorinėje sistemoje viena programa praktiškai neišnaudoja viso procesoriaus laiko;
  + iš anksto nustatyta veiksmu˛ tvarka kai kuriems taikymams yra nepriimtina.
  + Gamtos ir inžinerijos mokslų sudėtingų problemų sprendimas:
    - atmosferos reiškinių prognozavimas;
    - branduolinė ﬁzika, matematika;
    - kosminių aparatų˛ ir ginklų projektavimas;
    - Pramonė ir komercija:
    - naftos ir dujų gavyba;
    - farmacija;
    - ﬁnansai ir ekonominis modeliavimas;
* **Lygiagrečiojo programavimo problemos.** 
  + skirtingi rezultatai gali būti gauti vykdant programa˛net ir su tais pačiais duomenimis;
  + vartotoją tenkina tik tam tikri rezultatai, o tuo pačiu ir tik tam tikros veiksmų atlikimo sekos;
  + reikia uždrausti nenaudingas vykdymo sekas, t. y. sinchronizuoti procesus; procesai turi keistis informacija.
* **Atominiai veiksmai, kritinės sekcijos ir tarpusavio išskyrimas, badavimas, aklavietės ir begaliniai ciklai.**
  + - atominis veiksmas (atomic action) – veiksmas, kuriam vykstant nevyksta kitų procesų veiksmai;
    - kritinė sekcija (critical section) - programos kodo gabalas, kuris kitu˛ procesu˛ atžvilgiu turi būti atominiu veiksmu;
    - tarpusavio išskyrimas (mutual exclusion) - priemonės, neleidžiančios dviems procesams vienu metu vykdyti savo veiksmus (naudotis tuo pačiu resursu);
    - dviejų procesų˛ KS turi būti vykdomos su tarpusavio išskyrimu (mutual exclusion);
    - varžymosi būsena (race condition) – tai būsena, kai procesai vienu metu varžosi dėl kokių tai resursų
    - užimtas laukimas (busy wait) – būsena, kai procesas laukia kokio tai resurso pastoviai tikrindamas jo užimtuma˛;
    - badavimas (starvation) – tai būsena, kai procesui pastoviai neleidžiama naudotis bendrais resursais.
    - aklavietė (deadlock) – tai būsena, kai procesas laukia to, kas niekada neįvyks;
    - begalinis ciklas (livelock) – tai būsena, kai procesas cikliškai vykdo tuos pačius veiksmus be jokio progreso;

1. **Lygiagretieji procesai**

* **Lygiagrečiųjų procesų sąvoka.** 
  + tvarkos abstrakcija;
  + užrašymo tvarka nenurodo vykdymo tvarkos:
    - P ||R ↔∃v : ne P →R ir ne R →P (P,R – sakiniai)
    - P || R ↔∃v : ne x →y ir ne y →x (x,y – komandos)
  + tie patys duomenys – galimi skirtingi rezultatai; vartotoja˛tenkina tik tam tikri rezultatai.
  + procesas – tai programos kodo fragmentas, nuosekliai vykdomas viename procesoriuje;
  + LP: procesu˛ sk. ≥ﬁziniu˛ procesoriu˛ sk.,
  + procesu˛ sk. = loginiu˛ procesoriu˛ sk.
  + **Lygiagretumo variantai**
    - funkcinis lygiagretumas;
    - duomenu˛ lygiagretumas.
* **Lygiagrečiųjų procesų tipai.**
  + statiniai ir dinaminiai procesai
  + vieno ir daugelio lygiu˛ procesai
  + procesu˛ inicializavimo ir užbaigimo skirtumai
* **Lygiagrečiųju˛ procesų savybės.**

komutatyvumas: P || R ⇒R ||P



asociatyvumas: P || (R ||Q) ⇒ (P ||R) ||Q

tranzityvumas: P || R ir R || Q ⇒P || Q ⇒P || R || Q

* **Lygiagrečiujų procesų būsenos.**
  + neegzistuojąs (non-existing)
  + sukurtas (created)
  + pasiruošęs (ready, runnable)
  + vykstąs (running)
  + blokuotas (blocked)
  + nutrauktas (terminated, dead)
  + sunaikintas (destroyed)
* **Lygiagrečiųjų procesų sąveikos tipai.** 
  + **procesai nepriklausomi,** 
    - LP kalboje – tik priemonės procesams kurti, vykdyti ir baigti.
    - Priemonės procesų sąveikai valdyti – nereikalingos.
    - Variantas retai pasitaikantis.
  + **procesai varžosi tarpusavyje**
    - LP kalboje – priemonės procesams kurti, vykdyti ir baigti.
    - LP kalboje – priemonės procesų prioritetams nustatyti.
    - Variantas, būdingas sistemoms, kuriose procesai sprendžia nepriklausomus uždavinius.
  + **procesai bendradarbiauja tarpusavyje.**
    - LP kalboje – priemonės procesams kurti, vykdyti ir baigti.
    - LP kalboje – priemonės procesų sąveikai valdyti.
    - Variantas, būdingas programoms, kuriose procesai sprendžia bendrą uždavinį
* **Lygiagrečiųjų procesų komunikavimas ir sinchronizavimas.**
  + **OS:**
    - 1 programa⇒1 procesas
    - 1 procesas⇒daug (≥1) giju˛
    - procesai neturi bendros atminties giju˛
    - atmintis bendra
  + **LP:**
    - 1 programa⇒daug (≥1) procesu˛
    - 1 procesas≡1 gija
  + **Bendra procesų atmintis:**
    - procesai komunikuoja per bendrus kintamuosius;
    - procesai gali būti sinchronizuojami naudojant bendrus kintamuosius;
    - problema: bendrų kintamųju˛ apsauga.
  + **Atskira procesų atmintis:**
    - procesai komunikuoja siųsdami ir priimdami pranešimus;
    - procesai sinchronizuojami siunčiant sinchronizavimo signalus;
    - problema: priimti pranešimus iš kelių procesų˛.
  + **Atminties modelis ir kompiuterio architektūra:**
    - programos architektūros modelis (bendra ar atskira atmintis) turi būti nepriklausomas nuo kompiuterio architektūros;
    - programa vykdoma efektyviai, jei programos modelis sutampa su kompiuterio architektūros modeliu.

**Sinchronizavimas:**

* Procesai laukia, kartodami kokius nors veiksmus.
* Procesai laukia, neatlikdami jokių veiksmų (yra blokuojami).

