



BENDRASIS PROGRAMAVIMO DOKUMENTAS



EUROPOS SĄJUNGA
Europos socialinis fondas



ŠVIETIMO IR MOKSLO MINISTERIJA



KURKIME ATEITĮ DRAUGE!

Rimantas Vaicekauskas

Lygiagretūs ir išskirstyti skaičiavimai

Mokymo medžiaga

Vilnius

2007

Mokymo medžiaga parengta vykdant projektą “Programų sistemų magistrantūros isteigimas”, įgyvendinantį 2004-2006 metų bendrojo programavimo dokumento 2.5 priemonę “Žmogiškųjų išteklių kokybės gerinimas mokslinių tyrimų ir inovacijų srityje”, finansuojamą Europos Sąjungos struktūrinių fondų lėšomis ir Lietuvos Respublikos bendrojo finansavimo lėšomis.

Turinys

1	Ivadas	7
2	Lygiagretūs procesai ir gijos.....	9
2.1	Motyvacija	9
2.2	Kritinės sekcijos problema.....	10
2.2.1	KS sprendimas, naudojant dalomus kintamuosius.....	10
2.2.2	Operacija TestAndSet.....	13
2.3	Semaforai	14
2.3.1	Apibrėžimas	14
2.3.2	Procesų synchronizacija - KS sprendimas naudojant binarinį semaforą	15
2.3.3	Procesų synchronizacija – informavimas apie įvykį.....	15
2.3.4	Gamintojo-vartotojo (<i>producer-consumer</i>) sąryšis.....	15
2.3.5	Vienatipių resursų dalijimo problema.....	16
2.4	Įvykių skaitikliai	16
2.5	Monitoriai	17
2.5.1	Pirminės idėjos: kritinė sekcija (KS), salyginė KS	17
2.5.2	MONITORiai	18
2.5.3	Trajektorijų išraiškos (<i>path expressions</i>)	20
2.5.4	Trajektorijų išraiškų realizacija progr. kalboje ADA – <i>select</i> sakinsky	21
2.5.5	Monitoriai ir Java	21
2.6	Aklavietės	22
2.6.1	Aklavietės situacija	22
2.6.2	Keturios būtiniosios aklaviečių susidarymo salygos	24
2.6.3	Aklaviečių prevencija	24
2.6.4	Aklaviečių išvengimas	25
2.6.5	Bankininko algoritmas	26
2.6.6	Aklaviečių aptikimas	27
2.6.7	Aklaviečių atstatymas	28
2.7	Asynchroninių procesų modeliavimas Petri tinklais	28
2.8	Gijų panaudojimas, išlygiagretinant algoritmus	31
2.8.1	Programos darbo spartinimas daugiaprocesorinėje aplinkoje.....	31
2.8.2	Šeimininko - tarno schema.....	32
2.8.3	Darbų krūvos schema.....	34
2.8.4	Konvejerio schema.....	34

2.9	Gijų realizacijos. POSIX gijos.....	35
2.9.1	Standartai	35
2.9.2	Kas yra gija	36
2.9.3	Gijų nauda.....	38
2.9.4	Gijų programų panaudojimo modeliai.....	39
2.9.5	Pthreads sąsaja (API)	40
2.9.6	Muteksai (<i>mutex</i>)	44
2.9.7	Įvykių kintamieji (<i>condition variables</i>)	46
3	Lygiagretūs kompiuteriai	51
3.1	Didelio našumo skaičiavimų poreikis.....	51
3.2	Lygiagrečiųjų kompiuterių tipai	54
3.2.1	Bendrosios (shared) atminties daugiaprocesorinė sistema	54
3.2.2	Pranešimais grįstas multikompiuteris	56
3.2.3	Išskirstyta bendroji atmintis.....	58
3.2.4	MIMD, SIMD klasifikacijos.....	59
3.3	Architektūrinės pranešimų multikompiuterių ypatybės	60
3.3.1	Statinio tinklo pranešimų multikompiuteriai	60
3.3.2	Tinklo kompiuteriai, kaip kompiuterinė platforma.....	64
3.4	Kiekybinės skaičiavimų spartinimo charakteristikos.....	65
4	Pranešimų perdavimu grįsti skaičiavimai	68
4.1	Įvadas	68
4.1.1	Programavimo variantai.....	68
4.1.2	Procesų sukūrimas	68
4.1.3	Pranešimų perdavimo procedūros.....	70
4.2	Kompiuterių klasteriai	76
4.2.1	Programiniai instrumentai.....	76
4.2.2	PVM (Parallel Virtual Machine).....	76
4.3	MPI (Message Passing Interface)	79
4.4	Lygiagrečiųjų programų įvertinimas.....	86
4.4.1	Lygiagretaus vykdymo laiko įvertinimas.....	86
4.4.2	Spartinimo įvertinimas.....	88
4.4.3	Vizualizacijos instrumentai.....	88
4.4.4	Sąvadai	89
5	Idealiai išlygiagretinami skaičiavimai	99

5.1	Įvadas	99
5.2	Paveikslėlių transformavimas	101
5.3	Mandelbroto aibė.	104
5.4	Monte-Karlo metodai.....	110
6	Skaidymo (skirstymo) strategijos	113
6.1	Skaidymo strategijos.....	113
6.2	„Skaldyk ir valdyk“ strategija.....	116
6.3	M-aris medis	118
6.4	Skaldyk ir valdyk strategijos taikymai.....	119
6.4.1	Rūšiavimas kiberais (bucket).....	119
6.4.2	Lygiagrečioji realizacija.....	119
6.4.3	Skaitinis integravimas	121
6.4.4	Pavyzdys. Skaičiaus PI skaičiavimo programa.....	123
6.4.5	N-kūnų problema	125
7	Konvejeriniai skaičiavimai	129
7.1	Konvejerinė technika	129
7.2	Kompiuterių platformos konvejeriniams skaičiavimams.	133
7.3	Konvejerinių programų pavyzdžiai.....	133
7.3.1	Skaičių sumavimas.....	133
7.3.2	Skaičių rikiavimas.....	135
7.3.3	Pirminių skaičių generavimas	138
7.3.4	Tiesinių lygčių sistemos sprendimas.	139
8	Sinchroniniai skaičiavimai.....	142
8.1	Barjeras	142
8.2	Lokalioji sinchronizacija.....	146
8.3	Sinchronizuotų skaičiavimų klasifikacija	147
8.3.1	Visiškai sinchronizuotų skaičiavimų pavyzdžiai	147
8.3.2	Lokaliai synchroniniai skaičiavimai.	153
9	Apkrovos balansavimas ir baigmės aptikimas.....	163
9.1	Statinis balansavimas	164
9.2	Dinaminis apkrovos balansavimas.....	165
9.2.1	Centralizuotas dinaminis apkrovos balansavimas	165
9.2.2	Decentralizuotas dinaminis apkrovos balansavimas.....	166
9.3	Išskirstytos programos baigmės aptikimo sąlygos.....	170

9.4	Apkrovos balansavimo ir baigmės aptikimo pavyzdys: trumpiausio kelio grafe paieškos uždavinys.....	174
9.4.1	Paieška grafe	176
9.4.2	Moore'o algoritmas.....	176
10	Lygiagrečiųjų algoritmų taikymai – duomenų rikiavimas.....	182
10.1	Potencialus spartinimas.....	182
10.2	Palyginimo ir sukeitimo (Compare-and-Exchange) pavidalo algoritmai	182
10.3	Rūšiavimas burbulais (Bubble Sort).	185
10.4	Rikiavimas sąlaja (mergesort).....	187
10.5	Quicksort rikiavimas	188
10.6	Rikiavimas specializuotuose tinklų topologijose.....	190
10.6.3	Rikiavimas dviejų dimensijų tinkle.	190
10.6.4	Shearsort – rikiavimas.....	190
10.7	Algoritmai hiperkubo tinkle:quicksort.....	192
10.8	Kiti rūšiavimo algoritmai	192
10.8.1	Rūšiavimas rangais	193
10.8.2	Rikiavimas išskaičiuojant (<i>counting sort</i>)	195
10.8.3	Pozicinis rikiavimas (<i>radix sort</i>).....	196
11	Literatūra.....	198
12	Priedas. Laboratorinių darbų pavyzdys.....	200
12.1	Kritinės sekcijos problemos raiška	200
12.2	Gijas sinchronizuojančių objektų apibrėžtis	201
12.3	Lygiagretus sprendimas bendrosios atminties sistemai	204
12.4	Lygiagretūs skaičiavimai pranešimų sistemoje	207

1 Išskirstytų sistemų funkcionavimo principai

Lygiagretūs ir išskirstyti skaičiavimai, anksčiau laikyti egzotiška kompiuterinės veiklos sritimi, prieinama siauram kompiuterių mokslo teoretikų ratui ar naudojama tik labai specializuotose kompiuterinėse architektūrose, vis plačiau įsigali visose kompiuterių panaudojimo sferose. Tą įtakoja visuotinis Interneto paplitimas, kompiuterinių sistemų fizinis ir geografinis išskaidymas, aukšto našumo skaičiavimų poreikis bei daugiaprocesorinių sistemų plitimas kaip efektyvi didesnių skaičiavimo pajėgumų sukūrimo priemonė.

„Lygiagrečių ir išskirstytų sistemų“ kurso tikslas – supažindinti „Programų sistemų“ magistratūros kurso dalyvius su pagrindiniais lygiagrečių ir išskirstytų sistemų sistemų funkcionavimo bei efektyvaus jų panaudojimo principais, atsižvelgiant į šiuolaikines mokslo bei technologijų tendencijas.

Dėstymas suskaidytas į dvi pagrindines dalis. Pirmojoje aptariami pagrindiniai lygiagrečiųjų sistemų funkcionavimo principai, neatsižvelgiant į fizinę jų prigimti. Įvedama lygiagrečiųjų procesų kaip loginių esybių savoka, apibrėžiamos pagrindinės problemos, susijusios su lygiagrečiųjų procesų naudojimu, sinchronizacija, komunikacija, patekimo į aklavietes problemos, bei jų sprendimo priemonės. Angliškoje literatūroje ši tematika vadinama „*Concurrent Programming*“. Lygiagretūs saveikaujantys procesai taikomosiose programose yra dažniausiai panaudojami gijų (*threads*) pavidalu. Kurse nagrinėjamos standartinės gijų priemonės įgyvendintos plačiai paplitusiose programavimo Java bei POSIX –gijos. Pirmosios dalies pabaigoje aptariami gijų taikymo metodai (paradigmos), sprendžiant tipines uždavinį klases.

Antrojoje dalyje, aptariamos sistemos, kuriuose lygiagretūs skaičiavimai yra naudojami kaip priemonė kompiuterinės sistemos našumui padidinti. Šiuo atveju, negalima neatsižvelgti į fizinę procesų prigimtį, taigi tenka nagrinėti lygiagrečiųjų kompiuterinių sistemų architektūrines ypatybes: bendrosios atminties kompiuteriai ar pranešimų perdavimais funkcionuojančios sistemos. Šiuolaikinė tendencija – išskirstytos įtinklinto sistemos, kurios gali funkcionuoti kaip nepriklausomi (personaliniai) skaičiavimų įrenginiai bei dalyvauti bendrai vykdomame skaičiavimų procese. Dėl silpno susijimo, tokų sistemų panaudojimas kelia specifinių reikalavimų.

Aptarus pagrindinius uždavinius, kuriuos tenka spręsti, projektuojant ir naudojant išskirstytas lygiagrečias kompiuterines sistemas pereinama prie lygiagrečiųjų skaičiavimų panaudojimo specifikos, atsižvelgiant į sprendžiamus uždavinius. Nenagrinėjamos

specializuotos ar neišreikštinės lygiagrečiaus programavimo sistemos, bet remiamamasi de fakto standartu - MPI pranešimų sistema.

Tolesniame dėstyme nagrinėjami specifiniai skaičiavimų tipai: smulkiagrūdžiai, tarpusavyje beveik nesusiję skaičiavimai, duomenų dekompozicijos ypatumai, konvejeriniai skaičiavimai, sinchroniniai (pažingsniniai) bei nereguliarūs išskirstyti skaičiavimai. Konkretūs pavyzdžiai, iliustruojantys specifinius skaičiavimų principus, įgalina pritaikyti gautas žinias specializuotoms problemoms spręsti.

Manau, jog „Lygiagrečių ir išskirstytų skaičiavimų“ kursas padės kurso dalyviams sustiprinti ir pagilinti žinias bei įsisavinti paruoštus įvairių praktikoje iškyylančių uždavinių sprendimo receptus.

Mokomojoje medžiagoje naudojama [1,2,3,4,5] šaltinių medžiaga, kreditai – autoriams.

2 Lygiagretūs procesai ir gijos

Šiame skyriuje aptarsime pagrindinius lygiagrečiųjų procesų funkcionavimo ir sąveikos principus, neatsižvelgiant į jų fizinę prigimtį. *

2.1 Motyvacija

Nuoseklusis procesas (programa) susideda iš nuoseklių veiksmų. Lygiagretumas nusako kelis vienu metu vykstančius nuosekliuosius procesus. Pavyzdžiu i multiprograminė operacinė sistema geba vykdyti kelis procesus vienu metu. Gali būti ir taip, jog kelis procesus vykdo vienas procesorius, taigi iš esmės mes turime loginį (potencialų) lygiagretumą (kelios programos vykdomo vienu metu, bet kiekvienu laiko momentu vykdoma tik viena programa).

Taigi, apibendrinus:

- **nuoseklus** – einantis vienas paskui kitą.
- **lygiagretus** - vykstantis vienu metu,
- **potencialiai /logiškai lygiagretus** .

Nesumažindami bendrumo, nagrinėsime logiškai lygiagrečius procesus, angliskoje literatūroje vadinamus terminu *concurrent*.

Kodėl naudingas lygiagretumas ?

Pranašumai: sutrumpėjės bendras procesų darbo laikas, sumažėjusios individualios pastangos.

Trūkumai: reikalingas veiklos derinamas, galimos prastovos, nedeterminizmas, trikiai.

Kalbant apie procesus dažnai naudojami terminai:

- **Procesai** – lygiagretaus vyksmo subjekta;
- **Sinchronizacija** – procesų veiklos derinimas;
- **Resursas** – monopoliskai procesų naudojamas objektas;
- **Komunikacija** – duomenų perdavimas tarp procesų;
- **Prioritetai** – pranašumo suteikimas vieniems procesams kitų atžvilgiu. Užtikrina efektyvesnį resursų valdymą.

Concurrent programavimo priemonės naudojamos

- kuriant sudėtingas realaus laiko sistemas su prigimtiniu lygiagretumu;
- siekiant užtikrinti efektyvesnį skaičiavimo resursų panaudojimą;

* Skyriuje naudojama [1,2,3,4] medžiaga.

- siekiant padidinti uždavinių sprendimo spartą daugiaprocesinėse sistemosose.

Uždavinių sprendimo spartos didinimą aptarsime šio skyriaus pabaigoje, bei kituose skyriuose.

2.2 Kritinės sekcijos problema.

Kritinė sekcija (KS) – kodo sritis kurioje skaitomi arba modifikuojami dalomi duomenys.

Pavyzdys:

```
task INCREMENT;
task body INCREMENT is
begin
    N := N + 1;
    PUT (N);
end INCREMENT;

task DECREMENT;
task body DECREMENT is
begin
    N := N - 1;
    PUT (N);
end DECREMENT;
```

Jeigu abu procesus, kurie modefikuoja bendrą kintamąjį N vykdysime lygiagrečiai, galutinė kintamojo N reikšmė nėra determinuota.

Taigi, kritinės sekcijos problema – kaip užtikrinti, jog KS kodą vienu metu vykdytu tik vienintelis procesas.

2.2.1 KS sprendimas, naudojant dalomus kintamuosius

Esmė - bendrų (*shared*) keliems procesams kintamujų panaudojimas, realizuojant KS iėjimo/baigmės kodą^{*}:

```
<entry protocol>
<critical section>
<exit protocol>
```

Aplinkybės:

- Tai programinis sprendimas, nenaudojant jokių spec. aparatūrinių instrukcijų. Kiekviena instrukcija nedaloma.
- Sprendimas neturi remtis išankstinėmis prielaidomis apie relatyvų asynchroninių procesų vykdymo greitį.
- Procesas už KS ribų neturi trukdyti kitam procesui pakliūti į kritinę sekciją.
- Proceso iėjimas į kritinę sekciją negali būti be galo atidėliojamas.

* Skyrellyje naudojami programų tekstai pagal [2]

Pirmas korektiškas sprendimas: (Dekker, 1965).

1. Sprendimas (naivusis).

```
N : INTEGER := 1;
LOCKED : BOOLEAN := FALSE;

task INCREMENT;
task INCREMENT is
begin
    -- Entry Protocol
    while LOCKED loop
        null;
    end loop;
    LOCKED := TRUE;
    -- Now in CS
    N := N + 1;
    -- Exit Protocol
    LOCKED := FALSE;
end INCREMENT;

task DECREMENT;
task DECREMENT is
begin
    -- Entry Protocol
    while LOCKED loop
        null;
    end loop;
    LOCKED := TRUE;
    -- Now in CS
    N := N - 1;
    -- Exit Protocol
    LOCKED := FALSE;
end DECREMENT;
```

Trūkumas - užimtas laukimas (*busy waiting*), neužtikrinama KS.

2. Sprendimas (i kritinę sekciją procesai patenka vienas paskui kitą).

```
N      : INTEGER := 1;
TURN   : POSITIVE := 1;

task INCREMENT;
task INCREMENT is
ME    : constant := 1;
NEXT  : constant := 2;
begin
    -- Entry Protocol
    while TURN /= ME loop
        null;
    end loop;
    -- Now in CS
    N := N + 1;
    -- Exit Protocol
    TURN := NEXT;
end INCREMENT;

task DECREMENT;
task DECREMENT is
ME    : constant := 2;
NEXT  : constant := 1;
begin
    -- Entry Protocol
    while TURN /= ME loop
        null;
    end loop;
    -- Now in CS
    N := N - 1;
    -- Exit Protocol
    TURN := NEXT;
end DECREMENT;
```

Trūkumai: KS problema išspręsta, bet didelė kaina.

3. sprendimas. Įėjimo į CS paraiška.

```
N : INTEGER := 1;
INCR CLAIM, DECR CLAIM : BOOLEAN := FALSE;

task INCREMENT;                                task DECREMENT;

task INCREMENT is
begin
    -- Entry Protocol
    INCR CLAIM := TRUE;
    while DECR CLAIM loop
        null;
    end loop;
    -- Now in CS
    N := N + 1;
    -- Exit Protocol
    INCR CLAIM := FALSE;
end INCREMENT;

task DECREMENT is
begin
    -- Entry Protocol
    DECR CLAIM := TRUE;
    while INCR CLAIM loop
        null;
    end loop;
    -- Now in CS
    N := N - 1;
    -- Exit Protocol
    DECR CLAIM := FALSE;
end DECREMENT;
```

Trūkumai: KS problema išspręsta, bet galima aklavietė (*deadlock*).

3' sprendimas. Modifikacija: laikinas paraiškos atsiémimas.

```
...
begin
    -- Entry Protocol
    INCR CLAIM := TRUE;
    while DECR CLAIM loop
        INCR CLAIM := FALSE;
        delay 0.1;
        INCR CLAIM := TRUE;
    end loop;
    -- Now in CS
    ...
begin
    -- Entry Protocol
    DECR CLAIM := TRUE;
    while INCR CLAIM loop
        DECR CLAIM := FALSE;
        delay 0.11;
        DECR CLAIM := TRUE;
    end loop;
    -- Now in CS
    ...
```

Trūkumas – galimas neapibrėžtas įėjimo į KS atidėjimas.

4. Sprendimas. (Dekker). Apibendrinti 2,3 atvejai.

```

N : INTEGER := 1;
TURN : POSITIVE := 1;
INCR CLAIM, DECR CLAIM : BOOLEAN := FALSE;

task INCREMENT;                                task DECREMENT;
task INCREMENT is                            task DECREMENT is
    ME : constant := 1;                      ME : constant := 2;
    NEXT : constant := 2;                   NEXT : constant := 1;
begin                                         begin
    -- Entry Protocol                         -- Entry Protocol
    INCR CLAIM := TRUE;                     DECR CLAIM := TRUE;
    while DECR CLAIM loop                 while INCR CLAIM loop
        if TURN /= ME then                if TURN /= ME then
            INCR CLAIM := FALSE;          DECR CLAIM :=
        FALSE;                           while TURN /= ME
        while TURN /= ME loop           null;
        loop                             end loop;
        null;                           DECR CLAIM := TRUE;
        end loop;                       end if;
        INCR CLAIM := TRUE;             end loop;
        end if;                         -- Now in CS
        end loop;                      N := N - 1;
        -- Now in CS                   -- Exit Protocol
        N := N + 1;                   TURN := NEXT;
        -- Exit Protocol             DECR CLAIM := FALSE;
        TURN := NEXT;                end DECREMENT;
        INCR CLAIM := FALSE;          end INCREMENT;
end INCREMENT;                                 end DECREMENT;

```

KS problemos sprendimo dalomais kintamaisiais kritika.

- Sprendimas labai sudėtingas. Dalomų kintamujų skaičius proporcingsas procesų ir KS skaičiui.
- Neatsparus “paslėptoms” klaidoms.
- Netinkamas didelėms synchronizacijos schemoms.
- Procesai labai susiję tarpusavyje.
- “Užimtas laukimas”.

2.2.2 Operacija TestAndSet.

Postuluojamas nedalomas (nepertraukiama) reikšmės patikrinimas ir priskyrimas, pvz., aparatūriškai realizuojama instrukcija.

```

VOID TestAndSet(BOOL &a, BOOL &b) {
    a = b; b = TRUE; // Nedaloma operacijų seka
}

```

Kritinės sekcijos realizacija, naudojant `TestAndSet`. Visi procesai vykdo tapatų ižangos/baigmės kodą.

```

BOOL busy = FALSE; // Bendras

//. . . . .
BOOL wait = TRUE;
while (wait)
    TestAndSet(wait, busy);

// KS viduje

busy = FALSE;
//. . . . .

```

Akivaizdu, jog ši schema, palyginus su programiniu sprendimu, leidžia mažiau galimybių suklysti.

2.3 Semaforai

2.3.1 Apibrėžimas

Ivedė Dijkstra, 1965. Semaforas - sinchronizacinis primityvas su operacijomis P,V.

Iniciacija:

```
init(S, <skaičiumi>);
```

P(S):

```

if (S > 0)
    S--;
else
    <laukti S>;

```

V(S):

```

if (<Vienas ar daugiau procesų laukia S>)
    <vienu procesu atlaisvinti>;
else
    S++;

```

P,V – nedalomos (nepertraukiamos) operacijos.

Semaforai, kurie tegali išgauti reikšmes {0,1} vadinami binariniais semaforais. Kitu atveju semaforais - skaitliukais {0,1, 2,...}.

Galima aparatūrinė bei programinė semaforų realizacija. Dažniausiai jie realizuojamos kaip OS branduolio (kuris valdo procesus) funkcijos.

2.3.2 Procesų synchronizacija - KS sprendimas naudojant binarinį semaforą.

```
init(S, 1);  
P(S); < KS kodas>; V(S);
```

2.3.3 Procesų synchronizacija – informavimas apie įvykį.

```
init(S, FALSE); // Įvykis dar neįvykęs  
  
//...  
P(S) // <-- Laukiame S  
//...  
//...  
V(S); // --> Pranešame  
apie S  
//...  
//...
```

2.3.4 Gamintojo-vartotojo (*producer-consumer*) sąryšis.

Problema: suvartoti "pagamintus" duomenis tik po vieną kartą, jokio "neprapuldžius". Iš esmės tai – komunikacinis kanalas

2.3.4.1 Vienavietė „bendro naudojimo“ talpykla. Naudojami binariniai semaforai.

```
init(S_sunaudota, TRUE);  
init(S_pagaminta, FALSE);  
DATA d; // Dalomas  
  
// Gamintojas  
while(TRUE){  
    DATA duom =  
    pagaminti();  
    P(S_sunaudota);  
    d = duom;  
    V(S_pagaminta);  
}  
  
// Vartotojas  
while(TRUE){  
    P(S_pagaminta);  
    DATA duom = d;  
    V(S_sunaudota);  
    sunaudoti(duom);  
}
```

2.3.4.2 N-vietis buferis. Naudojami skaitl. semaforai ir “žiedinis” buferis.

```
init(S_sunaudota,N);
init(S_pagaminta,0);
DATA d[N]; // Bendras

// Gamintojas                                // Vartotojas
int k = 0;
while(TRUE) {
    DATA duom =
    pagaminti();
    P(S_sunaudota);
    d[k] = duom;
    V(S_pagaminta);
    k = (k+1) % N;
}
int k = 0;
while(TRUE) {
    P(S_pagaminta);
    DATA duom = d[k];
    V(S_sunaudota);
    k = (k+1) % N;
    sunaudoti(duom);
}
```

2.3.5 Vienatipių resursų dalijimo problema.

Sprendžiama semaforu-skaitliuku, nustatant lygiu turimų resursų skaičiui.

- $P(S)$ – resurso paėmimas iš krūvos;
- $V(S)$ – resurso gražinimas.

Semaforų kritika.

- Konkrečiame panaudojime galimai neaškus vaidmuo: išskyrimas / synchronizacija / resursų skaičiavimas.
- Glaudžiai susiję sudėtingose synchronizacijos schemose.
- Klaidų bei aklaviečių šaltinis.

Reziumė: semaforai – gana žemo lygio synchronizacinių primityvų (bet kitokių gali ir nebūti).

2.4 Įvykių skaitikliai.

Galimi kiti synchronizacijos primityvai kaip semaforų pakaitalai.

Ivedė Reed (1977) ir Kanodia (1979). *Eventcount* (įvykių skaitiklis), *sequencer* (rikuotojas).

Eventcount operacijos:

- `advance(eventcount)` - skaitiklis “nedalomai” padidinamas vienetu;
- `read(eventcount)` - nuskaitoma skaitliuko reikšmė;
- `await(eventcount, value)` - laukama, kol skaitliukas taps ne mažesnis negu value.

Gamintojo-vartotojo uždavinio realizacija, naudojant įvykių skaitiklius.

```
// Global
const int N = 5;
eventcount in, out;
int buf[N];

// Producer                                // Consumer
int i = 0;                               int i = 0;
while(TRUE){                           while(TRUE){
    i++;                                 i++;
    await(out, i-N);                  await(in, i);
    buf[i%N] = produce();           concume(buf[i%N]);
    advance(in);                   advance(out);
}
}
```

2.5 Monitoriai

Skyrelyje aptariamos aukšto lygio *concurrent* konstrukcijos programavimo kalbose.

Programavimo kalbos: Concurrent Pascal, Communicating Sequential Processes, Modula-2, Ada, Java, C#.

2.5.1 Pirminės idėjos: kritinė sekcija (KS), sąlyginė KS

Kritinė sekcija (*critical region*). Ši kalbinė konstrukcija „apskliaudžia“ nuosekliai prieinamą programos kodo vietą:

```
region shareddata do action
```

Sąlyginė kritinė sekcija leidžia išvengti “užimto laukimo”:

```
region shareddata do begin await condition; action end
```

Čia – shareddata – dalomas kintamasis, condition – sąlyga.

Pavyzdys: ribotos talpos automobilių parkavimo aikštélė (Ada stilius)

“Busy wait” atvejis

```
loop
    region SPACES is
        GOT_IN := SPACES > 0;
        If GOT_IN then
            SPACES := SPACES-1;
            End if;
        end region;
        exit when GOT_IN;
    end loop;
```

“Guarded” atvejis

```
region SPACES
    when
        SPACES > 0
    is SPACES := SPACES-1;
end region;
```

Problemos:

- sunku realizuoti informacijos slėpimo principą;
- idėtosios kritinės sekcijos yra problematiškos

2.5.2 MONITORiai

Modulinė konstrukcija užtikrinantį duomenų abstrakciją kartu su abipusiu išskyrimu. Duomenys prieinami (skaitomi/modifikuojami) naudojant viešas funkcijas. Tik vienas procesas gali vykdyti tą funkciją vidinį kodą (t.y., būti monitoriaus viduje). Nauda: kompiliatorius vykdo kontrolę ir generuoja kodą, naudodamas *run-time* aplinką.

Sąlygos kintamasis (*conditional variable*). Apibrėžia įvykį, kurio gali laukti procesas.

- **wait** (*conditionalVariable*) – laukti įvykio;
- **signal** (*conditionalVariable*) – pranešti apie įvykį (išlaisvinant vieną procesą, laukiantį eilėje, susijusioje su sąlygos kintamuju).

Pastabos:

- wait/signal operacijos kviečiamos, esant procesui monitoriaus viduje,-taigi duomenų neprieštarininga būsena užtikrinama.
- skiriasi nuo semaforo. Jeigu signalizuojama, nesant laukiančiam procesui, signalas sąlygos kintamujų atveju “prarandamas”.

2.5.2.1 Pavyzdys: resurso išskyrėjas

Resursas – tai objektas, kuriuo vienu metu gali operuoti tik vienas procesas. Naudojamos dvi bazinės operacijos – gauti resursą ir grąžinti resursą. Procesas, pateikęs paraišką resursui, blokuojasi, kai resursas yra užimtas.

```
monitor ResourceAllocator is
    ResourceInUse: Boolean := False;
    ResourceIsFree: Condition;
    procedure getResource is
    begin
        if ResourceInUse then
            Wait(ResourceIsFree);
        end if;
        ResourceInUse = True;
    end getResource;

    procedure releaseResource is
    begin
        ResourceInUse = False;;
        Signal(ResourceIsFree);
    end releaseResource;
end ResourceAllocator
```

2.5.2.2 Skaitytojų/rasytojų problemos sprendimas, naudojant monitorius.

Skaitytojų rašytojų uždavinio esmė – užtikrinti, kad kelių rūsių procesai įgautų vienas kito požiūriu neprieštaringą būseną. Kaip pavyzdį, galime nagrinėti duomenų bazę, kurios įrašus procesai – skaitytojai gali naudoti tik skaitymui, o kitiems procesams – rašytojams – leidžiama įrašus modifikuoti. Įrašų peržiūrą leidžiama vykdyti keliems procesams vienu metu, tačiau tegali būti „aktyvus“ tik vienas „rašytojas“. Kad užtikrinti tinkamą procesų sąveiką, apibrėžiame resursų naudojimo protokolą. Procesas – skaitytojas pradeda skaitymą operacija Start_Read ir užbaigia skaitymą kreipiniu End_Read. Atitinkamos proceso-rašytojo operacijos - Start_Write ir End_Write. Dabar būtų galima apibrėžti *neleistiną* (prieštaringą) procesų būseną tokia logine salyga: yra du procesai, vienas iš jų - rašytojas, kurie jau įvykdė operaciją Start_{Read|Write}, bet dar neiškvietė atitinkamos End_{Read|Write} operacijos. Vienas iš galimų sprendimų, kuris leidžia išvengti neapibrėžto proceso vykdymo atidėjimo, pateiktas žemiau (pagal [2]). Reikia atkreipti dėmesį, kad mums šiuo atveju nėra svarbu apsaugoto resurso prigimtis. Tai gali būti duomenų bazės įrašas, visa lentelė, ar netgi programinis bendro naudojimo kintamasis.

```

monitor Reader_Writer_Monitor is
    Readers: Integer := 0;
    Writing: Boolean := False;
    OK_to_Read, OK_to_Write: Condition;

    procedure Start_Read is
    begin
        if Writing or Non_Empty(OK_to_Write) then
            Wait(OK_to_Read);
        end if;
        Readers := Readers + 1;
        Signal(OK_to_Read);
    end Start_Read;

    procedure End_Read is
    begin
        Readers := Readers - 1;
        if Readers = 0 then Signal(OK_to_Write); end if;
    end End_Read;

    procedure Start_Write is
    begin
        if Readers <> 0 or Writing then
            Wait(OK_to_Write);
        end if;
        Writing := True;
    end Start_Write;

    procedure End_Write is
    begin
        Writing := False;
        if Non_Empty(OK_to_Read) then
            Signal(OK_to_Read);
        else
            Signal(OK_to_Write);
        end if;
    end End_Write;
end Reader_Writer_Monitor;

```

2.5.3 Trajektorijų išraiškos (*path expressions*)

Tai būdas deskriptyviai apibrėžti, kurie procesai ir kokia tvarka gali ar negali vykti vienu metu. Pagrindinės konstrukcijos (pavyzdžiai).

PATH read END

- “read” veiksmas išskirtinis.

PATH beginreading, read, endreading END

- nurodyti veiksmai vykdomi tik užduota tvarka (bet nebūtinai ją vykdo tas pats procesas)

PATH {read} END

- keletas veiksmų “read” gali vykti vienu metu;

PATH read | write END

- arba tik “read” arba “write” gali vykti tuo pačiu metu;

2.5.4 Trajektorijų išraiškų realizacija progr. kalboje ADA – *select* sakiny

```
select
  when CONDITION1 => accept ENTRY1
    sequence of statements;
  or when CONDITION2 => accept ENTRY2
    sequence of statements;
  or...
  else
    sequence of statements;
end select
```

Sąveika tarp kliento ir serverio įvyksta *rendezvous* () principu.

“Serverio” procesas priklausomai nuo sąlygos (*guard*) laukia “kliento” kreipinio. Klientas kviečia serverį nurodydamas iėjimo procedūrą (*ENTRY*). Įvyksta “rankos paspaudimas”.

2.5.5 Monitoriai ir Java

Su kiekvienu java objektu susietas monitorius.

```
synchronized(object) { /* Monitoriumi apsaugotas kodas */
}
```

Įvykio laukimas: *object.wait() / object.wait(millis)*

Pranešimas apie įvyki: *object.notify() / object.notifyAll()*

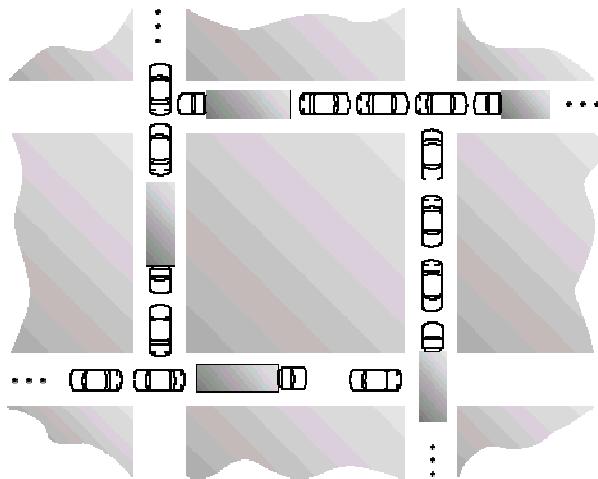
<pre>// Thread1: condition = true; object.notifyAll(); //----></pre>	<pre>// Thread2: while (!condition) object.wait();</pre>
---	--

```
/** Pavyzdys: reikšmė, kuri bus priskirta ateityje */  
  
public class Future {  
    Object value;  
    public Future(){value = this;}  
    /** Laukti, kol bus priskirta reikšmė */  
    synchronized Object getValue() {  
        while (this == value) {  
            try {  
                wait();  
            }  
            catch (InterruptedException exc) {}  
        }  
        return value;  
    }  
    /** Priskirti reikšmę ir informuoti visus laukiančiuosius */  
    synchronized void setValue(Object value) {  
        this.value = value;  
        notifyAll();  
    }  
}
```

2.6 Aklavietės

2.6.1 Aklavietės situacija

Aklavietė – tai situacija procesų sąveikoje, kai vienas ar daugiau procesų laukia įvykio, kuris niekada neįvyks. Artima aklavietei situacija - neapibrėžtas proceso vystymosi atidėjimas.



2.1 pav. Aklavietė sankryžoje

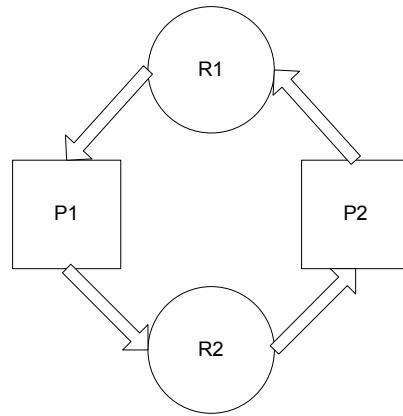
Pavyzdžiai: transporto kamšatis gatvėje, *spool'ing* sistemos, resursų išskyrimas.

Pagrindinės tyrimų kryptys

- aklaviečių prevencija,
- išvengimas,
- aptikimas
- atstatymas.

“Kova” su aklavietėmis iš esmės kompromisas tarp šių priemonių kaštų bei nuostolių, kuriuos iššaukia aklavietės padariniai.

Paprasčiausios aklavietės pavyzdys.



2.2 pav. Paprasčiausia aklavietė

Čia stačiakampiai žymi procesus P1 ir P2, o apskritimai – resursus R1 ir R2.

2.6.2 Keturios būtinosios aklaviečių susidarymo sąlygos

- Procesas išskirtinai valdo resursus (*monopolinio valdymo* sąlyga).
- Procesas laukia papildomo resurso, užlaikydamas jam išskirtus resursus (*laukimo* sąlyga).
- Resursai negali būti atimti iš procesų, kurie tuos resursus naudoja, iki resursai nebus pilnai panaudoti (*perskirstymo nebuvimas*).
- Egzistuoja uždara procesų *grandinė*, kurioje kiekvienas procesas, laiko vieną ar daugiau resursų kurių laukia sekantis resursas grandinėje.

2.6.3 Aklaviečių prevencija

Tai tokių procesų vystymosi sąlygų sudarymas, kai aklavietės negalimos iš princiopo. Taigi bent viena iš būtinujų sąlygų turi negalioti. [Havender, 1968].

2.6.3.1 Resursų monopolinio valdymo paneigimas.

Daugeliu atveju sunkiai įmanomas.

Pavyzdžiai. Pakartotinio naudojimo bibliotekos. Nesynchronizuoti konstantinių Java objektų metodai.

2.6.3.2 Laukimo sąlygos paneigimas.

Procesas turi pareikalauti visų resursų iš karto bei laukti, kol visi bus suteikti. Trūkumas – neefektyvus resursų panaudojimas.

2.6.3.3 Perskirstymo nebuvimo sąlygos paneigimas

Jeigu proceso pareiška resursui nepatenkinama (tuo metu laisvų resursų nėra) procesas turi atlaisvinti savo turimus resursus ir reikalauti resursų iš naujo. Trūkumai:

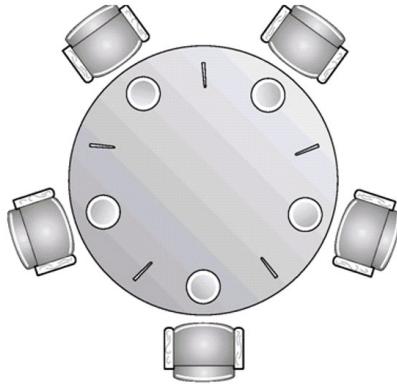
- prarasti darbo rezultatai atlaisvinant resursus;
- neapibrėžto proceso vystymosi atidėjimo galimybė.

2.6.3.4 Ciklinio laukimo sąlygos paneigimas

Resursai sunumeruojami. Kiekvienam procesui leidžiama reikalauti resurso tik su didesniu numeriu, negu jo valdomi resursai. Išnyksta ciklo susidarymo galimybė.

Trūkumas: sunkiai įgyvendinama.

2.6.3.5 „Pietaujančiųjų filosofų“ problemos sprendimai ?



2.3 pav. „Pietaujantys filosofai“

Tai klasikinis lygiagretaus programavimo uždavinys. Už stalo sėdi penki filosofai, kurie arba mąsto arba valgo. Valgymui reikalingos dvejos šakutės. Tarp bet kurių gretimų filosofų padėta tik viena šakutė, kurią valgydamas gali naudoti tik vienas. Uždavinio esmė - sugalvoti teisingą šakučių skirstymo strategiją. Pastebėsime, kad paprasčiausias skirstymo būdas, kai filosofas, pradėdamas valgyti, pradžioje ima kairiają šakutę, po to dešiniajają, - netinka, kadangi gali atvesti i aklavietę:

```
while(true){  
    imti_kairiaja();  
    imti_dešiniaja();  
    valgyti();  
    grąžinti_kairiaja();  
    grąžinti_dešiniaja();  
    mąstyti();  
}
```

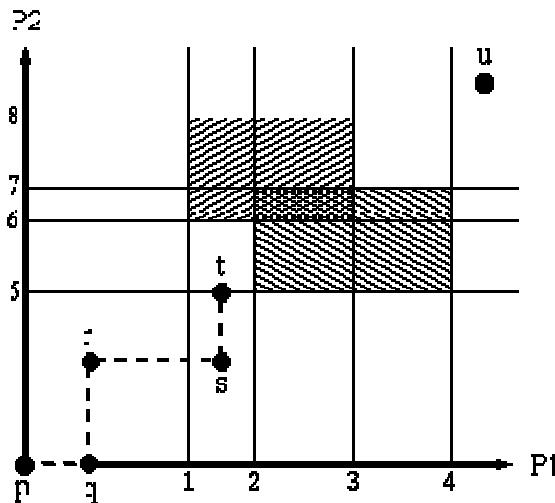
2.6.4 Aklaviečių išvengimas

Kalbama apie situacijas, kai visos keturios sąlygos gali (iš princiopo) egzistuoti, tačiau aklavietės apeinamos dinamiškai, renkantis ējimus tik į „saugias“ būsenas.

Būsena saugi, jeigu nėra aklavietės, ir egzistuoja bent viena procesų vystymosi trajektorija, „patenkinanti“ procesų poreikį resursams per baigtinį laiko tarap.

Saugios ir nesaugios būsenos 2-jų procesų ir 2 resursų atveju atskleidžiamos žemiau pateiktame paveikslėlyje. Pirmajam procesui laiko momentais (2,3) bei (3,4) reikalingi

atitinkamai pirmasis ir antrasis resursai, antrajam intervaluose (6,7) bei (7,8) reikalingi atitinkamai antrasis ir pirmasis resursai. Užbrūkšniuota dvigubais brūkšniais zona žymi aklavietę



2.4 pav. Saugios ir nesaugios būsenos

2.6.5 Bankininko algoritmas

Paskelbė Dijkstra 1965. Bankininkas turi tam tikrą pradinę pinigų sumą, kurią galimą skolinti klientams. Kiekvienas klientas pareiškia, kokios *maksimalios* sumos jam gali prieikti. Kaip įprasta, banke yra pinigų mažiau negu bendra galimų paraiškų suma. Jeigu klientas pasiskolino maksimalią pinigų sumą, jis privalo per *baigtinį* laiko tarpą gražinti visą paskolą. Problemos esmė - patenkinti klientų paraiškas per baigtinį laiką.

Palyginimui: bankininkas - OS, klientai - procesai, pinigai - resursai.

Bankininko algoritmas. Dijkstros resursų skirstymo algoritmo esmė – išskirti procesui reikiama kiekį resursų tik tuo atveju, jeigu išskyrimo rezultate susidaro *saugei būsena*. Apibrėžimas (rekursyvus):

- 1) galima patenkinti bent vieno proceso, kuriam mažiausiai trūksta iki maksimumo, paraišką resursams;
- 2) Ilykus 1-jam įvykiui, ir (vėliau) šiam procesui grąžinus visus turimus resursus, būsena išlieka saugi.

Bankininko algoritmo iliustracija (1)

Pradžioje:

Proc.Nr.	Panaudota	Maks.
I	0	6

II	0	5
III	0	4
IV	0	7
Liko: 10		

Bankininko algoritmo iliustracija (2) Po tam tikro laiko – ar saugios būsenos ?

Proc.Nr.	Panaudota	Maks.
I	1	6
II	1	5
III	2	4
IV	4	7
Liko: 2		

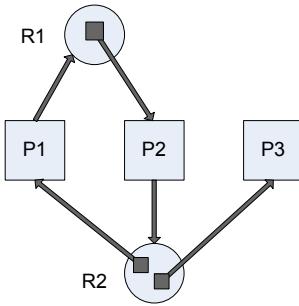
Proc.Nr.	Panaudota	Maks.
I	1	6
II	2	5
III	2	4
IV	4	7
Liko: 1		

Bankininko algoritmo trūkumai

- fiksuotas resursų skaičius;
- fiksuotas procesų skaičius;
- garantuotas aptarnavimas per baigtinių laiko tarpą, nors dažnai prireikia stipresnių garantijų;
- procesas privalo žinoti resursų poreikius iš anksto.