1. **Java procesai (gijos)**

* Gijos˛itrauktos˛i bazine˛ Java kalbos speciﬁkacija˛.
* Gija – lengvasvoris procesas, kurio viduje veiksmai vykdomi nuosekliai.
* Giju˛ naudojimas sudaro galimybe˛ tuo pačiu metu (lygiagrečiai) vykdyti kelias programos užduotis.
* Programoje visada yra bent viena gija: main funkcija – taip pat gija.
* Kiekviena Java virtualioji mašina (JVM) privalo palaikyti gijas.
* **Kūrimas ir inicializavimas.** 
  + Panaudojant (paveldint) java.lang.Thread klase˛.
  + Panaudojant (realizuojant) java.lang.Runnable sasaja˛

class Gija extends Thread

{

. . .

public void run()

{

. . .

}

} .

. . Gija g = new Gija();

g.start();

//-------------------------------

class Gija implements Runnable

{

. . .

public void run()

{

. . .

}

}

. . .

Thread g = new Thread(new Gija());

g.start();

* **Vykdymas.** 
  + Programa lygiagrečiai (concurrent) gali vykdyti bet kokį skaičių gijų.
  + Gijos vykdymas pradedamas, iškvietus gijos-objekto metoda˛ start.
  + Gijos vykdymo metu nuosekliai atliekami veiksmai, užrašyti gijos klasės metode run.
  + Visos programos gijos gali atlikti skirtingus veiksmus (jei jos yra skirtingu˛ gijų klasių objektai).
  + Jei programuotojas netaiko papildomu˛ priemonių, gijos-objekto vykdymas nepriklauso nuo kitų˛ gijų vykdymo.
  + Programose run metodas neturėtų būti kviečiamas tiesiogiai. Sistema iškvies jį pati.
  + Jei run metodas kviečiamas tiesiogiai, programos kodas vykdomas ne lygiagrečiai, bet nuosekliai.
* **Nutraukimas.**

public final void join() throws InterruptedException Waits for this thread to die.

* **Bendros atminties naudojimas.**
  + Java, OpenMP gijos gali tureti bendrus kintamuosius;
* **Komunikavimas ir sinchronizavimas.**
  + Tarpusavio išskyrimas (mutual exclusion) - "užrakinant" objekta˛su synchronized (realizuoja JVM).
  + Salygine sinchronizacija (condition synchronization) – naudojant Object klases wait, notify, notifyAll metodus. Sąlyginė sinchronizacija tai sincronizacijos procesas, kuomet viena gija siunčia pranešimą kitai gijai, kai tenkinama tam tikra sąlyga
  + notify() Wakes up a single thread that is waiting on this object’s monitor. notifyAll() Wakes up all threads that are waiting on this object’s monitor.
  + wait() Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
  + wait(long timeout) Waits for notiﬁcation or until the timeout period has elapsed. timeout is measured in milliseconds

1. **KS apsauga ir sąlyginės sinchronizacija**

Jei vienas procesas yra NKS, o kitas nori įeiti˛i KS, reikia leisti jam tai daryti.

Jei du procesai varžosi dėl perėjimo˛į KS, sprendimas, kuriam procesui leisti tai daryti, negali būti atidėtas neapibrėžtam laikui.

* **Peterson’o algoritmo analizė.**

Thread P1: run { | Thread P2: run {

while (true) { | while (true) {

NKS1; | NKS1;

flag1 = true; | flag2 = true;

n = 2; | n = 1;

while(flag2&&n==2) | while(flag1&&n==1)

{} | {}

KS; | KS;

flag1 = false; | flag2 = false;

NKS2; | NKS2;

} | }

} | }

|  |  |
| --- | --- |
| bool flag[2] = {false, false};  int turn; | |
| P0: flag[0] = true;  P0\_gate: turn = 1;  **while** (flag[1] && turn == 1)  {  *// busy wait*  }  *// critical section*  ...  *// end of critical section*  flag[0] = false; |  |

P1: flag[1] = true;

P1\_gate: turn = 0;

**while** (flag[0] && turn == 0)

{

*// busy wait*

}

*// critical section*

...

*// end of critical section*

flag[1] ž false;

Petersono algoritmas tarpusavio išskyrimui leidžia dviems ar daugiau procesų dalintis vienkartinio panaudojimo resursus išvengiant konfliktų, bendrą atmintį naudoja tik komunikavimui. Naudojami 2 kintamieji flag ir turn. Flag[n] true reikšmė nurodo, kad n procesas nori patektį į kritinę sekciją. Įėjimas leidžiamas procesui P0, jei

P1 nenori tuo metu į ją patekti arba jei P1 suteikė pirmenybę P0 nustatydamas turn reikšmę į 0.

Algoritmas patenkina pagrindinius kriteriujus kritinės sekcijos problemos išsprendimui, kadangi turn,flag[0] ir flag[1] reikšmės keičiasi iškart ir atomiškai. Trys kriterijai yra tarpusavio išskyrimas, progresavimas ir ribotas laukimas.

Tarpusavio išskyrimas(tik vienas procesas gali būti KS vienu metu):

P0 ir P1 niekada negali būti kritinėje sekcijoje tuo pačio momentu. Jei P0 sekcijoje tai flag[0] yra true. Taip pat

Progresas(no deadlock):

Jei nei vienas procesas nevykdo kritinės sekcijos ir yra norinčių į ją patekti, tuomet tik tie procesai, kurie tuo metu nevykdo likusių sekcijų gali dalyvauti sprendžiant, kuris procesas pateks į kritinę sekciją sekantis.

Apribotas laukimas(bounded waiting, no livelock)

Algoritme procesas niekada nelauks daugiau nei 1 kartą norėdamas patekti į KS.

* **Bakery algoritmo analizė.**

// procesu˛ skaičius

int n;

// rodo, kuris procesas ima bilieta˛

boolean choosing [n];

// procesu˛ bilietu˛ nr. (prad. reikšmės = 0)

int ticket [n];

// Thread Pk: prieš KS:

choosing[k] = true; // bilieto pasirinkimas

ticket[k] = maxValue(ticket) + 1;

choosing[k] = false;

for (int i=0; i<n; i++) {// laukti lyginant savo

if (i == k)

continue; // bilieta˛ su kt.proc.bil.

while (choosing [i]);

while (ticket[i]!=0 && ticket[i]<ticket[k]);

if (ticket[i] == ticket[k] && i<k)

while (ticket[i] != 0); }

// už KS:

ticket[k] = 0;

Lamporto kepyklos algoritmas yra skirtas apsaugoti bendrų resurų dąlinimąsi tarp kelių ar daugiau gijų, naudojant tarpusavio išskyrimą(mutual exclusion)

Tai leidžia išvengti duomenų sugadinimo( data corruption), kuris gali įvykti, kuomet dvi ar daugiau gijų bando rašyti į tą pačią atminties vietą. Algoritmas sukurtas taip,kad būtų išvengiama kelių gijų patekimo į kritinė sekciją tuo pačiu metu ir taip sumažinant duomenų sugadinimo tikimybę.