2.6.6 Aklaviečių aptikimas.

Vykdomas pagal resursų pasiskirstymo grafą. Gali brangiai kainuoti ($O(n^2)$ sudėtingumas).



2.5 pav. Resursų pasiskirstymo grafas be aklavietės

Pateiktajame paveikslėlyje procesai vaizduojami kvadratais, o resursai – apskritimais. Vienarūšių resursų gali būti keletas (žr. R2). Aklavietės dar nėra, kadangi procesui P3 gražinus resursą R2, procesas P2 turės visus reikalingus resursus, taigi ciklas – gali būti pertrauktas. Minėtasis procesas vadinamas procesu – resursų grafo redukcija.

2.6.7 Aklaviečių atstatymas

Gali remtis transakcijų mechanizmu. Susiję su dailies atlikto darbo praradimu. Naudojamas, pavyzdžiu, duomenų bazių valdymo sistemoje. Vienas iš procesų, kuris kreipdamasis į duomenų bazės įrašus sukuria aklavietės situaciją, pasirenkamas kaip „auka“ ir informuojamas apie nesėkmingą transakcijos rezultatą, todėl turi pakartoti savo nepatvirtintus veiksmus.

2.7 Asinhroninių procesų modeliavimas Petri tinklais

Paprasčiausios aklavietės pavyzdys (C pseudokodas, WIN32 posistemė).*

CRITICAL_SECTION cs1,cs2;	
// Thread1 while (TRUE){ EnterCriticalSection(&cs1); printf("T1: +1 -2.\n"); EnterCriticalSection(&cs2); printf("T1: +1 +2.\n"); LeaveCriticalSection(&cs2); printf("T1: +1 -2.\n"); LeaveCriticalSection(&cs1); printf("T1: -1 -2.\n");}	// Thread2 while (TRUE){ EnterCriticalSection(&cs2); printf("T2: -1 +2.\n"); EnterCriticalSection(&cs1); printf("T2: +1 +2.\n"); LeaveCriticalSection(&cs1); printf("T2: -1 +2.\n"); LeaveCriticalSection(&cs2); printf("T2: -1 -2.\n");}

* Skyrelyje naudojama [3] medžiaga.

Pastaba: nors aklavietė galima, ne visi scenarijai veda į aklavietę.

Tam, kad įrodyti aklavietės galimybę reikia pateikti tokio scenarijaus pavyzdį. Kur kas sunkiau (o dažnai ir neįmanoma) įrodyti aklavietės negalimumą.

CRITICAL_SECTION cs1,cs2;	
// Thread1 while (TRUE){ EnterCriticalSection(&cs1); Printf("T1: +1 -2.\n"); EnterCriticalSection(&cs2); Printf("T1: +1 +2.\n"); LeaveCriticalSection(&cs2); Printf("T1: +1 -2.\n"); LeaveCriticalSection(&cs1); Printf("T1: -1 -2.\n");}	// Thread2 while (TRUE){ EnterCriticalSection(&cs1); Printf("T1: +1 -2.\n"); EnterCriticalSection(&cs2); printf("T1: +1 +2.\n"); LeaveCriticalSection(&cs2); printf("T1: +1 -2.\n"); LeaveCriticalSection(&cs1); printf("T1: -1 -2.\n");}

Formalus įrodymas remiasi invariantų analize.

Teisinga:

- galima nustatyti ar procesai “pakliuvo” į aklaviete (pagal resursų priklausomybės grafa);
- invariantai gali būti apskaičiuojami;
- galima nustatyti ar tam tikras aklavietės scenarijus yra “įgyvendinamas” iš pradinės būsenos.

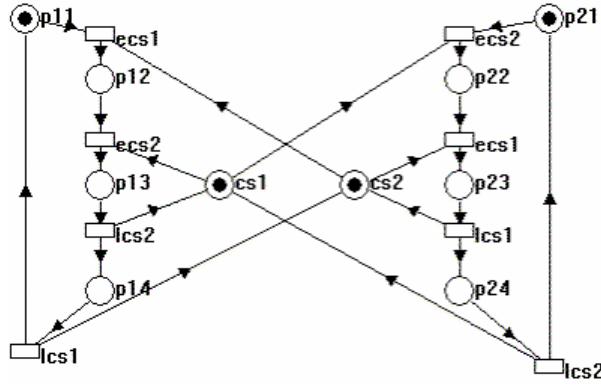
Petri (1970) tinklai įgalina formaliajų asinhroninių konkuruojančių procesų analizę ir vizualų modeliavimą.

Petri tinklas – orientuotas grafas, kurio viršūnės arba *vietos* (*places*) arba *perėjimai* (*transitions*). Briauna gali jungti tik vietą su perėjimu.

Tam tikros vietos yra pažymėtos. Prosesų galimi vystymosi keliai modeliuojami, perkeliant žymes pagal tam tikrą taisyklę. Jeigu visas vietos vedančios į perėjimą yra pažymėtos ir visas išėjimo iš duotojo perėjimo vietos yra žymiu, tai perėjimas gali būti aktyvuotas, nutrinant visas jėjimo vietose esančias žymes ir pažymint išėjimo vietas žymėmis.

Jeigu negalimas joks žymiu perkėlimas, vadinasi procesai atsidūrė aklavietėje.

Petri tinklo pavyzdys, iliustruojantis minėtasių kritines sekcijas (su aklaviete) pateiktas žemiau.



2.6 pav. Petri tinklo pavyzdys

Incidentiškumo matrica \mathbf{m} , atitinkanti šį grafi:

	ecs1	ecs2	lcs2	lcs1	ecs2	ecs1	lcs1	lcs2
p11	-1	0	0	1	0	0	0	0
p12	1	-1	0	0	0	0	0	0
p13	0	1	-1	0	0	0	0	0
p14	0	0	1	-1	0	0	0	0
cs1	0	-1	1	0	-1	0	0	1
cs2	-1	0	0	1	0	-1	1	0
p21	0	0	0	0	-1	0	0	1
p22	0	0	0	0	1	-1	0	0
p23	0	0	0	0	0	1	-1	0
p24	0	0	0	0	0	0	1	-1

Lygties $\mathbf{v}^* \mathbf{m} = \mathbf{0}$ sprendinys V išvardija invariantus. Kitaip tariant, jeigu v_0 – pradinė grafo būsena (pradinis žymėjimas), o v_n – bet kuris sprendinys, gautas pritaikius vieną ar daugiau perejimų (pridėjus vieną ar daugiau matricos m stulpelių):

$$\mathbf{v}^* v_0 = \mathbf{v}^* v_n = \text{const.}$$

Null-erdvės baziniai vektoriai:

Invariante Nr.	p11	p12	p13	p14	cs1	cs2	p21	p22	p23	p24
1		1	1	1	1	0	0	0	0	0
2		0	0	-1	0	-1	0	1	0	0
3		-1	0	0	0	0	1	0	0	1
4		1	0	1	0	1	-1	0	1	0

Arba:

$$\begin{aligned}
 p_{11} + p_{12} + p_{13} + p_{14} &= \text{constant} \\
 p_{21} - p_{13} - cs_1 &= \text{constant} \\
 cs_2 + p_{23} - p_{11} &= \text{constant} \\
 p_{11} + p_{13} + cs_1 - cs_2 + p_{22} + p_{24} &= \text{constant}
 \end{aligned}$$

Jeigu imti pradinį žymėjimą, kaip parodyta paveikslėlyje, gausime dešinės pusės konstantas:

$$\begin{aligned} 0*p_{11}+0*p_{12}+1*p_{13}+0*p_{14} &= 1 \\ 0*p_{21}-1*p_{13}-0*cs_1 &= -1 \\ 0*cs_2+1*p_{23}-0*p_{11} &= 1 \\ 0*p_{11}+1*p_{13}+0*cs_1-0*cs_2+0*p_{22}+0*p_{24} &= 1 \end{aligned}$$

Taigi invariantai turi pavidalą:

$$\begin{aligned} p_{11}+p_{12}+p_{13}+p_{14} &= 1 \\ p_{21}-p_{13}-cs_1 &= -1 \\ cs_2+p_{23}-p_{11} &= 1 \\ p_{11}+p_{13}+cs_1-cs_2+p_{22}+p_{24} &= 1 \end{aligned}$$

Bet kurios pasiekiamos būsenos žymėjimai turi tenkinti invariantus. Pavyzdžiu, aklavietės situacija (pažymėtos viršūnės p_{12} , p_{22}) duotąsias tapatybes tenkina, taigi aklavietė yra galima. Pastaba, invariantai nusako būtinąsias, bet ne pakankamąsias aklavietės sąlygas.

Visoms keturioms lygtims galima suteikti interpretacijas, pvz. lygtis

$$p_{11}+p_{12}+p_{13}+p_{14} = 1, \text{ atsižvelgiant į neneigiamas pių reikšmes, reiškia, jog pirmasis procesas gali būti tik vienoje iš duotujų būsenų } \{p_{11}, p_{12}, p_{13}, p_{14}\}.$$

2.8 Gijų panaudojimas, išlygiagretinant algoritmus

2.8.1 Programos darbo spartinimas daugiaiprocesorinėje aplinkoje

Daugiaiprocesorinėje aplinkoje programos vykdymą galime paspartinti, išskaidydam i mažai priklausomą jos dalių vykdymą į keletą gijų.

Yra įvairūs būdai tai atlikti, žymiai įtakojantys programos vykdymo efektyvumą.

Daugiagijęs programos efektyvumą riboja šie veiksnių:

- synchronizacijos nuostoliai,
- varžymasis
- balanso nebuvimas.

Synchronizacija užima laiko net ir tuo atveju, kai nėra varžymosi.

Varžymasis dėl bendrų resursų sukelia sistemos darbo sulėtėjimą.

Netolygios procesorių apkrovos iššaukia balanso nuostolius. Išbalansavimą dažnai sukelia darbo padalijimas į didelius "gabalus", tačiau per didelis darbų "susmulkinimas" gali grėsti varžymosi nuostoliais.

Bendrosios atminties kompiuteriuose taip pat reikia atsižvelgti į sparčiosios atminties (*cache*) netikslingo atnaujinimo (*false caching*) problemą.

Nagrinėjamos kelios paprastos paradigmos (išlygiagretinimo schemas), tinkančios daugeliui uždaviniių išskiriamos pagal tai,

- ar darbas yra dalijamas po lygiai tarp atskirų gijų, ir ar visos gijos vykdo tą patį kodą;
- kaip gijos sinchronizuojamos;
- kokie duomenys yra dalomi tarp gijų.

Nagrinėsime schemas:

- **šeimininko – tarno** (*master-slave*).
- **darbų krūvos.**
- **konvejerio.**

2.8.2 Šeimininko - tarno schema

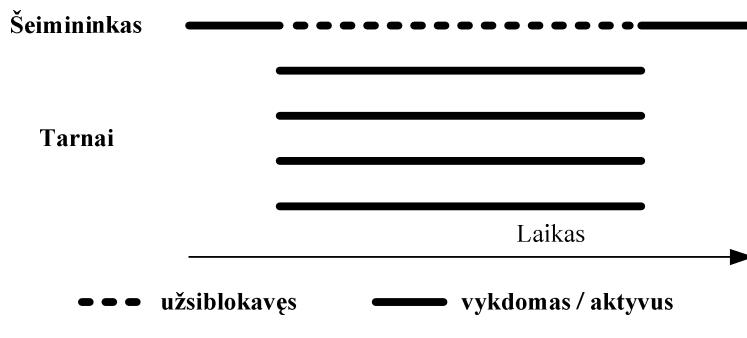
Viena gija (šeimininkas) startuoja gijas-tarnus, kurioms paskiria darbus po lygiai. Po to šeimininkas laukia, kiekvieno tarno sinchronizaciniame taške - *barjere*.

Surinkęs rezultatus, šeimininkas gali baigtis darbą ar kartoti šį procesą cikle.

Uždaviniai - dviejų matricų daugyba, paveikslėlio elementų lokalus apdrojimas ir kt., kuriuose atliekamo darbo apimtis žinoma iš anksto.

Programos spartėjimas priklauso nuo darbo "grūdėtumo." Kaip tipiška sinchronizacijos schema naudojamas barjeras.

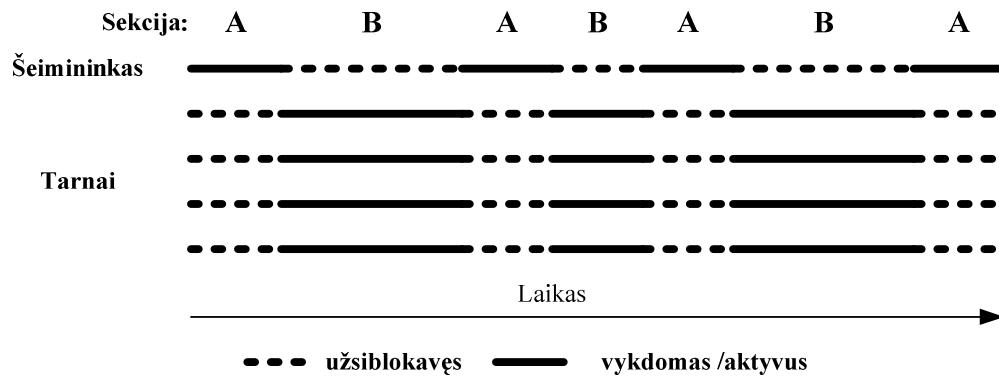
Barjeras inicijuojamas dalyvaujančiu barjere gijų skaičiumi. Kreipiantis į barjerą, gijos darbas yra sustabdomas tol, kol nurodytas gijų skaičius nepasieks barjero. Tada gijos gali pratęsti darbą. Žemiau esantis paveikslėlis pateikia gijų, dalyvaujančių barjero operacijoje, būsenas.



Šeimininko-tarno schema, naudojanti paprastą barjerą.

<u>Šeimininkas</u>	<u>// Tarnas</u>
<p><i>Paruošiamieji veiksmai.</i></p> <pre>for (i = 0; i < n; i++) { /* paleisti tarnus */ pthread_create(..., slave, ...); }</pre> <p><i>Imanomas Šeimininko apdorojimas.</i></p> <pre>for (i = 0; i < n; i++) pthread_join(...);</pre> <p><i>Baigiamieji veiksmai.</i></p>	<p><i>slave(void* params){</i></p> <p><i>Nustatyti indeksą į pagal params,</i> <i>kuris nustato kokį darbą imti iš</i> <i>globalios atminties.</i></p> <p><i>Apskaičiuoti rezultatus ir patalpinti į</i> <i>globalią atmintį.</i></p> <pre>pthread_exit();</pre>

Galimas ir kartotinis barjeras, kuriame gijas nebūtina startuoti iš naujo.



Žemiau pateiki programos fragmentai iliustruoja pasikartojančias barjero operacijas šeimininko - tarno schemaje.

<u>Šeimininkas</u>	<u>Tarnas</u>
<p><i>Paruošiamieji veiksmai.</i></p> <pre>for (i = 0; i < n; i++) { /* paleisti tarnus */ pthread_create(..., slave, ...);</pre>	<p><i>slave(void* params){</i></p>

<pre> } while (!done) { A: paruošti darbus tarnams barrier_wait(bar); B: Imanomas šeimininko darbas /* Laukti tarnų */ barrier_wait(bar); A: apdoroti rezultatą. } for (i = 0; i < n; i++) pthread_join(...); Baigiamieji veiksmai. </pre>	<pre> while (!done) { A: laukti darbo iš šeimininko barrier_wait(bar); B: nustatyti darbą pagal params. Apskaičiuoti rezultatus ir patalpinti į globalią atmintį. /* Synchronizuoti su liku- siais procesais*/ barrier_wait(bar); } pthread_exit(); } </pre>
--	--

2.8.3 Darbų krūvos schema

Darbinės gijos nuskaito darbus iš *darbų krūvos*, kurią paprastai atitinka synchronizuota eilė. Darbo pasėkoje gali atsirasti nauji darbai, kurie talpinami į tą pačią krūvą. Procesas baigiamas, kai visos gijos tampa laisvos, o darbų krūva - tuščia.

Skirtingai negu *Šeimininko - torno* schema, darbai yra skirtini, todėl darbų krūva sumažina gijų prastovas.

Pavyzdžiai - skaldyk-valdyk tipo algoritmai (*quicksort*, paieškos grafuose uždaviniai), ciklo išlygiagretinimas, spinduliu trasavimas (*ray tracing*) ir pan.

Darbų krūvos realizacija – synchronizuota eilė

```

while ( (job = queue -> getJob() ) != null) {
    Apdoroti job.
    [Galbūt, pridėti naujus darbus į darbų eilę.]
}

```

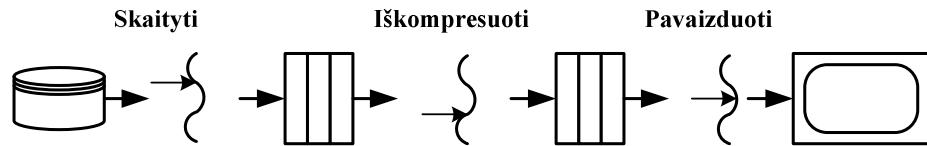
2.8.4 Konvejerio schema

Šioje schema kiekvienas darbas yra perduodamas gijų konvejeriui, kurioje kiekvienna gija atlieka tam tikrą darbo dalį.

Daugeliu atveju atliekami darbai kiekvienoje konvejerio grandyje yra skirtini, tačiau galimi ir homogeniniai konvejeriai.

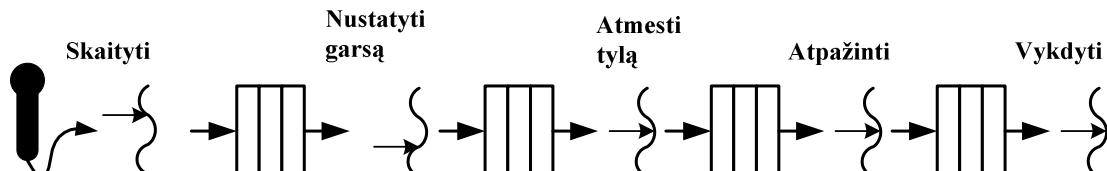
Realizuojant šia schemą kiekvienna gija pora sąveikauja kaip *gamintojas/vartotojas*.

Pavyzdžiai. Paveikslėlio pavaizdavimas.



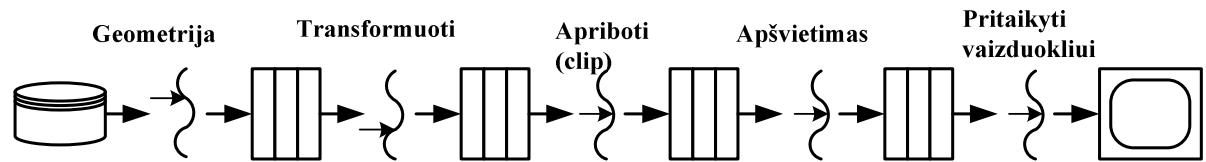
2.9 pav. Paveikslėlio pateiktis (*rendering*)

Audio-apdorojimas: nuskaityti signalą, išskirti garsus, apjungti į komandas, interpretuoti.



2.10 pav. Audio-apdorojimas

Geometrinio aprašo pateiktis.



2.11 pav. Geometrinio aprašo patektis

2.9 Gijų realizacijos. POSIX gijos

2.9.1 Standartai

Bendrai naudojamos atminties architektūrose, pavyzdžiui SMP, lygiagretusis programavimas gali būti vykdomas gijomis.* Gijų realizacijos nėra naujas dalykas. Istorikai kiekvienas techninės įrangos gamintojas pateikia savo specifinę realizaciją. Šių realizacijų skirtumai sudaro sunkumus, rašant pernešamas gijų programas. Pastangos standartizuoti gijų įgyvendinimą įtakojo dviejų skirtingų standartų atsiradimą: **POSIX Threads** ir **OpenMP**.

Standartas **POSIX Threads**

- Remiasi funkcijų biblioteka; reikalauja išreikštinio lygiagretumo kodavimo.
- Specifikuotas IEEE POSIX 1003.1c standartu (1995).
- skirtas tik C programavimo kalbai
- visuotinai primtas pavadinimas “Pthreads”.

* Skyriuje naudojama [4] medžiaga.

- Daugelis techninės įrangos tiekėjų pateikia POSIX gijų realizaciją kartu su nuosava.
- Lygiagretumas visiškai išreikštinis, todėl reikalauja žymaus programuotojo dėmesio kodo detalėms.

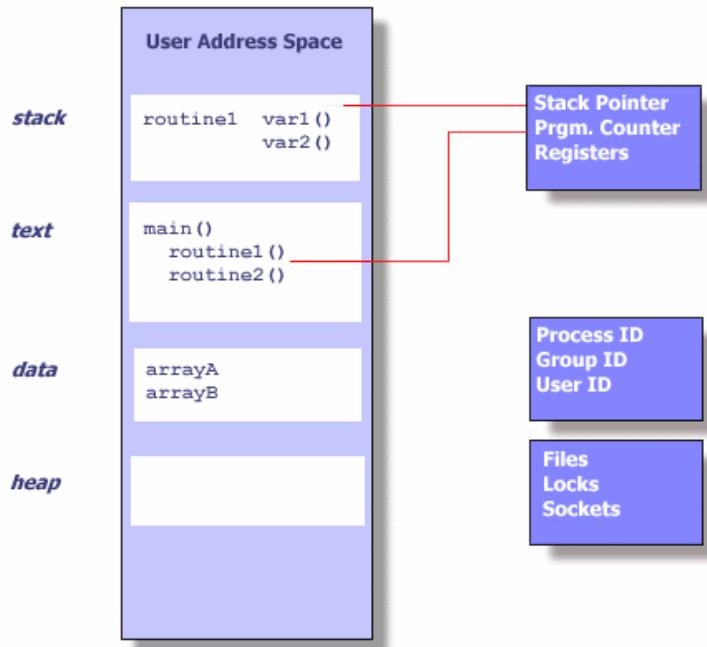
OpenMP

- Remiasi kompiliatoriaus direktyvomis; gali naudoti nuoseklujį kodą.
- Standartą apibrėžė jungtinė pagrindinių programinės ir techninės įrangos gamintojų grupė. OpenMP API (taikomujų programų sąsaja) Fortran programavimo kalbai buvo išleista 1997 spalio 28 d. C/C++ API išleista 1998-jų pabaigoje..
- Standartas pernešamas, daugiaplatforminis, išskaitant Unix ir Windows NT platformas.
- Tinka C/C++ ir Fortran programavimo kalboms.
- Gali būti labai lengvai panaudojimas – tinkamas igyvendinti “pažingsnių lygiagretumą”.

Microsoft korporacija turi nuosavą gijų realizaciją, Win32, kuri yra visiškai nesusijusi nei su UNIX POSIX standartu nei su OpenMP.

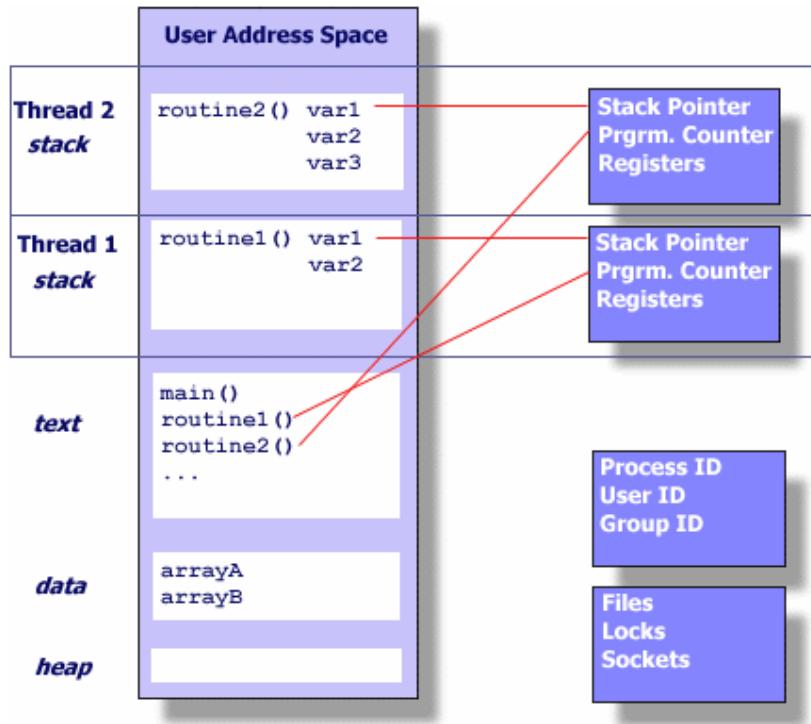
2.9.2 Kas yra gija

- Tai nepriklausomai vykdoma instrukcijų seka. UNIX aplinkoje gija egzistuoja proceso viduje ir naudoja proceso resursus. Tačiau gija užtikrina nepriklausomą komandų sekos vykdymo kontrolę. Proceso viduje gali būti kelios vykdomos gijos. Geriausiai būtų galima nusakyti giją kaip “procedūrą”, kuri vykdoma nepriklausoma tēkme proceso viduje.



2.12 pav. UNIX procesas [4]

- Tam, kad geriau suprasti gijos “sandarą”, vertėtų išsiaiškinti sąryšį tarp proceso ir gijos. Procesą sukuria operacijų sistema (OS). Procesas turi informaciją apie programos resursus bei programos vykdymo būseną, įskaitant:
 - proceso ID, proceso grupės ID, naudotojo ID bei naudotojo grupės ID;
 - aplinką (*environment*);
 - darbinį failų katalogą;
 - programos komandas;
 - registrus;
 - steką;
 - bendrą adresinę erdvę, duomenis ir atminties *heap*'ą;
 - failų deskriptorius;
 - signalų veiksmus;
 - bendro naudojimo(*shared*) bibliotekas;
 - tarpprocesinės komunikacijos priemones (pranešimų eiles, kanalus, semaforus, bendrają atmintį ir pan.).



2.13 pav. Gijos proceso ribose [4]

- Gijos naudojamos ir egzistuoja šių proceso resursų viduje ir gali būti vykdomos OS kaip nepriklausomi vienetai procese.
- Gija gali įgyvendinti nepriklausomą komandų srauto valdymą, kadangi ji turi nuosavą:
 - steką ;
 - vykdomumo (*scheduling*) ypatybes (tokias kaip saugumą, prioritetus)
 - signalų aibę;
 - gjai specifinius duomenis.
- Procesas gali turėti daugelį gijų, kurios visos dalijasi proceso resursais bei naudoja tą pačią adresinę erdvę. Daugelio gijų programoje vienu metu egzistuoja, keli tokie vykdymo vienetai.

2.9.3 Gijų nauda

- Pirminis motyvas – tai galimybė vykdyti programą efektyviau. Lyginant su procesu sukūrimo ir valdymo kaina, gija gali būti paleista su žymiai mažesnėmis OS išlaidomis negu procesams. Tarkime, žemiu pateikta lentelė sulygina komandos, skirtos paleisti procesus, - **fork()**- bei gijų paleidimo procedūros - **pthread_create()** -

paleidimo kaštus. Laikas nurodytas sekundėmis, paleidžiant procesą/gijas 50,000 kartų be jokios optimizacijos. Gijų paleidimo pranašumai akivaizdūs.

Platforma	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM 375 MHz POWER3	61.94	3.49	53.74	7.46	2.76	6.79
IBM 1.5 GHz POWER4	44.08	2.21	40.27	1.49	0.97	0.97
IBM 1.9 GHz POWER5 p5-575	50.66	3.32	42.75	1.13	0.54	0.75
INTEL 2.4 GHz Xeon	23.81	3.12	8.97	1.70	0.53	0.30
INTEL 1.4 GHz Itanium 2	23.61	0.12	3.42	2.10	0.04	0.01

- Visos gijos proceso viduje naudoja tą pačią adresinę erdvę, todėl komunikacija tarp gijų gali būti efektyvi bei paprastesnė negu tarpprocesinė komunikacija.
- Gijos gali dirbti efektyviau negu nuosekli programa, jeigu įvedimo/išvedimo operacijos persipina su skaičiavimais.
- Gijos tinkamos apdoroti asynchroninius įvykius, kai reikia laukti signalo bei tuo pačiu metu apdoroti pateiktąsias kliento paraiškas (web serveriai)
- SMP konfiguracijoje procesoriai gali efektyviaus keisti duomenimis paprastu atminties blokų kopijavimu negu MPI pranešimų perdavimo kreipiniai (žr. žemiau pateiktą lentelę).

Platforma	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
IBM 375 MHz POWER3	0.5	16
IBM 1.5 GHz POWER4	2.1	11
Intel 1.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

2.9.4 Gijų programų panaudojimo modeliai

Išskiriami šie (bendriausi) modeliai:

- Šeimininkas / tarnas (manager / worker):** viena gija, šeimininkas, paskiria darbus, kuriuos vykdo kitos gijos – tarnai. Paprastai šeimininkas užsiima duomenų įvedimu bei rezultatų pateiktimi. Galima išskirti pagrindines šio modelio variacijas: statinė gijų-tarnų krūva arba dinamiškai kintanti krūva.

- **Konvejeris (pipeline):** užduotis kurią reikia atlikti suskaidoma į suboperacijų serijas, bet kiekvieną serijos dalinę operaciją atlieka skirtingos gijos. Analogiją automobilių gamybos konvejeris.
- **Lygiaverčių gijų (peer):** modelis panašus į šeimininko torno modelį, tačiau pagrindinė gija, paleidusi “tarnus, dalyvauja darbe kartu su kitomis gijomis.

2.9.5 Pthreads sasaja (API)

Yra apibrėžta ANSI/IEEE POSIX standartu. Funkcijos, sudarančios Pthreads API, gali būti neformaliai suskirstytos į tris pagrindines klases :

1. **Valdymo:** šios funkcijos tiesiogiai operuoja gijos objektais: sukuria, atskiria (*detach*), sulieja (*join*), ir t.t.. Be to priklauso gijų atributų keitimo funkcijos (pavyzdžiu prioritetų keitimo).
2. **Mutex'ai:** ši funkcijų šeima skirta gijų sinchronizacijai taip vadinamais *mutex'ais* (angl. santrumpa žodžių *mutual exclusion*) – semaforai, naudojami užtikrinti kritines sekcijas. *Mutex* funkcijos užtikrina semaforų sukūrimą, sunaikinimą, užrakinimą ir atrakinimą. Taip pat priklauso funkcijos, modifikuojančios mutex'o atributus.
3. **Ivykių (condition)** kintamieji: trečioji funkcijų klasė skirta užtikrinti “švelnesnę”(finer) negu mutex'ai sinchronizaciją. Jeina funkcijos leidžiančios sukurti, sunaikinti ivykių kintamuosius, laukti ivykio, signalizuoti apie ivyklį. Taip pat priklauso funkcijos leidžiančios užklausti ir nustatyti ivykių kintamujų atributus.

Žymėjimo susitarimai: visi gijų bibliotekos identifikatoriai prasideda **pthread_**.

Priešdėlis	Funkcinė grupė
pthread_	Manipuliacija gijomis bei kitos funkcijos
pthread_attr_	Gijų atributai
pthread_mutex_	Mutex (semaforai)
pthread_mutexattr_	Mutex atributai
pthread_cond_	Ivykių kintamieji (semaforai)
pthread_condattr_	Ivykių atributai
pthread_key_	Gijoms specifiški duomenys

Bibliotekos *Pthreads* API sudaro virš 60 funkcijų.

2.9.5.1 Gijų sukūrimas/sunaikinimas

- Pirminė programos gija main() funkciją. Kitos gijos turi būti sukurtos išreikštiniu būdu.
- Funkcijos:

pthread_create (thread,attr,start_routine,arg)

- Sukuria naują giją ir paleidžia vykdymui. Naujai paleista gija gali sukurti naujas gijas.
- Funkcija “pthread_create” gražina naujai sukurtos gijos identifikatorių (ID), naudojant *thread* argumentą. Kvietėjas gali panaudoti gautąjį ID tam, kad atlikti su gija įvairias operacijas. Privalu ID patikrinti, ar iš tiesų gija buvo sėkmingai sukurta.
- Parametras *attr* yra panaudojamas, kad priskirti gijai atributus. Galima nurodyti konkretų atributų objektą arba NULL, siekian naudoti standartinius atributus.
- Funkcija *start_routine* yra C funkcija, kuri bus vykdoma, sukūrus giją .
- Vienintėlis rodyklės tipo argumentas *arg* gali būti perduodamas funkcijai *start_routine*.

2.9.5.2 Gijos vykdymo nutraukimas

Galimi keli būdai nutraukti *Pthread* :

- užbaigti paleistąją giją (arba pirminę giją) grįzimo sakiniu *return*. Pagal nutylėjimą, Pthread biblioteka atlaisvins visus sistemionius resursus, kuriuos naudojo gija. Tai panašu į proceso užbaigimą, grįžus iš main.
- Įvykdyti išreikštinių gijos darbo nutraukimą funkcija pthread_exit (nagrinėjama žemiau).
- Giją užbaigia kita gija, kviesdama funkciją pthread_cancel.
- Gija gauna signalą užbaigti giją.
- Užbaigiamas procesas, kuriam priklauso gija, dėl exec arba exit funkcijų kvietinių.
- Funkcijos:

pthread_exit (status)

- Ši funkcija naudojama išreikštinių būdu užbaigti gijos darbą..

Kodo pavyzdys: gijų sukūrimas, sunaikinimas, argumentų perdavimas.

```
#include <pthread.h>
#include <stdio.h>
```

```

#define NUM_THREADS      5

void *PrintHello(void *threadid)
{
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]){
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++){
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello,
(void *)t);
        if (rc)
            printf("ERROR; return code from pthread_create() is %d\n",
rc);
            exit(-1);
    }
    pthread_exit(NULL);
}

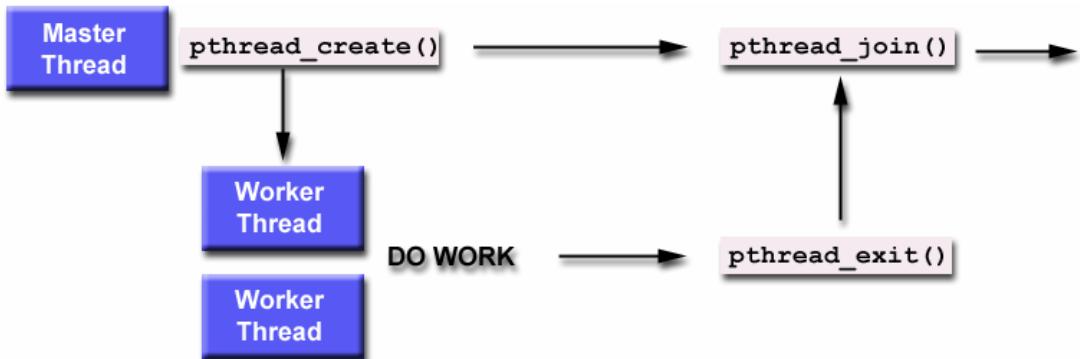
```

PThread identifikatoriai

[pthread_self\(\)](#)

[pthread_equal\(thread1,thread2\)](#)

2.9.5.3 Gijų „suliejimas” (*join*)



2.14 pav. Gijų suliejimas [4]

- "Suliejimas" yra vienas iš gijų sinchronizacijos būdų. Kiti būdai, kurie remiasi semaforais, bus aptarti vėliau.

- funkcijos:

pthread_join (threadid,status)

- Funkcija `pthread_join()` blokuoja kviečiančiąją giją tol, kol nurodyta gija `threadid` baigia darbą.
- Programuotojas gali užklausti gijos baigmės būseną, jeigu nurodyta naudojant `pthread_exit()`, `status` parametru.
- Yra neįmanoma sulieti atskirtą (*detached*) giją.

Sukuriant giją, galima nurodyti atributą, kuris apibrėžia ar gija gali būti suljeta ar ne. Atskirta (*detached*) - reiškia jog gija negali būti suljeta. Neatskirta (*undetached*) reiškia – suliejimas įmanomas.

Funkcijos:

pthread_attr_init (attr)

pthread_attr_setdetachstate (attr,detachstate)

pthread_attr_getdetachstate (attr,detachstate)

pthread_attr_destroy (attr)

pthread_detach (threadid,status)

```
/* Programa demonstruoja, kaip laukti giju baigmės */
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 3

void *BusyWork(void *null)
{
    int i;
    double result=0.0;
    for (i=0; i<1000000; i++)
    {
        result = result + (double)random();
    }
    printf("result = %e\n",result);
    pthread_exit((void *) 0);
}

int main (int argc, char *argv[])
{
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc, t, status;
```

```

/* Initialize and set thread detached attribute */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);

for(t=0; t<NUM_THREADS; t++)
{
    printf("Creating thread %d\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork,
NULL);
    if (rc)
    {
        printf("ERROR; return code from pthread_create()
is %d\n", rc);
        exit(-1);
    }
}

/* Free attribute and wait for the other threads */
pthread_attr_destroy(&attr);
for(t=0; t<NUM_THREADS; t++)
{
    rc = pthread_join(thread[t], (void **) &status);
    if (rc)
    {
        printf("ERROR; return code from pthread_join()
is %d\n", rc);
        exit(-1);
    }
    printf("Completed join with thread %d status=%d\n", t, status);
}

pthread_exit(NULL);
}

```

2.9.6 Muteksai (*mutex*)

Mutex'ų sukūrimas / sunaikinimas

- Funkcijos:

pthread_mutex_init (mutex,attr)
pthread_mutex_destroy (mutex)
pthread_mutexattr_init (attr)
pthread_mutexattr_destroy (attr)

- Mutex kintamieji turi būti apibrėžti, naudojant tipą `pthread_mutex_t`, ir, prieš naudojant turi būti iniciuoti. Yra du būdai iniciuoti mutex kintamaji:

- Statiškai, metu. Pavyzdžiu:

pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

- Dinamiškai, naudojant pthread_mutex_init() funkciją. Šis metodas įgalina priskirti mutex objekto atributus *attr*.

Pradžioje mutex yra *atrakintas*.

- Objektas *attr* object gali būti naudojamas tam, kad priskirti mutex objektui atributus, ir privalo būti pthread_mutexattr_t tipo, jeigu naudojamas (arba gali būti specifikuotas kaip NULL, kad priimti numatytais reikšmes).
- Funkcijos pthread_mutexattr_init() ir pthread_mutexattr_destroy() naudojamos atitinkamai mutex atributų sukūrimui arba sunaikinimui.
- pthread_mutex_destroy() funkcija naudojama nereikalingo mutex objekto atlaisvinimui.

Mutex užrakinimas (*lock*) / atrakinimas (*unlock*).

- Funkcijos:

pthread_mutex_lock (mutex)

pthread_mutex_trylock (mutex)

pthread_mutex_unlock (mutex)

Mutex objektas veikia kaip binarinis semaforas ir naudojamas kritinės sekcijos užtikrinimui. Funkcija pthread_mutex_lock() naudojama mutex semaforo užrakinimui. Jeigu tuo metu semaforą yra užrakinusi kita gija, šis kreipinys blokuosis tol, kol semaforas bus atrakintas. Kai ši funkcija grįžta, kviečiančioji gija tapo semaforo savininku.

Funkcija pthread_mutex_trylock() bandoma semaforą užrakinti. Bet skirtingai negu funkcijos pthread_mutex_lock() atveju, gija nesiblokuoja. Jeigu semaforas užrakinamas, funkcija gražins klaidos kodą , reiškiantį "užimta". Ši funkcija yra naudinga, vykdant aklaviečių prevenciją, arba situacijoje, kai keičiamos gijų prioritetai.

Funkcija pthread_mutex_unlock() atrakina mutex'ą. Privaloma, jog atrakintu gija – mutex'o savininkė. Semaforas paprastai atrakinamas, kai gija užbaigė savo darbą su kritiškais duomenimis. Bus iššaukta klaida, jeigu

- mutex'as jau buvo atrakintas,
- mutex'o savininkas yra kita gija.

```
#include <pthread.h>
pthread_mutex_t mutexsum;
//...
```

```

/* Užrakinti mutex'ą, prieš keičiant reikšmes kelių giju
naudojamame duomenų segmente */
pthread_mutex_lock (&mutexsum);
dotstr.sum += mysum;
pthread_mutex_unlock (&mutexsum);
//...

```

2.9.7 Įvykių kintamieji (*condition variables*)

Įvykių kintamujų sukūrimas/sunaikinimas. Funkcijos:

pthread_cond_init (condition,attr)
pthread_cond_destroy (condition)
pthread_condattr_init (attr)
pthread_condattr_destroy (attr)

- Įvykių kintamieji turi būti apibrėžti kaip tipo pthread_cond_t kintamieji, ir privalo būti inicializuojami prieš panaudojimą. Šis metodas leidžia priskirti pradinius atributus *attr*.
- Nebūtinės *attr* objektas naudojamas atributų priskyrimui. Šiuo metu žinomas tik vienas atributas "process-shared", kuris įgalina įvykių kintamajam būti matomu kituose procesuose. Atributo (jei naudojamas) tipas privalo būti pthread_condattr_t (arba NULL – nustatyta jai reikšmei).
- Funkcijos pthread_condattr_init() bei pthread_condattr_destroy() naudojamos atributų nustatymui ir sunaikinimui
- Kai įvykių kintamasis nereikalingas, jis reikia atlaisvinti funkcija pthread_cond_destroy().

Įvykių laukimas (*wait*) / pranešimas apie įvykį (*signal*). Funkcijos:

pthread_cond_wait (condition,mutex)
pthread_cond_signal (condition)
pthread_cond_broadcast (condition)

- Funkcija pthread_cond_wait() blokuoja kviečiančią giją tol, kol bus gautas signalas įvykiui *condition*. Prieš iškviečiant šią funkciją, gija turi būti nurodytojo *mutex* savininku. Laikotarpiui, kol gija laukia įvykio, mutex'as yra automatiškai atrakinamas. Aišku, kažkuriuo metu gija privalo atraktinti *mutex'q*, po to kai gautas signalas.
- Funkcija pthread_cond_signal() yra naudojama, kad signalizuoti (t.y., pažadinti) kitą giją, kuri laukia įvykio. Minėtoji funkcija turi būti iškviesta, esant *mutex'o* savininkui

bei turi atrakinti *mutex'q* tam, kad funkcija for pthread_cond_wait() įvykdytų savo darbą.

- Funkcija pthread_cond_broadcast() gali būti panaudojama pranešti apie įvykį , užuot naudojus pthread_cond_signal(), jeigu pranešimo gavėjas yra daugiau negu viena gija.
- Yra loginė klaida, jeigu kviečiamas pthread_cond_signal(), prieš kviečiant funkciją pthread_cond_wait().

„Teisingo“ įvykių semaforų panaudojimo pavyzdys

```
// WAIT
pthread_mutex_lock(&count_mutex);
while (count < COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("Thread %d Condition signal received.\n",
    *my_id);
}
pthread_mutex_unlock(&count_mutex);
//SIGNAL:
pthread_mutex_lock(&count_mutex);
count++;
if (count == COUNT_LIMIT)
    pthread_cond_signal(&count_threshold_cv);
pthread_mutex_unlock(&count_mutex);
```

Pthread bibliotekos funkcijų suvestinė.