Taip pat algorimte vyrauja tikimybė, kad kelios gijos gaus tuos pačius numerius, tad į pagalbą pasitelkiama ir gijos numeris, kuo mažesnis gijos numeris tuo greičiau ta gija bus aptarnauta.

Bakery koncepcija:

Galvokime apie populiarią parduotuvę su apkrautu klientų skaitikliu.

Žmonės ima bilietą iš mašinos.

Jei niekas nelaukia, bilietai neturi prasmės

Jei keli laukia, bilieto eiliškumas nusako, kieno eilė apsipirkti.

Bakery algoritmas gerai tinka kur busy-waiting nėra itin neefektyvus.

Pavyzdžiui, jei turima pakankamai CPU kad kiekviena gija turėtų savo.

Algoritmas isprendia overflow jeigu bilietas perdidelis nustatant ji i 0 ir bandant is naujo

* **Sąlyginės sinchronizacijos realizavimas nuoseklaus programavimo priemonėmis.**

// thread | // thread

class Siuntėjas { | class Gavėjas {

int m; | int d;

void run() { | void run() {

Formuoti(m); | while (!BK2.t);

while (BK2.t);| d=BK2.v;

BK2.v=m; | BK2.t=false;

BK2.t=true; | Naudoti(d);

} } | } }

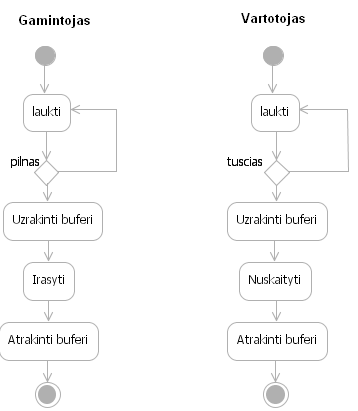
class BK2 {

int v; boolean t = false;

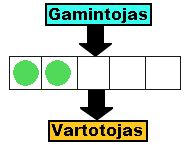
void main() {

... }

1. LP uždaviniai

* **Dekoratyvinis sodas.**
* **Gamintojas – Vartotojas.**
* 

Gamintojo-vartotojo problema – tai dar viena problema, parodanti, jog sistemose, kurios naudojasi bendrais resursais būtina gijų sinchronizacija. Tarkime turime bendrai naudojamą fiksuoto dydžio buferį. Viena gija – gamintojas – rašo informaciją į buferį, kita – vartotojas – skaito informaciją iš buferio. Abi gijos veikia lygiagrečiai. Problema susidaro tuomet, kai gija bando įrašyti į pilną buferį arba nuskaityti iš tuščio.



1. Gamintojo-vartotojo problema

Taigi tam, kad nekiltų problemų naudojantis bendru buferiu, t.y. nesusidarytų kova dėl resursų, gijos turi būti teisingai sinchronizuotos. Gamintojas gali įrašinėti į buferį tik tuomet, kai jis nėra pilnas. Vartotojas gali skaityti iš buferio tada, kai jis nėra tuščias. Taigi tiek gamintojas tiek vartotojas gali laukti, kol šios sąlygos bus tenkinamos.

* **Skaitantieji – Rašantieji.**

Readers/writers problem: – Allow multiple readers to concurrently access a data – Allow only one writer at a time • Two possible solutions using semaphores – Favor readers – Favor writers • Starvation is possible in either case!

Consider a situation where we have a file shared between many people.

* If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
* However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem**

Problem parameters:

* One set of data is shared among a number of processes
* Once a writer is ready, it performs its write. Only one writer may write at a time
* If a process is writing, no other process can read it
* If at least one reader is reading, no other process can write
* Readers may not write and only read

**Solution when Reader has the Priority over Writer**

Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: **mutex, wrt, readcnt** to implement solution

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers
2. **int** readcnt;  //    **readcnt** tells the number of processes performing read in the critical section, initially 0

**Functions for sempahore :**

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

**Writer process:**

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

do {

// writer requests for critical section

wait(wrt);

// performs the write

// leaves the critical section

signal(wrt);

} while(true);

**Reader process:**

1. Reader requests the entry to critical section.
2. If allowed:
   * it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
   * It then, signals mutex as any other reader is allowed to enter while others are already reading.
   * After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.
3. If not allowed, it keeps on waiting.
4. do {
6. // Reader wants to enter the critical section
7. wait(mutex);
8. // The number of readers has now increased by 1
9. readcnt++;
10. // there is atleast one reader in the critical section
11. **// this ensure no writer can enter if there is even one reader**
12. **// thus we give preference to readers here**
13. if (readcnt==1)
14. wait(wrt);
15. // other readers can enter while this current reader is inside
16. // the critical section
17. signal(mutex);
18. // current reader performs reading here
19. wait(mutex);   // a reader wants to leave
20. readcnt--;
21. // that is, no reader is left in the critical section,
22. if (readcnt == 0)
23. signal(wrt);         // writers can enter
24. signal(mutex); // reader leaves
25. } while(true);
26. Thus, the mutex ‘**wrt**‘ is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

* **Pietaujantys ﬁlosofai.**

N filosofų sėdi prie apvalaus stalo. Tarp kiekvieno greta sėdinčio filosofo yra šakutė. Kitaip tariant yra N šakučių. Kiekvienas filosofas arba galvoja, arba valgo. Filosofas galvoja tol, kol išalksta. Tam, kad jis galėtų valgyti reikalingos dvi šakutės. Taigi išalkęs jis pasiima dvi šakutes ir pradeda valgyti. Kai filosofas pavalgo, jis padeda šakutes atgal ir ima vėl galvoti.

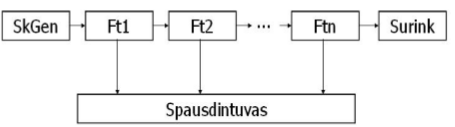
Esminis veiksmas yra „paimti šakutes“. Filosofai negali paimti tos pačios šakutės(resurso) tuo pat metu, kadangi susidarys kova dėl resursų. Be to filosofai negali laukti šakutės, kurios niekada negaus. Tam, kad išvengtume šių problemų tarkime, kad kiekviena šakutė yra bendrai naudojamas resursas, kuris turi būti apsaugomas užraktu. Tuomet filosofas gaus šakutę tik tuomet, jeigu ji bus laisvas, t.y. užraktas nebus užrakinti. Žemiau pateikiama veiklos diagrama.