Pthread funkcijos	
Gijų valdymas	<u>pthread_create</u>
	<u>pthread_exit</u>
	<u>pthread_join</u>
	<u>pthread_once</u>
	<u>pthread_kill</u>
	<u>pthread_self</u>
	<u>pthread_equal</u>
	<u>pthread_yield</u>
	<u>pthread_detach</u>
Gijų speciniai	<u>pthread_key_create</u>

duomenys (Specific Data)	pthread_key_delete pthread_getspecific pthread_setspecific
Gijų baigmė	pthread_cancel
	pthread_cleanup_pop
	pthread_cleanup_push
	pthread_setcancelstate
	pthread_getcancelstate
	pthread_testcancel
Gijų vykdomumas (scheduling)	pthread_getschedparam pthread_setschedparam
Signalai	pthread_sigmask

Mutex funkcijos	
Mutex valdymas	pthread_mutex_init
	pthread_mutex_destroy
	pthread_mutex_lock
	pthread_mutex_unlock
	pthread_mutex_trylock
Prioritetai	pthread_mutex_setprioceiling
	pthread_mutex_getprioceiling

Mutex Attribute Functions	
Pagrindinės operacijos	pthread_mutexattr_init
	pthread_mutexattr_destroy
Bendras naudojimas (Sharing)	pthread_mutexattr_getpshared
	pthread_mutexattr_setpshared
Protokolo atributai	pthread_mutexattr_getprotocol
	pthread_mutexattr_setprotocol
Prioritetai	pthread_mutexattr_setprioceiling

	pthread_mutexattr_getprioceiling
--	--

Ivykių kintamųjų funkcijos	
Bazinės funkcijos	pthread_cond_init
	pthread_cond_destroy
	pthread_cond_signal
	pthread_cond_broadcast
	pthread_cond_wait
	pthread_cond_timedwait

Ivykių kintamųjų atributų funkcijos	
Bazinės operacijos	pthread_condattr_init
	pthread_condattr_destroy
Bendras panaudojimas (Sharing)	pthread_condattr_getpshared
	pthread_condattr_setpshared

Pthread atributų funkcijos	
Bazinės operacijos	pthread_attr_init
	pthread_attr_destroy
Atskyrimas, suliejimas	pthread_attr_setdetachstate
	pthread_attr_getdetachstate
Steko informacija	pthread_attr_getstackaddr
	pthread_attr_getstacksize
	pthread_attr_setstackaddr
	pthread_attr_setstacksize
Gijos vykdymumo atributai	pthread_attr_getschedparam
	pthread_attr_setschedparam
	pthread_attr_getschedpolicy
	pthread_attr_setschedpolicy
	pthread_attr_setinheritsched

pthread_attr_getinheritsched

pthread_attr_setscope

pthread_attr_getscope

3 Lygiagretūs kompiuteriai

Šiame ir tolesniuose skyriuose nagrinėsime realiai lygiagrečiasias sistemas, kurių paskirtis – užtikrinti kiek įmanomą spartą užduočių atlikimą.*

3.1 Didelio našumo skaičiavimų poreikis

Aukšto našumo skaičiavimų poreikis auga greičiau negu galimybės. Gerai žinomas sritys, kurios reikalauja skaičiuojamojo greičio, - skaitinis modeliavimas, mokslo bei inžinierinių problemų simuliavimas. Šios problemos paprastai reikalauja didelės apimties pakartotinių skaičiavimų, kad pasiekti reikiamus rezultatus. Be to skaičiavimai turi būti atlikti per „priimtiną“ laiką.

Gamyboje, inžinieriniai skaičiavimai turi būti atlikti per sekundes ar minutes, bet kelios paros inžinierui yra dažniausiai nepriimtinas laikotarpis. Aišku, kuo simuliujamas reiškinys sudėtingesnis – tuo daugiau laiko užtrunkama. Egzistuoja specifinės problemos, kurių sprendimo laikas labai ribotas. Pavyzdžiui, skaičiavimams užtrukus dvi dienas, norint tiksliai nustatyti rytojaus orus, tokia prognozė taps neberekalinga. Esama tokų problemų, kaip didelių DNA struktūrų modeliavimas arba orų prognozė žemės rutuliu, kurios laikomos globalaus iššūkio problemomis. Kitaip tariant, tai tokios problemos, kurios negali būti efektyviai išspręstos šiuolaikiniais kompiuteriais per priimtiną laiką.

Kompiuterinė orų prognozė, yra gerai žinomas uždavinys, reikalaujantis pajėgių kompiuterių. Atmosfera yra modeliuojama, dalijant ją į trimates gardeles. Tam, kad išreikštį įvairius efektus, naudojamos matematinės lygtys. Iš esmės, kiekvienai ląstelei skaičiuojamos įvairios fizikinės charakteristikos (temperatūra, slėgis, drėgmė, vėjo greitis bei kryptis), atsižvelgiant į šias reikšmes praėjusių laiko momentu. Šie skaičiavimai, cikliškai kartojami daug kartų, modeliuojant laiko tėkmę. Šio uždavinio sprendimo greitį ir tikslumą esminiai įtakoja ląstelių kiekis bei jų dydis. Nuspėjant orą keletui dienų, atmosferinius reiškinius veikia toli viena nuo kitos esančios sritys, todėl reikia nagrinėti didelius regionus. Sakykime, imame domeną globalią (visos Žemės) atmosferą, padalytą į ląstelės $1 \text{ km} * 1 \text{ km} * 1 \text{ km}$ (iki $10 \text{ km} - 10$ ląstelių - aukščio). Taigi, turėsime maždaug 10^9 ląstelių. Tarkime, vienai ląstelei reikia 100 operacijų su slankaus kablelio skaičiais, taigi, sekančiam laiko žingsniniui apskaičiuoti atliksime 10^{11} operacijų. Jeigu numatome orą 10 dienų, naudodami 10 min. laiko intervalus, bus maždaug 10^4 laiko žingsnių arba iš viso – 10^{15} operacijų. Kompiuteris, kurio galia 1

* Skyriuje naudojama [5] medžiaga

Gflop/s – (gigafloras – 10^9 slankaus kablelio operacijų per sekundę) užtruks 10^6 sekundžių arba virš 10 dienų. Tam, kad atlikti skaičiavimus per 10 minučių prireiks kompiuterio, kurio galia apie 1,7 Tflop/s ($1,7 * 10^{12}$ slankaus kablelio operacijų per sekundę).

Kita gerai žinoma problema, reikalaujanti milžiniškos skaičiavimų apimties yra daugelio astrononinių kūnų judėjimo erdvėje modeliavimas (N-kūnų problema). Kiekvienas kūnas veikia kitą. Taigi N kūnų atveju turime apie N^2 sąveikų. Skaičiavimo procesas, žingsnis po žingsnio kiekvienam laiko intervalui skaičiuoja pasikeitusias pozicijas bei naujai veikiančias jėgas. Tegu galaktika turi 10^{11} žvaigždžių. Taigi prireiks 10^{22} sudétinių skaičiavimų naujai pozicijai rasti. Panaudojus efektyvesnį aproksimacijos algoritmą, prireiktų $10^{11} * \log 10^{11}$ veiksmų. Net jeigu vieną skaičiavimą sugebėtume atlikti per 100 ns (nano sekundžių - labai optimistinis įvertis, atsižvelgus į daugybos bei dalybos veiksmų gausą vienai operacijai atlikti - prireiks 10^8 metų, tam, kad įvykdyti vieną iteraciją, naudojant N^2 algoritmą arba apie 1 mėnesį, naudojant $N * \log 2 N$ algoritmą.

Galiausiai, žmogaus neriboti poreikiai, iššaukia naujų taikymų atsiradimą, kuriems nepakanka esamų skaičiavimo pajėgumų, pavyzdžiui, virtualiosios realybės modeliavimas. Šiuolaikinių kompiuterinių sistemų širdis – procesorius – yra nepaliaujamai tobulinamas. Tačiau procesoriaus spartos bei integryno elementų tankis artėja prie fizikinių ribų, nusakomų fizikos dėsniais - šviesos greičiu, atomų dydžiais ir pan.

Vienas iš būdų padidinti skaičiavimo greitį - naudoti daugelį procesorių, kartu sprendžiančių tą pačią problemą. Bendra problema yra skaidoma į dalis, o kiekvienu dalį sprendžia atskiras procesorius. Programų rašymas šio tipo skaičiavimams yra vadinamas **lygiagrečiuoju (parallel) programavimu**. Skaičiavimo platforma – **lygiagretusis kompiuteris**, - tai yra specialiai paruošta kompiuterinė sistema, kuri susideda iš keleto procesorių arba kelių nepriklausomų kompiuterių, sujungtų tam tikru būdu. Tokia sandara turėtų užtikrinti žymų našumo išaugimą. Idėja ta, kad N kartu dirbančių kompiuterių turėtų veikti N kartų sparčiau, negu vienas kompiuteris. Taigi, ta pati problema turėtų būti išspręsta per N kartų trumpesnį laiką. Aišku, tokia ideali situacija yra retai sutinkama. Dažnai problema negali būti padalinta lygiai į N dalis, sąveika tarp atskirų procesorių, apsikeičiant duomenimis bei derinant skaičiavimus, gali gaišinti problemas sprendimą. Tačiau daugeliu atveju keletos procesorių sistema bus pajėgesnė. Vadinasi lygiagretusis kompiuteris – tai iš esmės greitesnis kompiuteris. Tačiau, nepaisant to, visuomet atsiras naujų problemų – iššūkių, reikalaujančių kurti dar labiau pajėgesnes sistemas.

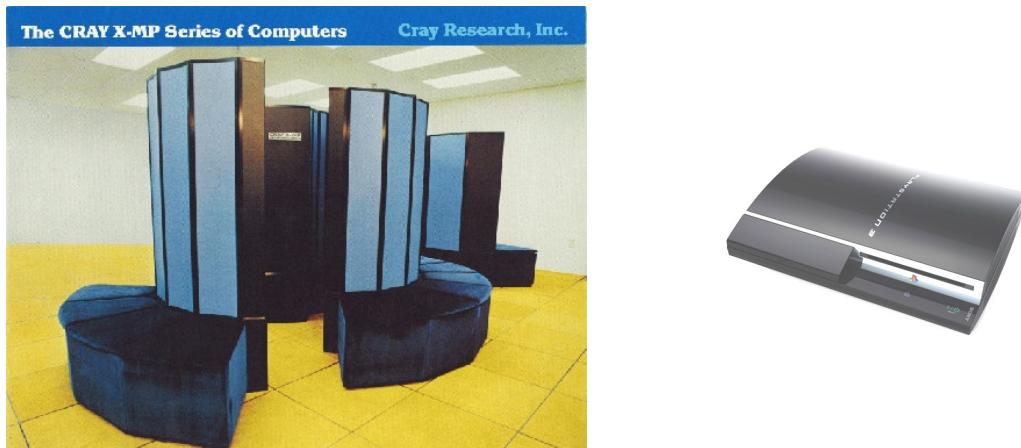
Lygiagrečiosios sistemos įgalina ne tik greičiau bet ir tiksliau išspręsti duotąją problemą, per priimtiną laiką. Pavyzdžiui, daugelis fizikinių problemų sprendžiamos, dalijant problemą į

diskrečius taškus. Priminsime, orų prognozės uždavinio sprendimas naudoja trimatę gardelę. Dvimatės arba trimatės gardelės sutinkamos daugelyje kitų uždavinių. Lygiagrečioji sistema įgalina rasti sprendinį „smulkesnės“ gardelės taškuose, taigi, uždavinio sprendimas tiksliau atspindi modeliuojamą reiškinį. Susijęs faktorius, ribojantis išprastą procesorinę sistemą, yra atminties talpa. Daugelio procesorių sistema turi daugiau tiesioginės prieigos atminties, taigi gali išspręsti sudėtingesnį, labiau imlų atminčiai uždavinį.

Lygiagretusis kompiuteris yra gana sena idėja. Jau 1958 Gill rašė apie lygiagretujį programavimą. Conway 1963 metais aprašė lygiagretujį kompiuterį ir jo programavimą. Straipsniai su panašiomis antraštėmis išleidžiami ir po 40 metų. 1996 m. Flynn ir Rudd rašė, kad “tebesitestantis vis aukštenio ir aukštesnio našumo sistemų siekis leidžia padartyti svarbią išvadą: ateitis priklauso lygiagretumui.”

Svetainėje www.top500.org galime rasti pačių sparčiausių pasaulyo superkompiuterių sąrašą. Pagal 2006 liepos mėn. duomenis pirmauja IBM firmos pagamintas **BlueGene/L** superkompiuteris, susidedantis iš 131072 Power PC 400 tipo procesorių, sujungtų greitaeigiu tinklu. Šio kompiuterio sparta, sprendžiant LINPACK testą – 280,600 Tflop/s. 2000-jų metų pirmojo dešimtmečio pabaigoje planuojama sukurti 1 Tflop/s našumo superkompiuterį.

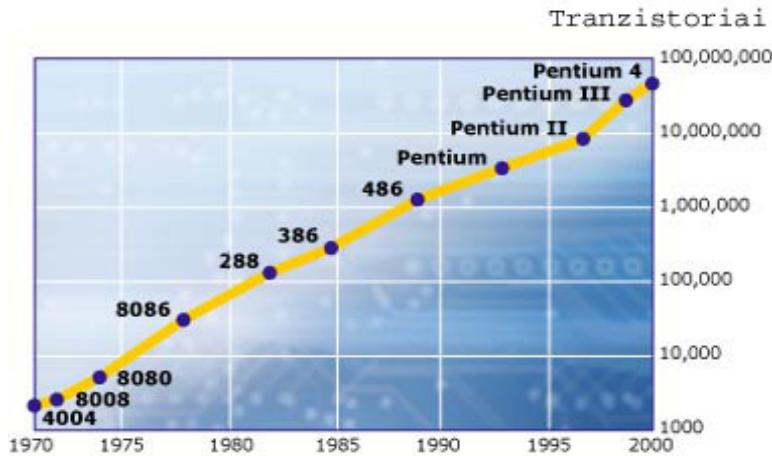
Palyginimui, *Sony PlayStation 3* – specializuoto žaidimų kompiuterio – sparta yra 1 Tflop/s eilės, o kaina neviršija 1000 \$, kai 1986 m. spartuolis - *CRAY X-MP* superkompiuteris – būdamas panašios galios kainavo apie 15 000 000 \$.



3.1 pav. *CRAY X-MP* [17] ir *Sony PlayStation 3*.

Kompiuterių grandyno sudėtingumo vystymosi tendencijas nuspėjo **Moore's** (“dėsnis”, pavadinimas autoriaus vardu): tranzistorių kiekis pigiausiam pagaminti grandyne padvigubėja kas du metus. Šis “dėsnis” tapo kelrodžių kompiuterinių grandynų gamintojams ištisus

dešimtmečius. Panašūs “dėsniai” formuluojami išorinių atmintinių talpų, taktiniui dažnių kitimo tendencijoms nusakyti.



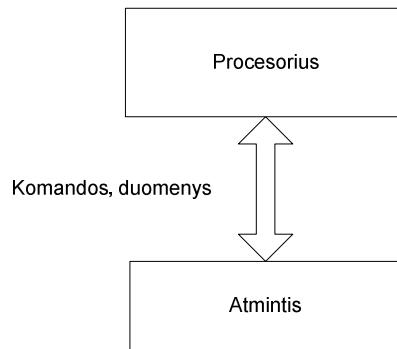
3.2 pav. Moore'o dėsnis tranzistorių skaičiui grandyne (pagal [16])

3.2 Lygiagrečiųjų kompiuterių tipai

Lygiagretusis programavimas reikalauja tinkamos kompiuterių platformos, kurią galime nusakyti arba kaip vieną kompiuterį su daugeliu vidinių procesorių, arba kaip keletą kompiuterių, sujungtų į tinklą. Išsamiau išnagrinėkome šias galimybes.

3.2.1 Bendrosios (shared) atminties daugiaprocesorinė sistema

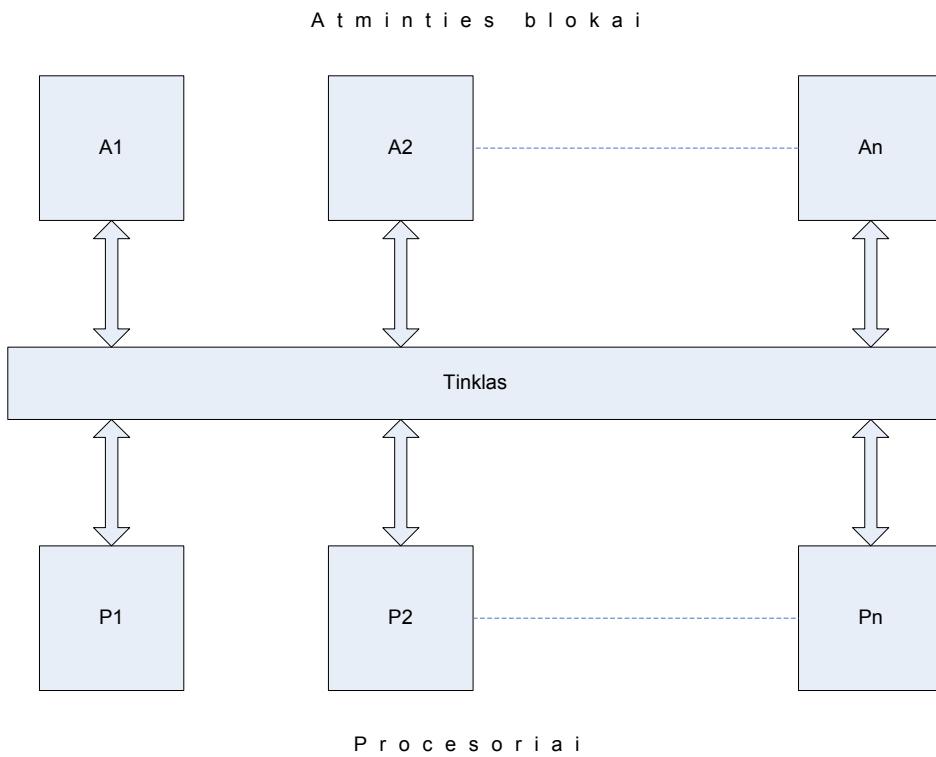
Iprastas kompiuteris susideda iš procesoriaus, kuris vykdo programą, patalpintą pagrindinėje atmintyje. Atmintis pasiekiamā, naudojant adresą. Adresų erdvė – nuo 0 iki 2^{n-1} , kur n – adreso ilgis bitais.



3.3 pav. Nuoseklusis kompiuteris [5]

Natūralus kelias išplėsti šį požiūrį – turėti daugelį procesorių, sujungtų su atminties moduliais taip, kad kiekvienas procesorius galitu prieiti prie bet kurios atminties vietas.

Turime taip vadinamą bendrosios atminties konfigūraciją. Procesorių sujungimai su atmintimi suformuoja tam tikrą sujungimų tinklą (*interconnection network*). Bendrosios atminties kompiuteris turi vieningą adresinę erdvę, tai reiškia, jog kiekvienas atminties elementas turi savo adresą ir yra vienodai gerai pasiekiamas iš bet kurio procesoriaus.



3.4 pav. Bendrosios atminties multiprocesorius [5]

Daugelis vienaprocesoriinių sistemų įgyvendina virtualiosios atminties koncepciją. Virtualioji atmintis “paslepia” tą faktą, jog realiai atmintis sudaro hierarchinę struktūrą greitaeigišumo požiūriu. Virtualizacija įgyvendinama, automatiškai perkeliant greitaeigėje atmintyje esančius duomenis į lėtesnę (bet talpesnę) atmintį, pavyzdžiu, iš pagrindinės atminties į diską, ir atvirkšciai. Šis procesas naudoja adresų transliaciją tarp realios ir virtualiosios atminties. Toks požiūris gali būti išplėstas ir į daugelio procesorių kompiuteri, kai reali atminties komponentė turi savo fizinį adresą, kuris susietas su virtualiuoju.

Programavimas bendrosios atminties multiprocesoriu reiškia, jog kiekvienas procesorius atmintyje saugo nuosavą vykdomą kodą kartu su bendrojoje atmintyje esančiais duomenimis. Būdai, kaip programuotojas pateikia programą ir duomenis multiprocesoriui gali būti įvairūs. Galima naudoti lygiagretaus programavimo kalbas su specialiais lygiagretujų vyksmą bei

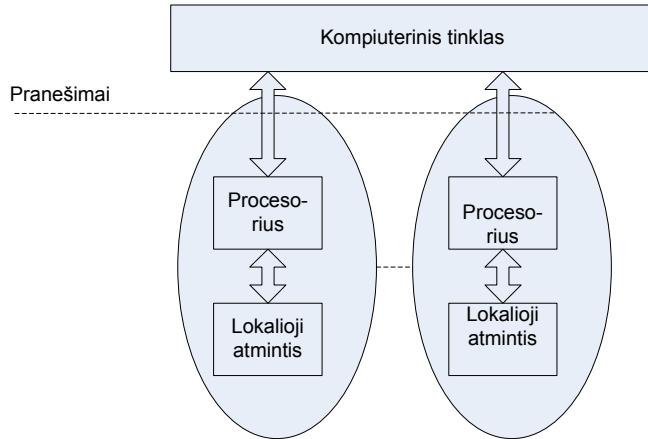
bendrus kintamuosius išreiškiančiaisiais sakiniais. Galutinį kodą, paruoštą vykdymui sukuria kompiiatorius. Kartais lygiagretaus programavimo kalbos remiasi jau egzistuojančiomis nuoseklaus programavimo kalbomis, pavyzdžiu FORTTRAN ar C++, įvedant trūkstamas lygiagretumui konstrukcijas. Kita alternatyva - naudoti taip vadinamasias gijas (*thread*), kurios naudoja įprastus nuoseklaus programavimo kreipinius. Gijos vykdomos vienu metu keliuose procesoriuose ir gali pasiekti bendrus kintamuosius.

Programavimo požiūriu bendrosios atminties modelis yra patrauklus, kadangi bendrai naudojamą kintamųjų idėja yra aiški ir suprantama. Tačiau, bendrosios atminties sistemas yra labai sunku realizuoti aparatūriškai, užtikrinant spartą vienalaikį bendrosios atminties prieinamumą iš daugelio procesorių. Todėl daugelis didelių daugiaprocesorinių sistemų naudoja vienokio ar kitokio pavidalo hierarchinę ar išskirstytą atminties struktūrą, kurioje procesorius gali pasiekti arti esančios atminties elementus greičiau negu tolimesnius. Tokios sistemas nusakomas terminu NUMA – nevienalytės atminties prieigos sistemomis (*non uniform memory access*)-, kaip priešingybė vienalytės prieigos atminties sistemoms - UMA (*uniform memory access*).

Iprastos vienaprocesorinės sistemas naudoja sparčiąją atmintį (*cache*), kurioje saugo ką tik naudotos pagrindinės atminties srities turinį, taip sumažinant kreipčių į lėtesnę pagrindinę atmintį skaičių. Šio princiopo naudojimas daugiaprocesinėse sistemose, gali padėti kompensuoti nevienalytės, nutolusios ir artimos, atminties prieigos laiko skirtumus, tačiau išsaukia naują problemą – tų pačių duomenų, esančių skirtingu procesorių sparčiojoje atmintyje tapatumo problemą. Dažniausiai ji sprendžiama labai neracionaliai. Vienam procesoriui atnaujinus duomenis savo sparčiojoje atmintyje, atitinkami duomenų blokai visuose kituose procesuose turi būti pripažinti negaliojančiais.

3.2.2 Pranešimais grįstas multikompiuteris

Bendrosios atminties lygiagretusis kompiuteris yra specializuota sistema. Kita alternatyva – tai sistema susidedanti iš įprastų nuosekliųjų kompiuterių, sujungtų į kompiuterinį tinklą.



3.5 pav. Pranešimų multikompiuteris [5]

Kiekviena šios sistemos komponentė susideda iš procesoriaus ir lokaliosios atminties, tiesiogiai neprieinamos kitoms komponentėms. Taigi šiame multikompiuteryje atmintis yra išskirstyta tarp nuosekliųjų kompiuterių, turinčių nuosavą adresinę erdvę. Kompiuterinis tinklas užtikrina procesoriams galimybę siusti vienas kitam pranešimus. Perduodami bei priimdami pranešimus, procesoriai apsikeičia duomenimis, reikalingais atliliki skaičiavimus. Tokia lygiagrečioji sistema paprastai yra vadinama pranešimų siuntimu grįstu multiprocesoriumi (*message-passing multiprocessor*) arba tiesiog multikompiuteriu, ypač jeigu jį sudaro kompiuteriai, kurie galėtų funkcionuoti ir nepriklausomai vienas nuo kito.

Programavimas pranešimų multikompiuteriui taip pat kaip ir bendrosios atminties kompiuterio atveju organizuoojamas dalijant problemą į dalis, kurios galėtų būti vykdomos nepriklausomai atskiruose kompiuteriuose. Rašant programas taip pat gali būti naudojamomos lygiagrečiosios arba išplėstos nuosekliosios programavimo kalbos, bet dažniausiai yra naudojamos pranešimų siuntimo bibliotekos, kurios kviečiamos iš įprastų nuosekliųjų programų. Dažnai naudojamas terminas procesas tam, kad apibrėžti lygiagrečiosios programos dalį vykdomą lokaliai, nuosekliai ir dalinai nepriklausomai nuo kitų procesų. Taigi programa yra suskirstoma į tam tikrą skaičių lygiagrečiuju procesus. Kiekvieną procesą vykdo individualus procesorius. Prosesorius gali užtikrinti ir kelių procesų vykdymą laiko dalijimo būdu. Vienintelis būdas procesoriams apsikeisti duomenimis – siunčiant pranešimus. Pranešimų siuntimas kaip kelių procesų bendravimo būdas gali būti naudojamas (modeliuojamas) ir bendrosios atminties multikompiuteriuose.

Laikoma, kad pranešimų multikompiuteris fiziškai yra lengviau plečiamas (*scale*) negu bendrosios atminties multikompiuteris. Kitaip tariant šią sistemą lengviau išplėsti, proporcingai padidinant našumą. Prosesorių varžymasis dėl atminties prieigos bendrosios atminties multikompiuteryje yra pagrindinė menko plečiamumo priežastis. Yra sukonstruota

specializuotų pranešimų sistemų, pvz., *transputer*‘iai, kuriems skirtos specializuotos aukšto lygmens programavimo kalbos, pvz. *occam*. Šiuo metu vis plačiau diegiamos išskirstytos sistemos, kurias sudaro dešimtys, šimtai ar tūkstančiai įprastų personalinių kompiuterių, sujungti į greitaeigį tinklą. *GRID*.

Dažnai manoma, kad pranešimų multikompiuteriai yra ne tokie patrauklūs programuotojams negu bendrosios atminties sistemos, kadangi reikalauja išreikšinio aprašymo, kaip duomenys bus persiunčiami iš vieno proceso į kitą. Toks išreikštinis duomenų kopijavimas atliekamas žemo lygio programavimo primityvais, kurie gali būti sulyginami su asemblerio kalba, ir gali išsaukti klaidų. Bendrosios atminties multikompiuterio programavimas gali būti iš dalies automatinis, nenusakant duomenų išskirstymo tarp atskirų procesorių detalių.

Visgi, iš technologinės pusės, į pranešimus orientuotų multikompiuterių pranašumas yra tas, kad jiems nereikia specialių mechanizmų, kurie užtikrintu atminties prieigos tarp skirtinį procesų sinchronizaciją, kuri gali žymiai prailginti programos vykdymą. Labiausiai reikšminga ypatybė - galimybė sudaryti multikompiuterį iš įprastų kompiuterių, sujungtų į tinklą. Daugeliui specializuotų bendrosios atminties kompiuterių sistemų nebuvo lemta ilgai egzistuoti, dėl nepaliaujamo techninių priemonių spartos vystymosi. Aišku, geriau naudoti vieną šiuolaikinį nuoseklujį kompiuterį, negu senstelėjusį lygiagretujį kompiuterį, kurio sparta tokia pati. Apjungti kelis kompiuterius į tinklą reikalauja žymiai mažiau sąnaudų, negu suprojektuoti ir pagaminti specializuotą daugiaprocesorinę sistemą. Be to kompiuteriai tinkle gali būti naudojami ir kitiems poreikiams, o kaip lygiagretusis multikompiuteris jis gali būti panaudotas tik tam tikru metu (pavyzdžiui, naktį).

3.2.3 Išskirstyta bendroji atmintis

Kadangi bendrosios atminties architektūra yra labai patogi programavimo požiūriu, buvo pasiūlytas išskirstytos bendrai naudojamos atminties architektūrinis modelis, kuriamė kiekvienas procesorius gali pasiekti bendrąjį atmintį, naudodamas vieningą adresų erdvę. Tam, kad procesorius nuskaitytų (ar įrašytų) nutolusią atminties vietą, atitinkamam procesoriui turi būti nusiūstas (ar gautas) pranešimas. Šis procesas gali būti automatizuotas, tokiu būdu įvedant virtualiosios atminties sąvoką, kuri paslepinia nutolusios atminties egzistavimo faktą. Aišku kreipiniai į nutolusią atmintį užtruks žymiai ilgiau, negu lokalūs kreipiniai. Dar vienas sutinkamas išskirstytos atminties organizavimo lygiagrečiuose kompiuteriuose būdas yra procesoriai tik su spartinančiaja atmintimi, kurie keičiasi duomenimis tarp šių atminčių, esant reikalui.

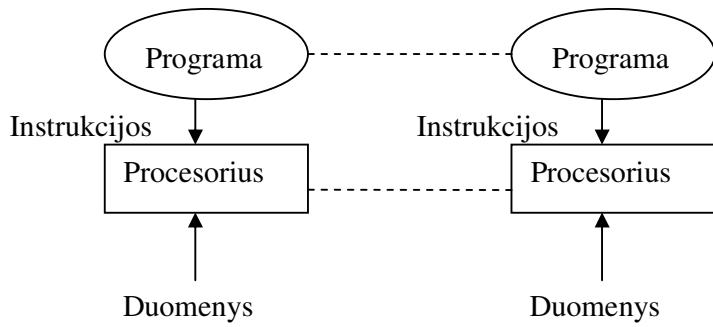
3.2.4 MIMD, SIMD klasifikacijos

Vienaprocesorinėje sistemoje programos vykdymas nusakomas vienu vykdomu instrukcijų srautu, kuris operuoja duomenimis. 1966 m. Flynn'as sukurė kompiuterių klasifikaciją, kurioje tokį kompiuterį pavadino vieno instrukcijų srauto (*Single Instructions stream*) vieno duomenų srauto (*Single Data stream*) kompiuteriu, sutrumpintai - SISD. Bendrosios paskirties multikompiuteryje, kiekvienas procesorius gali vykdyti atskirą programą, taigi turėti atskirą komandą srautą bei operuoti su atskirais duomenimis. Flynn'as šio tipo kompiuterius klasifikavo kaip daugelio instrukcijų srauto daugelio duomenų srautų (MIMD – *Multiple Instruction stream - Multiple Data stream*) kompiuterius. Abiejų tipų – bendrosios atminties bei pranešimų multikompiuteriai - pakliūna į šią klasifikaciją.

Be šių ekstremumų, SIMD bei MIMD būtų naudingas kompiuteris, kuris vykdo vieną programą, t.y., vieną programų srautą, bet yra keli duomenų srautai. Programos komanda yra persiunčiamā visiems procesoriams. Iš esmės, kiekvienas procesorius yra tik aritmetinis įrenginys (be valdymo įrenginio). Atskiras valdymo įrenginys atsakingas už eilinės instrukcijos nuskaitymą iš atminties ir jos perdavimą procesoriams. Kiekvienas procesorius sinchroniškai vykdo tą pačią instrukciją, bet su skirtingais duomenimis. Dažnai duomenys suformuoja masyvą, ir viena operacija atlieka grupinį veiksma su visais masyvo elementais. Flynn'as šio tipo kompiuterį klasifikavo kaip vieno instrukcijų srauto, daugelio duomenų srautų kompiuterį (*single instruction stream – multiple data streams*), trumpiau – SIMD. Šio tipo kompiuterio atsiradimo priežastis yra įvairios taikymosios programos, kurios operuoja masyvais. Tai fizikinių sistemų (pvz., molekulių) modeliavimo programos ar paveikslėlių manipuliavimo programos. SIMD sistemų panaudojimas tokiomis problemomis spręsti yra nesudėtingas tiek aparatūros kūrimo, tiek ir programavimo prasme.

Ketvirtoji Flynnno sistemos klasė – MISD – daugelio instrukcijų srauto, vieno duomenų srauto (*multiple instruction stream single data stream*) sistema nėra įgyvendinta, išskyrus, galbūt, kompiuterines sistemas, atsparias trikiams (*fault tolerant systems*) ar specialaus taikymo konvejerines sistemas.

MIMD sistemose, kiekvienas procesorius vykdo nuosavą programą. Kitais žodžiais, mes turime daugelio programų - daugelio duomenų - MPMD (*multiple program multiple data*) struktūrą. Suprantama, kai kurios iš tų programų bus identiškos. Pavyzdžiu, rašant šeimininko-tarno (*master - slave*) pavidalo programas, turime dvi skirtinges programas.



3.6 pav. MPMD struktūra [5]

SIMD klasifikaciją atitinkę „viena programa skirtini duomenys“ (SPMD) struktūra. Šiuo atveju, visi procesoriai vykdo tą pačią programą, tačiau nepriklausomai ir, galbūt, ne sinchroniškai, be to su skirtiniais duomenimis. Galbūt, viena programa galime apjungti skirtinas MPMD dalis.

3.3 Architektūrinės pranešimų multikompiuterių ypatybės

3.3.1 Statinio tinklo pranešimų multikompiuteriai

Statinis tinklas- tai kompiuterių sujungimo būdas, kuriame tinklo mazgai - kompiuteriai yra susieti fizinėmis jungtimis. Kitas variantas,- tinklo mazgai turi komutatorius (*switch*), įgalinančius perduoti pranešimą, nepertraukiant procesoriaus darbo. Jungties tipai tarp dviejų mazgų gali būti įvairūs: dvikryptė/vienakryptė jungtys, kiekvieno bito perdavimui gali būti skirtas atskiras jungties laidas, įgalinant perduoti vienu metu ne vieną bitą, o tarkime 32 bitus lygiagrečiai. Lygiagrečios jungtis dažnai naudojamos glaudžiai susijusiose sistemoje, kuriose atskiri procesoriai yra arti vienas kito. Vėlesnėje analizėje fizinių jungties charakteristikų neimsime domėn.

Tinklo ypatybės. Kompiuterinį tinklą (*network*) nusako šios pagrindinės charakteristikos: **greitaveika** (*bandwidth*), **inercija** (*latency*) ir **kaina** (*cost*) kaip jungčių skaičius tinkle. Greitaveika matuojama bitų skaičiumi perduodamu per sekundę. Inercija – tai nulinio ilgio pranešimo siuntimo trukmė. Iš esmės tai programinių bei techninių priemonių išlaidos pranešimo paruošimo siuntimui: (maršruto radimas, fizinės jungties komutacija, pranešimo

duomenų supakavimas, išpakavimas ir pan.). Šios papildomos išlaidos turi būti įskaičiuotos į pranešimo perdavimą.

Jungčių skaičius tarp dviejų kompiuterinių mazgų – esminis faktorius, lemiantis pranešimo perdavimo trukmę. Žemiau pateikuose apibrėžimuose traktuosime kompiuterinę tinklą kaip grafą, kuriame viršūnės – kompiuteriniai mazgai, o briaunos – ryšio jungtys. Tegu atstumas tarp dviejų viršūnių – trumpiausio jungiančiojo kelio briaunų skaičius. **Skersmeniu** (*diameter*) vadinsime atstumą tarp tolimiausią grafo viršūniu. Skersmuo kompiuteriniui tinklui išreiškia didžiausią pranešimų perdavimo trukmę – tolimiausią kelią, kurį turi įveikti perduodamas pranešimas – ir gali būti naudojamas įvertinti komunikacijos trukmę iš viršaus tam tikruose algoritmuose. Pavyzdžiui, daugelyje rūšiavimo algoritmų, procesoriai turi apsikeisti duomenimis, priklausomai nuo jų tvarkos. Blogiausiu atveju tenka apsikeisti duomenimis, esančiais tolimiausiose tinklo viršūnėse. Taigi, esant tinklo diametrui d , blogiausiu atveju reikės atlikti d duomenų perdavimo žingsnių perduodant kiekvieną duomenį.

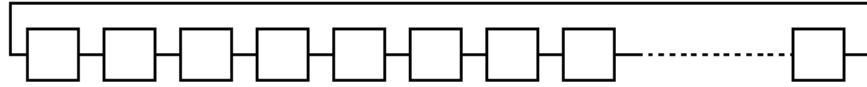
Tinklo (*bisekcijos*) **plotis** – tai briaunų skaičius, kurias “perkirtus” tinklą galima padalyti į du tarpusavyje nejungius tinklus su tuo pačiu viršūnių skaičiumi. Yra algoritmai, kurie reikalauja kad dvi tinklo dalys apsikeistų duomenimis. Jeigu n - duomenų skaičius, o w – tinklo plotis, o jungtis yra dvikryptės, tai perduodant duomenis lygiagrečiai tarp dviejų tinklo dalių reikės atlikti mažiausiai n/w komunikacijos žingsnių. Ši reikšmė gali tarnauti kaip apatinis komunikacijos trukmės įvertis.

Žemiau apžvelgsime tipiškas kompiuterinio tinklo topologijas.

Visiškai jungiamie (*exhaustive/completely connected*) tinkle bet kurios dvi viršūnes iš n sujungtos briauna. Taigi, nors šis tinklas yra ypač efektyvus, perduodant pranešimus, tačiau tinklo kaina (jungčių skaičius) yra $n(n-1)/2$, kas ir yra pagrindinis tokio tinklo trūkumas. Todėl praktikoje naudojami labiau riboti kompiuteriniai tinklai - tiesinis sujungimas (*line*), žiedas (*ring*), tinklas (*mesh*), hiperkubas (*hypercube*) bei medžio (*tree*) pavidalo tinklai. Gali būti ir kitų tinklo variantų tačiau minėtieji dažniausiai naudojami, analizuojant algoritmus.

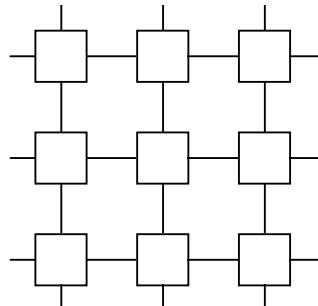
Tiesinis kompiuterinis tinklas / žiedas susideda iš jungčių, kurios jungia gretimus procesorius. Sujungus abu galus gauname žединį tinklą. Tinklo plotis lygus 1. Taigi šis tinklas ypač ekonomiškas, tačiau neefektyvus, kadangi tinklo skersmuo yra lygus n ar $n/2$ (žiedui). Kai tinklas nėra visiškai jungus, svarbi tinklo ypatybė – maršrutizavimo algoritmo sudėtingumas. Tiesinio/žiedinio tinklo atveju- maršrutas trivialus – eiti į dešinę/kairę,

priklausomai nuo procesoriaus eilės numerio tinkle. Be to šis tinklas nesunkiai plečiamas, išterpiant naują procesorių su savo jungtimis vietoje vienos iš jungčių.



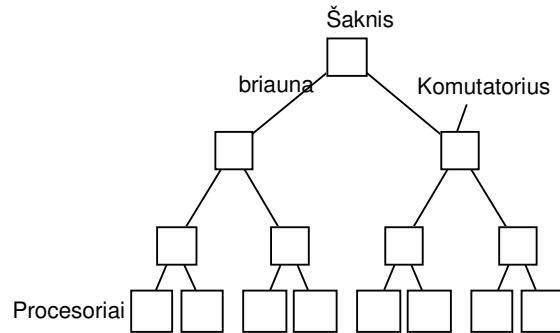
3.7 pav. Tiesinis/Žiedinis tinklas [5]

Tinklas. Dvieju dimensiju tinklas – tai gardelės pavidalo tinklas, kuriame viduriniai elementai turi po keturius kaimynus. Jeigu sujungsime kraštinius elementus eilutėse bei stulpeliuose tarpusavyje gausime toro pavidalo tinklą. Viršūnes galime pažymeti įprastomis (x,y) koordinatėmis, įgalinančiomis trivialią maršrutizaciją. Jeigu n – viršūnių skaičius, tai jungčių skaičius yra $2n$, skersmuo lygus $2 (\sqrt{n}-1)$, o tinklo plotis \sqrt{n} . Dvieju bei trijų dimensiju tinklas yra plačiai naudojamas dėl paprastos topologijos bei plečiamumo.



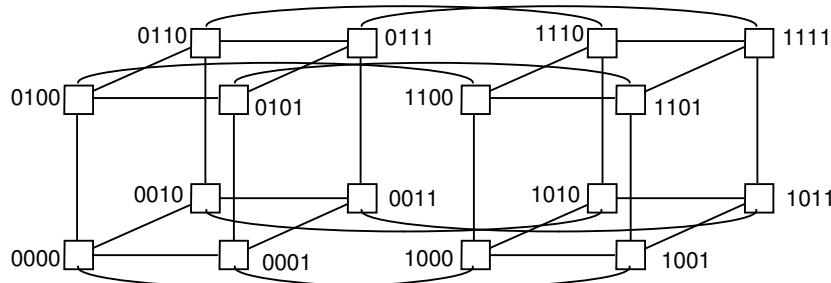
3.8 pav. 2D-tinklo konfigūracija [5]

Medžio pavidalo tinklas. Tinklas turi šaknį (0-nio lygmens viršūnė), o kiekviena viršūnė sujungta su keliomis žemesniojo lygmens viršūnėmis. Binariniame pilname tinkle kiekviena viršūnė turi lygiai du vaikus, taigi bendras viršūnių skaičius yra 2^n , kur n – lygmenų skaičius. Medžio pavidalo struktūros yra tinkamos sprendžiant taip vadinamus „skaldyk ir valdyk“ tipo uždavinius, tačiau dažnai mažas briaunų skaičius viršutiniuose lygmenyse gali būti siaura vieta (iš šaknies į vaikus perduodami duomenys, o atgal paprastai persiunčiami rezultatai). Kad išspręsti šį ribotumą yra pasiūlytos medžio modifikacijos (*fat tree network*), turinčios eksponentiškai augantį jungčių skaičių, kylant iš nuo lapų į šaknį.



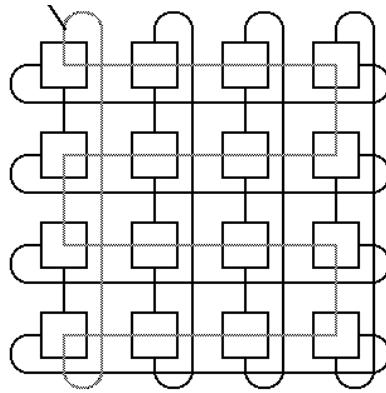
3.9 pav. Medžio pavidalo tinklas [5]

Hiperkubas. k -matavimų hiperkubiniame tinkle kiekviena viršūnė sujungta briauna su k gretimų viršūnių, taigi viršūnių skaičius lygus 2^k . Kiekvienai viršūnei galima priskirti k bitų adresą. Pastebėsime, kad dvi viršūnės sujungtos briauna tada ir tik tada, kai jų adresų bitai skiriasi tik vienoje pozicijoje. Ši ypatybė užtikrina efektyvų maršrutizacijos algoritmą, kuris leidžia išvengti aklaviečių. Hiperkubo skersmuo yra lygus k , t.y., auga lėtai, palyginus su viršūnių skaičiumi. k -matis hiperkubas, susideda iš dviejų $(k-1)$ -mačių hiperkubų, kuriuos jungia k briaunų. Taigi, hiperkubo plotis lygus k . Hiperkubo pavidalo tinklas tapo populiarus 80-siais, šiuo metu nėra toks paplitęs. Kaip vėliau pamatysime analizuodami pranešimų perdavimo hipercube algoritmus, ši konfigūracija yra efektyvumo prasme ideali daugeliui algoritmu.



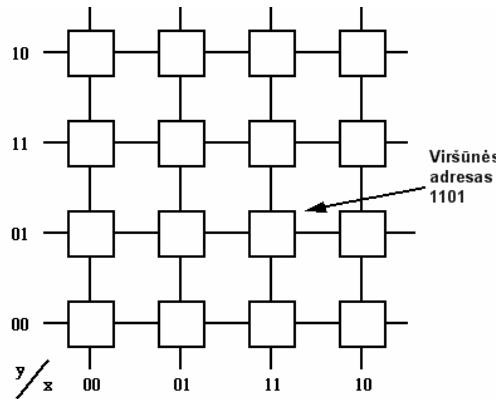
3.10 pav. Keturių matavimų hiperkubas [5]

Tinklų įterptimi (*embedding*) vadinsime tinklo viršūnių atvaizdį į kitokios konfigūracijos tinklą. Tinkamiausias būtų „idealus“ įterimas, kada pirminio tinklo viršūnės, sujungtos briauna, atvaizduojamos į tiesiogiai briauna sujungtas atvaizdžio viršūnes. Idealus įterimas įgalina algoritmą bei jo analizę, atliktą vienam statiniam tinklui, pritaikyti kitam tinklui. Piešinyje pavaizduotas „idealus“ žiedo atvaizdis į torą.



3.11 pav. Žiedo įterptis i torą [5]

Tinklas arba toras gali būti idealiai atvaizduojamas i reikiama matavimo hiperkubą, kaip pateikta paveikslėlyje. Kiekvienai viršūnei suteikiame adresą (x,y). Kadangi sujungtos viršūnės skiriasi tik vienu bitu,- jungtys idealiai atitinka hiperkubą.