****

Kiekviena šakutė yra bendrai naudojamas resursas. Todėl gali susidaryti kova dėl resursų, t.y. jeigu filosofas bandys paimti šakutę vienu metu. Kadangi filosofai tarpusavyje nebendrauja, gali susidaryti aklavietė, pavyzdžiui jeigu visi filosofai išalks vienu metu pasiims po šakutę ir lauks antros. Taip pat gali susidaryti badavimas: jeigu visą laiką valgys vienas prieš kitą sėdintys filosofai, o kiti trys nespės paimti šakučių, tuomet jie niekuomet nevalgys. Filosofų atliekami vieksmai yra ciklikliški. Filosofas paima šakutes, pavalgo, padeda šakutes, masto ir vėl kartoja tuos pačius veiksmus.

1. Lygiagretieji algoritmai

* **Pirminiu˛ skaičių generavimas (1 alg.).**



Vienas iš pirminių skaičių genervimo algoritmų yra naudojant Eratosteno rėtį. Tai yra senai žinomas algoritmas pirminų skaičių radimui. Jo esmė yra išrašyti skaičius nuo 2 iki pasirinkto n. Tada pašaliname 2 kartotinius, sekantis skaičius 3 yra pirminis. Tuomet šaliname 3 kartotinius, tuomet 5 yra pirminis , šaliname 5 kartotinius ir t.t. Algoritmą galime realizuoti naudojant CSP kanalus.

Pradedame sukurdami skaičių generatoriaus procesą, jo funkcija yra sugeneruoti skaičių x , kuris yra nuo 2 iki n ir jį išsiųsti.

Kiekvienam gautam skaičiui x sukuriame filtruojanti procesą, jo paskirtis yra perduoti pirminį skaičių x spausdinančiam procesui ir pašalinti visus x skaičiaus kartotinius, tuomet perduoti likusias reikšmes toliau.

Filtravimas vykdomas kol gaunami visi skaičiai.

* **Matricos ir vektoriaus daugyba (1 alg.).**

Jeigu turime matricą [ 𝑎00 … 𝑎0𝑛 … ⋱ … 𝑎𝑚0 … 𝑎𝑚𝑛 ] ir vektorių [ 𝑥0 … 𝑥𝑛 ], tai jų sandauga yra vektorius [ 𝑎00𝑥0 + 𝑎01𝑥1 + ⋯ + 𝑎0𝑛𝑥𝑛 … 𝑎𝑚0𝑥0 + 𝑎𝑚1𝑥1 + ⋯ + 𝑎𝑚𝑛𝑥𝑛 ]. Turime n\*m gijų ir n kanalų, kur gijai suskaičiuoja sandaugą aij \*xj ir siunčia ją į j-ąjį kanalą. Toliau yra dar n gijų, kur gijaisuskaičiuoja k-ojo kanalo skaičių sumą ir padeda į k-ąją rezultatų vektoriaus poziciją.

* **Matricu˛ daugyba, Tiesiniu˛ lygčių˛ sistemų sprendimas, Masyvų tvarkymas, Grafo trumpiausių kelių radimas ir kt.(1 alg. pasirinktinai).**
* **Masyvų tvarkymas (2 alg.).**

1. Bendra atmintis. Monitoriai

* **Monitorių paskirtis ir realizavimas.** 
  + kritinis regionas, monitorius – programos kodo dalis, kuri visada vykdoma su tarpusavio išskyrimu.
  + Jis užtikrina, jog tik vienai gijai leidžiama vykdyti monitoriaus kodą vienu metu.
  + Monitor =įspėjėjas (lot.) –įtaisas kam nors stebėti, testuoti, užrašinėti.
  + Monitorių sudaro:
    - bendrieji saugomi duomenys;
    - atominių veiksmų (metodų) skirtų saugomiems duomenims apdoroti, rinkinys;
    - sąlyginių kintamųjų˛ rinkinys.
      * condition tipas; // nonbusy:condition;
      * nonbusy.wait; // išlaisvinti monitoriu˛ ir blokuoti procesa˛,

// vykdantį wait, prie sąlygos nonbusy

* nonbusy.signal; // išlaisvinti viena˛procesa˛iš laukiančiųjų˛ // prie sąlygos nonbusy
* visi procesu˛ atominiai veiksmai yra vienoje vietoje;

monitoriaus viduje gali būti kreipiniai˛i kitus monitorius.

* metodas, vykdomas vieno monitoriaus viduje, gali kviesti kito monitoriaus metoda˛;
* procesas, vykdantis signal, yra monitoriuje, o procesas, išblokuojamas signal, turi grįžti˛i monitoriu˛ (signal-and-exit ar signal-and-continue).
* **Realizacija**
* Kiekvienas Java objektas turi su juo susietą monitorių.
* Kiekviena Java klasė turi su ja susietą monitorių.
* Monitorius nerealizuojamas, jei nenaudojami sinchronizuoti (synchronized) metodai.

Jeigu monitoriaus saugomas kodas yra užimtas, tuomet java gija pereina į laukimo būseną, kol jis bus atlaisvintas. Yra du būdai objekto užrakinimui[4].

1. Kviečiant sinchronizuotą bloką.

synchronized (anObject) { ... }

Čia užrakinamas obejktas anObject. Užraktas atlaisvinamas tuomet, kai baigiamas kodo vykdymas sinchronizuotame bloke. Jeigu kita gija jau vykdo kodą sinchronizuotame bloke, tuomet gija yra blokuojama kol blokas atsilaisvins.

1. Metodo sinchronizavimas

public synchronized void aMethod() { ... }

Metodo sinchronizavimas, tai tas pats kad objekto *this* užrakinimas sinchronizuotame bloke.

* **Monitorių naudojimas KS apsaugai ir sąlyginei sinchronizacijai**.

Kiekvienas monitorius turi užraktą. Jeigu gija bando naudoti monitorių ir jeigu užraktas yra užimtas tuomet gija turi laukti kol jis bus atlaisvintas arba baigia darbą.