3.12 pav. Toro 4x4 įterpimas i 4-mati hiperkubą [5]

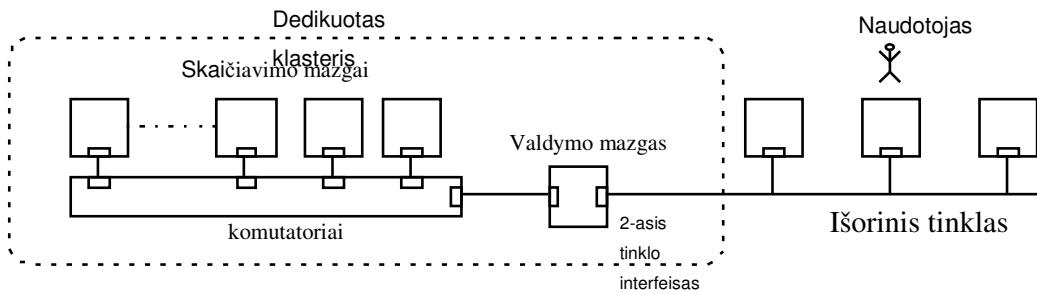
3.3.2 Tinklo kompiuteriai, kaip kompiuterinė platforma

Dabartiniu metu labai paplitę taip vadinami kompiuterių klasteriai – įprastiniai kompiuteriai sujungti i tinklą, kaip nebrangi superkompiuterių alternatyva. Pagrindinės ypatybės:

1. aukšto našumo personaliniai kompiuteriai yra prieinami už nedidelę kainą;
2. galima naudoti naujausias sistemas (pajėgiausius procesorius);
3. galima nesunkiai panaudoti egzistuojančią programinę įrangą arba nesunkiai ją modifikuoti.

1980 m. tokios sistemos pradėtos naudoti lygiagrečiuosiuose skaičiavimuose stimuliavo tokių pranešimais besiremiančių programinių sistemų kaip PVM ar vėliau MPI atsiradimą. (Šios sistemos bus aptariamos vėliau).

Personaliosios sistemos paprastai apjungiamos *Ethernet* pavidalo tinklu, kuriame pranešimai perduodami duomenų paketais. Tinklo greitaveika nusakoma standartais. Šiuo metu pereinama nuo 100 Mb/s prie 1Gb pralaidumo tinklo. Pagrindinė problema, kuri atsiranda naudojant tokį tinklą, tai palyginus ilgas paketo paruošimo persiuntimui laikas. Todėl išprasta įdiegti našesnius tinklus, užtikrinančius spartesnę greitaveiką. Viena iš galimų tipinių konfigūracijų pateikta paveikslėlyje.



3.13 pav. Dedikuotos skaičiavimams klasterinės sistemos struktūra [5]

3.4 Kiekybinės skaičiavimų spartinimo charakteristikos

Nepaisant to, kokią lygiagrečiąją sistemą naudojame, esminis jos principas - kelios programos dalys turi būti vykdomos lygiagrečiai. Taigi galime kalbėti apie lygiagrečius procesus, kurie apdoroja lokalius duomenis ir gali keistis duomenis su kitais procesais (nutolusios operacijos). Proceso apimtis nusako skaičiavimų grūdėtumą (*granularity*). Smulkiagrūdžiai procesai gali atlikti tik kelias nepriklausomas operacijas, o stambiagrūdžių procesų veikimas – žymią skaičiavimų dalį. Dažnai grūdėtumas apibrėžiamas kaip skaičiavimų apimtis tarp procesų komunikacijos ar sinchronizacijos taškų. Stambiagrūdžiai procesai mažiaus sugaiš komuniuodami tarpusavyje, tačiau jų išlygiagretinimas bei apkrovos subalansavimas gali būti problematiškas, lyginant su smulkaigrūdžiais. Taigi sprendžiant kiekvieną problemą reikia rasti šių tendencijų balansą.

Spartinimas (*speedup*), tai lygiagrečiosios sistemos našumo kiekybinis matas. Jis nusakomas kaip santykis tarp uždavinio sprendimo nuosekliaja sistema ir lygiagrečiaja. Taigi spartinimas sistemoje su n procesorių nusakomas formulė:

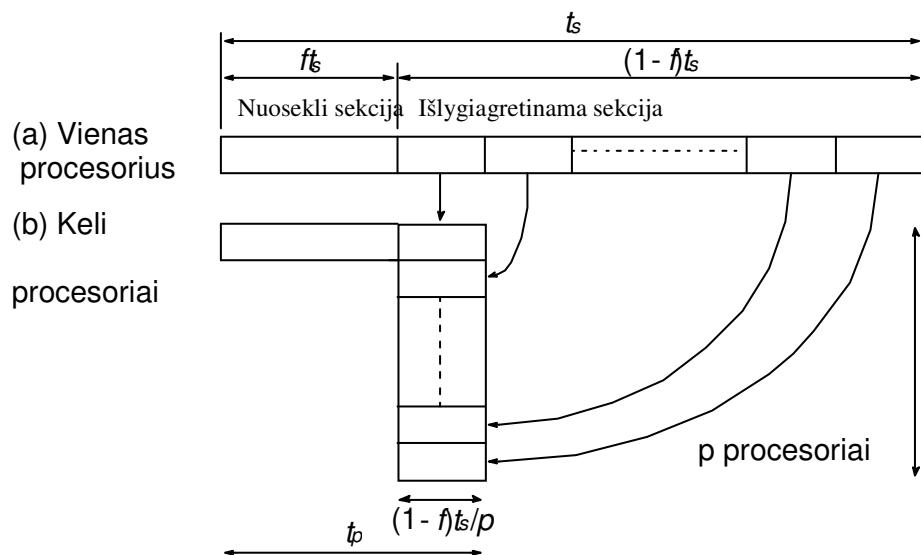
$$S(n) = T_s/T_p,$$

kur T_s – nuoseklaus vykdymo laikas (naudojant efektyviausių algoritmą), o T_p – multiprocesoriaus darbo laikas, sprendžiant tą patį uždavinį. Kadangi paprastai lygiagretusis

algoritmas turi "pridėtinį išlaidą", lyginant su nuosekliuoju (prastovos, pranešimų perdavimas, lokalų reikšmių perskaičiavimas), todėl spartinimas

$$S(n) \leq n.$$

Atvejis $S(n) > n$ (virštiesinis spartinimas) atrodo paradoksalus, kadangi visuomet lygiagrečiojo algoritmo darbą galime modeliuoti nuosekliaja sistema, todėl reikštū-egzistuoja optimalesnis nuoseklusis algoritmas, nei buvo vertinimas. Tačiau ši situacija gali pasireikšti, dėl didesnio suminio multisistemos atminties kieko, ar gali būti susijusi, pavyzdžiui, paieškos uždaviniuose, su kitokiu paieškinės erdvės peržiūros būdu lygiagrečiajame algoritme.



3.14 pav. Amdahl'o dėsnis [5]

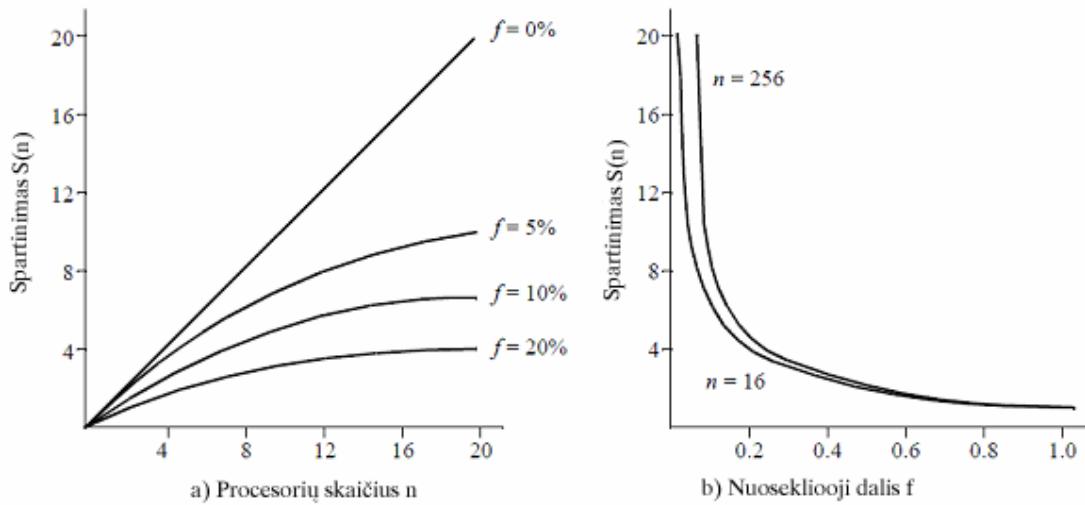
Maksimalaus spartinimo ribas, taikant konkretų algoritmą, nusako *Amdahl'o* dėsnis. Jame atsižvelgiama į tą uždavinio sprendimo algoritmo dalį, kuri negali būti išlygiagretinama (žr. paveikslėli). Taigi, jeigu f – nuoseklioji algoritmo dalis, *Amdahl'o* dėsnis sako:

$$S(n) = T_s / (f T_s + (1-f) T_s / n) = n / (1 + (n-1)f).$$

Riboje, turint neribotą proc. skaičių:

$$S(n) \rightarrow 1/f, \text{ kai } n \text{ artėja į begalybę.}$$

Tarkime, jeigu neišlygiagretinama algoritmo dalis yra 20%, vadinasi spartinimas nebus didesnis už 5 kartus.



3.15 pav. Spartinimo priklausomybė nuo nusekljosios dalies f ir procesorių skaičiaus n [5]

Atvirkštinius spartinimui dydis, padalytas iš procesorių skaičiaus vadinamas efektyvumu.

Efektyvumas

$$E(n) = Ts / (Tp n).$$

Kur Ts – nuoseklusis laikas, Tp – lygiagretusis, o n procesorių skaičius.

Trečioji charakteristika - **kaina** – apibrėžiama, kaip algoritmo vykdymo laiko bei procesorių skaičiaus sandauga.

$$C(n) = Tp n = Ts n / S(n) = Ts/E.$$

Dar viena charakteristika, naudojama įvertinant skaičiavimus - plečiamumas (*scalability*). Architektūrinis (hardware'inis) plečiamumas išreiškia galimybę, išplėtus sistemą, padidinti jos efektyvumą. **Algoritminis** plečiamumas nusako, kaip pasikeis algoritmo vykdymo laikas, sprendžiant „didesnį“ uždavinį (pavyzdžiu, naudojant daugiau pradinių duomenų, ar skaičiuojant didesniu tikslumu). Dažnai sutinkami algoritmai, kuriuose išlygiagretinamoji uždavinio dalis didėja, augant uždavinio dydžiui, taigi tokis algoritmas – gerai plečiamas.

Gustafson'o dėsnis skaičiavimų plečiamumui. Plečiantysis spartinimas

$$S_s(n) = \frac{s + np}{s + p} = s + np = n + (1 - n)s$$

Čia remiamasi tuo, kad vykdymo laikas yra fiksotas ($s + p = 1$), o nuoseklioji dalis nepriklauso nuo procesorių skaičiaus ir problemos dydžio.

4 Pranešimų perdavimu grįsti skaičiavimai

4.1 Ivadas

4.1.1 Programavimo variantai

Pranešimais grįstas programavimas multikompiuteriams galimi remtis^{*}

1. specializuota programavimo kalba;
2. aukšto lygio nuosekliosios programavimo kalbos sintaksiniai plėtiniai, užtikrinančiais pranešimų perdavimą;
3. bibliotekiniais kreipiniais pranešimų perdavimui.

Pirmojo sprendimo pavyzdys – programavimo kalba *occam*, skirta specialiam pranešimų procesoriui – *transputer*'ui. Tokios programavimo kalbos kaip *CC+* bei *Fortran M* iliustruoja antrajį požiūrį. Taip pat galime kalbėti apie specialius išlygiagretinančius kompiliatorius, gebančius išlygiagretinti įprastas nuosekliasias programas, pvz., parašytas progr. kalba *Fortran*.

Šiame skyriuje didžiausią dėmesį skirsimė trečiajam variantui, kuris naudoja įprastą progr. kalbą, tokią kaip C, kartu su bibliotekinėmis funkcijomis, užtikrinančiomis tiesioginį pranešimų perdavimą tarp atskirų procesų. Duotojų atveju reikia išreikštiniu būdu nurodyti

- kokie procesai ir kaip bus kuriami,
- kokius pranešimus jie perdavinės.

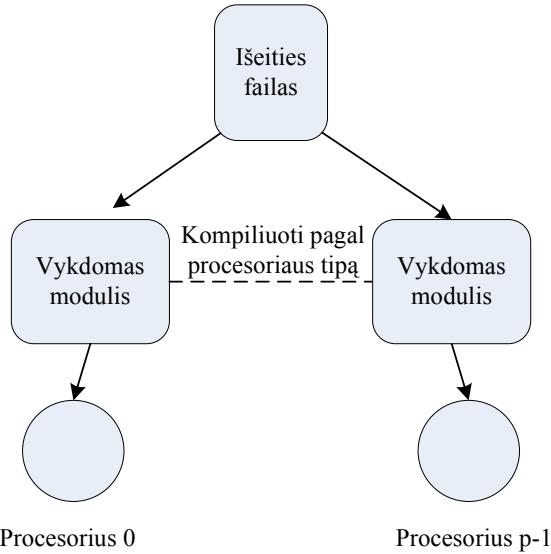
4.1.2 Procesų sukūrimas.

Kiekvieną procesą vykdo atskiras procesorius, o procesas kaip programa vykdomas nuosekliai. Testavimo režime, arba siekiant paslėpti tinklo inercija - latentiškumą, vienas procesorius gali vykdyti kelis procesus. Prieš vykdant procesus, jie turi būti sukurti.

Statiniu procesų sukūrimo atveju procesų skaičius nustatomas prieš lygiagrečiosios programos paleidimą ir nesikeičia vykdymo metu. Procesų skaičius gali būti pateikiamas sistemai komandine eilute. Dažniausiai visi procesai néra skirtinti. Pavyzdžiui, šeimininko-tarno pavidalo programoje, yra dviejų tipų procesai: valdantysis procesas – šeimininkas, kuris valdo likusius procesus – tarnus. Vienos programos multi-duomenų (single program multiple data - *SPMD*) modelyje, kelis skirtintus procesus galime sulieti į vieną programą. Šioje

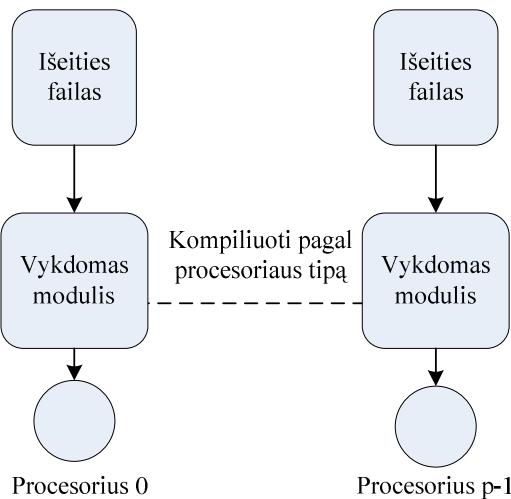
^{*} Skyriuje naudojama [5] medžiaga

programoje skirtinį procesų vykdomą kodą nusako skirtinės programos šakos, kurios parenkamos, priklausomai nuo proceso numerio. Plačiausiai naudojamas tokio pavidalo instrumentas yra MPI sistema.



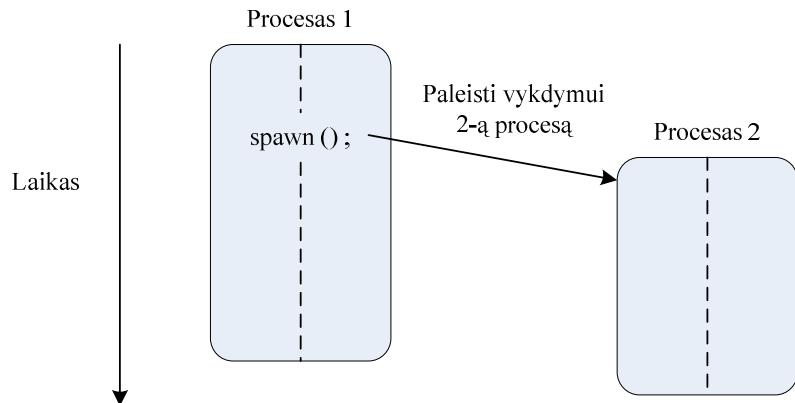
4.1 pav. MPI sistema [5]

Procesai gali būti paleidžiami ir *dinamiškai* programos vykdymo metu. Taigi vykdomų procesų skaičius gali kisti. Pats bendriausias modelis atitinkantis dinaminį procesų sukūrimą yra multi-programų multi-duomenų (*multiple program, multiple data* - MPMD) modelis, kuriame skirtinės procesai gali vykdyti skirtinės programos. Ši modelių įgyvendina gerai žinoma pranešimų sistema PVM.



4.2 pav. MPMD procesų modelis [5]

Naujo proceso paleidimas MPMD modelyje pavaizduotas žemiau pateiktoje schemaje. Paleidus naują procesą, abu procesai pradeda veikti lygiagrečiai.



4.3 pav. Naujo proceso paleidimas [5]

4.1.3 Pranešimų perdavimo procedūros.

Pranešimų perdavimas. Prosesai sąveikauja tarpusavyje siušdami pranešimus. Pagrindinės tiesioginio (“point-to-point”) pranešimų siuntimo (*send*) ir gavimo (*receive*) operacijos užrašomos forma:

```

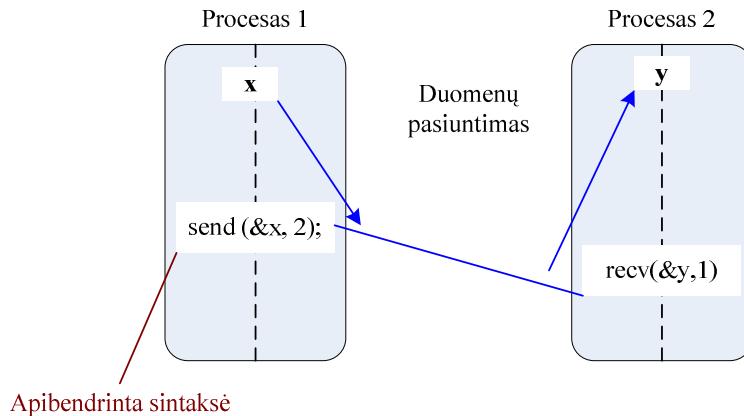
send (parametru_sarašas) ;
recv (parametru_sarašas) .
    
```

Parametru sarašas priklauso nuo naudojamų bibliotekų. Paprasčiausiu atveju siunčiant reikia nurodyti siunčiamajį duomenį ir proceso – adresato identifikatorių, o priimant – proceso-siuntejó identifikatorių bei adresą kintamojo, kuriame bus patalpintas siunčiamas duomuo. C pseudo-kodu:

```

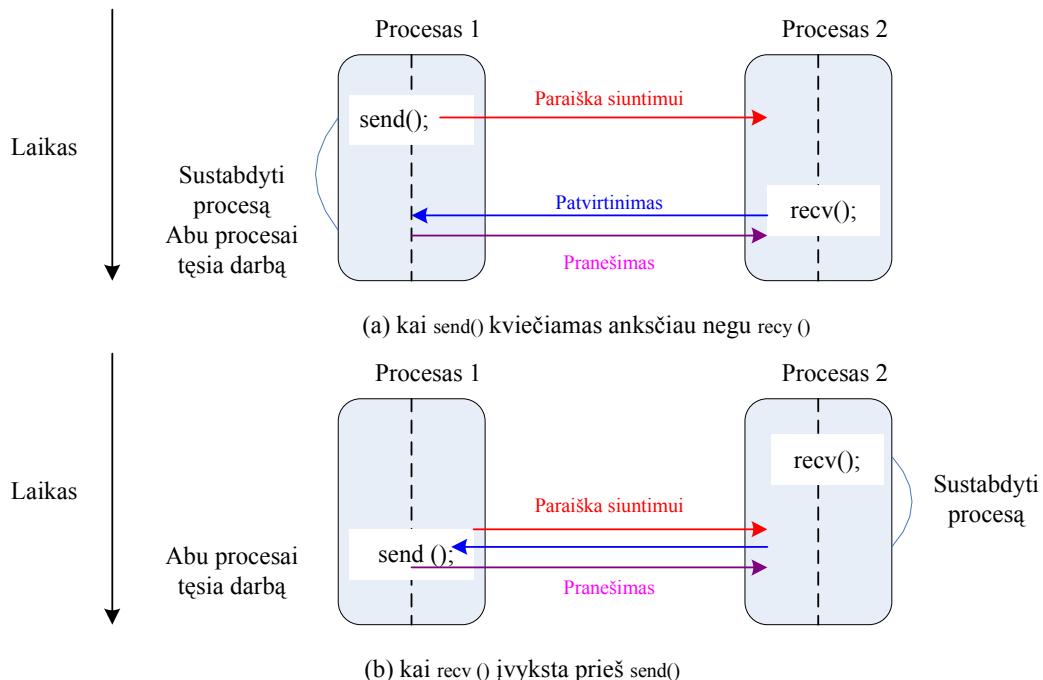
send (&x, gavėjo_id) ;
recv (&y, siuntėjo_id) .
    
```

Dažnai siunčiamas ne vienas duomuo, o visas masyvas arba jo dalis. Be to, siunčiami duomenys gali būti įvairių tipų, taigi, realiai, parametru nurodoma žymiai daugiau.



4.4 pav. Paprasto pranešimo perdavimas [5]

Sinchroninis pranešimo perdavimas. Tai pranešimų perdavimo būdas, kai grįztama iš pranešimo siuntimo/priėmimo procedūros tik tada, kai pranešimas pilnai perduotas ar priimtas. Taigi synchroninė siuntėjo procedūra lauks, iki gavėjas pranešimą priims, o synchroninė gavėjo procedūra lauks, iki siuntėjo duomenys bus išsiuisti. Taigi du procesai yra pilnai synchronizuojami, be to nereikalingas papildomas buferis pranešimo saugojimui. Tokios komunikacijos realizacija gali remtis trijų lygių komunikacijos protokolu, pateiktu brėžinyje. Pastebėsime, kad pranešimas yra saugomas procese siuntėjuje, iki nebus realiai išsiuistas synchronizacijos taške.



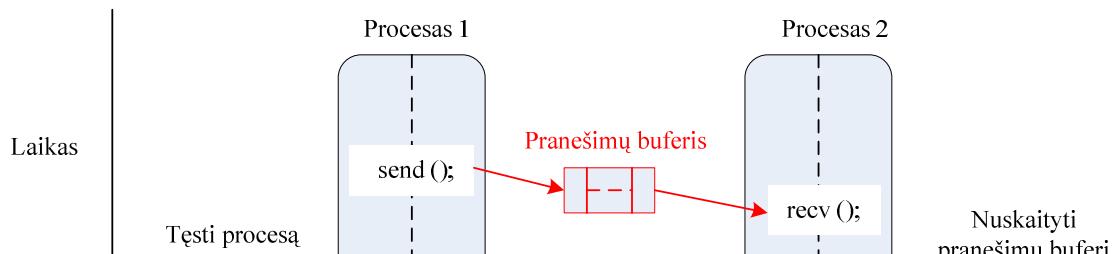
4.5 pav. Sinchroninis send() and recv(), naudojant trijų lygių sąveikos protokolą [5]

Asinchroninis pranešimų perdavimas.

- Procedūros neprivalo laukti, kol bus baigtas pranešimo persiuntimas- gali būti grįžtama anksčiau. Paprastai reikalauja papildomos atminties pranešimų saugojimui.
- Kelios realizacijų versijos, priklausomai nuo grįžimo iš procedūros semantikos.
- Iš esmės procesų nesynchronizuojasi, taigi – leidžia procesams judėti pirmyn anksčiau. Turi būti naudojamos atsargiai.

Blokujantieji ar neblokuojantieji pranešimų perdavimai.

Blokavimasis reiškia, jog iš pranešimo siuntimo ar gavimo procedūros negrįztama, kol nebus pilnai baigtas duomenų perdavimas. Taigi tuo aspektu, blokavimosi ir synchroninio perdavimo semantikos sutampa. Tačiau kai kuriose pranešimų perdavimo sistemoje, pvz., MPI, blokavimosi prasmė yra kitokia. Prieš pradēdami nagrinėti MPI subtilybes, pasižiūrėkime, kaip gali pranešimų perdavimo procedūros grįžti, prieš užbaigiant pranešimo perdavimą. Reikalingas pranešimo buferis, kuriame būtų patalpintas pranešimas, prieš jį priimant.



Paveikslėlyje parodyta, jog buferis naudojamas tam, kad patalpinti pranešimą, išsiųstą su `send()`, prieš priimant `recv()` kreipiniu. Šiuo atveju pirmasis procesas gali saugiai grįžti iš siuntimo procedūros ir toliau testi darbą. Tokiu būdu lygiagrečiosios programos darbas tampa efektyvesnis. Praktiškai, tokio buferio ilgis yra baigtinis, taigi siunčiantysis procesas gali būti vis vien pristabdytas, kai buferis užsipildys. Kita vertus, gali prieikti papildomų komunikacijų, kad sinchronizuoti abu procesus.

Toliau naudosime MPI blokavimosi terminus. Procedūras, kurios naudoja buferius ir grįžta, užbaigus lokalius veiksmus, nors realiai pranešimo perdavimas dar nepasibaigęs, vadinsime blokuojančiomis (lokaliai). Tas procedūras, kurios grįžta iš karto, vadinsime nesiblokuojančiomis. Tuo atveju laikoma, kad duomenys, kurie nurodyti siuntimo

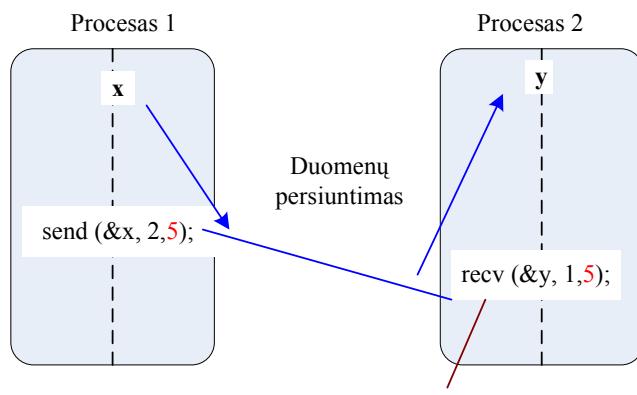
procedūroje privalo būti nemodifikuoti iki jų panaudojimo, ir tai užtikrinti turi programuotojas. Kitose (ne MPI) sistemoje blokavimo terminai gali turėti ir kitas prasmes.

Pranešimų išrinkimas.

Pranešimo siuntimo ar gavimo operacijose nurodomas konkretus procesas. Operacija `recv()` priima pranešimus tik iš nurodyto parametru proceso. Specialus proceso identifikatorius (pvz., skaičius -1) gali būti naudojamas, kad nurodyti kaip procesą siuntėja bet kurį procesą.

Kad užtikrinti didesnį lankstumą, perduodant pranešimą, gali būti naudojamos pranešimo žymė (*message tag*). Paprastai tai neneigiamas sveikasis skaičius, kuris gali būti naudojamas tam, kad išskirti pranešimus pagal prasmę ar tipą. Pavyzdžiui, žymė 0 gali būti naudojama priimant duomenis, o 1 – valdančią informaciją. Žymė įtraukiamā į pranešimo perdavimo procedūrų parametrus bei perduodama kartu su pranešimu. Gali būti naudojama speciali žymė, kad priimti pranešimus su visomis žymėmis.

Pavyzdys. Reikia persiusti pranešimą x su žyme iš 1-jo proceso į paskirties proceso nr. 2 patalpinant rezultatą į kintamąjį y.

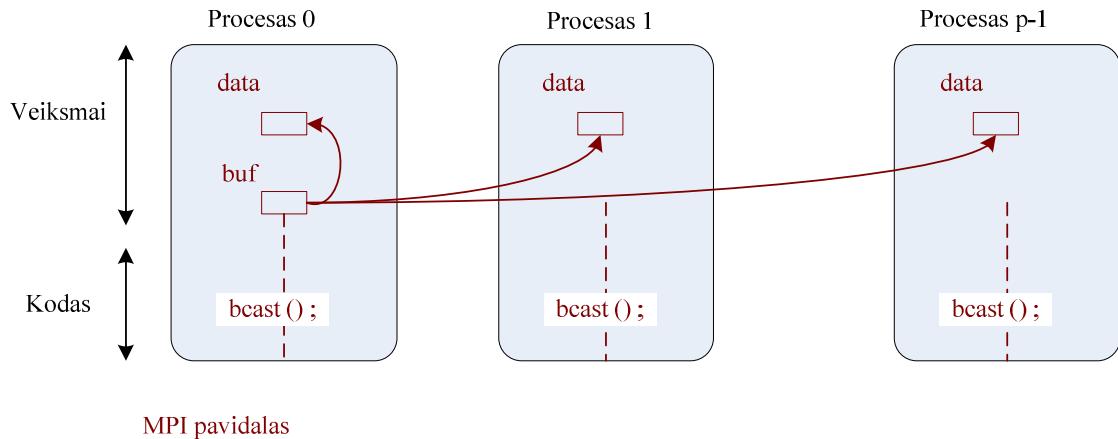


4.7 pav. Pranešimo su žyme perdavimas [5]

“Grupinės” pranešimų perdavimo operacijos.

Be minėtujų “primityviųjų” pranešimų perdavimo operacijų sutinkama ir daugelis kitų. Minėtinis procedūros, įgalinančios persiusti tuos pačius duomenis tam tikrai procesų grupei. Tai, pavyzdžiui, duomenų paskleidimo visiems procesams operacija (*broadcast*). Nors ši operacija galėtų būti realizuota `send()`, `recv()` primityvais, tačiau priklausomai nuo pranešimo tinklo topologijos galimos ir efektyvesnės realizacijos.

Paprastai paskleidimo operacijoje dalyvaujantys procesai sudaro procesų grupę, kurios identifikatorius nurodomas kaip `bcast()` procedūros parametras. Paveikslėlyje pavaizduota, jog procesas su identifikatoriu 0 yra šakninis (*root*) procesas. Šakniniu procesu gali būti bet kuris iš grupinėje operacijoje dalyvaujančių procesų. Šiuo atveju, šakninis procesas laiko persiunčiamus duomenis buferyje *buf*. Kiekvienas procesas kviečia tą pačią procedūrą `bcast()`, kas yra tipiška SPMD modeliui, kuriame kiekvienas procesas vykdo tą pačią programą.

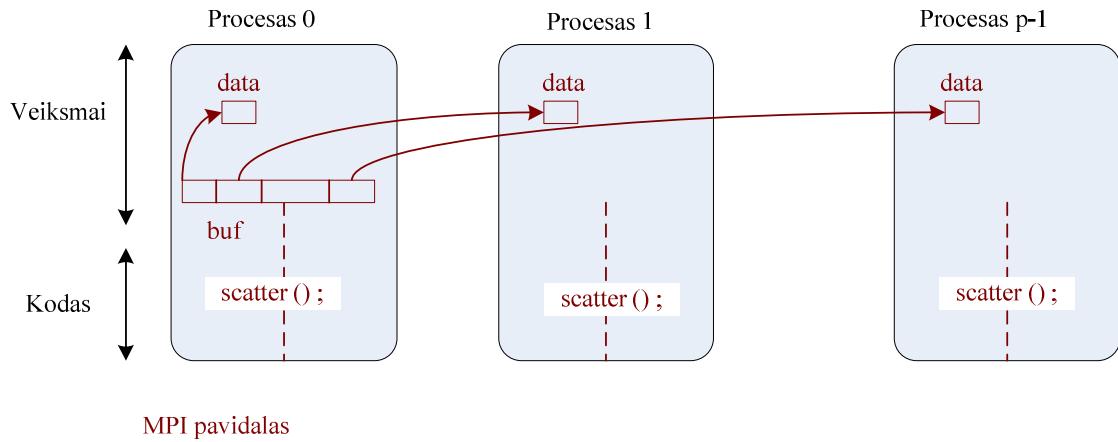


4.8 pav. Duomenų paskleidimo (broadcast) operacija [5]

Šakninis procesas taip pat gauna rezultatą datą, kad būtų visiška simetrija tarp procesų. Pastebėsime, jog `bcast()` procedūra nepradedama tol, kol visi procesai neiškviečia `bcast()`. Taigi šios operacijos pradžia sinchronizuojia procesus.

Duomenų išbarstymas (*scatter*).

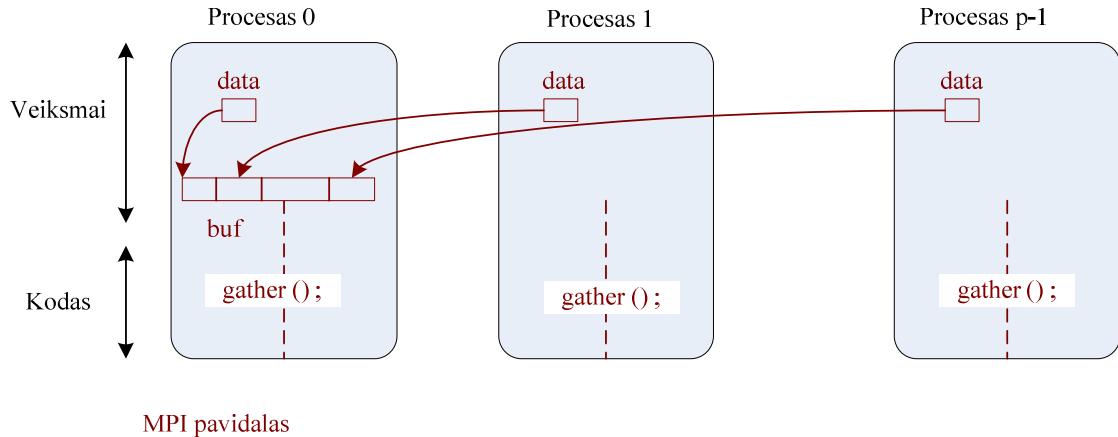
Ši sąvoka naudojama aprašyti atskirų masyvo dalij siuntimą iš šakninio proceso atskiriems procesams. I-sios pozicijos masyvo dalis yra nusiunčiama i-jam procesui. Paveikslėlyje parodoma, kad šakninis procesas taip pat gauna savo duomenų dalį, kas būdinga SPMD modeliui.



4.9 pav. Duomenų išbarstymo (scatter) operacija [5]

Duomenų surinkimas (gather).

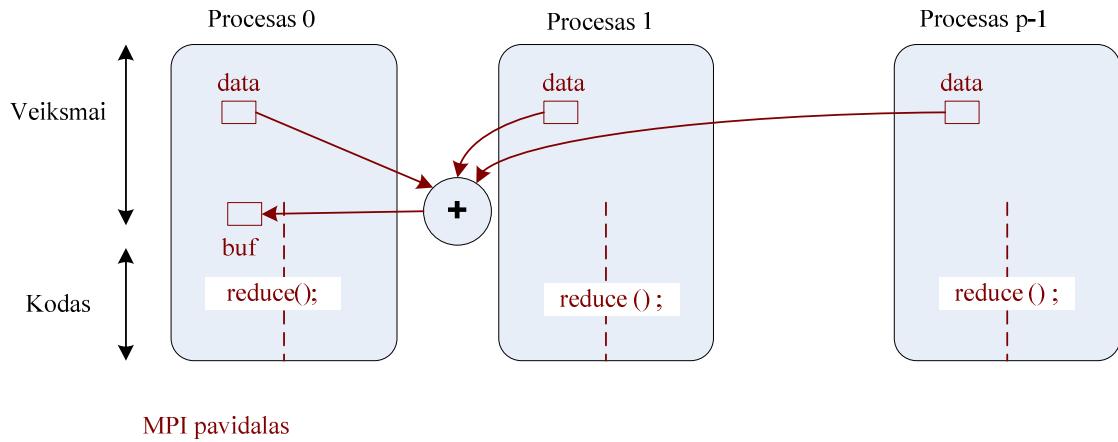
Nusako, kaip vienas procesas surenka duomenis iš visų procesų. Paprastai operacija kviečiama programos darbo baigmėje, surenkant visų procesų darbo rezultatus. Iš esmės, surinkimo operacija yra atvirkštinė išbarstymui. I-jo proceso duomenys yra patalpinami į i-ją šakninio proceso poziciją. 0-nis procesas, kaip parodyta pateiktajame paveikslėlyje, taip pat pateikia duomenis.



4.10 pav. Surinkimo operacija (gather)

Redukavimo operacija (reduce).

Iš esmės tai surinkimo operacija kartu su pateiktaja aritmetine ar logine operacija. Pavyzdžiu, surenkamos reikšmės gali būti susumuojamos. Kitos galimos operacijos – *min*, *max*, *and*, *or* ir t.t. Pastebėsime, jog redukcijos operacijos gali būti realizuotos dar efektyviau negu surinkimo, kadangi gali būti perduodamas žymiai mažesnis duomenų kiekis.



4.11 pav. Redukcijos (reduce) operacija [5]

4.2 Kompiuterių klasteriai

4.2.1 Programiniai instrumentai.

Aptarsime, kokie instrumentai užtikrina pranešimų perdavimo panaudojimą kompiuterių klasteriams (*clusters*). Pirmoji, labiausiai propažinta priemonė, kurią galima naudoti kompiuterių klasteriuose buvo PVM sukurta *Oak Ridge* laboratorijoje. PVM tai programų aplinka, pranešimų perdavimui homogeninėse ir heterogeninėse kompiuterių aplinkose ir yra sudaryta iš bibliotekinių procedūrų, kurias naudotojai gali kvieсти iš C kalbos ar FORTRAN programų. PVM tapo plačiai naudojama sistema, nes buvo nemokama bei egzistuojanti įvairiomis (netgi Microsoft Windows) sistemomis.

Yra ir specializuotų (*proprietary*) pranešimų sistemų, tokų kaip IBM'o MPL pranešimų perdavimo sistema RS/6000 SP/2 multiprocesoriams. Tam, kad standartizuoti pranešimų perdavimo sistemas, akademiniai ir industrijos partneriai susijungę, sukurdami tam tikrą pranešimų sistemų „standartą“, pavadintą MPI, kuris iš tiesų paskutiniu metu tapo plačiai vartojuamu. Šiuo metu egzistuoja pakankamai daug realizacijų, išskaitant LAM, MPICH, CHIMP, UNIFY ir t.t.

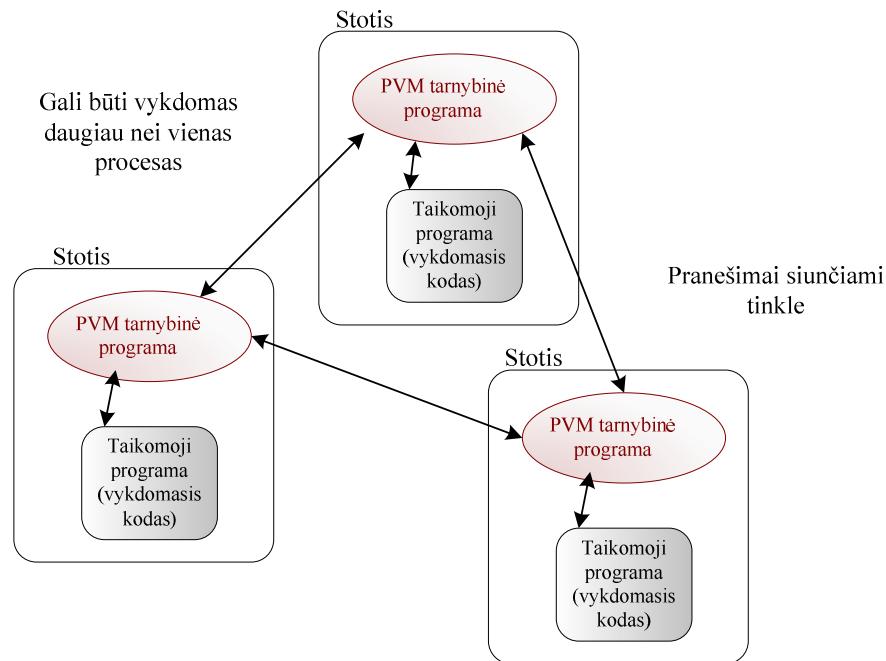
Panagrinėkime smulkiau PVM ir MPI sistemas.

4.2.2 PVM (Parallel Virtual Machine).

Naudodamas PVM, programuotojas išskaido problemą į atskiras programas. Kiekviename programa, parašyta C ar FORTRAN programavimo kalba kompiliuojama taip, kad galėtų būti

vykdoma specifiniams kompiuterių, esančių tinkle, tipams. Kiekviena stotis (*workstation*) turi savo lokalią failų sistemą, taigi programos binarinis kodas turi būti prieinamas lokalai.

Kompiuterių aibė, kurioje bus vykdoma lygiagreti programa turi būti aprašyta iš anksto. Dažniausiai sukuriamas tekstinis failas su stočių aprašu – *host-failas*. Ši failą vėliau nuskaito PVM. Alternatyvus kelas yra startuoti vieną mašiną ir vėliau pridėti likusių mašinų vardus, naudojant PVM komandas.



4.12 pav. MPI pranešimų siuntimas tarp atskirų stočių [5]

Pranešimų siuntimas tarp kompiuterių atliekamas PVM *pagalbiniu* (*daemon*) procesu, instaliuotu kiekviename kompiuteryje. Kiekvienas demono procesas turi turėti pakankamai informacijos, kad nustatyti tinkamą pranešimo perdavimo maršrutą.

Gali atsitikti, kad procesų skaičius yra didesnis negu procesorių skaičius. Pavyzdžiui, mes galime norėti testuoti lygiagrečią programą viename kompiuteryje, prieš perduodant vykdymui multiprocesorinei sistemai. PVM įgalina sukurti kiek norima procesų, nekreipiant dėmesi į procesorių skaičių, ir priskirti procesus procesoriams automatiškai (jeigu to nenurodė programuotojas išreikštiniu būdu).

Procesų sukūrimas ir vykdymas.

PVM sistemoje procesai gali būti startuojami dinamiškai. Dažnai procesai organizuojami šeimininko-tarno principu. Pradžioje paleidžiamas „šeimininko“ procesas, o po to – dinamiškai - „tarnų“ procesai. Tam, kad startuoti nauja procesą naudojamas PVM kreipinys `pvm_spawn()`. Vienas iš nurodomų parametru yra proceso vardas. Jeigu reikia, galime perduoti procesui parametrus. Taip pat galime specifikuoti, kuriame kompiuteryje leisti procesą. Procedūra `pvm_spawn()` grąžina paleistojo proceso identifikatorių, kurį galima panaudoti nurodant procesą – pranešimų gavėją. Paleistasis procesas turi „užsiregistruoti“ PVM sistemoje kviesdamas `pvm_mytid()` procedūrą, kuri grąžina proceso ID. Darbas užbaigiamas, kviečiant procedūrą `pvm_exit()`.

Pagrindinės pranešimų perdavimo procedūros.

Programos komunikuoja, kviesdamos pranešimų perdavimo funkcijas, tokias kaip `pvm_send()` ir `pvm_recv()`. Visos PVM siuntimo procedūros yra neblokuojančiosios (asynchroninės). Tą užtikrina buferiai, į kuriuos talpinami siunčiami ar gaunami pranešimai. PVM naudoja pranešimų žymes, kad išskirti gaunamus pranešimus. Žymė -1 naudojama priimant pranešimą su bet kuria žyme.

Kai perduodami duomenys yra to paties tipo, naudojamos PVM funkcijos `pvm_psend()`, `pvm_precv()`, kuriose nurodomi siuntimo/priėmimo masyvai. Bendras funkcijų pavidalas yra

```
pvm_psend (int dest_id, int mshtag, char* buf, int len, int datatype);
pvm_precv (int source_id, int mshtag, char* buf, int len, int datatype).
```

Tuo atveju, kai reiki siusti įvairiatipius duomenis, pvz., skaičius bei simbolių eiliutes, duomenys turi būti supakuoti į PVM išsiuntimo buferį. Procesas – gavėjas, gavęs pranešimą turi išpakuoti duomenys, tokiu pat formatu kaip jie buvo supakuoti. Kadangi tai néra trivialus veiksmas, galima panaudoti specialias supakavimo/išpakavimo funkcijas, tokias kaip `pvm_pkint()`, `pvm_upkint()`, naudojamas sveikujų skaičių pakavimui, ar `pvm_pkstr()`, `pvm_upkstr()` – simbolių eilutėms. Taigi visa pranešimo perdavimo procedūra, išskaitant ir buferio paruošimą atrodo taip.

<code>/*Process_1*/ pvm_initsend();</code>	<code>/*Process_2*/ pvm_recv(process_1, ...); //----></code>
--	---

pvm_pkint(..., &x, ...); pvm_pkstr(..., &s, ...); pvm_send(process_2); //---->	pvm_upkint(..., &x, ...); pvm_upkstr(..., &s, ...);
--	---

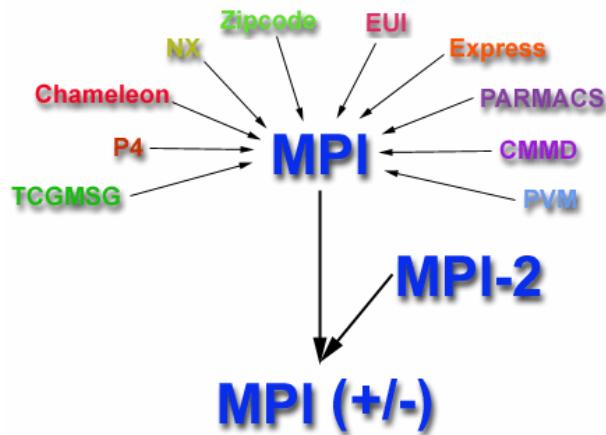
4.13 pav. PVM supakuotų pranešimų perdavimas.

Grupinės operacijos: Broadcast, multicast, Scatter, Gather, Reduce.

Grupinis pranešimų perdavimas atliekamas suformuotos procesų grupės viduje. Prosesai suformuoja vardinę grupę kviesdami pvm_joingroup(). Iškiesta funkcija pvm_bcast() perduos pranešimą kiekvienam grupės nariui.

4.3 MPI (Message Passing Interface)

Tai standartas, apibrėžiantis bibliotekines funkcijas ar procedūras, skirtas pranešimų perdavimui. MPI turi virš 120 funkcijų ir ši aibė plečiasi. Mes aptarsime tik nedidelę bet reikšmingą dalį. MPI dizainas įgalina nesudėtingą pagrindinių funkcijų panaudojimą.



4.14 pav. MPI standarto šaltiniai.

Pirmaoji MPI versija buvo paskelbta 1994 m. Dabar standartizuota MPI-2. Didelis bibliotekai priklausančių funkcijų skaičius buvo numatytas todėl, kad įgalintų programuotojus rašyti efektyvų kodą. Tačiau plačiai naudojama tik nedidelė funkcijų dalis. Užtenka tik šešių pagrindinių, kad užtikrinti lygiagrečiosios programos funkcionavimą. Taip pat kaip ir PVM sistemoje, palaikomos C ir FORTRAN programavimo kalbos. Kalbų ratas, bėgant laikui, plečiasi. Visos MPI funkcijos prasideda MPI priešdeliu. Daugelis funkcijų gražina diagnostinę informaciją (kurią mes pavyzdžiuose ignoruosime).

Procesų sukūrimas ir paleidimas vykdymui.

Kaip ir PVM sistemoje, lygiagretūs skaičiavimai vykdomi procesuose. MPI sistemoje procesų sukūrimas ir paleidimas nėra specifikuojamas, todėl priklauso nuo realizacijos. Esminis skirtumas nuo PVM sistemos – MPI q-je versijoje leidžiamas tik statinis procesų skaičius. Net jeigu būtų leidžiama paleisti procesus vykdyti dinamiškai, tai galėtų sumažinti skaičiavimų efektyvumą dėl naujo proceso paleidimo kaštų. Taigi MPI įgyvendina SPMD skaičiavimų modelį. Procesų skaičius paleidžiant dažniausiai nurodomas komandinės eilutės parametrais. Pavyzdžiu, keturių procesų paleidimas galėtų būti užduotas:

```
mpirun prog1 -np 4
```

Ši komanda paleistų keturias programos prog1 kopijas. Procesų atvaizdavimas į procesorius taip pat MPI standarte nėra apibrėžtas, ir jeigu toks atvaizdis reikalingas, jo apibrėžimas priklauso nuo realizacijos ir gali būti užduotas komandine eilute arba failiniu aprašu. MPI turi priemones apibrėžti tinklo topologiją (pvz. tinklą ir t.t.), taigi potencialiai procesų išdėstymas procesoriams gali būti atliktas automatiškai.

Prieš bet kurį MPI kreipinį, kodas turi būti inicijuotas funkcija **MPI_Init(Komandinės-Eilutės-Parametrai)** ir pabaigoje užbaigtas kreipiniu **MPI_Finalize()**.

Komunikatoriai.

Komunikatoriai apibrėžia pranešimų siuntimuose dalyvaujančių procesų aibę. Kiekvienas procesas turi savo rangą – sveikajį skaičių, susietą su komunikatoriumi. Skaičiavimų pradžioje užduotas globalus komunikatorius - **MPI_COMM_WORLD**, kuris apima visus procesus, numeruojamus nuo p iki p-1, kur p – procesų skaičius. Daugeliui programų pakanka šio komunikatoriaus, tačiau sudėtingesnėms, ypač sudarančioms bibliotekines procedūras, prireikia specializuotų procesų aibių.

SPMD skaičiavimo modelio naudojimas.

SPMD modelis yra idealus, kai procesai vykdo identiškas programas. Tačiau dažnai, vienas ar keli procesai turi vykdyti skirtinį kodą. Tuo atveju MPI programa turi apjungti visus skirtingesius kodus (pavyzdžiu, “šeimininko” bei “tarno” procesų kodą). Tai iliustruoja žemiau pateikta programa.

```
main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    .
```

```

    .
    /* rasti proceso ranga */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0)
        master();
    else
        slave();
    .
    .
    MPI_Finalize();
}

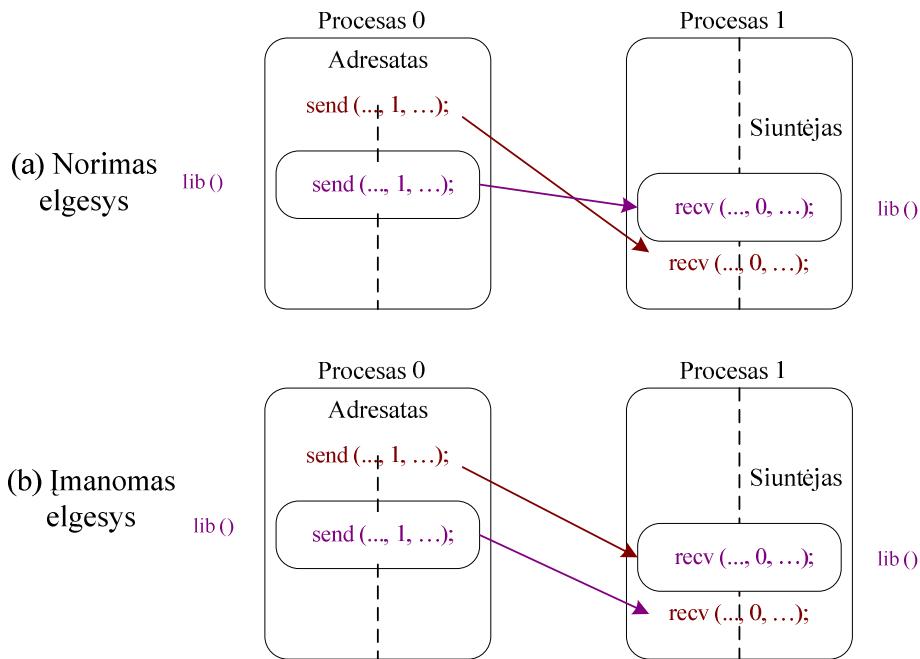
```

kur `master()` ir `slave()`, tai procedūros, kurias turi vykdyti atitinkamai šeimininko bei tarno procesai.

Nereikia pamiršti, kad esant SPMD modeliui, globalūs kintamieji egzistuos kaip kopija kiekviename procese. Jeigu kopijos nereikalingos, prasminga kintamuosius apibrėžti kaip lokalius kintamuosius. Atminties segmentai, kurių ilgiai gali būti skirtini skirtinguose procesuose turi būti išskiriami dinamiškai naudojant `malloc()` ar kitas procedūras.

Pranešimų perdavimo funkcijos.

Pranešimų perdavimas gali būti klaidų šaltiniu. MPI buvo kuriamas kaip saugi aplinka. Tačiau nereikia pamiršti, kad be programuotojo išreikštinai nurodytų pranešimo perdavimo operacijų yra ir bibliotekinės funkcijos taip pat galimai naudojančios MPI komunikaciją. Taigi, kaip parodyta paveikslėlyje galimas nenuspėjamas ir klaidingas elgesys.



4.15 pav. Nesaugi komunikacija [5]

MPI sprendimai, siekiant išvengti tokių klaidų.

“Komunikatoriai”.

Apibrėžia tarpusavyje komunikuojančių procesų aibę. Bibliotekos procedūros gali apibrėžti specifinius komunikatorius ir tokių būdu vartotojo programos komunikacija bus atskirta nuo bibliotekos. Komunikatoriai naudojami tiek vienas su vienu (point-to-point) saryšio metu tiek ir grupinio komunikavimo operacijose. Iš tiesų yra dviejų tipų komunikatoriai: intrakomunikatoriai, užtikrinantys procesų bendravimą grupės viduje ir interkomunikatoriai komunikavimui tarp grupių. Kiekvienas procesas komunikatoriaus viduje turi savo unikalų identifikatorių – rangą. Pradinis komunikatorius yra MPI_COMM_WORLD , kiti komunikatoriai sukuriami, remiantis jau egzistuojančiais komunikatoriais.

MPI Point-to-Point komunikacija.

Vykdoma įprastais siūsti-gauti kreipinius. Kaip parametrai nusakomi komunikatorius, proceso ID, bei pranešimo žymė. Siekiant priimti pranešimus iš visų procesų naudotinas MPI_ANY_SOURCE proceso ID bei žymė MPI_ANY_TAG. PVM stiliaus duomenų pakavimas iš esmės yra išvengiamas, naudojant duomenų tipus, kurie yra arba standartiniai, kaip MPI_INT, MPI_FLOAT arba formuojant norimo sudėtingumo išvestinius tipus. Taigi, tampa nereikalingi ir išreikštiniai siuntimo buferiai.

MPI komunikacijos baigtis.

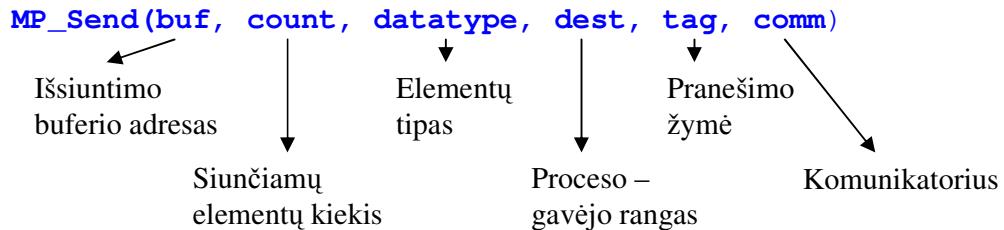
Yra variacijos: *lokaliai* baigtos bei *globaliai* baigtos duomenų persiuntimo operacijos. Laikysime operaciją lokaliai baigta, jeigu visi lokalūs veiksmai atlikti (bet procesas partneris gal būt neužbaigė savo operacijos). Operacija yra globaliai užbaigta, jeigu visi dalyvaujantys procesai baigė savo lokalius veiksmus

Blokuojančios funkcijos.

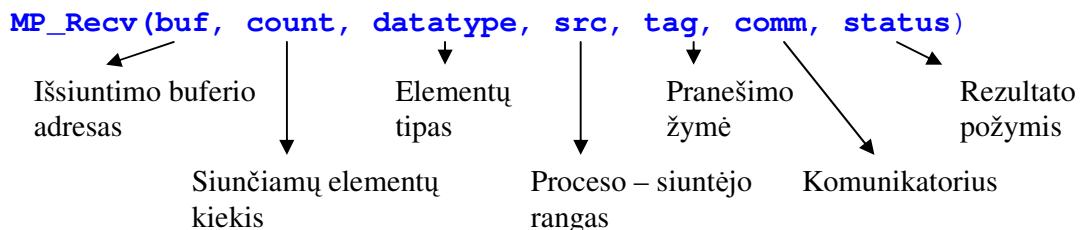
MPI požiūrius duomenų perdavimo operacijos yra blokuojančios, jeigu iš operacijos grįžtama atlikus lokalius veiksmus. Kaip lokalaus blokavimo išdava, duomenų buferis, kuris buvo naudojamas siuntimo operacijoje gali būti pakartotinai panaudojamas programoje, nepakeičiant siunčiamų duomenų. Tai nereiškia, kad pranešimas jau išsiustas arba, kad jį

gavo procesas gavėjas. Tai reiškia, jog procesas gali vykdyti tolimesnius veiksmus, neįtakodamas siunčiamų duomenų.

Blokuojančio išsiuntimo parametrai:



Blokuojančio pranešimo gavimo parametrai:



Atkreipsime dėmesį, kad maksimalus pranešimo ilgis yra apibrėžtas MPI_Recv() funkcijoje. Jeigu persiusta per daug duomenų, kurių ilgis viršija priemimo buferį, įvyksta perpildymo klaida.

Pavyzdys. Siunčiame kintamąjį x iš 0-nio proceso į 1-jį.

```

MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* rasti rangą */
if (myrank == 0) {
    int x;
    MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, stat);
}
  
```

MPI neblokujančios funkcijos.

Neblokujančios funkcijos grįžta iš karto, t.y., leidžia vykdyti tolimesnį kodą, nepriklausomai nuo to, ar lokalūs proceso veiksmai, perduodant/priimant pranešimą, jau atlikti ar dar ne. Taigi siuntimo procedūra MPI_Irecv(), kur I – reiškia „iš karto“ (*immediate*) grįš tada,

kai duomenų buferio pakeitimai dar nėra saugus. Atitinkamai neblokuojantis MPI_Irecv() grįš, net jeigu pranešimas dar negautas. Formatas:

- **MPI_Isend(buf,count,datatype,dest,tag,comm,request)**
- **MPI_Irecv(buf,count,datatype,source,tag,comm, request)**

Ar operacija pilnai atlikta galima išitikinti operacijomis **MPI_Wait()** and **MPI_Test()**.

- **MPI_Wait()** laukia kol operacija bus užbaigta ir tik tada grįžta.
- **MPI_Test()** gražina požymį, ar kreipimosi metu duomenų ankstesnė komunikacijos operacija yra pasibaigusi.

Tam, kad nusiųsti skaičių x iš proceso 0 į procesą 1 naudosime žemiau pateiktą pseudo-kodą:

```
MPI_Comm_rank (MPI_COMM_WORLD, &myrank); /* gauti proceso rangą */
if (myrank==0) {
    int x;
    MPI_ISend(x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
    compute();
    MPI_Wait(req1, status);
} else if (myrank==1) {
    int x;
    MPI_Recv(x, 0, MPI_INT, 1, msgtag, MPI_COMM_WORLD, status);
    .
    .
}
```

Keturi persiuntimo funkcijos **komunikacijos režimai**.

- *Standartinis* režimas. Nereiškia, kad atitinkama duomenų priėmimo funkcija jau startavusi. Buferizavimo apimtis MPI sistemoje nėra specifikuota. Jeigu sistemos realizacijoje buferizavimas numatytas, send funkcija grįš anksčiau, negu pranešimas bus priimtas.
- *Buferinis* režimas. Send gali grižti anksčiau negu bus iškviestas atitinkamas receive. Būtina nurodyti buferį naudojant funkciją **MPI_Buffer_attach()**.
- *Sinchroninis* režimas. Send ir receive gali startuoti nepriklausomai, tačiau grįžta tik užbaigus komunikacijos operaciją.
- *Pasiruošimo (ready)* režimas. Send gali startuoti tik tuo atveju, jeigu jau iškviestas atitinkamas receive. Naudoti atsargiai.

Kiekvienas iš nurodytų keturių režimų gali būti naudojamas tiek su blokuojančia, tiek ir su nesiblokuojančia *send()* funkcija. 3-jų nestandardinių režimų mnemonika: buffered – b, synchronous – s, ready – r. Pavyzdžiui `MPI_Issend()` – nesiblokuojanti sinchroninė *send()* funkcija.

Kolektyvinės komunikacijos funkcijos, tokios kaip broadcast, vyksta tarp procesų susietų tuo pačiu intrakomunikatoriumi. Skirtingai negu *point-to-point* atveju, pranešimų žymės nenaudojamos. Pagrindinės kolektyvinės komunikacijos funkcijos yra:

<code>MPI_Bcast()</code>	- Išplatina pranešimą iš šakninio proceso visiems likusiems
<code>MPI_Gather()</code>	- Surenka reikšmes iš procesų grupės
<code>MPI_Scatter()</code>	- Išbarsto šakninio proceso buferio dalis kitiems procesams
<code>MPI_Alltoall()</code>	- Persiunčia duomenų iš visų procesų visiems
<code>MPI_Reduce()</code>	- “Redukoja” visų procesų reikšmes į vieną reikšmę
<code>MPI_Reduce_scatter()</code>	- “Redukoja” reikšmes ir gautąjį išbarsto visiems proc.
<code>MPI_Scan()</code>	- atlieka visiems procesams prefix-redukcijas (?).

Programos fragmentas, kuris surenka elementus iš visų procesų į 0-nį procesą:

```
int data[10]; /*duomenys, kuriuos reikia surinkti iš proceso */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*rasti proceso rangą*/
if (myrank == 0) {
    MPI_Comm_size(MPI_COMM_WORLD, &grp_size);
    /*grupės dydis*/
    buf = (int *)malloc(grp_size*10*sizeof (int));
    /*išskirti atmintį*/
}
MPI_Gather(data, 10, MPI_INT, buf, grp_size*10, MPI_INT, 0, MPI_COMM_WORLD);
```

Barjeras (Barrier). Tai proceso sustabdymas, iki visi duotosios grupės (nusakomos komunikatoriumi) procesai iškvies barjero operaciją.

MPI programos pavyzdys. Programos paskirtis sudėti skaičių aibę saugomą faile.

```
#include mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000
```

```

void main(int argc, char *argv)
{
    int myid, numprocs;
    int data[MAXSIZE], i, x, low, high, myresult,
result;
    char fn[255];
    char *fp;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    if (myid == 0) { /* Atidaryti įėjimo failą */
        strcpy(fn,getenv(HOME"));
        strcat(fn,"/MPI/rand_data.txt");
        if ((fp = fopen(fn,"r")) == NULL) {
            printf("Can't open the input file:
%s\n\n", fn);
            exit(1);
        }
        for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d",
&data[i]);
    }
    MPI_Bcast(data, MAXSIZE, MPI_INT, 0,
MPI_COMM_WORLD); /* Paskleisti duomenis */
    x = n/proc; /* Susumuoti savo dalį */
    low = myid * x;
    high = low + x;
    for(i = low; i < high; i++)
        myresult += data[i];
    printf("I got %d from %d\n", myresult, myid); /* Rasti
globalią sumą */
    MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM,
0, MPI_COMM_WORLD);
    if (myid == 0) printf("The sum is %d.\n", result);
    MPI_Finalize();
}

```

4.4 Lygiagrečiųjų programų įvertinimas.

4.4.1 Lygiagretaus vykdymo laiko įvertinimas

Nuoseklaus algoritmo vykdymo laikas t_s įvertinamas skaičiuojant žingsnių skaičių vykdant optimaliausią algoritmą.

Skaičiuojant lygiagrečiosios programo vykdymo laiką t_p reikia iškaityti ne tik skaičiavimų laiką t_{comp} bet ir įvertinti komunikacijos nuostolių laiką t_{comm} :

$$t_p = t_{\text{comp}} + t_{\text{comm}}$$

Skaičiavimo laiką galima aproksimuoti atliktų kompiuterinių žingsnių skaičiumi. Jeigu veikia daugiau nei vienas procesorius, imsime ilgiausiai veikiančio (sudėtingiausio) proceso laiką. Bendru atveju, vykdymo laikas yra n (apdorojamų duomenų skaičiaus) ir p (procesų skaičiaus) funkcija, t.y.,

$$t_{\text{comp}} = f(n, p)$$

Dažnai algoritmas susideda iš etapų, taigi galime vertinti atskirus etapus:

$$t_{\text{comp}} = t_{\text{comp}1} + t_{\text{comp}2} + t_{\text{comp}3} + \dots$$

Paprastai remiamės prielaida, kad skaičiavimų sparta yra ta pati visuose proceasoriuose. Aišku heterogeniniuose klasteriuose analizė yra žymiai sudėtingesnė. Reikia naudoti įvairius apkrovos balansavimo metodus, kad užtikrinti maksimalią procesorių apkrovą.

Komunikacijos laiką nelengva tinkamai aproksimuoti. Jį lemia daugelis faktorių, išskaitant tinklo apkrovą. Pirmuoju priartėjimu galime laikyti:

$$t_{\text{comm}} = t_{\text{startup}} + nt_{\text{data}},$$

kur

- t_{startup} – yra pranešimo siuntimo paruošimo laikas – kiek užtrunkama išsiusti tuščią pranešimą. Daugeliu atveju galime laikyti konstanta.
- t_{data} – pranešimo perdavimo laikas, sugaištantamas perduodant minimalų duomenų vienetą (konstantinis), o n – duomenų skaičius.

Taigi idealizuotame artinyje laikysime, kad komunikacijos laikas proklauso tiesiškai nuo perduodamų duomenų skaičiaus. Kadangi paprastai programoje procesai bendrauja vienas su kitu daugelį kartų, suminis komunikacijos laikas t_{comm} būtų atskirų komunikacijos laikų suma. T.y.,

$$t_{\text{comm}} = t_{\text{comm}1} + t_{\text{comm}2} + t_{\text{comm}3} + \dots + t_{\text{comm}k}$$

Tipiškai daugelis procesų naudoja tuos pačius komunikacijos šablonus, taigi galime nagrinėti tik vieną iš tokų procesų.

Kadangi pranešimo paruošimo t_{startup} ir perdavimo t_{data} laikus matavome tais pačiais vienetais (vienos duomenų operacijos atlikimo laiku) jų suma sudaro bendrą vykdymo laiką t_p .

Aišku realiose sistemoje, idealizuota analizė ne visada tinkta. Prireikia atsižvelgti į aplinkybę, jog ne visi klasterio mazgai sujungti tiesiogiai, ir pranešimų siuntimo laikas priklauso nuo mazgų padėties tinkle.

4.4.2 Spartinimo įvertinimas.

Išmatavę ar apskaičiavę laikus t_s , t_{comp} , t_{comm} galime apskaičiuoti spartinimo (speedup) koeficientą bei skaičiavimų/komunikacijos (*computation/communication*) reitingą konkrečiai algoritmo realizacijai.

$$[\text{Spartinimas}] = t_s / t_p = t_s / (t_{\text{comp}} + t_{\text{comm}})$$

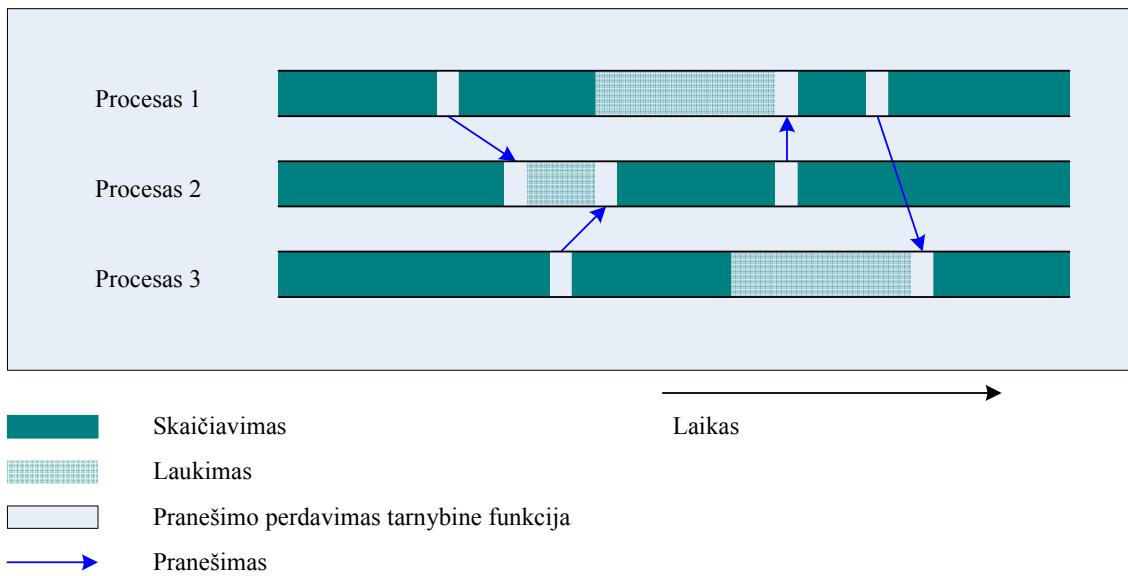
$$[\text{Skaičiavimų ir komunikacijos santykis}] = t_{\text{comp}} / t_{\text{comm}}$$

Abi funkcijos priklauso nuo procesorių skaičiaus p ir duomenų skaičiaus n. Keičiant n galima nustatyti lygiagrečiosios programos plečiamumą (*scalability*) kaip algoritmo ypatybę, esant didesniams procesorių skaičiui apdoroti daugiau duomenų.

Skaičiavimų/komunikacijos reitingas gali padėti įvertinti komunikacijos efektą, priklausomai nuo problemos bei sistemos dydžių.

4.4.3 Vizualizacijos instrumentai

Programų vykdymą galima pateikti kaip laiko – erdvės diagramas (arba procesų-laiko diagramas)



4.16 pav. Laikinė diagrama [5]

Laikinės diagramos leidžia aptikti multi-programmos siaurasias vietas, kurias nulemia nesėkmingai pasirinkti komunikacijos taškai, arba netinkamas procesorių apkrovos subalansuotumas.

4.4.4 Sąvadai

4.4.4.1 MPI primityvieji duomenų tipai

C duomenų tipai		Fortran duomenų tipai	
MPI_CHAR	signed char	MPI_CHARACTER	character(1)
MPI_SHORT	signed short int		
MPI_INT	signed int	MPI_INTEGER	integer
MPI_LONG	signed long int		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short int		

MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long int		
MPI_FLOAT	float	MPI_REAL	real
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	double precision
MPI_LONG_DOUBLE	long double		
		MPI_COMPLEX	complex
		MPI_DOUBLE_COMPLEX	double complex
		MPI_LOGICAL	logical
MPI_BYTE	8 binary digits	MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

Pastaba: Programuotojas gali susikurti išvestinius duomenų tipus: nuosekliuosius (*contiguous*), vektorinius, indeksuotus, struktūrinius.

4.4.4.2 Nuosekliai einačių duomenų tipo apibrėžimas C kalba

/* Tikslas – sukurti duomenų tipą išreiškiantį masyvo eilutę ir perduoti visiems procesams. */

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];
{
    int numtasks, rank, source=0, dest, tag=1, i;
```

```

float a[SIZE][SIZE] =
{1.0, 2.0, 3.0, 4.0,
 5.0, 6.0, 7.0, 8.0,
 9.0, 10.0, 11.0, 12.0,
 13.0, 14.0, 15.0, 16.0};
float b[SIZE];

MPI_Status stat;
MPI_Datatype rowtype;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

if (numtasks == SIZE) {
    if (rank == 0) {
        for (i=0; i<numtasks; i++)
            MPI_Send(&a[i][0], 1, rowtype, i, tag,
                     MPI_COMM_WORLD);
    }

    MPI_Recv(b, SIZE, MPI_FLOAT, source, tag,
             MPI_COMM_WORLD, &stat);
    printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f\n",
           rank,b[0],b[1],b[2],b[3]);
}

else
    printf("Must specify %d processors.
Terminating.\n",SIZE);

MPI_Type_free(&rowtype);
MPI_Finalize();
}

```

4.4.4.3 Grupinės komunikacijos funkcijų su parametrais sąvadas

[MPI_Barrier](#)

```

MPI_Barrier (comm)
MPI_BARRIER (comm,ierr)

```

[MPI_Bcast](#)

```

MPI_Bcast    (&buffer,count,datatype,root,comm)
MPI_BCAST    (buffer,count,datatype,root,comm,ierr)

```

[MPI_Scatter](#)

```
MPI_Scatter (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype,  
             root, comm)  
MPI_SCATTER (sendbuf, sendcnt, sendtype, recvbuf,  
             recvcnt, recvtype, root, comm, ierr)
```

[MPI_Gather](#)

```
MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf,  
            recvcount, recvtype, root, comm)  
MPI_GATHER (sendbuf, sendcnt, sendtype, recvbuf,  
            recvcount, recvtype, root, comm, ierr)
```

[MPI_Allgather](#)

```
MPI_Allgather (&sendbuf, sendcount, sendtype, &recvbuf,  
               recvcount, recvtype, comm)  
MPI_ALLGATHER (sendbuf, sendcount, sendtype, recvbuf,  
               recvcount, recvtype, comm, info)
```

[MPI_Reduce](#)

```
MPI_Reduce (&sendbuf, &recvbuf, count, datatype, op, root, comm)  
MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm, ierr)
```

[MPI_Allreduce](#)

```
MPI_Allreduce (&sendbuf, &recvbuf, count, datatype, op, comm)  
MPI_ALLREDUCE (sendbuf, recvbuf, count, datatype, op, comm, ierr)
```

[MPI_Reduce_scatter](#)

```
MPI_Reduce_scatter (&sendbuf, &recvbuf, recvcount, datatype,  
                     op, comm)  
MPI_REDUCE_SCATTER (sendbuf, recvbuf, recvcount, datatype,  
                     op, comm, ierr)
```

[MPI_Alltoall](#)

```
MPI_Alltoall (&sendbuf, sendcount, sendtype, &recvbuf,  
              recvcnt, recvtype, comm)
```

```
MPI_ALLTOALL (sendbuf, sendcount, sendtype, recvbuf,
               recvcnt, recvtype, comm, ierr)
```

MPI_Scan

```
MPI_Scan (&sendbuf, &recvbuf, count, datatype, op, comm)
MPI_SCAN (sendbuf, recvbuf, count, datatype, op, comm, ierr)
```

4.4.4.4 Redukcijos (*reduce*) operacijų sąrašas

MPI Redukcijos operacija	C Duomenų tipas	Fortran tipas
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bit-wise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bit-wise OR	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double
MPI_MINLOC	min value and location	float, double and long double

4.4.4.5 Grupinių ir komunikatorių funkcijų panaudojimo C kalba pavyzdys

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8
int main(argc,argv)
int argc;
char *argv[]; {
int rank, new_rank, sendbuf, recvbuf, numtasks,
ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
MPI_Group orig_group, new_group;
MPI_Comm new_comm;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
if (numtasks != NPROCS) {
printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
MPI_Finalize();
exit(0);
}
sendbuf = rank;
/* Nustatyti originalų grupės deskriptorių */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
/* Padalyti užduotis į dvi grupes, priklausomai nuo rango */
if (rank < NPROCS/2) {
MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
}
else {
MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
}

/* Sukurti naują komunikatorių ir atlikti kolektyvius apsikeitimus duomenimis */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
```

```

MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n", rank, new_rank, recvbuf);

MPI_Finalize();
}

```

4.4.4.6 Virtualiosios topologijos panaudojimo pavyzdys

Sukuriamas Dekarto 4×4 topologija 16-kai procesorių ir visi gretimi procesai apsikeičia savo rangu su kaimynais.

```

#include "mpi.h"
#include <stdio.h>
#define SIZE 16
#define UP    0
#define DOWN  1
#define LEFT   2
#define RIGHT  3

int main(argc,argv)
int argc;
char *argv[]; {
int numtasks, rank, source, dest, outbuf, i, tag=1,
inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL, },
nbrs[4], dims[2]={4,4},
periods[2]={0,0}, reorder=0, coords[2];

MPI_Request reqs[8];
MPI_Status stats[8];
MPI_Comm cartcomm;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder,
&cartcomm);
    MPI_Comm_rank(cartcomm, &rank);
    MPI_Cart_coords(cartcomm, rank, 2, coords);
    MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
    MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

    outbuf = rank;

    for (i=0; i<4; i++) {
        dest = nbrs[i];
        source = nbrs[i];
        MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
                  MPI_COMM_WORLD, &reqs[i]);
        MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,

```

```

        MPI_COMM_WORLD, &reqs[i+4]);
}

MPI_Waitall(8, reqs, stats);

printf("rank= %d coords= %d %d neighbors(u,d,l,r)= %d %d %d
%d\n",
      rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],
      nbrs[RIGHT]);
printf("rank= %d           inbuf(u,d,l,r)= %d %d %d %d\n",
      rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);
}
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}

```

4.4.4.7 MPI funkcijų suvestinė

Aplinkos valdymo		
MPI_Abort	MPI_Errhandler_create	MPI_Errhandler_free
MPI_Errhandler_get	MPI_Errhandler_set	MPI_Error_class
MPI_Error_string	MPI_Finalize	MPI_Get_processor_name
MPI_Init	MPI_Initialized	MPI_Wtick
MPI_Wtime		
“Point-to-Point” komunikacija		
MPI_Bsend	MPI_Bsend_init	MPI_Buffer_attach
MPI_Buffer_detach	MPI_Cancel	MPI_Get_count
MPI_Get_elements	MPI_Ibsend	MPI_Iprobe
MPI_Irecv	MPI_Irsend	MPI_Isend
MPI_Issend	MPI_Probe	MPI_Recv
MPI_Recv_init	MPI_Request_free	MPI_Rsend
MPI_Rsend_init	MPI_Send	MPI_Send_init
MPI_Sendrecv	MPI_Sendrecv_replace	MPI_Ssend

MPI_Ssend_init	MPI_Start	MPI_Startall
MPI_Test	MPI_Test_cancelled	MPI_Testall
MPI_Testany	MPI_Testsome	MPI_Wait
MPI_Waitall	MPI_Waitany	MPI_Waitsome

Grupinės komunikacijos funkcijos

MPI_Allgather	MPI_Allgatherv	MPI_Allreduce
MPI_Alltoall	MPI_Alltoally	MPI_Barrier
MPI_Bcast	MPI_Gather	MPI_Gatherv
MPI_Op_create	MPI_Op_free	MPI_Reduce
MPI_Reduce_scatter	MPI_Scan	MPI_Scatter
MPI_Scatterv		

Procesų grupės

MPI_Group_compare	MPI_Group_difference	MPI_Group_excl
MPI_Group_free	MPI_Group_incl	MPI_Group_intersection
MPI_Group_range_excl	MPI_Group_range_incl	MPI_Group_rank
MPI_Group_size	MPI_Group_translate_ranks	MPI_Group_union

Komunikatoriai

MPI_Comm_compare	MPI_Comm_create	MPI_Comm_dup
MPI_Comm_free	MPI_Comm_group	MPI_Comm_rank
MPI_Comm_remote_group	MPI_Comm_remote_size	MPI_Comm_size
MPI_Comm_split	MPI_Comm_test_inter	MPI_Intercomm_create
MPI_Intercomm_merge		

Išvestiniai tipai

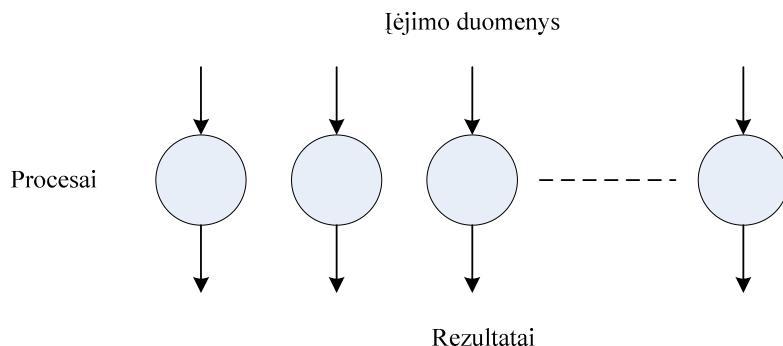
MPI_Type_commit	MPI_Type_contiguous	MPI_Type_count
MPI_Type_extent	MPI_Type_free	MPI_Type_hindexed
MPI_Type_hvector	MPI_Type_indexed	MPI_Type_lb

MPI_Type_size	MPI_Type_struct	MPI_Type_ub
MPI_Type_vector		
Virtualiosios topologijos		
MPI_Cart_coords	MPI_Cart_create	MPI_Cart_get
MPI_Cart_map	MPI_Cart_rank	MPI_Cart_shift
MPI_Cart_sub	MPI_Cartdim_get	MPI_Dims_create
MPI_Graph_create	MPI_Graph_get	MPI_Graph_map
MPI_Graph_neighbors	MPI_Graph_neighbors_count	MPI_Graphdims_get
MPI_Topo_test		
Kitos		
MPI_Address	MPI_Attr_delete	MPI_Attr_get
MPI_Attr_put	MPI_DUP_FN	MPI_Keyval_create
MPI_Keyval_free	MPI_NULL_COPY_FN	MPI_NULL_DELETE_FN
MPI_Pack	MPI_Pack_size	MPI_Pcontrol
MPI_Unpack		

5 Idealai išlygiagretinami skaičiavimai

5.1 Ivadas

Skaičiavimus, kurie gali būti suskirstyti į visiškai arba beveik visiškai nepriklausomas dalis vadinsime *idealai išlygiagretinamais*.^{*} Šiu problemų sprendimas lygiagrečiaisiais kompiuteriais yra mažiausiai komplikuotas. Visiškai lygiagretūs uždaviniai nereikalauja jokios komunikacijos tarp atskirų procesų, kaip pavaizduota paveikslėlyje. Taigi, tokio pavidalo skaičiavimai pasižymi maksimaliai įmanomu spartinimu. Vienintelė konstrukcija, reikalinga atlikti idealai lygiagretiemis skaičiavimams, yra pradinių duomenų perdavimas procesams. Kadangi daugeliu atveju skirtini procesai atlieka identiškus veiksmus, SPMD modelis yra tinkamiausias.

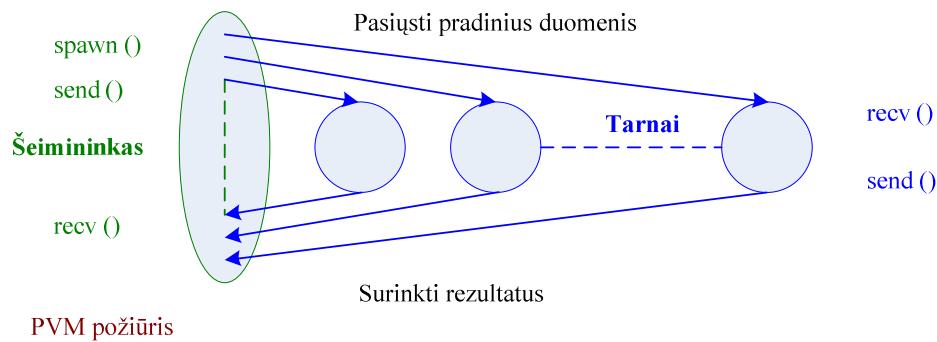


5.1 pav. Nesusiję, visiškai išlygiagretinami skaičiavimai [5]

Dažnai sutinkami uždaviniai, kurių sprendimas būdamas artimu ideliai išlygiagretinamiems, reikalauja ne tik pradinio duomenų paskirstymo procesams, bet ir tam tikro rezultatų iš atskirų procesų surinkimo bei apdorojimo. Taigi tinkamiausia procesų funkcionavimo schema – šeimininko-tarnų shema. Dinamiškų procesų naudojimo atveju ir pradžioje ir pabaigoje funkcionuoja vienintelis procesas – „šeimininkas“. Kaip pavaizduota paveikslėlyje, pradinis procesas paleidžia pagalbinius procesus, nusiųsdamas jiems duomenis ir pabaigoje surinkdamas rezultatus.

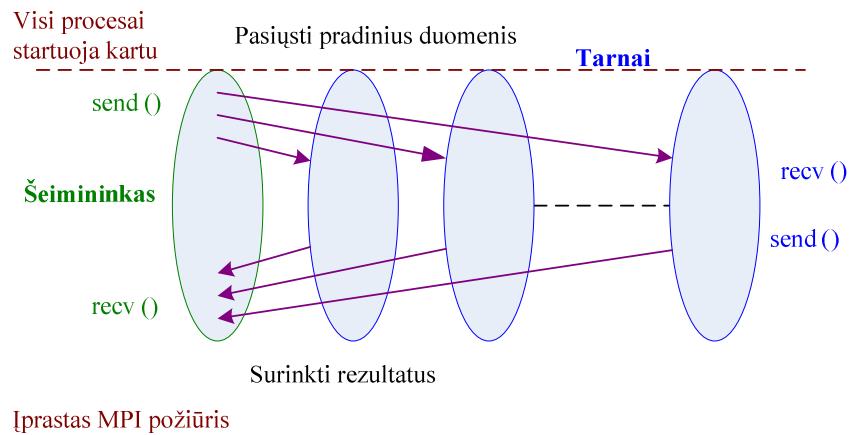
* Skyriuje naudojama [5] medžiaga

Pradžioje startuoti šeimininką



5.2 pav. Šeimininko - torno modelis dinaminio procesų sukūrimo atveju (PVM) [5]

Šeimininko - torno paradigma, kaip žinome, gali būti panaudota ir statinių procesų atveju (MPI). Tokiu atveju „šeimininko“ ir „tarnų“ procesų kodas sudedamas į vieną programą, ir priklausomai nuo proceso identifikatoriaus vykdoma viena ar kita šaka.



5.3 pav. Šeimininko - torno modelis statiniams procesams (MPI) [5]

Toliau nagrinėsime uždavinius, kurių sprendimo shema yra artima idealiai išlygiagretinamiems, t.y. tarp procesų egzistuoja minimali sąveika. Šiuo atveju, netgi esant vienodiems „tarnų“ procesams statinis procesų paskirstymas procesoriams gali būti ir neoptimalus, ypač jeigu procesoriai yra nevienodos galios, kaip paprastai būna kompiuterių klasterio atveju. Tada, norint užtikrinti efektyvius skaičiavimus, reikia naudoti apkrovos balansavimo technikas. Šiame skyriuje nagrinėsime tokius balansavimo metodus, kuriems nereikalinga „darbinių“ procesų tarpusavio komunikacija. Gilesnė balansavimo analizė bus pateikta vėlesniuose skyriuose.

5.2 Paveikslėlių transformavimas

Vienas iš daugelio uždavinių sutinkamų kompiuterinėje grafikoje – paveikslėlių transformavimas. Kompiuteriniu pavidalu saugomą paveikslėlį gali tekti pastumti, padidinti, pasukti, invertuoti, „apšvesti“, kitaip tariant- atliki geometrinę ar vaizdo transformaciją.

Kai kurios transformacijos gali būti itin sudėtingos analize arba skaičiavimo galių poreikiu, pavyzdžiui, skaitmenine kamera neryškiai nufotograuoto („nesufokusuoto“) objekto pirminio vaizdo atstatymas.

Šiame skyriuje nagrinėsime kompiuterinius paveikslėlius saugomus vaizdo elementų (*pixel*) masyvais. Kiekvienas elementas gali turėti tam tikrą skaitinių reikšmių intervalą. Pavyzdžiui, juodai-baltų paveikslėlių saugojimui pakanka dviejų reikšmių {0, 1} – vieno bito. Pilkų spalvų paveikslėliams paprastai naudojami aštuoni bitai, o spalvotiesiems - 24 bitai.

Vykstant geometrinę transformaciją, kiekvieno pikselio koordinatės (bet ne spalvinė informacija) yra keičiamos individualiai, nepriklausomai nuo kitų pikselių. Taigi – turime idealiai išlygiagretinamo skaičiavimo pavyzdį.

Žemiau pateiktos kai kurios plačiausiai naudojamos geometrinės transformacijos plokštumoje.

Poslinkis kryptimi (dx, dy). Transformuoto taško koordinatės (x',y') nusakomos priklausomybėmis:

$$x' = x + dx$$

$$y' = y + dy$$

čia (x,y) yra pradinės pikselio koordinatės.