* + Tarpusavio išskyrimas (mutual exclusion) - "užrakinant" objekta˛su synchronized (realizuoja JVM).
  + Sąlyginė sinchronizacija (condition synchronization) – naudojant Object klasės wait, notify, notifyAll metodus.
  + Monitorius – tai užraktas (lock), nustatantis objekto (klasės) panaudojimo tvarką ir vienu metu leidžiantis jį naudoti tik vienai gijai.
* **Monitorių naudojimas LP uždaviniams spręsti.**
* **Monitorių (kritinių regionų) realizavimo Java ir OpenMP ypatybės.**
* Multiple ways of guarding critical section.
* **1. synchronized methods:**
* public synchronized void incrementCounter(){
* ++counter;
* }
* **2. synchronized statements**
* public int getCounter(){
* synchronized(this){
* return counter;
* }
* }
* **3. Lock API**
* class X {
* private final ReentrantLock lock = new ReentrantLock();
* // ...
* public void m() {
* lock.lock(); // block until condition holds
* try {
* // ... method body, which is critical section
* } finally {
* lock.unlock()
* }
* }
* }

**OpenMP**

#pragma omp parallel shared(c, n)

{

for (int i=0; i<n; i++)

{

#pragma omp critical

{

int a = c; a++; c = a;

}

}

}

KS su lock

omp\_lock\_t \*raktas;

raktas = new omp\_lock\_t;

omp\_init\_lock(raktas);

// ---- Lygiagretus kodas ---

#pragma omp parallel shared(c, cikluSk) {

for (int i=0; i<cikluSk; i++) {

omp\_set\_lock(raktas);

int a = c; a++; c = a;

omp\_unset\_lock(raktas);

}

}

omp\_destroy\_lock(raktas);

1. Bendra atmintis. Semaforai ir užraktai

* Semaforų paskirtis ir tipai.

Kiekviena gija, prieš tai darydama, turi gauti iš semaforo leidimą (kviesdama semaforo leidimo pasiėmimo metodą), o baigusi darbą su saugomu objektu – leidimą grąžinti (kviesdama leidimo grąžinimo metodą). Jei semafore daugiau šiuo metu leidimų nebėra (jo likusių leidimų skaitiklis lygus nuliui), leidimo prašanti gija blokuojama kol kokia nors kita gija anksčiau pasiimtą leidimą grąžins.

Jei semaforas turi tik vieną leidimą, jo darbas daug nesiskiria nuo [užrakto](https://lt.wikipedia.org/wiki/U%C5%BEraktas_(programavimas)). Tačiau, skirtingai nuo užraktų, semaforai paprastai leidžia grąžinti leidimą ir ne tai gijai, kuri jį pasiėmė. Tai gali būti reikalinga, pavyzdžiui, išsprendžiant kai kuriuos sinchronizacijos aklavietės atvejus. Tačiau šiai galimybei reikalingi sinchronizavimo veiksmai gali sąlygoti lėtesnį darbą nei naudojant užraktus.

Skirtingai nuo užraktų, semaforai neskirti pakartotinam rakinimui ta pačia gija bei atskiriems rakinimams skaitymui bei rašymui.

Kaip ir užraktas, semaforas gali būti garbingas ([angl.](https://lt.wikipedia.org/wiki/Angl%C5%B3_kalba) *fair*) arba ne. Garbingas semaforas pirmiausia suteikia leidimą tai gijai, kuri jo laukė ilgiausiai. Garbingo semaforo kodas yra sudėtingesnis ir vykdomas ilgiau, todėl naudoti jį verta ne visada.

* struktūra, turinti ją sukuriant nustatytą leidimų skaičių vykdymo gijoms dirbti su saugoma duomenų struktūra ar vykdyti saugomą kodo sekciją.
* Semaforai naudojami, kuomet pageidaujama, jog su tam tikra duomenų struktūra ar kompiuterio įrenginiu nedirbtų daugiau nei nurodytas ribotas skaičius lygiagrečių gijų.

**Tipai:**

priklausomai nuo reikšmiu˛ rinkinio: bendrieji (general, counting semaphore); dvejetainiai (binary semaphore);

priklausomai nuo veikimo: "šališki"(unfair) "beveik bešališki"(weakly fair) "bešališki"(fair)

* **Semaforų savybės, galimi veiksmai.**
  + blocked waiting laukimas:
  + FIFO: P –˛i eiles gala˛, V – eil˙es pr.,
  + RANDOM: P – atsitikt., V – atsitikt.;
  + daugumoje kalbų realizacijų: s ≥0;
  + s = s0+ #V −#P;
  + **veiksmai:**
  + s.P() operacija: jei s reikšme = 0, blokuoti procesą ties s, priešingu atveju – sumažinti s reikšme˛;
  + s.V() operacija: jei yra blokuotų procesų ties s, išblokuoti vieną iš jų, priešingu atveju – padidinti s reikšmę˛;
  + s.init(a) operacija: semaforui s suteikti pradinę reikšmę a.
* **Semaforų taikymas KS apsaugai ir sąlyginei sinchronizacijai realizuoti.**

class G extends Thread {

void run() {

while (true) {

NKS1;

s.P(); // veiksmai prieš KS;

KS;

s.V(); // veiksmai už KS;

NKS2; }

}

}

* **Semaforų naudojimas LP uždaviniams spręsti.**

// thread | // thread

class Siuntejas { | class Gavejas {

int m; | int d;

void run() { | void run() {

Formuoti(m); | ??? // while (!BK2.t);

??? // while (BK2.t); | d=BK2.v;

BK2.v=m; | ??? // BK2.t=false;

??? // BK2.t=true; | Naudoti(d);

} } | } }

class BK2 {

int v; ??? //

boolean t = false;

void main()

{ ... }

* **Semaforų (užraktų) realizavimo Java ir OpenMP ypatybės.**
  + import java.util.concurrent.Semaphore;
  + Semaphore s = new Semaphore(p);

Creates s with the given number of permits (int p) and nonfair fairness setting. Semaphore s = new Semaphore(p, f);

Creates s with the given number of permits (int p) and the given fairness (boolean f) setting.

* + s.acquire();

Acquires a permit from s, blocking until one is available, or the thread is interrupted.

* + s.acquire(p);

Acquires the given number of permits (int p) from s, blocking until all are available, or the thread is interrupted. s.release(); Releases a permit, returning it to the s.

* + s.release(p);

Releases the given number of permits (int p), returning them to the s.

* + s.isFair();

Returns true, if s has fairness set true.

* + s.availablePermits();

Returns the current number (int) of permits available in s.

1. Žinučių perdavimas kanalais

* Žinučių perdavimo modeliai ir galimi variantai.
* **Kanalai: tipai, savybės, galimi veiksmai.**

CSP procese tiek duomenys, tiek ju˛ apdorojimo metodai yra privatus.

**"many-to-one"**

Komunikavimas "many-to-one": galima supaprastinti programos koda˛.

* public class Channel
  + public static Any2OneChannel any2one()
  + public static Any2OneChannelInt any2oneInt()
* public interface Any2OneChannel
  + AltingChannelInput in()
  + ChannelOutput out()
* public interface Any2OneChannelInt
  + AltingChannelInputInt in()
  + ChannelOutputInt out()

**"point-to-point"**

Komunikavimas "**one-to-one**": galima sužinoti, kas siunte.