Mastelio keitimas (*scaling*), kur kx, ky – koeficientai x bei y kryptimi:

$$x' = x * kx$$

$$y' = y * ky.$$

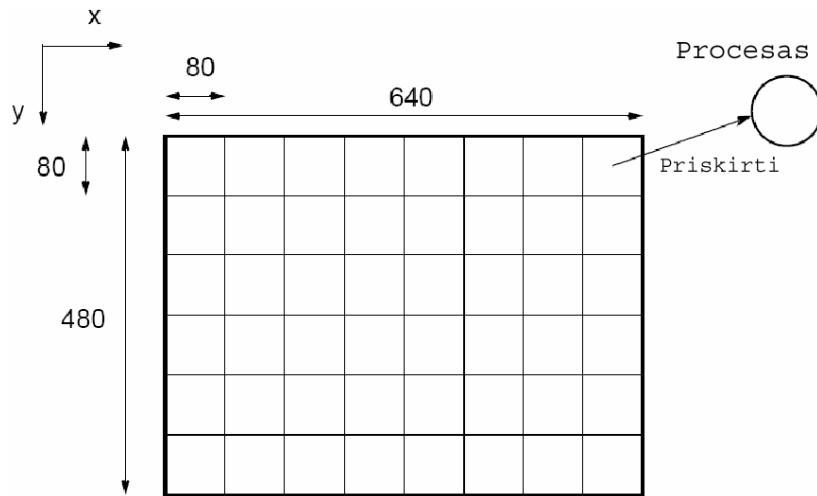
Pasukimas kampu t apie koordinačių sistemos pradžią:

$$x' = x * \cos t + y * \sin t$$

$$y' = -x * \sin t + y * \cos t.$$

Pradinis paveikslėlis dažniausiai laikomas faile, ir jį nuskaito pagrindinis procesas. Pagrindinis programavimo uždavinys lieka pikselių paskirstymas pagalbiniams procesams (procesoriams). Tam naudojami du pagrindiniai pikselių grupavimo metodai: skirtysti pradinį masyvą stačiakampėmis sritimis arba eilutėmis (stulpeliais). Tarkime, jeigu turime, kaip parodyta brėžinyje, 640 x 480 dydžio paveikslėlį bei 48 procesorius, paveikslėli galime

suskirstyti į 48 kvadratinės 80×80 pikselių dydžio sritis, ir kiekvienai sričiai priskirti po vieną procesorių.



5.4 pav. Srities suskaidymas stačiakampiais gabalais [5]

Kitas sprendimas,- išskaidyti masyvo eilutes į 48 dalis po 640×10 elementų kiekvienoje juostoje. Mūsų atveju, kai gretimi procesai neturi keistis duomenimis, pateiktieji sprendimai yra lygiaverčiai.

Žemiau pateiksime pseudo-kodą poslinkio transformacijai spręsti. Tarkime, naudosime vieną šeimininko procesą bei 48 "tarnus". Kiekvienam tarno procesui skirtiame apdoroti 80×80 masyvo sritį. Šeimininkas nusiunčia reikiamas srities koordinates tarnams bei priima pakeistas koordinates. Tarnai gauna pradinius duomenis, apskaičiuoja poslinkį ir gražina rezultatus šeimininkui.

```
/* Šeimininkas */

for (i = 0; i < 8; i++)      /* kiekvienas iš 48 procesų */
for (j = 0; j < 6; j++) {
    p = i*80; /* pradinės koordinatės */
    q = j*80;
    /* patalpinti koordinates į masyvus x[], y[]*/
    for (i = 0; i < 80; i++)
        for (j = 0; j < 80; j++) {
            x[i] = p + i;
            y[i] = q + j;
        }
}
```

```

z = j + 8*i;      /* proceso numeris */
/* išsiųsti koordinates tarnui */
send(Pz, x[0], y[0], x[1], y[1] ... x[6399], y[6399]);
}

for (i = 0; i < 8; i++)          /* kiekvienam iš 48 procesų */
for (j = 0; j < 6; j++) { /* gauti naujas koordinates */
    z = j + 8*i;           /* proceso numeris */
    /* gauti naujas koordinates */
    recv(Pz, a[0], b[0], a[1], b[1] ... a[6399], b[6399]);
    for (i = 0; i < 6400; i += 2) {/* atnaujinti paveikslėli */
        map[ a[i] ][ b[i] ] = map[ x[i] ][ y[i] ];
    }
}

/* Tarnas (i-sis procesas) */

recv(Pmaster, c[0] ... c[6400]);/* gauti pikselių bloką */
for (i = 0; i < 6400; i += 2) { /* transformuoti pikselius */
    c[i] = c[i] + delta_x;     /* pastumti x kryptimi */
    c[i+1] = c[i+1] + delta_y; /* pastumti y kryptimi */
}
/* nusiųsti transformuotus pikselius šeimininkui */
send(Pmaster, c[0] ... c[6399]);

```

Pastebėsime, kad šiame algoritme naudojamos paprasčiausios duomenų perdavimo operacijos. Grupinio duomenų perdavimo operacijos *scatter/gather* galėtų būti tam tikrose architektūrose efektyvesnės. Be to apdorotų duomenų surinkimo operacijoje galėtų būti panaudotas grupinis duomenų šaltinio proceso vardas – *recv(P_{ANY},...)*, kadangi duomenų surinkimo tvarka yra nesvarbi.

Algoritmo analizė. Tegu vienas skaičiavimo žingsnis - vieno pikselio apdorojimas. Nuosekliai apdorojant, matricą iš $n \times n$ pikselių, reikės atlikti $n \times n$ veiksmų, taigi nuoseklaus vykdymo laikas

$$t_s = n^2$$

taigi nuoseklaus vykdymo laiko įvertinimas yra $O(n^2)$.

Tegu turime p procesorių. Lygiagreti realizacija (eilutėmis/stulpeliais ar stačiakampėmis sritimis) suskirsto regioną į grupes po n^2/p pikselių. Lygiagreitai skaičiavimo laikas

$$t_{\text{comp}} = n^2/p$$

taigi laiko sudėtingumas išreiškiamas $O(n^2/p)$.

Prieš "tarnams" pradedant skaičiavimus, jiems turi būti persiusti duomenis, be to pabaigoje rezultatai turi perduoti procesui – "šeimininkui". Komunikacijos trukmė perduodant pranešimą yra nusakoma formule

$$t_{\text{comm}} = t_{\text{startup}} + m t_{\text{data}}.$$

Čia t_{comm} – konstantinis pranešimo iniciavimo (paruošimo) laikas, m – perduodamų duomenų kiekis, t_{data} – duomenų vieneto perdavimo trukmė. Pranešimo perdavimo laiką galime nusakyti kaip $O(m)$, tačiau praktikoje pasiruošimo laiko t_{startup} ignoruoti negalime, ypač kai komunikacijų skaičius tarp procesų yra didelis. Paveikslėlio transformacijos uždavinio sprendime

$$t_{\text{comm}} = 2p (t_{\text{startup}} + n^2/p t_{\text{data}}) = O(p + n^2)$$

Bendras algoritmo lygiagreitai vykdymo įvertis:

$$t_p = t_{\text{comp}} + t_{\text{comm}} = O(n^2/p) + O(n^2),$$

kuris yra $O(n^2)$ eilės, jeigu procesų skaičius fiksuotas.

5.3 Mandelbroto aibė.

Kitas paveikslėlio apdorojimo uždavinys yra taip vadinamosios *Mandelbroto* aibės pavaizdavimas. Mandelbroto aibė – tai kompleksinės plokštumos taškų aibė, kuri yra kvazistabili,- didės arba mažės, bet neviršys tam tikros ribinės reikšmės,- skaičiuojant funkcijos iteracijas:

$$z_{k+1} = z_k^2 + c$$

kur z_{k+1} yra $(k+1)$ -ji kompleksinio skaičiaus $z = a + bi$ iteracija, o c yra kompleksinis skaičius, užduodantis taško pozicija kompleksinėje plokštumoje. Pradinė z reikšmė yra 0. Iteracijos yra tęsiamos tol, kol vektoriaus z ilgis viršija skaičių 2, arba iteracijų skaičius viršija tam tikrą ribą. Priminsime, jog vektoriaus ilgis:

$$z_{\text{length}} = \sqrt{a^2 + b^2}$$

Iteracijos $z_n = z_{n-1}^2 + c$ skaičiavimas kompleksiniams skaičiams gali būti supaprastintas, išreiškus $z = a + bi$:

$$z^2 = a^2 + 2abi + (bi)^2 = a^2 - b^2 + 2abi,$$

kitaip tariant, jog realioji dalis yra $a^2 - b^2$, o menamoji $2abi$. Jeigu išreikštume

$$z = z_rea + z_imag \mathbf{i},$$

nauja iteracija galėtų būti apskaičiuojama:

$$\begin{aligned} z_real_n &= z_real_{n-1}^2 - z_imag_{n-1}^2 + c_real \\ z_imag_n &= 2z_real_{n-1}z_imag_{n-1} + c_imag, \end{aligned}$$

kur $c = c_real + c_imag \mathbf{i}$.

Kompleksinį skaičių galėtume nusakyti struktūra:

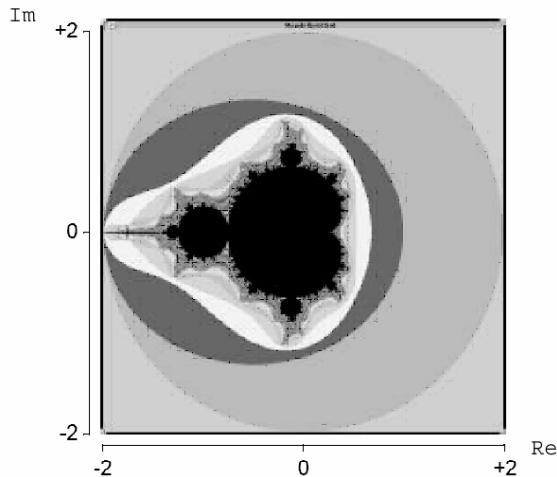
```
structure complex {
    float real;
    float imag;
};
```

Funkcija, skaičiuojanti iteracijų skaičių duotajam taškui c atrodytu taip.

```
int cal_pixel(complex c)
{
    int count, max;
    complex z;
    float temp, lengthsq;
    max = 256;
    z.real = 0; z.imag = 0;
    count = 0; /* iteracijų skaičius */
    do {
        temp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2 * z.real * z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real * z.real + z.imag * z.imag;
        count++;
    }
    while ((lengthsq < 4.0) && (count < max));
    return count;
}
```

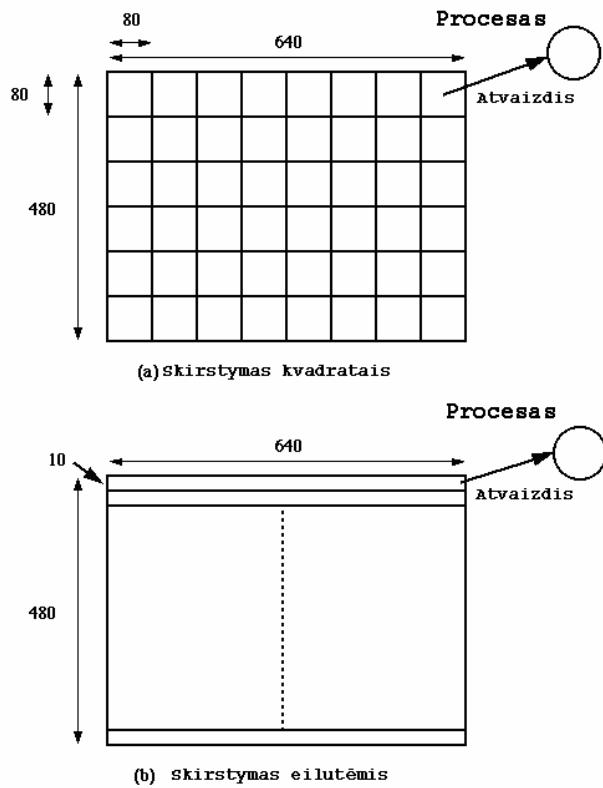
Norint pavaizduoti Mandelbroto aibę, reikės pasirinkti tam tikrą kompleksinės erdvės sritį, pavyzdžiui, stačiakampio formos, skiriamą gebą, t.y., pikselių skaičių horizontalia bei vertikalia kryptimi. Vėliau, kiekvienam pikseliui apskaičiavę jo koordinates $c=(x,y)$, iškviesime funkciją $cal_pixel(c)$. Tipinis rezultatas parodytas paveikslėlyje.

Mandelbroto aibės sudarymo uždavinys yra plačiai naudojamas lygiagrečiųjų sistemų testavimui, kadangi jo sprendimas reikalauja intensyvių skaičiavimų. Pastebėsime, kad skirtiniems kompleksinės plokštumos taškams, skaičiavimų apimtis gali labai skirtis.



5.5 pav. Mandelbroto aibė kompleksinėje srityje $[-2,2] \times [-2,2]$

Skaičiavimų išlygiagretinimas. Mūsų nagrinėjamai temai šis uždavinys gerai tinkamas, kadangi kiekvieno pikselio reikšmė yra skaičiuojama nepriklausomai nuo kitų pikselių. Išnagrinėsime ne tik *statinį* (iš anksto nustatyta) paveikslėlio srities priskyrimą procesoriui, bet ir *dinaminį*, kai procesorius gali apdoroti įvairias sritis.



5.6 pav. Paveikslėlio sričių priskyrimas procesoriams [5]

Statinis užduočių priskyrimas. Gali remtis skirstymu stačiakampėmis sritimis arba eilutėmis/stulpeliais, kaip parodyta paveikslėlyje. Žemiau pateiktas sprendimas, naudojantis 80x80 dydžio pikselių sritis, statiskai paskirtas kiekvienam procesoriui.

```
/* Master-procesas */
for (i = 0; i < 8; i++) /* kiekvienam iš 48 procesų */
    for (j = 0; j < 6; j++) {
        x = i*80; /* bitmap'o pradinės koordinatės */
        y = j*80;
        z = j + 8*i; /* proceso numeris */
        send(Pz, x, y); /* nusiųsti proadžios koordinates tarnui */
    }
for (i = 0; i < (640 * 480); i++) /* iš procesų, bet kuria tvarka */
    recv(ANY, x, y, color); /* gauti koordinates/spalvą */
    display(x, y, color); /* pavaizduoti pikselį ekrane */
}
```

```

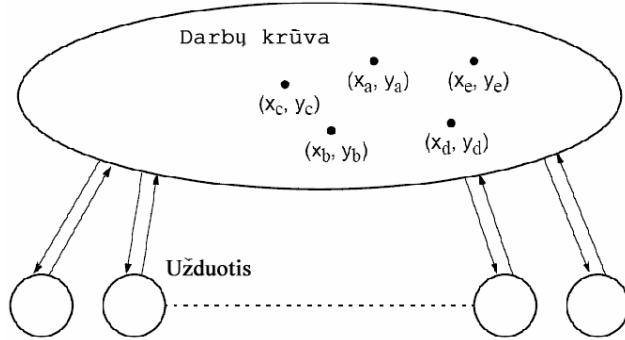
/* Slave -procesas (i-sis) */
recv(Pmaster, x, y); /* gauti pradines koordinates */
for (i = x; i < (x + 80); i++) {
    for (j = y; j < (y + 80); j++) {
        a = -2 + (x * 3.25 / 640); /* mastelis */
        b = -1.25 + (y * 2.5 / 480);
        color = cal_pixel(a, b); /* skaičiuoti spalvą */
        send(Pmaster, i, j, color); /* pasiųsti koordinates, spalvą master-procesui */
    }
}

```

Pastebėsime, kad patektoje realizacijoje, duomenų perdavimo vienetas yra vienas pikselis. Nesunku būtų šia programą modifikuoti, perduodant iš karto visą masyvo fragmentą, tuo būdų sumažinant komunikacijos kaštus.

Dinaminis užduočių išskirstymas. Mandelbroto uždavinio sprendimas ypatingas tuo, jog įteracijų skaičius taigi ir skaičiavimų laikas, apdorojant vieną pikselį, gali labai skirtis. Be to heterogeniuose kompiuterių klasteriuose procesorių greitis taip pat gali būti įvairus. Taigi, nustačius kiekvienam procesoriui užduotis iš anksto, procesoriai gali užbaigti darbą skirtingu laiku. Reiškia - sistemos efektyvumas nebus 100%. Ši problema yra vadinama apkrovos balansavimo (*load balancing*) problema. Idealiu atveju reikėtų pilnai apkrauti procesorius visą darbo laiką. Pavyzdžiui, paskiriant skirtiniems procesoriams skirtingo dydžio sritis, atsižvelgiant į skaičiavimų sudėtingumą. Deja, daugeliu atveju negalima tiksliai įvertinti "apkrovos" dydžio iš principio, neatlikę visų skaičiavimų. Be to, pasikeitus uždavinio pradiniam duomenims (pvz., skaičiavimo tikslumui Mandelbroto aibės atveju), turėtume visiškai naujają situaciją.

Paprasčiausias balansavimo sprendimas, naudoti taip vadinamą "darbų krūvą" (*work pool*), iš kurios procesorius, užbaigę turimą darbą, galėtų pasiimti naujų duomenų "porciją". Mandelbroto aibės atveju darbų krūva – tai pikselių koordinacių sąrašas, kurį patiekia procesas šeimininkas. Procesas – tarnas – nusiskaito eilines koordinates, atlieka iteracijas bei persiunčia rezultatą šeimininkui, šiuos veiksmus atlikdamas cikle. Kad sumažinti intensyvią komunikaciją duomenų vienetas gali būti sustambintas, apjungiant kelis taškus į masyvą, perduodamą vienu kreipiniu. Paveikslėlyje pateikta darbų krūvos iliustracija.



5.7 pav. Darbu krūva [5]

```

/* Master – procesas */
for (k = 0; k < nproc; k++) { /* Pasiųsti procesui koordinates */
    x = k * 80;
    y = k * 80;
    send(Pk, data_tag, x, y);
}
unsent_blocks = nproc;
while (unsent_blocks < BLOCKS) {
    recv (Pi, ANY, result_tag, x, y, color);
    /* Priimti iš bet kurio tarno id in Pi */
    display (x, y, color); /* pavaizduoti pikselį ekrane */
    x = unsent_blocks * 80;
    y = unsent_blocks * 80;
    send(Pi, data_tag, x, y); /* pasiųsti daugiau koordinačių */
    unsent_blocks++;
}
for (last = 0; last < nproc; last++) { /* gauti galutinius rezultatus*/
    recv(Pi, ANY, result_tag, x, y, color);
    display(x, y, color);
    send(Pi, ANY, terminator_tag);
}

/* Prosesas - Slave (i-sis) */
while (!recv(Pmaster, terminator_tag)) { /* kol ne visi pikseliai paimti */
    recv(Pmaster, data_tag, x, y); /* gauti pradines koordinates */
    for (i = x; i < (x + 80); i++)
        for (j = y; j < (y + 80); j++) {

```

```

a = -2 + (x * 3.25 / 640);      /* mąstelis */
b = -1.25 + (y * 2.5 / 480);
color = cal_pixel(a, b);/* spalva */
send(Pmaster, result_tag, i, j, color);
/* pasiųsti koordinates bei spalvą master-procesui */

}

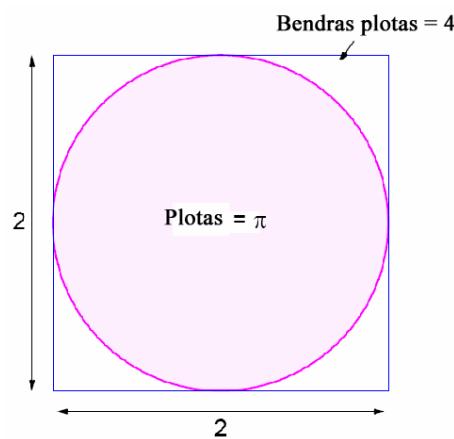
```

Šiame pavyzdje panaudota speciali baigmės žymė, kad procesai tarnai aptiktų darbo pabaigą. Procesai dirba asynchrone, t.y., gražinami rezultatų tvarka gali skirtis kiekvieno vykdymo metu. Siekiant užtikrinti optimalų balansą, vykdant šį uždavinį konkrečioje sistemoje, reikia nustatyti optimalų užduoties dydį (*grain size*), kuris įgalintų minimizuoti pranešimo paruošimo išsiuntimui kaštus, mažinant užduoties dydį, bei minimizuotų potencialias procesorių „prastovas“, kai darbų krūva tampa tuščia.

5.4 Monte-Karlo metodai

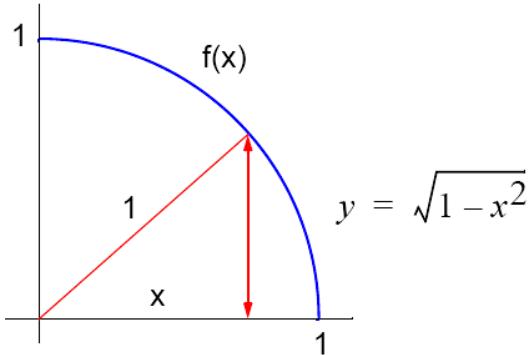
Tai skaičiavimai įprasti fizikiniuose bei matematiniuose modeliuose, kurie remiasi atsitiktinių dydžių kompiuteriniu generavimu. Tipiškas uždavinys – apskritimo kvadratūros uždavinys – t.y., ploto radimas. Jeigu apskritimą spinduliu $r = 2$ patalpinsime į 2×2 dydžio

kvadratą, santykis tarp apskritimo bei apskritimo ploto bus išreiškiamas: $\frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}$. Taigi, jeigu atsitiktinai mėtysime į kvadratą taškus, santykis tarp patekusių į apskritimą N_a ir visų taškų N_{total} bus lygus $\pi/4$. Brėžinyje pateiktos abi figūros.



5.8 pav. Į kvadratą įbrėžtas apskritimas [5]

Ar taškas su koordinatėmis (xr,yr) patenka į teigiamajį apskritimo ketvirtį galime nustatyti, naudodami formulę $y_r \leq \sqrt{1 - x_r^2}$; that is, $y_r^2 + x_r^2 \leq 1$. Pastebėsime, kad „ketvirčio“ apskritimo plotas lygus integralui: $\int_0^1 \sqrt{1 - x^2} dx = \frac{\pi}{4}$. Žr. paveikslėli.



5.9 pav. Integralo skaičiavimas [5]

Taigi, Monte-Karlo metodu galime apskaičiuoti ir apibrėžtinį integralą. Pastebėsime, kad vienamačio integralo skaičiavimas yra itin neefektyvus vienamatės funkcijos atveju, tačiau plačiai naudojamas, skaičiuojant daugiamatiškus integralus sudėtingose srityse. Bendru atveju vieno argumento funkcijos $f(x)$ integralas *Area* intervale $[x1, x2]$ randamas:

$$\text{Area} = \int_{x_1}^{x_2} f(x) dx = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N f(x_i)(x_2 - x_1)$$

Pavyzdžiui, nuoseklusis algoritmas integralo

$$I = \int_{x_1}^{x_2} (x^2 - 3x) dx$$

skaičiavimui gali būti toks.

```

sum = 0;
for (i = 0; i < N; i++) {           /* N atsiktinių x reikšmių */
    xr = rand_v(x1, x2);           /* gauti atsitiktinį dydį */
    sum = sum + xr * xr - 3 * xr; /* apskaičiuoti f(xr) */
}
area = (sum / N) * (x2 - x1);

```

Lygiagrečioji realizacija. Akivaizdu, kad duotasis uždavinys priklauso idealiai išlygiagretinamų uždavinių klasei, kadangi integralinės sumos gali būti skaičiuojamos nepriklausomai. Didžiausia problema, nepriklausomų atsitiktinių dydžių generavimas. Iprasta

C funkcija *rand()* čia netinka. Vienas iš sprendimo būdų – patikėti atsitiktinių dydžių generavimą procesui šeimininkui, kuris perduoda juos tarnams ir surenka rezultatus – integralines sumas. Šiuo atveju dideli komunikacijos kaštai. Antrasis kelias – panaudoti lygiagretujį a.d. generatorių, kuris sukonstruotas taip, jog ištisinėje generuojamų atsitiktinių skaičių sekoje kiekvienas procesas gali, pritaikydamas paprastą formulę, apskaičiuoti tam procesui skirtą reikšmę („peršokdamas“ kitiems procesams skirtas reikšmes).

6 Skaidymo (skirstymo) strategijos

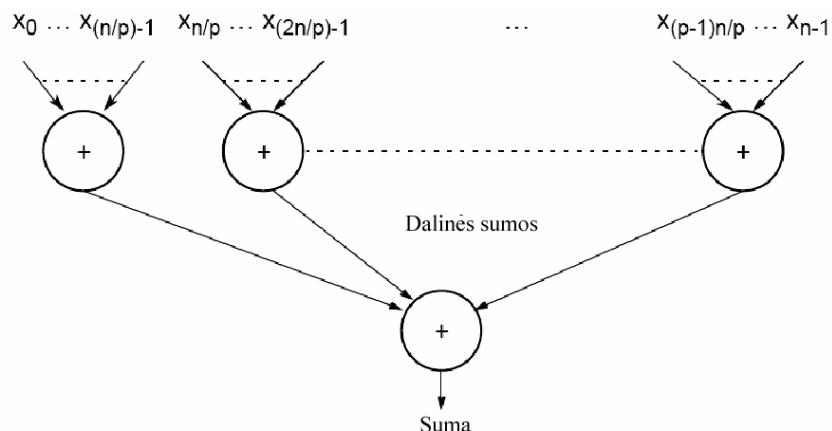
Skaidymą (*partitioning*) suprasime kaip problemos skaidymą į atskiras dalis, kurios gali būti sprendžiamos nepriklausomai.* Natūralus skaidymo išplėtinys yra „skaldyk ir valdyk“ strategijos įgyvendinimas, kai sprendžiama problemos dalis rekursyviai skaidoma į smulkesniąsias.

6.1 Skaidymo strategijos

Skaidydami problemą į dalis, mes įgaliname lygiagretų problemos sprendimą. „Idealiai“ išlygiagretinamiems uždaviniams, šių dalių sprendimas buvo iš esmės nepriklausomas. Dažniau sutinkami uždaviniai, kurie reikalauja papildomo atskirų dalių rezultatų apjungimo. Tokių uždavinių sprendimas ir bus nagrinėjamas šiame skyriuje.

Skaidyti galima duomenis – tai vadinama *duomenų dekompozicija*, arba funkcijas – *funkcinė dekompozicija*, kuri sutinkama rečiau.

Skaičių aibės sumavimas – paprasčiausia problema, sprendžiama, skaidant aibę į atskirus poaibius, kuriose sumos skaičiuojamos nepriklausomai. Jeigu turime n skaičių ir p procesorių, paprasčiausias sprendimas – pateikti kiekvienam procesoriui skaičiuoti n/p skaičių poaibio sumą.



6.1 pav. Skaičių sekos skaidymas sumuojujant [5]

* Skyriuje naudojama [5] medžiaga

Programos, įgyvendinančios ši sprendima struktūra galėtų būti „šeimininko-tarno“ pavidalo schema, kai šeimininkas perduoda tarnams pradinus duomenis, ir, galiausiai, surenka rezultatus.

Realizacijos naudojant *send/receive* pseudo kodas.

Šeimininkas – *Master*

```
s = n/m; /* skaičių kiekis tarnui*/
for (i = 0, x = 0; i < m; i++, x = x + s)
    send(&numbers[x], s, P1); /* siųsti skaičių tarnams */
sum = 0;
for (i = 0; i < m; i++){ /* laukti rez-tų iš tarnų*/
    recv(&part_sum, PANY);
    sum = sum + part_sum; /* susumuoti dalin. rez-tus */
}
```

Tarnas- *Slave*

```
recv(numbers, s, Pmaster); /* gauti s skaičių iš šeimininko*/
part_sum = 0;
for (i = 0; i < s; i++) /* sudėti skaičius */
    part_sum = part_sum + numbers[i];
send(&part_sum, Pmaster); /* pasiųsti sumą šeimininkui */
```

Realizacija, naudojant *Broadcast* operaciją

Šeimininkas - *Master*

```
s = n/m; /* skaičių kiekis tarnams */
bcast(numbers, s, Pslave_group); /* siųsti visus skaičius tarnams */
sum = 0;
for (i = 0; i < m; i++){ /* laukti rezultatų iš tarnų */
    recv(&part_sum, PANY);
    sum = sum + part_sum; /* susumuoti dalin rez-tus */
}
```

Tarnas - *Slave*

```
bcast(numbers, s, Pmaster); /* gauti visus skaičius iš šeimininko*/
start = slave_number * s; /* pradžia */
end = start + s;
part_sum = 0;
for (i = start; i < end; i++) /* susumuoti */
    part_sum = part_sum + numbers[i];
send(&part_sum, Pmaster); /* nusiųsti sumą šeimininkui */
```

Realizacija naudojant *scatter*, *reduce* operacijas

Šeimininkas – *Master*

```
s = n/m; /* skaičių kiekis */
scatter(numbers, &s, Pgroup, root=master); /* nusiųsti skaičius tarnams */
reduce_add(&sum, &s, Pgroup, root=master); /* priimti rezultatus iš tarnų */
```

Tarnas - *Slave*

```
scatter(numbers, &s, Pgroup, root=master); /* gauti s skaičių */
..... /* sudėti */
reduce_add(&part_sum, &s, Pgroup,
           root=master); /* pasiųsti sumas šeimininkui */
```

Sudėtingumo analizė

Nuoseklaus algoritmo sudėtingumas lygus $O(n)$.

Lygiagreatus algoritmo sudėtingumas (naudojant individualias *send*, *receive* operacijas) susideda iš:

1. Komunikacijos fazės. Sudėtingumas:

$$t_{comm1} = m(t_{startup} + (n/m)t_{data})$$

2. Skaičiavimo fazės. Sudėtingumas:

$$t_{comp1} = n/m - 1$$

3. Komunikacijos fazė: dalinių rezultatų surinkimas:

$$t_{comm2} = m(t_{startup} + t_{data})$$

4. Galutinio sumavimo fazė:

$$t_{comp2} = m - 1$$

Suminis sudėtingumas:

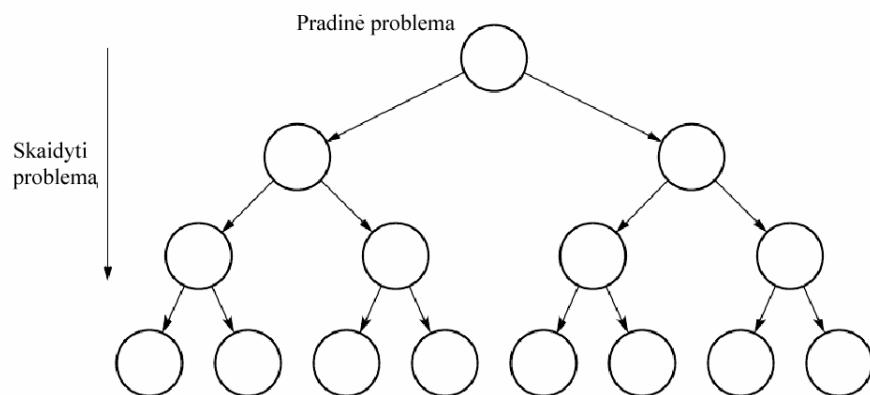
$$\begin{aligned} tp &= (t_{comm1} + t_{comm2}) + (t_{comp1} + t_{comp2}) = \\ &= 2mt_{startup} + (n + m)t_{data} + m + n/m - 2 = \\ &= O(n + m) \end{aligned}$$

Taigi lygiagreatus vykdymo sudėtingumas blogesnis negu nuosekliosios programos.

6.2 „Skaldyk ir valdyk“ strategija

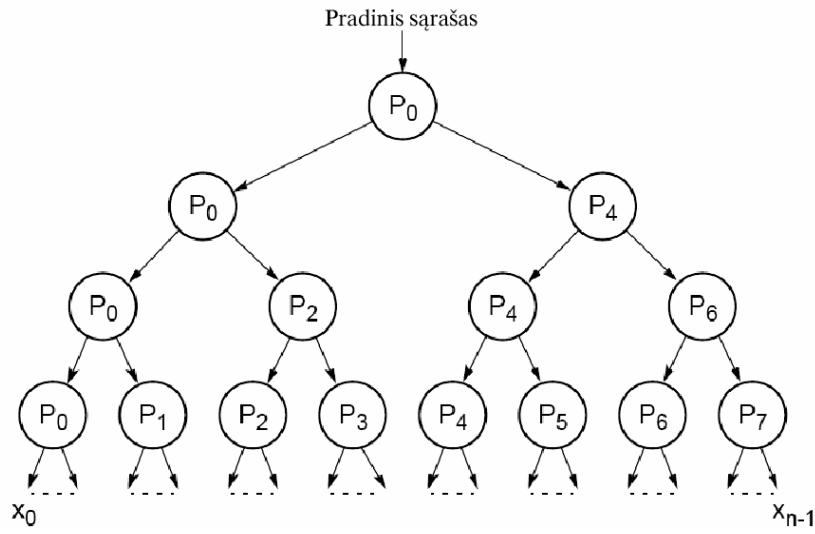
Ši strategija išgyvendinama skaidant problemą į dalines problemas, kurios turi tokią pačią formą kaip ir pradinė problema, taigi gali būti skaidomos toliau. Nuosekliajame programavime tai atliekama rekursyviaisiais kreipiniais. Pavyzdžiu, skaičių sąrašo suma gali būti apskaičiuota, kaip dviejų posarašių sumos suma. Kiekvienas posarašis gali būti rekursyviai skaidomas toliau. Rekursija baigiamā, kai posarašis susideda tik iš vieno skaičiaus. Rekursyvus sumavimas paprastai nėra naudojamas, kai egzistuoja paprastas iteracinis sprendimas, tačiau tokia sprendimo forma gali būti taikoma sudėtingesnėms problemoms, pavyzdžiu rūšiavimo suliejimu arba greitojo rūšiavimo (*quicksort*) algoritmuose.

Kai rekursyvaus skaidymo algoritmas dalo duomenis į lygiai dvi dalis, suformuojama binarinio medžio pavidalo procedūra, pavaizduota paveikslėlyje.



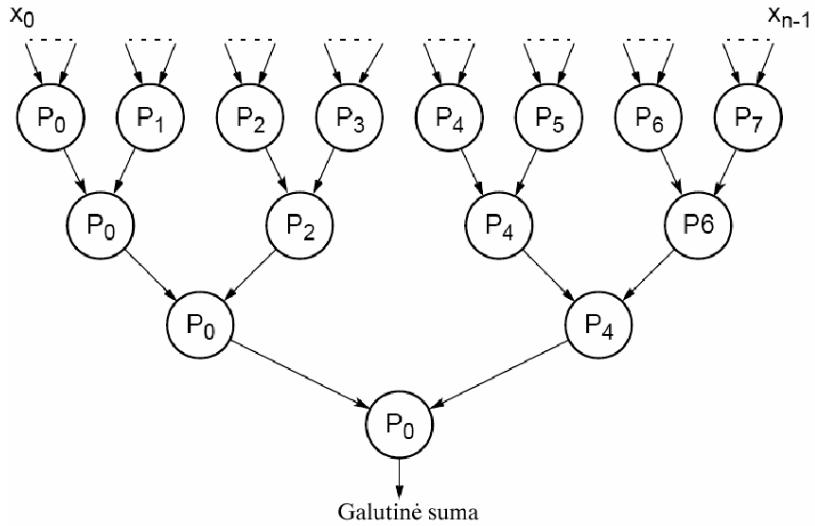
6.2 pav. Medžio pavidalo struktūra kaip rekursyvaus skaidymo išdava [5]

Lygiagrečioji realizacija įgalina peržiūrėti (konstruoti) tokį medį keliomis šakomis vienu metu. Paprasčiausiu atveju, vienai medžio viršūnei galime priskirti po procesorių, tačiau šis sprendinys nėra optimalus, nes apie pusę „dalyvių“ prastovės. Efektyvesnis sprendimas, panaudojantis tuos pačius procesorius keliose fazėse, pateikiamas schemae.



6.3 pav. Sąrašo skaidymo į dalis komunikacinė struktūra [5]

Apskaičiavus rezultatus medžio lapuose, jie turi būti apjungiami – susumuojami. Komunikacinė struktūra – analogiškas medis, peržiūrimas iš lapų į šaknį.



6.4 pav. Dalinių sumų surinkimas [5]

Lygiagretusis kodas proceso P_0 proceso požiūriu galėtų atrodyti taip.

```
/* skaidymo fazė */
divide(s1, s1, s2); /* suskirstyti s1 į s1 ir s2 sąrašus */
send(s2, P4); /* siusti vieną dalį kitam procesui */
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P1};
part_sum = *s1; /* apjungimo fazė */
recv(&part_sum1, P1);
```

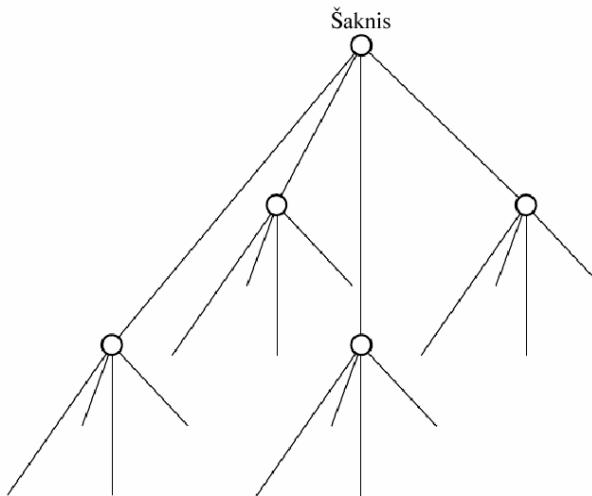
```

part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
part_sum = part_sum + part_sum1;

```

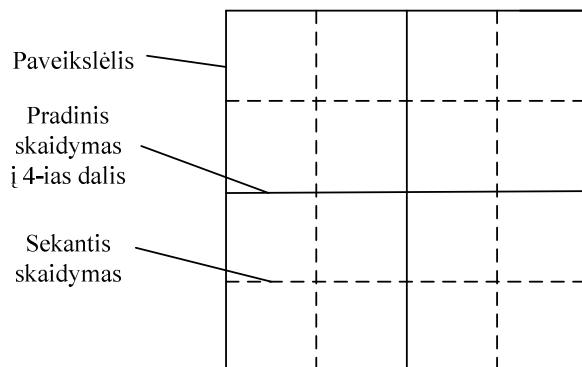
6.3 M-aris medis

Skaldyk ir valdyk technika gali būti pritaikoma, skaidant duomenis į daugiau dalių negu 2. Galime gauti, pavyzdžiui, ketvirtainį medį.



6.5 pav. Ketvirtainis medis [5]

Ketvirtainiai medžiai plačiai naudojami, atliekant stačiakampės srities – pavyzdžiui paveikslėlio koordinatinę dekompoziciją.

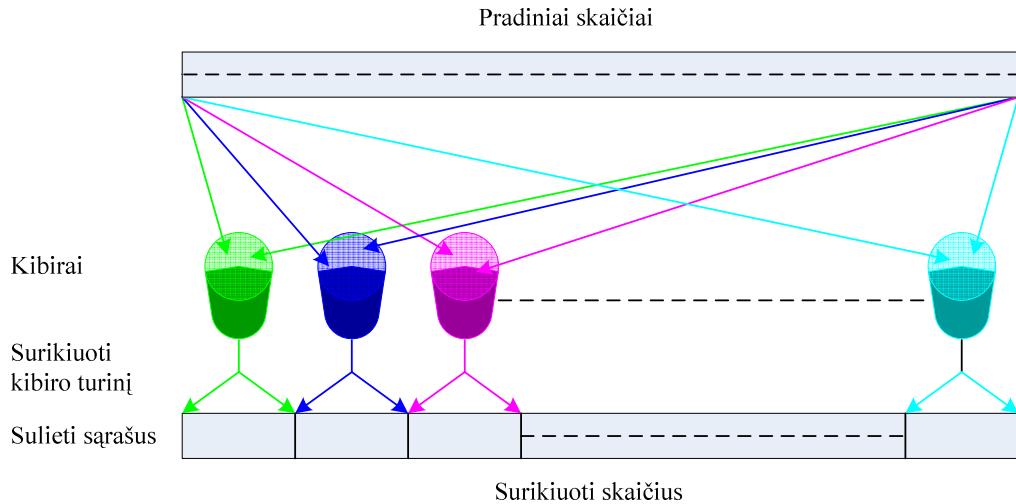


6.6 pav. Paveikslėlio koordinatinis skaidymas [5]

6.4 Skaldyk ir valdyk strategijos taikymai

6.4.1 Rūšiavimas kibirais (bucket)

Daugelis nuoseklių rūšiavimo (rikiavimas) algoritmų yra taip vadinami „palygink ir sukeisk“ pavidalo (compare and exchange) algoritmai. Šiame skyriuje nagrinėsime rūšiavimo „kibirais“ algoritmą, kuris nepriklauso minėtajai klasei, tačiau pasižymi geru skaidomumu.

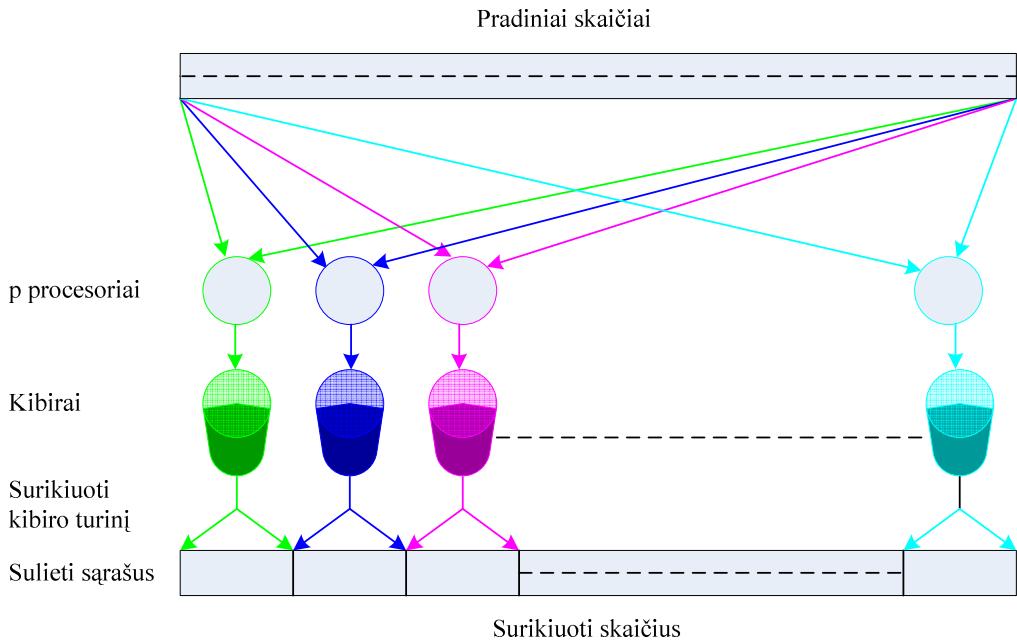


6.7 pav. Rūšiavimas kibirais [5]

Rūšiavimo algoritmo esmė, skaičių aibės suskirstymas į klasės – „kibirus“, remiantis paprastu požymiu, taip, kad surūšiuotame saraše vienos klasės skaičiai būtų šalia. Vėliau vienos klasės elementai gali būt toliau (rekursyviai) skirstomi arba rūšiuojami, naudojant kokį nors nuoseklų rūšiavimo metodą, pavyzdžiui *quicksort*. Skirstymas gali remtis, pavyzdžiui, vyriausiaisiais skaičiaus bitais. Rūšiavimas kibirais efektyvus tik tada, kada skaičiai yra tolygiai pasiskirstę minėtose klasėse. Pavyzdžiui, metodo efektyvumas yra $O(n)$, jeigu rūšiuojami visi skaičiai iš duotojo intervalo. Bendru atveju, jeigu žinoma, kad n skaičių tolygiai išsidėstę intervale I, \dots, m , algoritmo sudėtingumas $O(n \log(n/m))$.

6.4.2 Lygiagrečioji realizacija

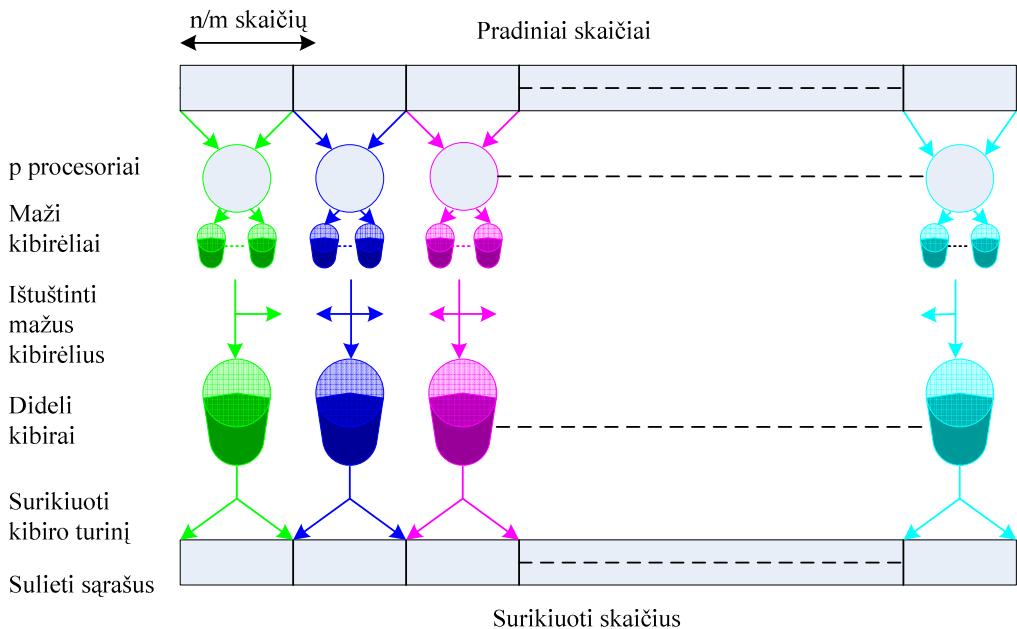
Paprasčiausiu atveju priskirkime vienam kibirui vieną procesorių.



6.8 pav. Rūšiavimo kibirais lygiagrečioji versija [5]

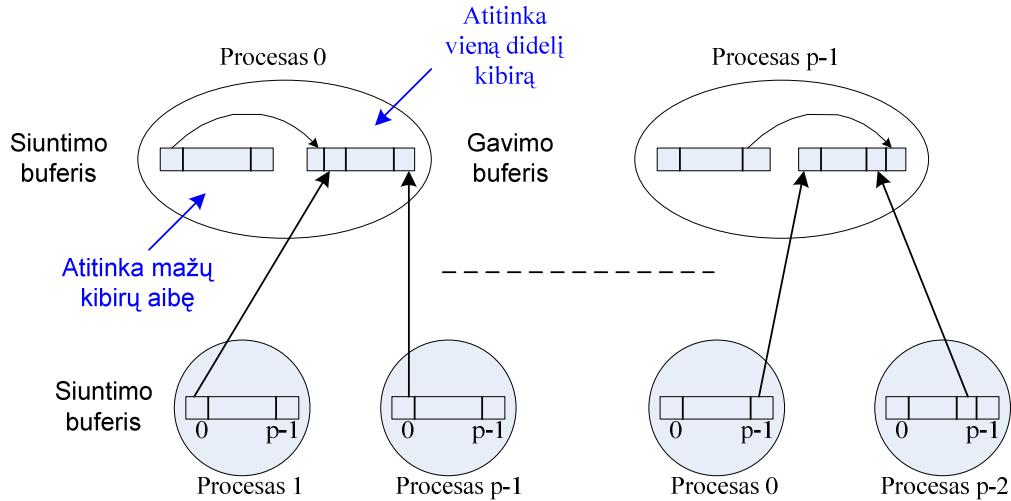
Kiekvienas procesorius „filtruoja“ pradinį skaičių sąrašą, išrinkdamas skirtus savo turimam „kibirui“.

Tobulesnė realizacija suskirsto skaičių sąrašą į m regionų, kiekvienam procesoriui po vieną regioną. Kiekvienas procesorius turi p „mažų“ kibirélių ir savo regionui priklausančius skaičius atskiria į reikiamus kibirélius. Galų gale maži kibireliai suliejami į galutinius kibirus.



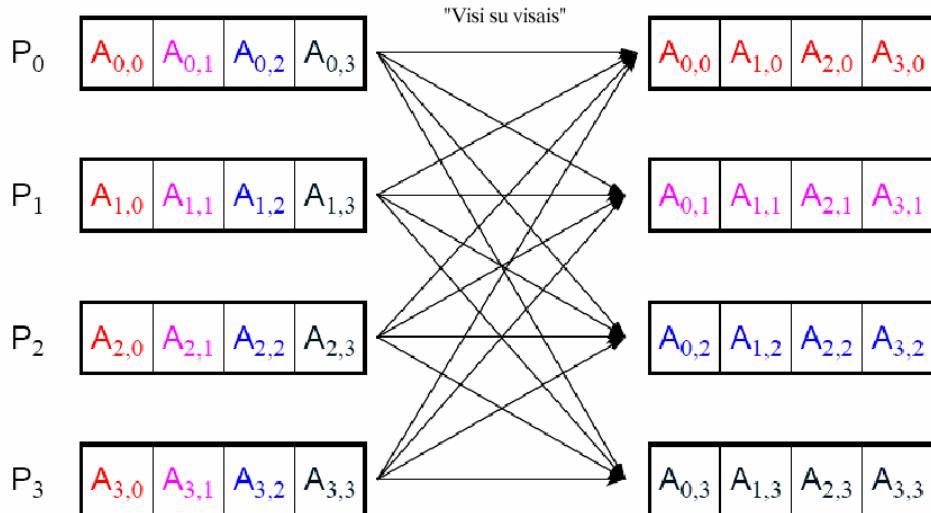
6.9 pav. Patobulinta rūšiavimo kibiraus lygiagrečioji versija [5]

Vykstant paskutiniąją algoritmo dalį prireikia atlikti „visi su visais“ procesorių komunikacijas. Žemiau pateikta tokios komunikacijos schema.



6.10 pav. „Visi – visiems“ duomenų paskleidimas [5]

Iš esmės ši procedūra atlieka matricos transponavimą, kaip parodyta schemae.

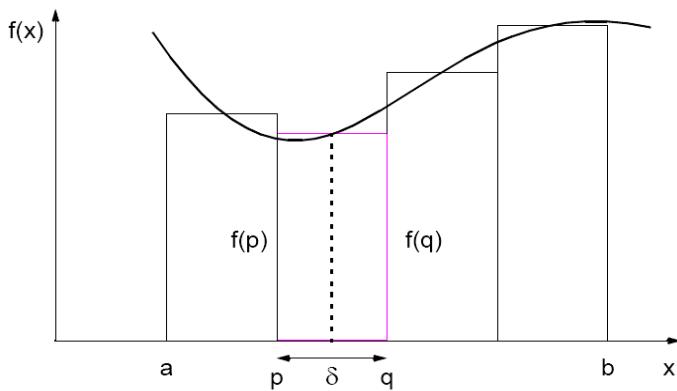


6.11 pav. „Visi – visiems“ duomenų paskleidimo atvaizdis masyve [5]

6.4.3 Skaitinis integravimas

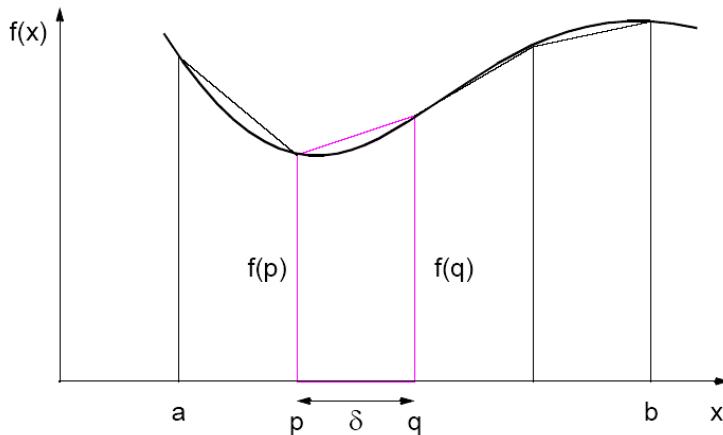
Funkcijos $f(x)$ integralo skaičiavimas gali būti pakeistas integralinių sumų skaičiavimu. Čia, skirtingai negu, rūšiavimo krūvomis uždavinyje, mes iš anksto galime nežinoti, kaip giliai reikia atlikti skaidymo procedūrą. Tai nustatoma įvertinant gautojo sprendinio tikslumą.

Paprasčiausiu atveju integralą aproksimuojame stačiakampių plotų suma, kaip pateikta brėžinyje (kvadratūrų metodas).



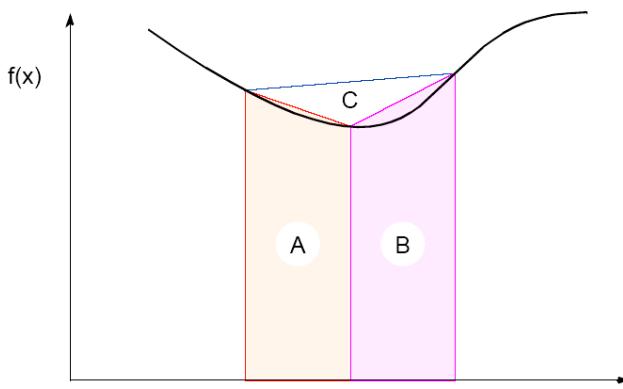
6.12 pav. Centrinis stačiakampių metodas integravimui [5]

Tikslesnius rezultatus gali duoti trapezijų metodas.



6.13 pav. Trapecijų metodas [5]

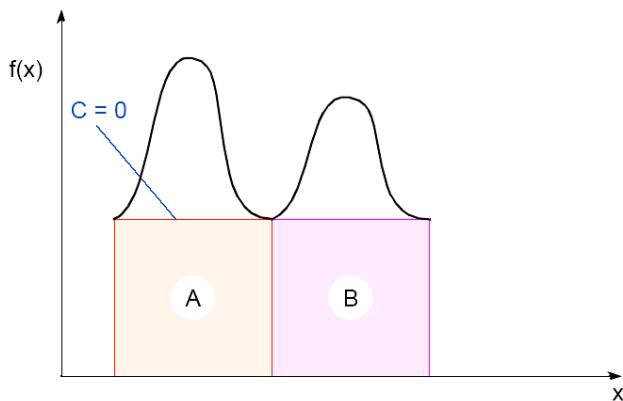
Adaptyvieji kvadratūriniai metodai, leidžia įvertinti aproksimacijos tikslumą lokaliai ir priimti sprendimą – testi tolimesnį intervalo skaidymą ar ne. Šie algoritmai tarsi prisitaiko prie kreivės formos.



6.14 pav. Adaptyviosios kvadratūros konstravimas [5]

Vienas iš metodų – apskaičiuoti plotus A, B, ir C, kaip pavaizduota brėžinyje. Skaičiavimus galime nutraukti, kai plotų $C/(A+B)$ santykis pakankamai mažas.

Nagrinėjamoji euristika gali kai kuriais atvejais nesuveikti, kaip parodyta žemiau pateiktoje schemaoje.



6.15 pav. Klaidingai nustatyta iteracijos baigtis [5]

6.4.4 Pavyzdys. Skaičiaus PI skaičiavimo programa

```
*****
pi_calc.cpp skaičiuoja pi reikšmę, sulyginant apskaičiuotą su tikraja (25
ženklu tikslumu) pi reikšme. Integruiama funkcija:  $f(x)=4/(1+x^2)$ .
Pagal: , 2001 K. Spry CSCI3145
*****  

#include <math.h> //include files
#include <iostream.h>
#include "mpi.h"
void printit(); //function prototypes
int main(int argc, char *argv[])
{
```

```

{
    double actual_pi = 3.141592653589793238462643; //sulyginimui vėliau
    int n, rank, num_proc, i;
    double temp_pi, calc_pi, int_size, part_sum, x;
    char response = 'y', respl = 'y';
    MPI::Init(argc, argv); //initiate MPI
    num_proc = MPI::COMM_WORLD.Get_size();
    rank = MPI::COMM_WORLD.Get_rank();
    if (rank == 0)
        printit(); /* Spausdina šaknininis procesas */
    while (response == 'y') {
        if (respl == 'y') {
            if (rank == 0) { /* Šaknininis procesas */
                cout << "_____" << endl;
                cout << "\nEnter the number of intervals: (0 will exit)" << endl;
                cin >> n;
            }
        } else n = 0;
        MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0); //broadcast n
        if (n==0) break; //ar išeiti
        else {
            int_size = 1.0 / (double) n; //skaičiuoti intervalo dydi
            part_sum = 0.0;
            for (i = rank + 1; i <= n; i += num_proc) { //dalinės sumos
                x = int_size * ((double)i - 0.5);
                part_sum += (4.0 / (1.0 + x*x));
            }
            temp_pi = int_size * part_sum;
            // Surenka visas dalines sumas, skaičiuoja pi
            MPI::COMM_WORLD.Reduce(&temp_pi,&calc_pi,
                                   1, MPI::DOUBLE, MPI::SUM, 0);
            if (rank == 0) { /* Šaknininis proc.*/
                cout << "pi is approximately " << calc_pi
                    << ". Error is " << fabs(calc_pi - actual_pi)
                    << endl
                    << "_____" << endl;
            }
        } //end else
        if (rank == 0) { /*Šaknininis*/

```

```

    cout << "\nCompute with new intervals? (y/n)" << endl; cin >> respl;
}
//end while
MPI::Finalize(); // užbaigtis MPI
return 0;
} //end main
//functions
void printit()
{
cout << "\n*****\n"
<< "Pi calculator!" << endl
<< "*****\n";
}//end printit

```

6.4.5 N-kūnų problema

Reikia nustatyti kūnų, kurie vienas kitą veikia gravitacijos jėga, greičius bei pozicijas erdvėje, naudojant Niutono fizikos dėsnius. Pateikiame lygtis.

Traukos jėga tarp dviejų kūnų turinčius masę m_a and m_b yra lygi

$$F = \frac{Gm_a m_b}{r^2}$$

Čia - G yra gravitacijos konstanta, o r – atstumas tarp kūnų. Remiantis antruoju Niutono dėsniu, traukos jėga nusakoma formule:

$$F = ma,$$

kur m – kūno masė, F – kūna veikianti traukos jėgą, o a – jėgos sukeliamas pagreitis.

Laiko momentu t kūną, kurio masė m , veiks traukos jėga:

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}.$$

Sekančiu laiko momentu $t+1$ greitis bus lygus:

$$v^{t+1} = v^t + \frac{F\Delta t}{m},$$

kur v_t yra greitis laiko momentu t .

Praėjus laiko intervalui Δt , pozicijos pasikeitimą nusakomas:

$$x^{t+1} - x^t = v\Delta t$$

kur, x^t yra kūno pozicija laiko momentu t .

Kūnui pasislinkus pasikeičia ir jį veikianti jėga, kuri turi būti perskaičiuota. Taigi – iteracijos tesiamos.

Iš esmės, N -kūnų problemos skaičiavimai nusakomi šiuo kodo fragmentu.

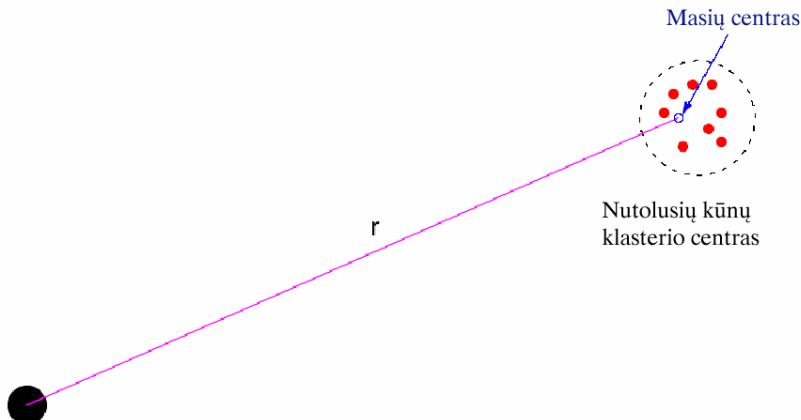
```

for (t = 0; t < tmax; t++)           /* kiekvienam laiko vienetui*/
    for (i = 0; i < N; i++) {        /* kiekvienam kūnui */
        F = Force_routine(i); /* Apskaičiuoti kūną veikiančią jėgą */
        v[i]new = v[i] + F * dt / m;
        /* rasti naują greitį */
        x[i]new = x[i] + v[i]new * dt; /* bei naują poziciją */
    }
    for (i = 0; i < nmax; i++) {      /* kiekvienam kūnui */
        x[i] = x[i]new;                /* atnaujinti greitį ir poziciją */
        v[i] = v[i]new;
    }
}

```

Nuoseklaus algoritmo sudėtingumas yra $O(N^2)$, kadangi kiekviename kūne iš N yra veikiamas kiekvienu iš likusių $N - 1$. Tai reiškia, jog duotasis algoritmas praktiškai nepritaikomas daugeliui iš įdomiausių N -kūnų problemų, kai kūnų skaičius didelis.

Algoritmo sudėtingumas gali būti žymiai sumažintas, atsižvelgiant į tai, jog iš didelio nuotolio greta esančių kūnų klasteris galėtų būti traktuojamas kaip jį sudarančių masių suma klasterio centre.



6.16 pav. Klasterizavimas

Barnes-Hut algoritmas.

Tam, kad efektyviai aptikti klasterius bei jais manipoliuoti gali būti naudojamas Barnes'o-Hut'o algoritmas. Tegu turime kubą, kurio viduje yra visi kūnai. Pradžioje kubas suskaidomas į 8-nis mažesnius kubus. Tada:

1. Jeigu kubo viduje nėra nė vieno kūno, kubas (sritis) yra pašalinama iš tolesnio nagrinėjimo.
2. Jeigu kubas turi tik vieną kūną, jis isimenamas, bet toliau nenagrinėjamas.
3. Jeigu kubui priklauso daugiau negu vienas kūnas, jis toliau rekursyviai skaidomas į 8-nias dalis tol, kol bus teisinga 1-ji arba 2-ji sąlyga.

Pateiktoji procedūra sukuria aštuntainį (*octtree*) medį, tiksliau medį, kuriame iš kiekvienos viršūnės išeina daugiausią 8-nios briaunos. Medžio lapai nusako kiekvieną kūną.

Po to, kai medis sukurtas, kiekvienoje medžio viršūnėje apskaičiuojami bei išsaugomi viršūnę atitinkančio subkubo suminė masė bei masių centras.

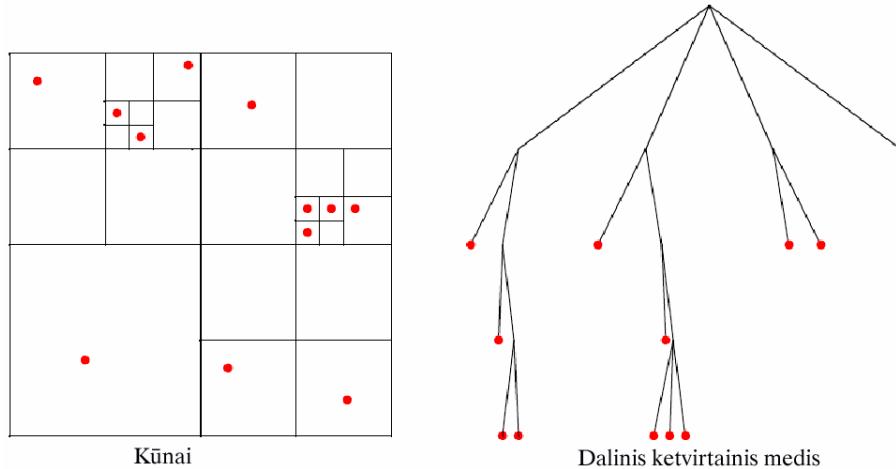
Sukonstravus tokį medį, kiekvieną viršūnę veikianti jėga randama, peržiūrint medį nuo šaknies link lapų ir sostojant, radus viršūnę, kuriai gali būti pritaikyta klasterizavimo aproksimacija, t.y., kai teisinga sąlyga:

$$r \geq \frac{d}{\theta},$$

kur tipiškai (*opening*) kampus $\theta < 1$.

Tokio medžio sukūrimo sudėtingumas yra $O(n \log n)$, ir tiek užtrunka apskaičiuoti visas jėgas, taigi bendras šio metodo sudėtingumas yra $O(n \log n)$.

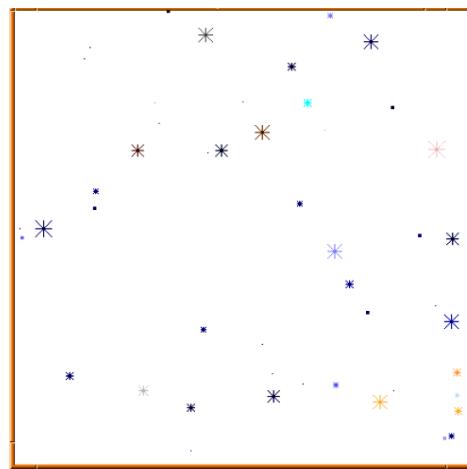
Balansavimas. Sprendžiant n-kūnų uždavinį dėl klasterizavimosi, medis gali būti blogai išbalansuotas, taigi atsiranda procesų apkrovos balansavimo problema. Ji gali būti sprendžiama kas tam tikrą laiko tarpą perskaičiavus procesorių apkrovas. Yra įvairių balansavimo metodų, čia paminėsime taip vadinamąjį ortogonalios rekursyviosios bisekcijos (*Orthogonal Recursive Bisection*) algoritmą, kuris taip pat gali būti išlygiagretinamas, naudojantis "skaldyk ir valdyk" principu. Pateikiame brėžinį dvimačiam atvejui.



6.17 pav. Ortogonaliai rekursyvijoji bisekcija 2D atveju [5]

Algoritmas veikia taip. Pradžioje randame vertikalią liniją (pjūvį), kuri suskaido pradinę sritį į dvi su vienodu kūnų skaičiumi. Toliau, kiekvienai daliai atliekame horizontalų „pjūvį“ su vienodu kūnų skaičiumi. Prosesą tęsiame tol, kol pasieksime reikiama kūnų skaičių keikviename stačiakampyne.

Vieno N-kūnų problemos modeliavimo rezultatai pateikti paveikslėlyje.

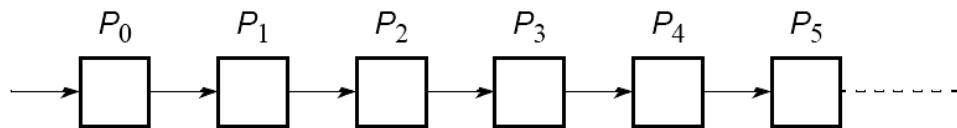


6.18 pav. Modeliavimo rezultatai

7 Konvejeriniai skaičiavimai

7.1 Konvejerinė technika

Naudojant konvejerinę techniką, skaičiavimai išskirstomi į atskiras dalis, apdorojamas nuosekliai viena po kitos kaip parodyta paveikslėlyje.* Kiekvienu metu atskirą dalį vykdo atskiras procesas, kuris dažnai vadinamas konverio etapu (grandimi). Iš esmės - konvejeris įgyvendina funkcinę programos dekompoziciją .



7.1 pav. Konvejeriniai procesai.

Pavyzdys. Tegu turime ciklą, sudedantį visus masyvo a elementus.

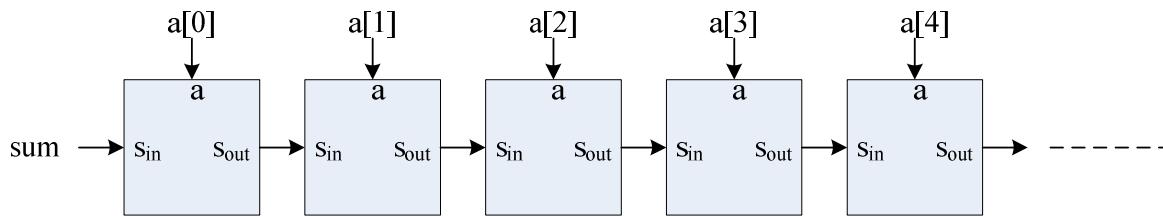
```
for (i= 0; i< n; i++)
    sum = sum + a[i];
```

Ciklas gali būti išskleistas tokiu būdu:

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
```

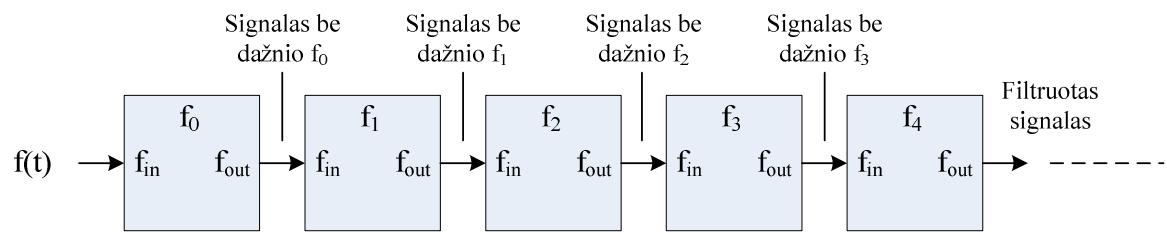
Kiekvieną sumavimą galėtų atitikti viena konvejerio grandis, kaip parodyta paveikslėlyje, kuri kaip iėjimą gautų akomuliujamą sumą bei vieną masyvo elementą, o išėjime perduotų naujai gautą sumą.

* Skyriuje naudojama [5] medžiaga



7.2 pav. Sumavimo ciklo išskleidimas konvejeriu

Kitas pavyzdys – dažnių filtras. Jo paskirtis,- pašalinti dažnius f_1, f_2, \dots, f_n iš skaitmeninio signalo $f(t)$. Signalo duomenys patenka į filtrą iš kairės, kaip parodyta paveikslėlyje, o kiekvienas etapas atsakingas už konkretaus dažnio pašalinimą.

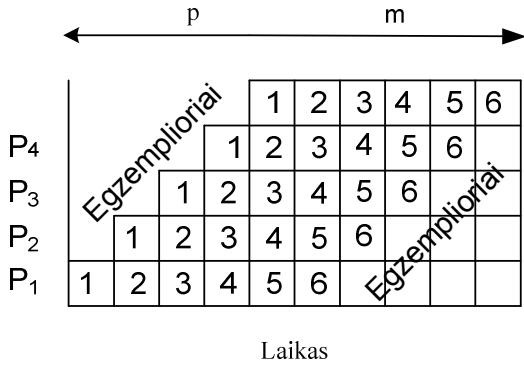


7.3 pav. Spektrinių dažnių filtras.

Aišku, mus domina tokie konvejerio panaudojimo atvejai, kurie įgalina programą įvykdysti sparčiau. Išskirime tris pagrindines situacijas.

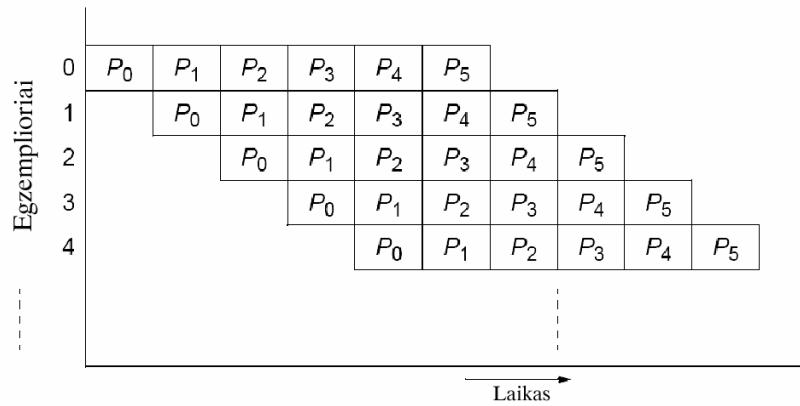
1. Kai reikia vykdyti daugiau negu vieną tos pačios problemos egzempliorių.
2. Kai turi būti apdorojama duomenų serija, kiekviena reikalaujanti daugelio operacijų.
3. Kai informacija, reikalinga sekančio proceso startavimui, gali būti perduodama pilnai neužbaigus ankstesniojo proceso darbo.

Pirmasis (1) atvejis dažnai sutinkamas kompiuterių aparatinės dalies realizacijose. Taip pat dažnas, sprendžiant simuliacines problemas, kurioms būdingas pakartotinis programos vykdymas su daugeliu parametru. Šio pavidalo konverejį galima iliustruoti erdvės-laiko diagrama, pateikta paveikslėlyje. Šioje diagramoje laikoma, kad kiekvienas procesas sugaišta tiek pat laiko, kad užbaigtį savo darbą. Kiekvienas laiko periodas yra vienas vienas konvejerio ciklas. Šiuo atveju kiekvienas duomenų egzempliorius reikalauja šešių nuoseklių procesų P0, P1, ..., P5 darbo. Po to, kai apdorojami penki pirmieji duomenų egzemploriai konvejeris yra pilnai užpildomas, t.y., visos konvejerio grandys yra pilnai apkraunamos.



7.4 pav. Konvejerio erdvės – laiko diagrama [5]

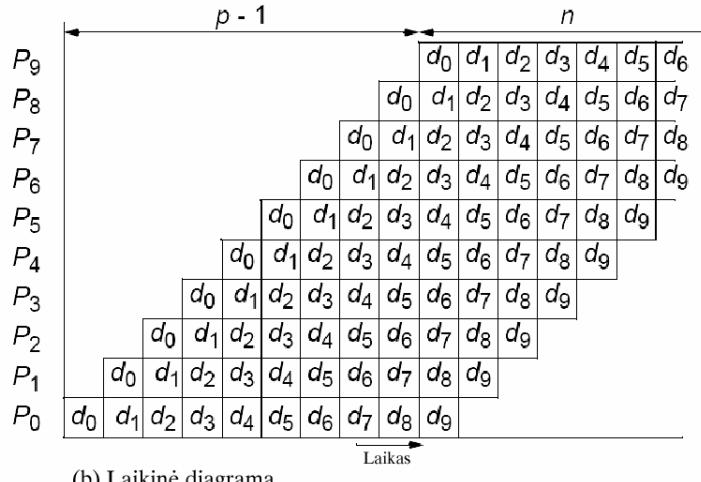
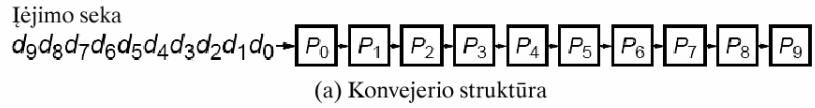
Alternatyvioji diagramma pateikta žemiau (5 procesams), kurioje užduotys išdėstomos vertikalioje ašyje.



7.5 pav. Alternatyvioji erdvės-laikinė diagrama [5]

Jeigu turime konvejerį iš p procesų, vykdantį m užduočių, tam kad įvykdyti visas užduotis proreiks $m+p-1$ ciklų arba vidutiniškai $(m+p-1)/m$ ciklų kiekvienam egzemplioriui. Taigi, didėjant m , ciklų skaičius vienam egzemplioriui artės prie 1.

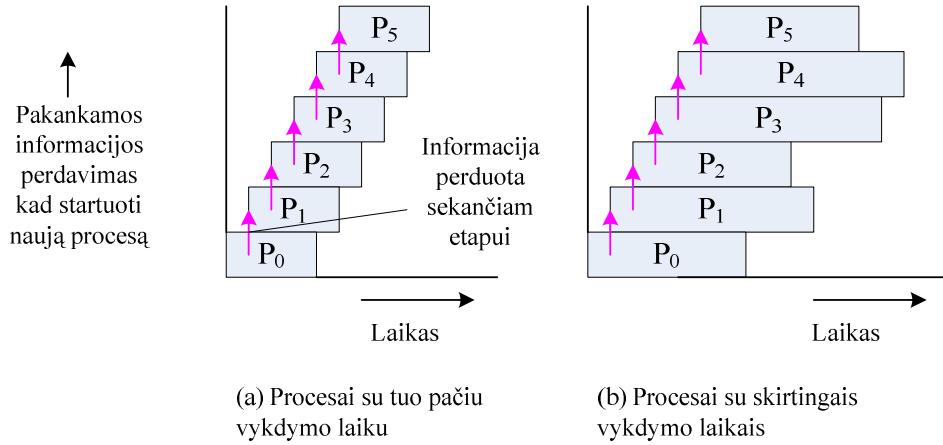
Antroji (2) situacija, kuri aprašo serijų apdorojima konvejeryje, dažnai sutinkama aritmetiniuose skaičiavimuose, pavyzdžiu, dauginant matricos elementus, kurie perduodami konvejeriu serijomis, kaip parodyta paveikslėlyje.



(b) Laikinė diagrama

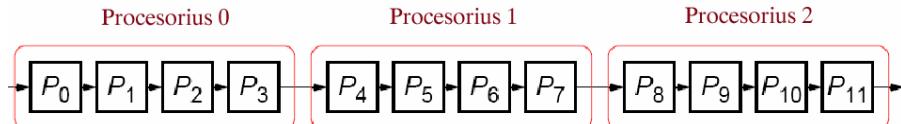
7.6 pav. Konvejeris, apdorojantis 10 elementų.

Dažnai sutinkama (3) situacija, kurioje tėra tik vienas vykdomas problemos egzempliorius, bet kiekvienas procesas gali perduoti informaciją sekančiam procesui, dar nebaigęs darbo, kaip parodyta paveikslėlyje.



7.7 pav. Trečiojo tipo laikinė-erdvės diagrama

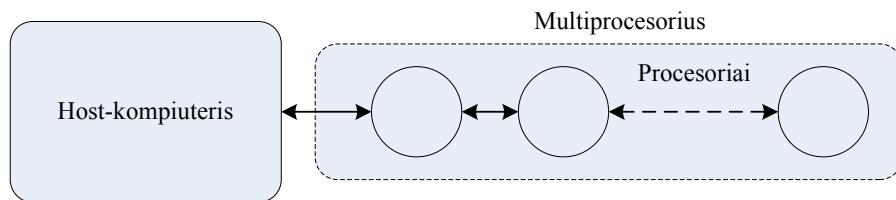
Jeigu konvejerio pakopų skaičius viršija procesorių skaičių, tai kelias pakopas galima apjungti į vieną, kaip parodyta brėžinyje. Aišku, šios pakopos vieno proceso rėmuose vykdomos nuosekliai.



7.8 pav. Prosesų atvaizdis į procesorius

7.2 Kompiuterių platformos konvejeriniams skaičiavimams.

Esminė konvejerio realizacijos ypatybė yra gebėjimas efektyviai perduoti pranešimus tarp dviejų gretimų konvejerio procesų. Tai, nulemia natūralią pranešimų tinklo konfigūraciją – tiesinį (žiedinį) tinklą, su kuriuo sujungiamas centrinis (host) kompiuteris kaip parodyta paveikslėlyje. Aišku, tinklas ar hiperkubas, į kuriuos puikiai gali būti įterpiamas tiesinis tinklas, taip pat gali būti sėkmingai naudojami.



7.9 pav. Daugelio procesorių sistema sujungta tiesišku tinklu

Specializuotos pranešimų perdavimo sistemos, pavyzdžiu transputeriai, sukurti 1980-siais, ar kitos sistemos pasižymintys, instrukcijų lygmenys lygiagretumu, gali būti sujungiamos į žiedinį tinklą, užtikrinant efektyvų “smulkaus grūdo” konvejerinių programų veikimą.

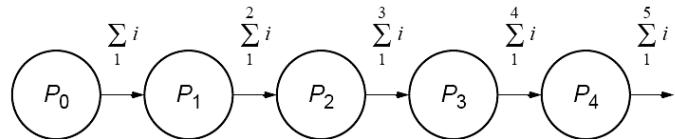
Taigi, Ethernet tipo tinklu susietas kompiuterių klasteris nėra pati tinkamiausia platforma konvejeriniams skaičiavimams, kadangi efektyvus vienalaikis pranešimų perdavimas, tarp kaimyninių mazgų tokiane tinkle gali būti problematiškas, nors nesiblokuojančių send(), receive() funkcijų panaudojimas galėtų suteikti tam tikrą laikiną lankstumą. O klasteris su greitaeigiais tiesioginiais tinkliniai sujungimais gali būti tinkamas.

7.3 Konvejerinių programų pavyzdžiai

7.3.1 Skaičių sumavimas

Pirmame pavyzdje nagrinėsime skaičių sumos radimo problemą. Sprendimas tinka ir kitoms asociatyviosioms operacijoms su skaičiais. Konvejerinis sprendinys susideda iš procesų,

kurie, sudėdami skaičius, kaupia sumą kaip parodyta paveikslėlyje. Kiekvienas procesas saugo po vieną sudedamą skaičių.



7.10 pav. Konvejerinis sumavimas

Bazinis proceso P_i kodas.

```

recv(&accumulation, Pi-1);
accumulation = accumulation + number;
send(&accumulation, Pi+1);
  
```

Išskyrus 0-ąjį procesą, kuris vykdo:

```
send(&number, P1);
```

O paskutinis procesas vykdo kodą:

```

recv(&number, Pn-2);
accumulation = accumulation + number;
  
```

Taigi, SPMD programa atrodys taip:

```

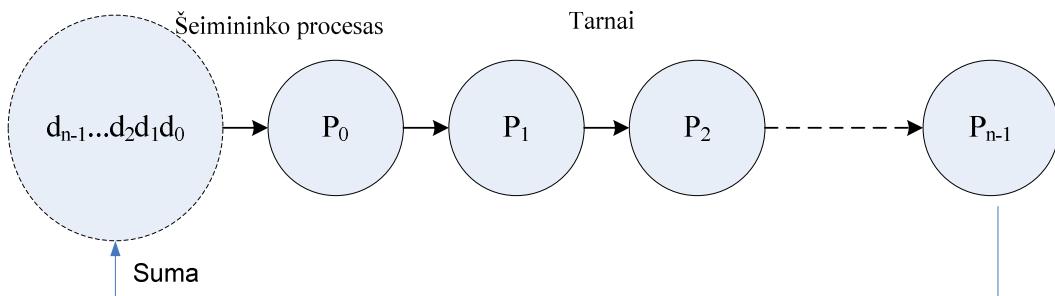
if (process > 0) {
    recv(&accumulation, Pi-1);
    accumulation = accumulation + number;
}
if (process < n-1)
    send(&accumulation, P i+1);
  
```

Galutinis rezultatas bus paskutiniame procese.

Vietoj sumos galima panaudoti ir kitas operacijas. Tarkime, skaičiuojant x^5 , galime atlikti 5 daugybas. Taigi konvejeris susidės iš penkių dauginančiųjų procesų.

Dažnai operacijų rezultatą, perduodamą konvejeriu, reikia grąžinti atgal į centrinį procesą.

Tuo atveju būtų labai žiedinė tinklo struktūra, kuri pateikta paveikslėlyje.



7.11 pav. Konvejerinis skaičių sumavimas, naudojant centrinių procesų ir žiedo pavidalo tinklą.

Analizė. Nagrinėjamas konvejeris yra (1) tipo. Kitaip tariant, konvejeris gali būti efektyvus tik tuo atveju, jeigu turime daugiau negu vieną problemos egzempliorių (daugiau negu vieną skaičių aibę, kuri turi būti susumuota). Analizuojant konvejerio darbo trukmę, reikštų išreikšti programos darbo trukmę konvejerio ciklais ir, nustačius vieno ciklo trukmę, rasti vykdymo laiką. Bendra programos darbo trukmę:

$$T_{total} = (T_{comp} + T_{comm})(m+p-1),$$

kur m – problemos egzempliorių skaičius, o p – procesų skaičius. Skaičiavimo ir komunikacijos laikas yra atitinkamai T_{comp} ir T_{comm} . Vidutinis skaičiavimų vienam problemos egzemplioriui laikas lygus:

$$Ta = T_{total}/m$$

Jeigu turime **vieną problemos egzempliorių**, apdorojamą kiekviename etape, skaičiavimų laikas (įskaitant, kad $T_{comp}=1$)

$$T_{total} = (2(T_{startup} + T_{data}) + 1)*n,$$

t.y., sudëtingumas yra eiles $O(n)$.

Apdorojant **m grupių iš n skaičių**, konvejerio ciklo ilgis išlieka tas pats, bet po $m+n-1$ ciklų turėsime:

$$T_{total} = (2(T_{startup} + T_{data}) + 1)(m+n-1).$$

Dideliems m vidutinis vykdymo laikas bus apytiksliai lygus

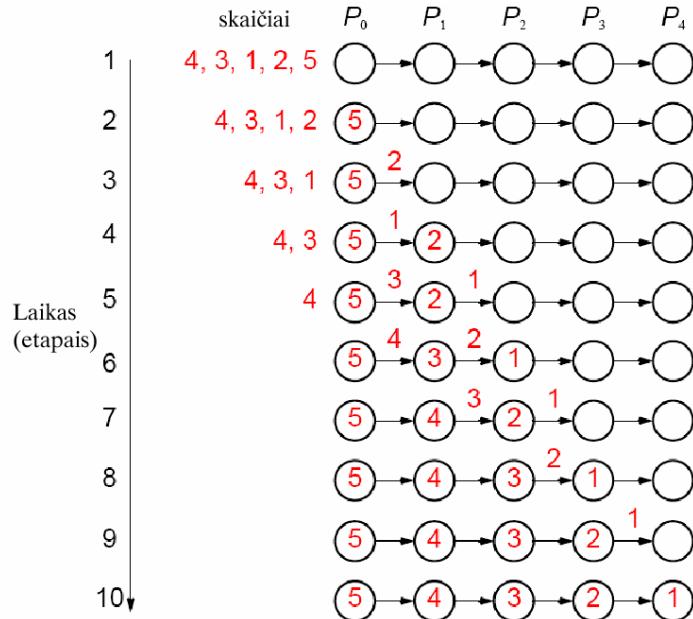
$$Ta = 2(T_{startup} + T_{data}) + 1,$$

t.y., lygus vienam konvejerio ciklui.

7.3.2 Skaičių rikiavimas.

Priminsime, kad skaičių rikiavimas – tai skaičių sekos sutvarkymas didėjančia (mažėjančia) tvarka. Konvejerinis rikiavimo sprendimas susideda iš proceso P_0 , kuris gauna skaičių seką, įsimena iki tol gautą didžiausią skaičių ir perduoda visus skaičius, mažesnius negu įsimintasis

tolesniems procesams. Tuo atveju, jei gautas skaičius didesnis negu įsimintasis, šis skaičius įsimenamas, o ankstesnis perduodamas tollyn. Kiekvienas tolesnis procesas Konvejeryje įgyvendina šį algoritmą, įsimindamas iki tol gautą didžiausią skaičių.



7.12 pav. Žingsniai, atliekant 5 elementų lygiagretujį rikiavimą įterpimais [5]

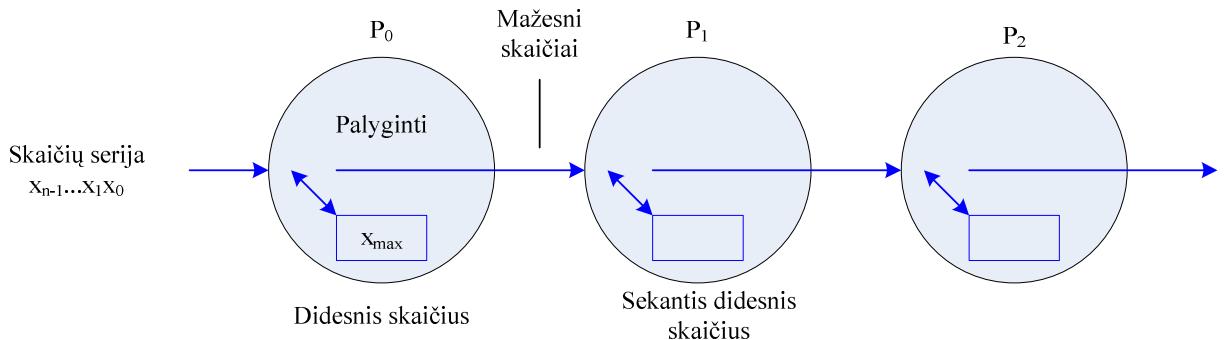
Kai visis skaičiai bus apdoroti, procesas P_0 turės didžiausią skaičių P_1 – didžiausią iš likusiųjų ir tt. Paskutinis procesas išsaugos mažiausią sekos elementą. Šis algoritmas yra lygiagreti rikiavimo įterpimais (*insertion sort*) versija. Iliustracija pateikta paveikslėlyje. Procesui P_i bazinį algoritmą galima nusakyti tokiu būdu.