* public interface ChannelOutput
  + void write(Object objektas)
* public interface ChannelOutputInt
  + void write(int i)
* public interface ChannelInput
  + Object read()
* public interface ChannelInputInt
  + int read()
* public class Channel
  + public static One2OneChannel one2one()
  + public static One2OneChannelInt one2oneInt()
* public interface One2OneChannel
  + AltingChannelInput in()
  + ChannelOutput out()
* public interface One2OneChannelInt
  + AltingChannelInputInt in()
  + ChannelOutputInt out()
* **Selektyvus laukimas, alternatyvos ir saugai.**
* public abstract class Guard
  + Guard();
* public class CSTimer extends Guard
  + public void after(long msecs);
  + public long read();
  + public void sleep(long msecs);
  + public void setAlarm(long msecs);
  + public long getAlarm();
* public class Skip extends Guard
  + public void run();
* public class Stop extends Guard
  + public void run();
  + public abstract class AltingChannelInput extends Guard
    - public abstract boolean pending();
  + public abstract class AltingChannelInputInt extends Guard
    - public abstract boolean pending();
  + public class Alternative
    - Alternative(Guard[] guard);
    - int select();
    - int fairSelect();
    - int priSelect();
    - int select(boolean[] preCondition);
    - int fairSelect(boolean[] preCondition);
    - int priSelect(boolean[] preCondition);
* **Kanalų naudojimas LP uždaviniams spręsti.**

1. Ada kalbos žinučių˛ perdavimo modelis

* **Modelio realizavimas ir ypatybės.** 
  + Remote Invocation (RI)
  + dvikryptis duomenu˛ perdavimas;
  + komunikavimo tipas: "daug-vienam" ("many-to-one");
  + komunikavimas tarp procesu˛ - be tarpininku˛;
  + galimas selektyvus (alternatyvus) laukimas.
* **Įeigos ir ju˛ naudojimas; skirtumai tarp įeigu˛ ir procedūrų.** 
  + naudojimas:
    - ta˛pacia˛procedura˛galima kviesti skirtingose vietose,
    - procesas gali priimti kvietimus iš kitu˛ procesu˛;
  + informacijos perdavimas:
    - kreipinys˛i procedura˛gali perduoti informacija˛tiek˛i procedura˛, tiek iš proceduros,
    - Remote Invocation metu informacija˛galima perduoti tiek procesui-priemejui, tiek procesui-siuntejui;
  + vykdymo salygos:
    - proceduros veiksmai vykdomi visada, kai tik aptinkamas kreipinys, RI kvietimas vykdomas tik tada, kai procesas-savininkas tai leidžia;
  + vykdytojas:
    - proceduros veiksmai atliekami jos kvietejo vardu,
    - kvietimo metu nurodytus veiksmus vykdo procesas-savininkas
* **Selektyvus laukimas ir įeigos.** 
  + entry\_aprašas ::= entry entry\_vardas[(formal¯us\_parametrai)] ;
  + entry\_kreipinys ::= proceso\_vardas.entry\_vardas[(fakt\_parametrai)];
  + accept\_sakinys ::= accept entry\_vardas[(formal¯us\_parametrai)] do sakinys
  + entry aprašų sąrašas nurodo,į kurias proceso dalis gali kreiptis kiti procesai;
  + entry kreipinys kreipiasi į kito proceso atitinkamą accept sakinį;
  + accept sakinys gali būti tik tarp proceso sakinių;
  + accept formali dalis turi pilnai sutapti su atitinkamo entry aprašo formalia dalimi;
  + accept parametrai matomi tik už accept užrašytame sakinyje;
  + accept E1 sakinys gali būti accept E2 sakinio viduje;
  + įeigos kreipinio faktiniai parametrai turi atitikti formaliems įeigos aprašo parametrams;
  + negalima kreiptis į pasibaigusių procesų įeigas; į tą pačią įeigą vienu metu gali kreiptis keli procesai
* **Ada modelio taikymas LP uždaviniams spręsti.**
  + procesas-kvietejas ir procesas-vykdytojas yra sinchronizuojami;
  + procesas-kvietejas blokuojamas, kol vykdomas accept sakinys;
  + procesas, bandantis vykdyti accept, yra stabdomas, jei nera atitinkamo kreipinio; accept sakinys gali būti tuščias.

1. **LP priemonių palyginimas**

* **Priemonių ypatybės, taikymo galimybės, privalumai ir trūkumai.**
* **Vienos priemonės modeliavimas kita priemone.**
* **Baigtinis (begalinis) buferis, pašto dežutė ir kt.: realizacija įvairiomis priemonėmis.**

1. Programavimas, naudojant OpenMP biblioteka˛

* **OpenMP paskirtis, ypatybės.**
  + OpenMP – tai papildomos priemonės, suteikiančios galimybes kurti lygiagrečias programas Fortran, C, C++ kalbomis;
  + OpenMP skirta kurti lygiagrečias programas, kuriose procesai (gijos) naudojasi bendra atmintimi;
  + OpenMP sudaro: kompiliatoriaus direktyvos, specialių funkciju˛ biblioteka, aplinkos kintamųju˛ rinkinys.
* **Pagrindinės OpenMP direktyvos ir funkcijos.**
  + Nuo programos darbo pradžios iki pabaigos vykdoma pagrindine gija (master thread).
  + Naujos gijos kuriamos kompiliatoriaus direktyvomis.
  + Vienu metu vykdomu˛ giju˛ rinkinys sudaro lygiagrecia srit (parallel region).
  + Lygiagrečioje srityje vykdomu˛ giju˛ skaičiu˛ galima nurodyti programos vykdymo metu.
  + Visos lygiagrečios srities gijos vykdo tą patį struktūrinį bloka˛.
  + Visos lygiagrečios srities gijos pradedamos vykdyti tuo pačiu metu ir vykdomos lygiagrečiai.
  + Lygiagreti sritis baigiama vykdyti, kai baigiamos vykdyti visos tos srities gijos. Kiekvienai lygiagrečios srities gijai suteikiamas unikalus numeris; master gijos numeris – 0.
  + Programos vykdymo metu nėra galimybiu˛ keisti gijos numerį.
  + Lygiagrecioje srityje vykdomu˛ giju˛ skaicius nurodomas pragma omp parallel atributu num\_threads() arba prieš lyg. sriti funkcijos omp\_set\_num\_threads() parametru.
  + Vykdomu˛ giju˛ skaiciu˛ gražina funkcija omp\_get\_num\_threads().
  + Gijos numeri gražina funkcija omp\_get\_thread\_num()
* **Programavimo naudojant OpenMP priemones pavyzdžiai.**

int main(){

int gijuSk = 2, gijosNr = 0;

omp\_set\_num\_threads(gijuSk);

// ------ Lygiagretus kodas -------

#pragma omp parallel

private(gijosNr) {

gijosNr = omp\_get\_thread\_num();

if (gijosNr == 0) vykdyti("Pirma");

else vykdyti("Antra"); }

cout << " Programa baig˙e darba˛";

1. **Programavimas, naudojant MPI biblioteka˛**

* MPI paskirtis, ypatybės.
  + MPI – Message Passing Interface.
  + MPI – tai pranešimu˛ perdavimo funkciju˛, skirtu˛ realizuoti lygiagreˇciuosius algoritmus, standartas.
  + MPI realizacija – tai MPI priemoniu˛ biblioteka klasteriui, paskirstytos atminties superkompiuteriui ar heterogeniniam tinklui.
  + Programavimo kalbos: C, C++, FORTRAN ir kt.
  + SPMD – Single Program Multiple Data.
  + Procesai (programos kopijos) sukuriami prieš programos vykdyma˛.
  + Procesu˛ sk. ≥superkompiuterio (klasterio) procesoriu˛ sk.
  + Procesai neturi bendros atminties.
  + Ryšys tarp procesu˛ – siunčiant ir priimant pranešimus (message).
* **Pagrindinės MPI funkcijos. MPI pranešimu˛ perdavimo variantai ir ypatybės.** 
  + Init() – nurodo MPI darbo pradžia˛.
  + Finalize() – nurodo MPI darbo pabaiga˛.
  + Comm\_size(), Get\_size() – grąžina procesu˛, vykdančių˛ skaičiavimus, kiekį.
  + Comm\_rank(), Get\_rank() – Grąžina proceso numerį nurodytame komunikatoriuje:
  + Send() – siunčia pranešimą˛
  + Recv() – priima pranešimą
  + Point-to-Point komunikavimas
    - Komunikavimas vyksta tarp dvieju˛ procesu˛ (Point-to-Point).
    - Source procesas siuncia žinute˛ (pranešima˛) destination procesui.
    - Destination procesas priima pranešima˛.
    - Komunikavimas vyksta communicator viduje.
    - Destination procesa˛identiﬁkuoja jo numeris (rank) komunikatoriuje
    - galimi variantai: Fully Synchronized (Rendezvous) – pilnas siuntejo ir gavejo sinchronizavimas,
    - Buffered – gavejas laukia pranešimo, siuntejas laukia, kol pranešimas bus patalpintas˛i buferi.
    - Collective komunikavimas
      * void Comm::Bcast(const void\* buf, int count, const Datatype& datatype, int root) const
      * root – siunciantis procesas; kiti parametrai: žiureti Send.
      * Scatter (išsklaidyti): duomenis, saugomus viename procese, persiuncia dalimis visiems komunikatoriaus procesams.
      * Gather (surinkti): duomenis, saugomus skirtinguose procesuose, surenka viename procese i bendra duomenu˛ rinkini
* **Programavimo naudojant MPI priemones pavyzdžiai.**

#include <mpi.h>

...

int main(int argc, char \*\*argv){

...

// Negalima kviesti jokiu˛ MPI funkciju

// prieš šia˛ eilute

MPI\_Init(&argc, &argv);

...

MPI\_Finalize();

// Negalima kviesti jokiu˛ MPI funkciju˛

// už šios eilut˙es

...

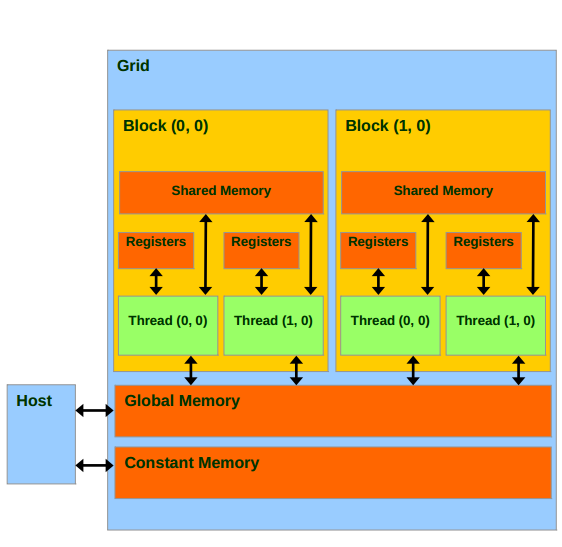
}

Open MPI

Projektas, apjungiantis kituose projektuose (FT-MPI, LA-MPI, LAM/MPI ir PACX-MPI) taikytas technologijas. Naudojama daugelyje TOP500 superkompiuteriu˛. Atviro kodo programinė įranga, realizuojanti MPI-3 standarta˛. Palaikoma˛ivairiose lygiagrečiųjų skaičiavimų aplinkose.

1. CUDA lygiagretumo modelis

* **Atminties modelis, skyrimas, atsisakymas.**



* + **Skyrimas:**
    - cudaError\_t cudaMalloc(void \*\*devPtr, size\_t size)

Skiria GPU atmintį.

devPtr - pradžios rodykle, size - dydis baitais.

* + **Atlaisvinimas** 
    - cudaError\_t cudaFree(void \*\*devPtr)
    - Atlaisvina GPU atmintį. devPtr - pradžios rodykle.CUDA paskirtis, ypatybės.
  + CUDA -– tai lygiagrečiųjų˛ skaičiavimų˛ architektūra, kurioje didelis skaičiavimų greitis pasiekiamas naudojant GPU;
  + CUDA skirtas kurti lygiagrečias programas C/C++, Fortran kalbomis;
  + CUDA realizuoja duomenų lygiagretumo modelį;
  + CUDA C/C++ sudaro: kompiliatorius, matematikos bibliotekos, derinimo ir optimizavimo˛irankiai;
  + naujausia versija: 8.0 (2016 m. rugsejis.)
* **Duomenų mainai GPU/CPU.**
  + cudaError\_t cudaMemcpy(void \*dst, const void \*src, size\_t count, enum cudaMemcpyKind kind)
  + Kopijuoja iš src˛i dst. kind - viena iš reikšmiu˛: cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, arba cudaMemcpyDeviceToDevice.
* **Gijų kūrimas ir vykdymas.** 
  + CPU gijos vykdomos pagrindiniame procesoriuje, GPU (graﬁnio procesoriaus) gijos vykdomos graﬁneje plokšteje;
  + CPU efektyviai gali buti vykdoma tik nedidelis kiekis giju˛, o GPU - tūkstančiai; sąvokos: host = CPU, device = GPU, kernel – funkcija, kuri vykdoma GPU;
  + daug GPU giju˛ tuo pačiu metu gali vykdyti ta˛pačia˛ kernel’io funkcija˛.
  + lygiagretus GPU giju˛ vykdymas (<<< >>>);
  + Gijos funkcija, kviečiama iš CPU, vykdoma GPU. Pradedama raktažodžiu \_\_global\_\_:
  + Funkcija, kviečiama iš GPU, vykdoma GPU. Pradedama raktažodžiu \_\_device\_\_:
  + // GPU (kernel’io) funkcija \_\_global\_\_ void MasSud(int\* A, int\* B, int\* C) { int i = threadIdx.x;

C[i] = A[i] + B[i];

}

int main()

{

...

// Sudeda masyvus A(n) ir B(n); rezultatas - C(n)

// vykdo n giju˛

MasSud<<<1, n>>>(A, B, C);

... }

* visos gijos vykdo tą pati programini kodą;
* gijos grupuojamos˛į blokus;
* gijos bloko viduje gali naudoti bendra˛atmintį, vykdyti atominius veiksmus, gali būti sinchronizuojamos;
* gijos skirtinguose blokuose nesąveikauja.
* kiekviena gija turi savo unikalu˛ ID (numerį):1D, 2D arba 3D (threadIdx.{x,y,z});
* **Thrust vektoriai.**
  + Duomenų struktūros:
    - Thrust::device\_vector
    - Thrust::host\_vector
    - Thrust::device\_ptr
  + Algoritmai
    - Thrust::sort
    - Thrust::reduce
    - Thrust::exclusive\_scan
  + Thrust – tai galinga greitu˛ lygiagrečiųjų algoritmų ir duomenų struktūrų˛ biblioteka. Bibliotekos organizavimo ir naudojimo principai panašūs C++ STL.
  + Thrust biblioteka http://thrust.github.io/

#include <thrust/host\_vector.h>

#include <thrust/device\_vector.h>

...

// thrust host vektoriai

// (saugomi operatyvioje atmintyje)

thrust::host\_vector<Duom> H1(n1);

skaityti(fd, H1);

rodyti(H1, "host 1-as vektorius:");

thrust::host\_vector<Duom> H2(n2);

skaityti(fd, H2);

rodyti(H2, "host 2-as vektorius:");

// thrust device vektoriai

// (saugomi GPU atmintyje)

// Kopijuoja host\_vector H i˛ device\_vector D

thrust::device\_vector<Duom> D1 = H1;

thrust::device\_vector<Duom> D2 = H2;

thrust::device\_vector<Duom> D(n1);

for(int i = 0; i < D.size(); i++)

{

D[i] = Plus(D1[i], D2[i]);

} thrust::host\_vector<Duom> H = D; rodyti(H, "VEKTORIU˛ SUMA:");

* + // thrust algoritmai

//

...

int x[6] = { -5, 0, 2, 3, 20, 40 };

int y[6] = { 3, 6, -2, 1, 2, 3 };

int z[6];

thrust::plus<int> operacija;

thrust::transform(x, x + 6, y, z, operacija);

// z= { -2, 6, 0, 4, 22, 43 }

1. LP kalbų ypatybės (procesų˛ kūrimas ir komunikavimas)

* **Java,** 
  + Panaudojant (paveldint) java.lang.Thread klase˛.
  + Panaudojant (realizuojant) java.lang.Runnable sasaja˛.
  + notify() Wakes up a single thread that is waiting on this object’s monitor.
  + notifyAll() Wakes up all threads that are waiting on this object’s monitor.
  + wait() Causes current thread to wait until another thread invokes the
  + notify() method or the notifyAll() method for this object. wait(long timeout) Waits for notiﬁcation or until the timeout period has elapsed. timeout is measured in milliseconds.
* **C++,** 
  + konstruktorius su parametrais: thread(fn, args), kur fn – gijoje vykdoma funkcija, args – funkcijai perduodamu˛ argumentu˛ sa˛rašas;
  + gijos darbo pabaigos laukimo metodas: join()
* **C#;**
  + using System.Threading;
  + Thread InstanceCaller = new Thread(

new ThreadStart(serverObject.InstanceMethod));

InstanceCaller.Start();

sleepingThread.Start();

Thread.Sleep(2000);

* [Thread.Interrupt](https://msdn.microsoft.com/en-us/library/system.threading.thread.interrupt(v=vs.110).aspx) wakes a thread out of any wait it might be in and causes a [ThreadInterruptedException](https://msdn.microsoft.com/en-us/library/system.threading.threadinterruptedexception(v=vs.110).aspx) to be thrown in the destination thread.
* [Thread.Abort](https://msdn.microsoft.com/en-us/library/system.threading.thread.abort(v=vs.110).aspx) wakes a thread out of any wait it might be in and causes a [ThreadAbortException](https://msdn.microsoft.com/en-us/library/system.threading.threadabortexception(v=vs.110).aspx) to be thrown on the thread. For details, see [Destroying Threads](https://msdn.microsoft.com/en-us/library/cyayh29d(v=vs.110).aspx).
* **Ada;**

procesas-kvietejas ir procesas-vykdytojas yra sinchronizuojami;

procesas-kvietejas blokuojamas, kol vykdomas accept sakinys;

procesas, bandantis vykdyti accept, yra stabdomas, jei nera atitinkamo kreipinio; accept sakinys gali buti tušcias.

* **occam;**

Palaiko point to point komunikavimą

Visas komunikavimas vykdomas per kanalus

Komunikavimas yra sinchronizuotas

Palaiko CSP

Naudoja Saugus(guards) Saugai

Lygiagretuma apraso sakinys PAR

SEQ

PAR

...knit body .

..knit left sleeve

...knit right sleeve

...knit neck .

..sew sweater

ALT

count1 < 100 & c1 ? data

SEQ

count1 := count1 + 1

merged ! data

count2 < 100 & c2 ? data

SEQ

count2 := count2 + 1

merged ! data

status ? request

SEQ

out ! count1

out ! count2

* **Google Go.**
  + go sakiniuose kviečiamos funkcijos vykdomos kaip atskiros gijos, t.y., lygiagrečiai (concurrent)
  + nera klasių, taciau kiekvienam tipui galima kurti metodus (funkcijas);
  + komunikavimui tarp goroutines taikomas Hoare CSP (occam, JavaCSP) modelis; komunikavimui naudojami tipizuoti kanalai:
    - nebuferizuotas: ch := make(chan Tipas)
    - buferizuotas: ch := make(chan Tipas, n)
    - rašymas˛i kanala˛(dvinare operacija): ch <- a
    - skaitymas iš kanalo (vienanare operacija): <- ch
  + sinchronizavimui naudojami nebuferizuoti (sinchroniniai) kanalai.