```
recv(&number, Pi-1);
if (number > x) {
    send(&x, Pi+1);
    x = number;
} else send(&number, Pi+1);
```

Turint n skaičių, skaičių kiekis, kurį reikės priimti i-jam procesui nusakomas dydžiu $n - i$.

I priekį reikės pasiūsti $n - i - 1$ skaičių, kadangi vienas skaičius lieka.

Rikiuojantysis konvejeris yra pateiktas paveikslėlyje.



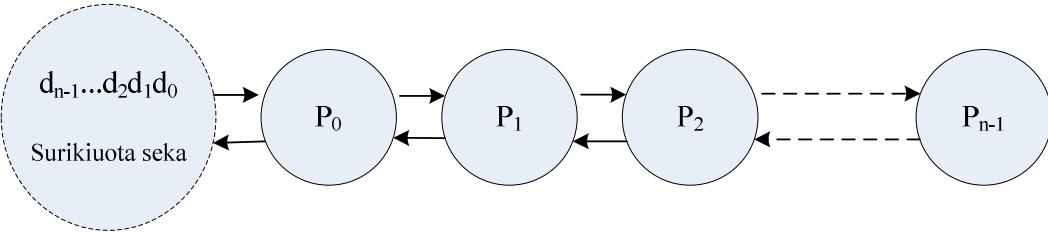
7.13 pav. Konvejeris lygiagrečiam rikiavimui įterpimais [5]

Naudojant dvikryptį pranešimų žiedą, rezultatą galima grąžinti pradiniam procesui.

Konvejeris yra pateiktas paveikslėlyje. Matyti, kad tarp pranešimų gavimo ir siuntimo vienintelė prasminga operacija, kurią gali atliliki procesas, yra dvių skaičių sulyginimas.

Mes galime padidinti operacijų skaičių vienam procesoriui paskiriant kelis procesus, t.y. apdorojant duomenis seriomis.

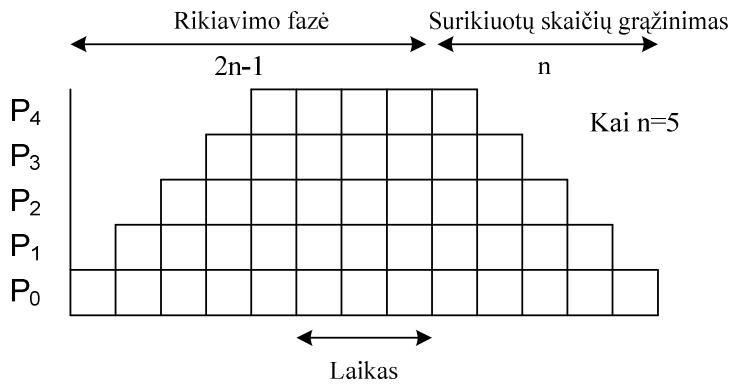
Šeimininko procesas



7.14 pav. Konvejeris, rūšiuojantis įterpimais, su rezultatų grąžinimu.

Rūšiavimo rezultatai gali būti gaunami naudojant žединę konfiguraciją arba dvikryptę tiesinę. Antrasis variantas pranašesnis, nes procesas gali grąžinti rezultatą iš kart po to, kai yra gautas paskutinysis skaičius.

Rūšiavimo įterpimais su rezultatu grąžinimu laikinė diagrama pateikta žemiau.



7.15 pav. Rūšiavimo įterpimais su rezultatų grąžinimu laikinė diagrama

7.3.3 Pirmių skaičių generavimas

Naudosime Eratosthenes'o rėčio algoritmą. Pradedant skaičiumi 2, generuojama didėjančiu natūrinių skaičių seka. Pirmasis skaičius laikomas pirmiu, todėl išsaugomas. Visi kiti, kartotiniai šiam pirmiam skaičiui – išbraukiami. Procesas kartojamas su kiekvienu likusiu pirmiu skaičiumi. Taigi visi likę neišbraukti skaičiai – ieškomi pirminiai.

Pastebėsime, kad algoritmą užtenka kartoti tol, kol randame pirmą skaičių \sqrt{n} .

Pavyzdys. Tegu turime skaičius nuo 2 iki 20.

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Išbraukus kartotinius 2, gausime

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

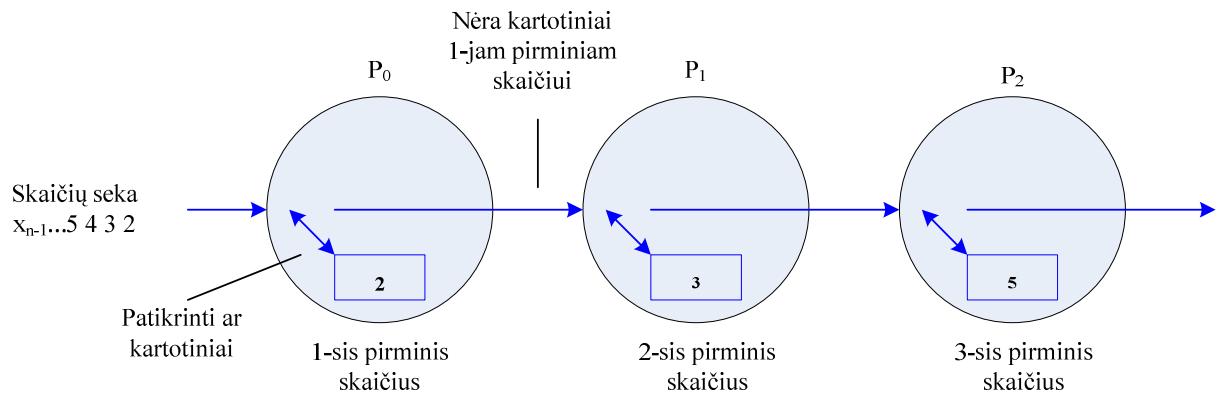
Sekantis pirmis skaičius – 3. Išbraukę kartotinius 3 gausime:

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Tolesni braukimai, sekos jau nepakeis. Taigi likę skaičiai – pirminiai.

Nuoseklusis algoritmas paprastai naudoja masyvą, saugantį išbraukimo požymį. Jo sudėtingumas yra eilės n^2 .

Lygiagrečiajame konvejeriniame algoritme, generuojama natūrinių skaičių seka patenka į konvejerį. Pirmoji grandis įsimina pirmąjį gautą skaičių kaip pirmąjį, ir persiunčia gretimajam procesui tik nelyginius skaičius. Taigi kiekvienas procesas atsakingas už jam priskirto pirmino skaičiaus kartotinių “išbraukimą”. Skirtingai negu nuosekliojoje versijoje tas pats skaičius negali būti “išbrauktas” keletą kartų.



7.16 pav. Konvejeris Eratosthenes'o rėčiui

Pi proceso kodas užrašomas taip:

```
recv(&x, Pi-1);
/* Kartoti kiekvienam skaičiui */
recv(&number, Pi-1);
if ((number % x) != 0)
send(&number, P i+1);
```

Kiekvienas procesas gaus skirtingą skaičių kiekį, todėl reikia naudoti skaičių sekos baigmės žymę.

```
recv(&x, Pi-1);
for (i = 0; i < n; i++) {
    recv(&number, Pi-1);
    if (number == terminator) break;
    (number % x) != 0) send(&number, P i+1);
}
```

7.3.4 Tiesinių lygčių sistemos sprendimas.

Tegu sistemos koeficientai yra viršutinio trikampio pavidalo.

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

.

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 = b_1$$

$$a_{0,0}x_0 = b_0$$

Čia dydžiai a bei b yra konstantos, o x – ieškomi nežinomieji. Tokio pavidalo sistemos sprendimas remiasi besikartojančiais atgaliniais keitiniais (*back substitution*). Iš paskutinės lygties gausime:

$$x_0 = \frac{b_0}{a_{0,0}}$$

Gautą reikšmę galime panaudoti skaičiuodami x1:

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

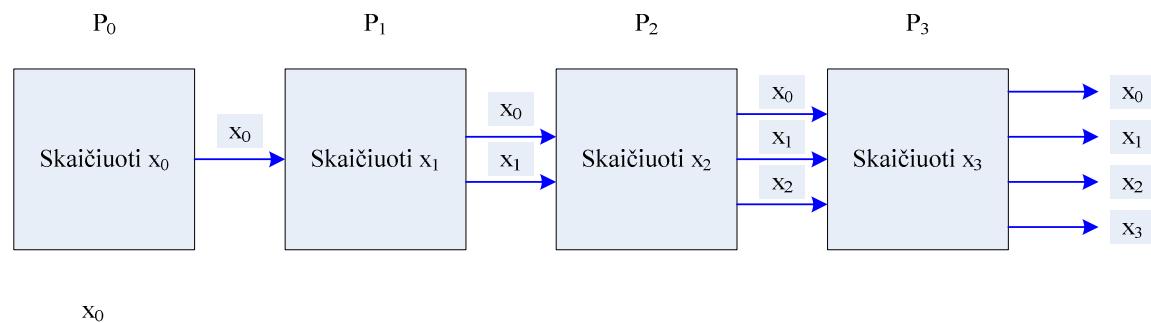
Abi (x1,x2) reikšmės gali būti panaudojamos skaičiuojant trečiąją.

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

Bendru atveju galime užrašyti:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j}x_j}{a_{i,i}}$$

Taigi, nesunku matyti, kaip sudaryti (3-jo tipo) konvejerį.



7.17 pav. Triištrijainės tiesinių lygčių sistemos sprendimas, naudojant konvejerį

Nuosekliosios versijos pseudo-kodas pateiktas žemiau.

`x[0] = b[0]/a[0][0]; /* Pradinis paruošimas */`

```

for (i = 1; i < n; i++) { /* Likusiems kintamiesiems */
    sum = 0;
    for (j = 0; j < i; j++)
        sum = sum + a[i][j]*x[j];
    x[i] = (b[i] - sum)/a[i][i];
}

```

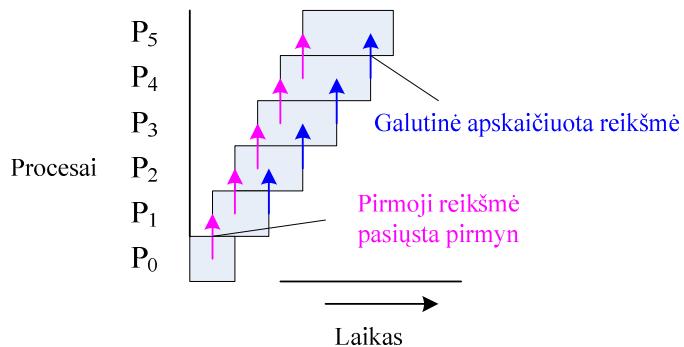
Lygiagrečiosios versijos pirminis kodas:

```

for (j = 0; j < i; j++) {
    recv(&x[j], Pi-1);
    send(&x[j], Pi+1);
}
sum = 0;
for (j = 0; j < i; j++)
    sum = sum + a[i][j]*x[j];
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], Pi+1);

```

Norint, kad procesas atliktų skaičiavimus, nebaigęs visų komunikacijų, reikia pertvarkyti ciklą, kad konvejerio struktūra būtų tokia, kaip parodyta paveikslėlyje.



7.18 pav. Konvejeris atvirkštiniam keitiniui

Galutinė programa procesui P_i :

```

sum = 0;
for (j = 0; j < i; j++) {
    recv(&x[j], Pi-1);
    send(&x[j], Pi+1);
    sum = sum + a[i][j]*x[j];
}
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], Pi+1);

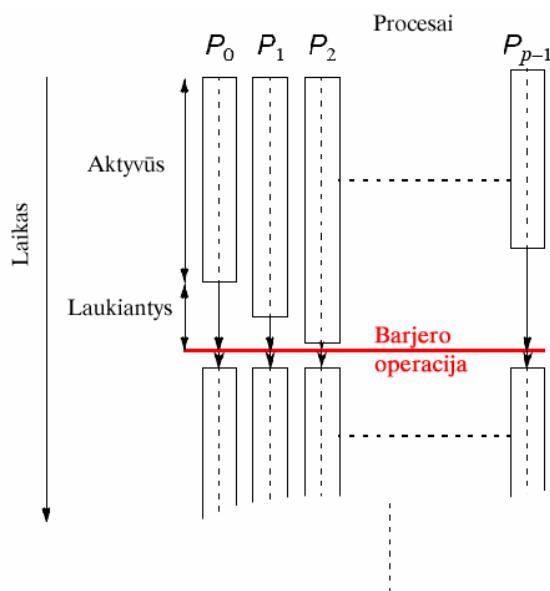
```

8 Synchroniniai skaičiavimai

Visiškai synchronizuotoje lygiagrečiojoje programoje visi procesai synchronizuojami reguliariuose taškuose.* Dažnai tokia programa operuoja reguliarios struktūros duomenimis (pavyzdžiu dvimačiais masyvais), ir kiekvienam duomenų elementui ar jų grupei atliekama ta pati (tos pačios) operacija (-os), žingsnis po žingsnio, panašiai kaip SIMD programose.

8.1 Barjeras

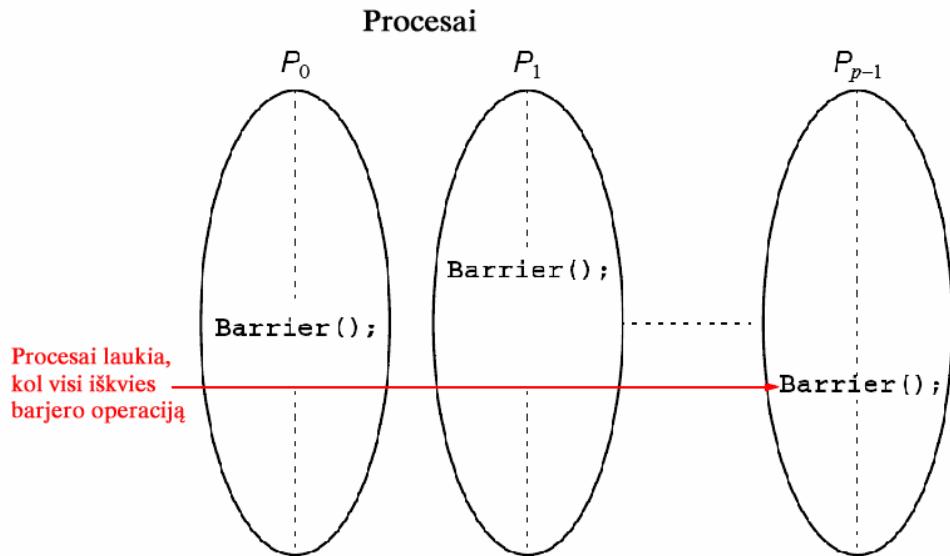
Tai vienas pagrindinių procesų synchronizavimo mechanizmų. Barjeras žymi proceso sostojimo tašką. Procesas gali pajudėti pirmyn tik tuo atveju, jeigu visi procesai (ar tam tikras numatytais procesų skaičius) pasiekė šį tašką.



8.1 pav. Prosesai pasiekiantys barjerą skirtingu metu [5]

Pranešimais grįstose lygiagrečiosiose sistemoje barjeras realizuotas bibliotekine funkcija.

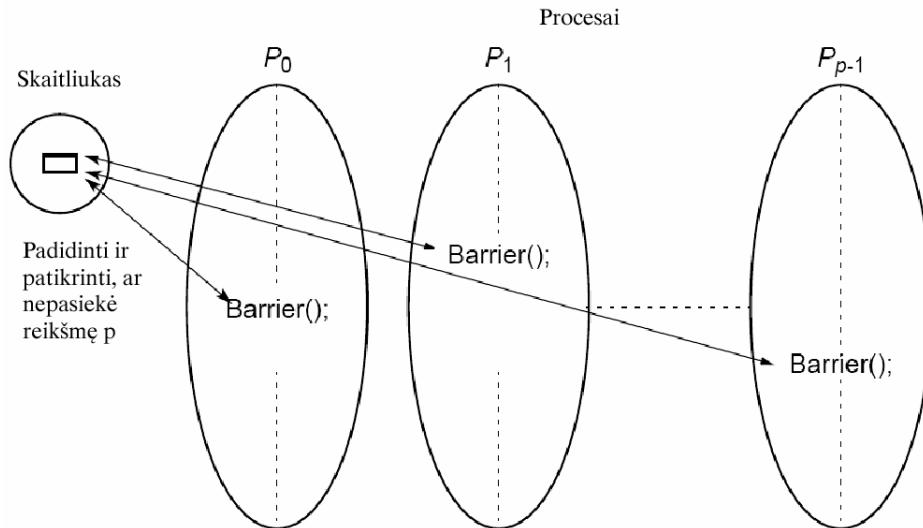
* Skyriuje naudojama [5] medžiaga



MPI sistemoje barjeras kviečiamas MPI_BARRIER() funkcija, kurios parametras – komunikatorius, nustatantis procesų grupę, dalyvaujančią barjere. Funkcijos kvietėjai blokuojami tol, kol kiekvienas grupės procesas iškvies MPI_BARRIER(), ir tik po to grįžtama.

Barjero įgyvendinimo būdai.

1. Centralizuota (tiesinė) barjero realizacija. Remiasi centralizuotu skaitliuku.



Kiekvienas procesas, įeinantis į barjera’ą padidina skaitliuką ir blokuojasi. Skaitliukui įgavus reikšmę, sutampančią su procesų skaičiumi, barjeras atlaisvinamas.

Tinkama realizacija turėtų atsižvelgti į galimą barjero pakartotinį panaudojamumą. T.y., reikėtų korektiškai apdoroti situaciją, kai į barjerą „ižengiama“ antrą kartą, o ne visi procesai spėjo palikti barjerą.

Dažnai skaitliukinis barjeras susideda iš dviejų fazių:

- atvykimo fazė, kai procesas atvyksta į barjerą ir nepalieka jo iki pirmają fazę išgyvendins visi procesai;
- išvykimo fazė, kurioje vis procesai atlaisvinami.

Dviejų fazių išgyvendinimas leidžia pakartotinio barjero panaudojimo scenarijų.

Realizacijos kodo pavyzdys:

Procesas – šeimininkas:

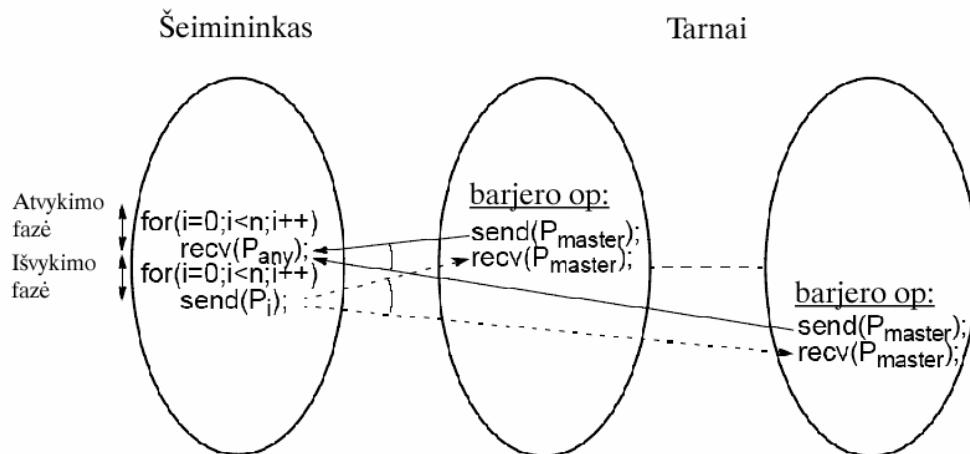
```
for (i = 0; i < n; i++)
    recv(Pany); /* Skaičiuoti tarnus, pasiekusius barjera */

for (i = 0; i < n; i++)
    /* Atlaisvinti tarnus */
    send(Pi);
```

Procesai – tarnai:

```
send(Pmaster);
recv(Pmaster);
```

Pranešimų sistemos barjero realizacija galėtų atrodyti taip:



8.4 pav. Barjero išgyvendinimas pranešimų sistemose

Akivaizdu, jog šios schemos išgyvendinimo sudėtingumas priklauso tiesiškai nuo procesų skaičiaus.

Barjero įgyvendinimas medžio pavidalo komunikacine struktūra.

Žymiai efektyvesnis: žingsnių skaičius $O(\log p)$.

Tegu turime 8 procesus, $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7$:

1 lygmuo: P_1 siunčia pranešimą procesui P_0 ; (kai P_1 pasiekia barjerą)

P_3 siunčia pranešimą procesui P_2 ; (kai P_3 pasiekia barjerą)

P_5 siunčia pranešimą procesui P_4 ; (kai P_5 pasiekia barjerą)

P_7 siunčia pranešimą procesui P_6 ; (kai P_7 pasiekia barjerą)

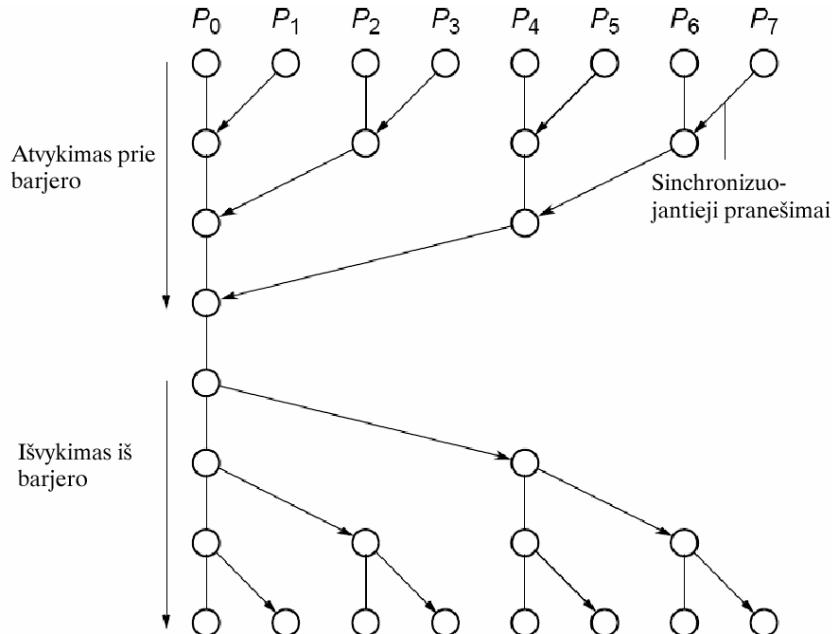
2 lygmuo: P_2 siunčia pranešimą procesui P_0 ; (P_2 ir P_3 pasiekia jų barjerus)

P_6 siunčia pranešimą procesui P_4 ; (P_6 ir P_7 pasiekia jų barjerus)

3 lygmuo: P_4 siunčia pranešimą procesui P_0 ; (P_4, P_5, P_6 , & P_7 pasiekia jų barjerus)

P_0 baigia atvykimo fazę, (kai P_0 pasiekia barjerą ir gauna pranešimą iš P_4).

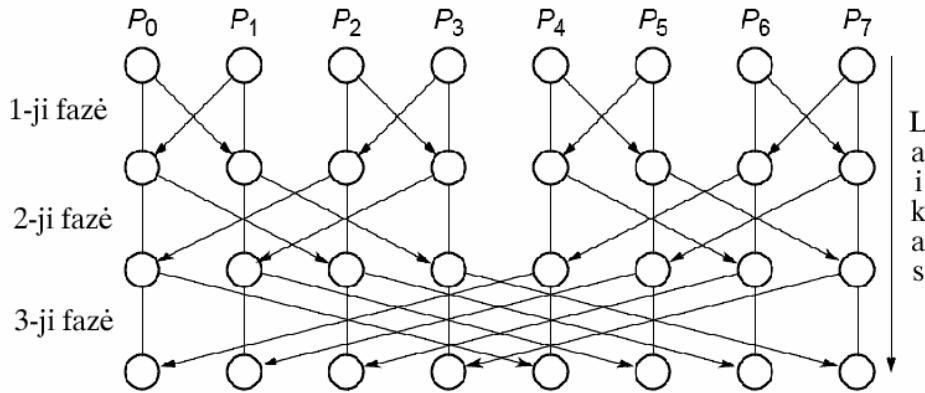
Procesų išlaisvinimas vyksta atvirkščia tvarka. Grafiškai:



8.5 pav. Medžio pavidalo barjeras

Kita gerai žinoma schema – drugelio schema (*butterfly*).

1-ji fazė	$P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$
2-ji fazė	$P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$
3-ji fazė	$P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$

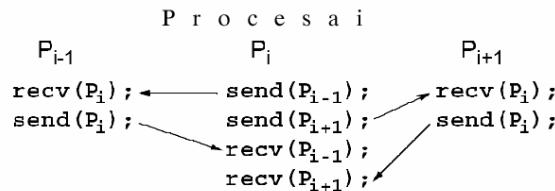


8.6 pav. Barjero įgyvendinimas "drugeliai"

8.2 Lokalioji synchronizacija

Dažnai sutinkama situacija, kada procesai turi būti synchronizuoti tik su nedideliu skaičiumi “kaimyninių” procesų.

Tarkime, procesas P_i turi būti synchronizuotas su procesais P_{i-1} ir P_{i+1} tam, kad apsikeisti duomenimis prieš visiems trims procesams pratęsiant darbą. Galima schema:



nėra ideali, kadangi procesas P_{i-1} bus synchronizuotas tik su P_i ir pratęs darbą, leidus procesui P_i . Atitinkamai procesas P_{i+1} synchronizuotas tik su P_i . Tačiau daugeliu atveju ši schema yra pakankama.

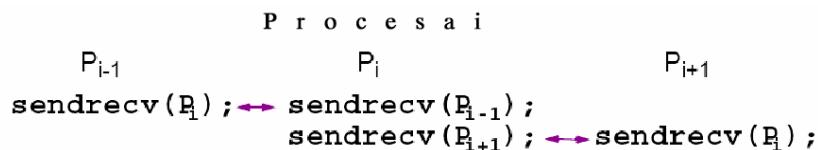
Aklavietė.

Gali pasitaikyti kai pora procesų siunčia / gauna pranešimus iš vienas kito. Pavyzdžiu, aklavietė atsiras tuo atveju, jeigu abu procesai vykdys operacija `send()` naudodami synchronines (blokuojančias) funkcijas. Nė viena iš šių funkcijų negrįš, kadangi lauks neįvyksiančios `receive()` operacijos.

Galimas sprendimas – vieną iš procesų įpareigoti pradžioje kvieсти duomenų priėmimo operaciją, o tik po to siuntimo.

Kitas pavyzdys. Jeigu turime tiesinį konvejerį, galime sutvarkyti procesus taip, kad lyginiai pradeda duomenų apsikeitimą siuntimo, o nelyginiai – priėmimo - operacija.

Kadangi duomenų apsikeitimas tarp dviejų procesų – gana dažnas reiškinys, pranešimų sistemos naudojamos specialios sendrecv() operacijos.



MPI parūpina operacijas **MPI_Sendrecv()** ir **MPI_Sendrecv_replace()**, kurios, aišku, turi dvigubai daugiau parametru negu įprastos.

8.3 Sinchronizuotų skaičiavimų klasifikacija

Išskiriami *visiškai* sinchronizuoti ir *lokaliai* sinchronizuoti skaičiavimai.

Visiškai sinchronizuotais laikysime tokius skaičiavimų, jeigu visi procesai dalyvaujantys skaičiavime turi būti sinchronizuoti.

Kitas variantas – lokaliai sinchronizuoti skaičiavimai – reikalauja tik logiškai artimų procesų sinchronizacijos.

8.3.1 Visiškai sinchronizuotų skaičiavimų pavyzdžiai

Duomenimis lygiagretūs skaičiavimai.

Šiuo atveju ta pati operacija atliekama su skirtiniais duomenų elementais vienu metu – lygiagrečiai.

Ypatybės:

- paprastas programavimas;
- geras programos plečiamumas (scaling);
- nemažai skaitiniu bei kitu problemų sprendimai gali būti užrašyti šiuo pavidalu.

Pavyzdys – konstantos sumavimas prie visų masyvo elementų.

Programos kodas.

```
for (i = 0; i < n; i++)
    a[i] = a[i] + k;
```

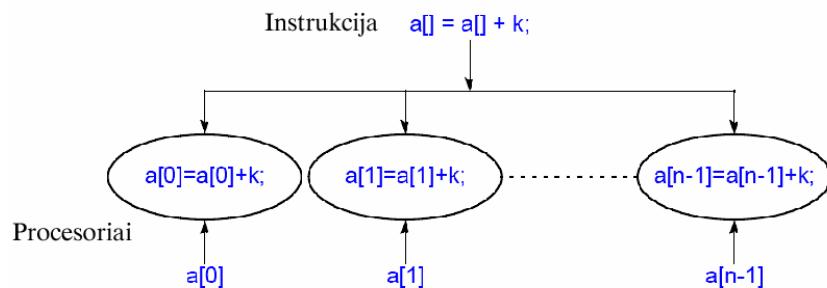
Sakinys:

```
a[i] = a[i] + k;
```

gali būti vykdomas lygiagrečiai. Svarbu, kad kiekvienas procesorius naudotų skirtinę indeksą

$i \ (0 < i \leq n)$.

Schematiškai:



8.7 pav. Duomenimis lygiagretūs skaičiavimai [5]

Užrašant lygiagretumą duomenimis, lygiagrečiosios programavimo kalbos naudoja **forall** konstrukciją.

Tarkime, sakinys

```
forall (i = 0; i < n; i++) {
    <kūnas>
}
```

reiškia, jog n egzempliorių ciklo kūno sakinių gali būti vykdomi vienu metu. Taigi, kiekvienas kūno egzempliorius turi nuosavą skaitliuko i reikšmę: 0,1, 2 ir t.t..

Kita pavyzdys. Tam, kad pridėti skaičių k prie kiekvieno masyvo a elemento, galėtume parašyti:

```
forall (i = 0; i < n; i++)
    a[i] = a[i] + k;
```

Išlygiagretinimo duomenimis techniką galime pritaikyti multikompiuteriams, ir aukščiau pateiktą sprendimą užrašyti taip:

```
i = myrank;
a[i] = a[i] + k;
barrier(mygroup);
```

kur myrank yra proceso rangas tarp 0 and n - 1.

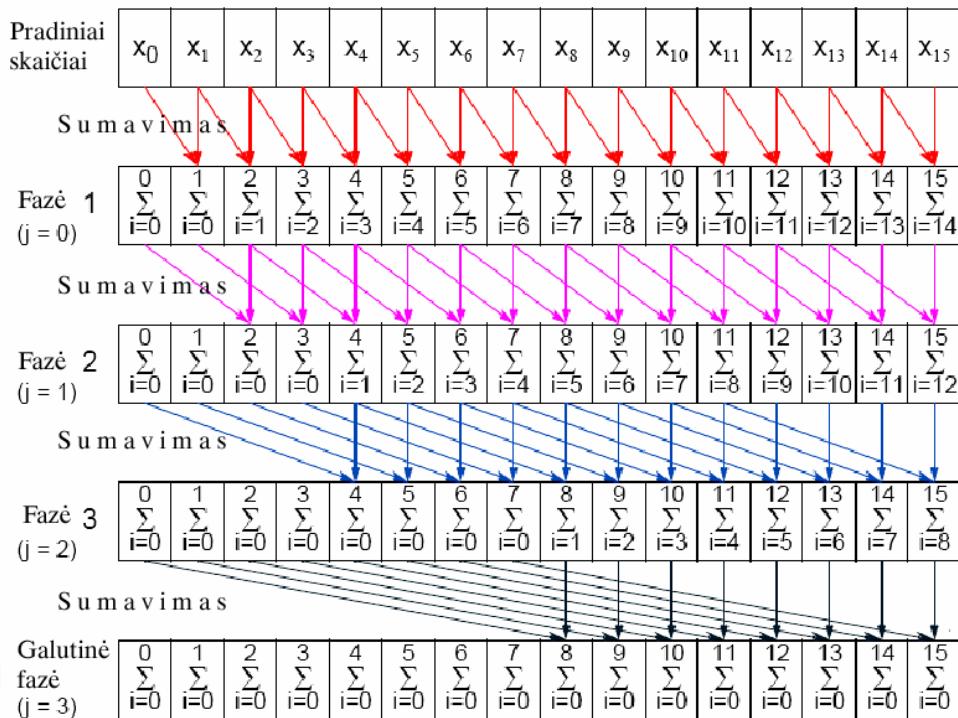
Prefiksinių sumų problema.

Tai vienas iš duomenimis lygiagretaus algoritmo pavyzdžių. Duotas skaičių sąrašas: x_0, \dots, x_{n-1} . Reikia rasti visas dalines sumas (t.y., $x_0 + x_1; x_0 + x_1 + x_2; x_0 + x_1 + x_2 + x_3; \dots$).

Vietoje sumavimo gali būti pateikta kita asociatyvi operacija.

Ši problema placiai išnagrinėta, konkretūs taikymai liečia tokias sritis kaip procesorių paskirstymas, duomenų spaudimas, rūšiavimas, polinomų skaičiavimas.

Žemiau pateikta komunikacinė schema iliustruoja 16 dėmenų atvejį.



8.8 pav. Prefiksinių sumų skaičiavimas

Pastebėsime, kad algoritmo sudėtingumas yra $O(\log n)$.

Žemiau pateiktas nuosekliosios ir lygiagrečiosios programos kodai.

Nuoseklusis kodas.

```
for (j = 0; j < log(n); j++)
    for (j = 2**j; i < n; i++)
        x[i] = x[i] + x[i-2];
```

Lygiagretusis kodas.

```
for (j = 0; j < log(n); j++)
    forall (i = 0; i < n; i++)
        if (i >= 2**j) x[i] = x[i] + x[i-2]
```

Sinchronizuotos iteracijos.

Nusakoma procesų aibe, kurie atlieka kiekvieną iteraciją sinchroniškai (vienu metu).

Naudojant forall

konstrukcija:

```
for (j = 0; j < n; j++) /* kiekvienai synch. iteracijai */
    forall (i = 0; i < N; i++) { /* N proc., kiekvienas naudoja */
```

```

        body(i);           /* specifinę i reikšmę */
    }

```

Naudodami pranešimų barjerą, turime kodą:

```

for (j = 0; j < n; j++) {           /* kiekvienai synchr.iteracijai */
    i = myrank;                   /* gauti naudojamą i reikšmę */
    body(i);
    barrier(mygroup);
}

```

Tiesinės lygčių sistemos sprendimas iteracijomis.

Tai dar vienas visiškai sinchronizuotų skaičiavimų pavyzdys. Tegu turime n lygčių su n nežinomujų.

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

.

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \dots + a_{2,n-1}x_{n-1} = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \dots + a_{1,n-1}x_{n-1} = b_1$$

$$a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \dots + a_{0,n-1}x_{n-1} = b_0$$

kur $x_0, x_1, x_2, \dots, x_{n-1}$ ($0 \leq i < n$) yra nežinomieji.

Pertvarkę i-ją lygtį:

$$a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \dots + a_{i,n-1}x_{n-1} = b_i$$

i

$$x_i = (1/a_{i,i})[b_i - (a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \dots + a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} \dots + a_{i,n-1}x_{n-1})]$$

gausime:

$$x_i = \frac{1}{a_{i,i}} \left[b_i - \sum_{j \neq i} a_{i,j}x_j \right] .$$

Gavome iteracinę formulę, leidžiančią apskaičiuoti tikslesnes nežinomujų reikšmes pagal jau turimas.

(Visos x_i reikšmės turi būti įstatomos vienu metu).

Šis tiesinių lygčių sistemos sprendimo algoritmas vadinamas Jacobi metodu. Irodyta, jog Jakobi metodas konverguoja, jeigu matricos a istrižainė yra dominuojanti, t.y., kiekvienas ištrižainės elementas a_{ii} yra moduliu didesnis negu likusių eilutės elementų modulių suma.

Kitaip tariant, salyga

$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|$$

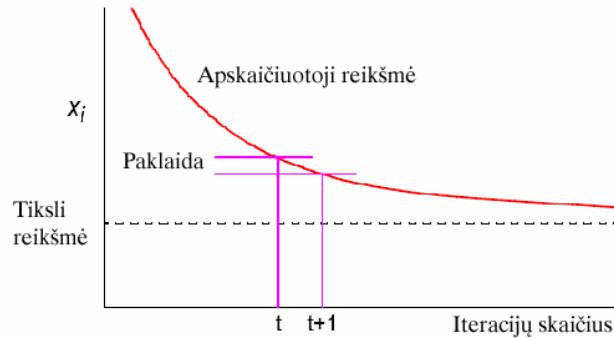
yra pakankama Jacobi iteracijų konvergavimo sąlyga.

Iteracijų baigmė paprastai apibrėžiama sąlyga:

$$|x_i^t - x_i^{t-1}| < \text{error}$$

kuri turi būti teisinga kiekvienam i , o t – iteracijos numeris.

Deja, pastaroji nelygybė ne visuomet gali reikšti pakankamą iteracinio sprendinio artimumą tikrajam sprendiniui, kaip parodyta žemiau pateiktoje schemae.



8.9 pav. Konvergavimo greitis [5]

Lygiagretusis sprendimo kodas, procesui P_i .

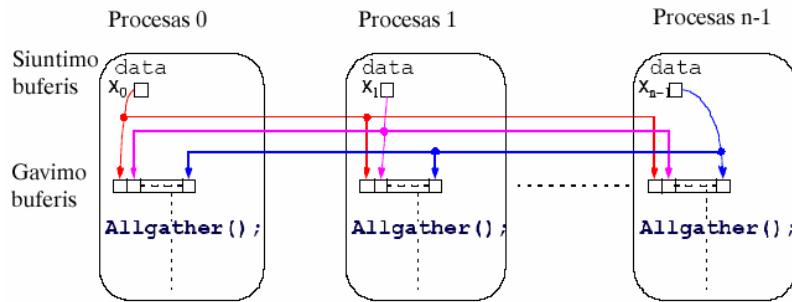
```

x[i] = b[i];                                     /* iniciuoti nežinomuosius */
for (iteration = 0; iteration < limit; iteration++) {
    sum = -a[i][i] * x[i];
    for (j = 0; j < n; j++)                      /* sumuoti */
        sum = sum + a[i][j] * x[j];
    new_x[i] = (b[i] - sum) / a[i][i];           /* skaičiuoti nežinomuosius */
    allgather(&new_x[i]);                         /* bcast/recv reikšmes */
    global_barrier();                            /* laukti visų procesų */
}

```

Čia panaudota funkcija `allgather()` perduoda naujai apskaičiuotą reikšmę $x[i]$ iš proceso i kiekvienam kitam procesui bei surenka broadcast'intus duomenis iš kitų procesų.

Funkcijos veikimas parodytas schemae:



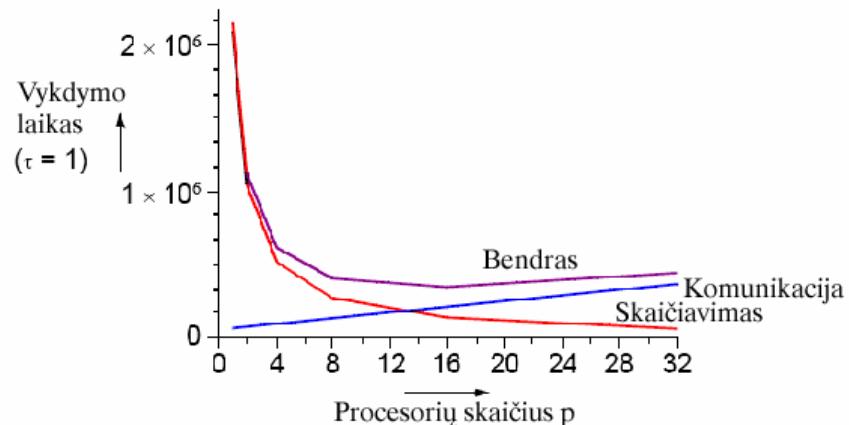
8.10 pav. Operacijos **allgather()** schema [5]

Paskirstymas. Paprastai procesorių skaičius yra žymiai mažesnis negu turimų apdorojamų duomenų skaičius. Taigi problemą reikia suskirstyti taip, kad vienas procesorius apdorotų duomenų aibę:

- *blokinis* paskirstymas – paskirti grupę vienas po kito einančių nežinomujų procesoriams didėjimo tvarka.
- *ciklinis išdėstymas* – kintamieji procesoriams paskiriami cikliškai, t.y., procesoriui P_0 priskiriami kintamieji $x_0, x_p, x_{2p}, \dots, x((n/p)-1)p$, procesoriui P_1 priskiriami $x_1, x_{p+1}, x_{2p+1}, \dots, x((n/p)-1)p+1$, ir t.t.

Ciklinis išdėstymas šiuo atveju neduoda jokios naudos.

Eksperimentinis skaičiavimų/komunikacijos santykis ir bendras vykdymo laikas pateiktas schema:



8.11 pav. Skaičiavimų ir komunikacijos sąryšis vykdant Jacobi iteracijas

8.3.2 Lokaliai sinchroniniai skaičiavimai.

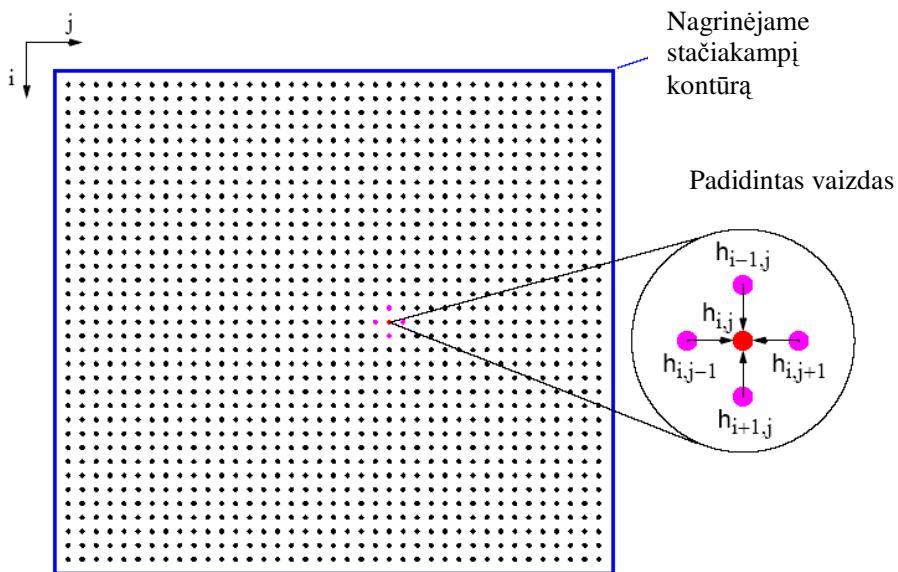
8.3.2.1 Šilumos pasiskirstymo problema

Tegu žinome temperatūros pasiskirstymą srities kontūre. Reikia rasti temperatūros pasiskirstymą šioje srityje.

Sprendimas. Padalykime sritį taškų tinklu $h_{i,j}$. Temperatūra vidiniame taške imkime kaip keturių aplinkinių taškų vidurkį. Kiekviename taške temperatūra nusakoma išraiška:

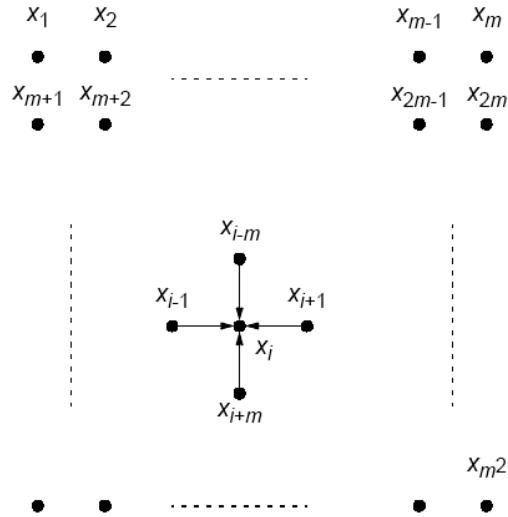
$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4},$$

kurią galima iteruoti ($0 < i < n, 0 < j < n$) fiksuotą kartą skaičių, arba kol skirtumas tarp dviejų iteracijų bus pakankamai mažas. Brėžinyje parodytas tokios srities pavyzdys.



8.12 pav. Šilumos pasiskirstymo problema

Taškų galima numeracija pateikta brėžinyje.



8.13 pav. Natūralioji sričių numeracija

Tada kiekvienam taškui galime užrašyti lygtį:

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-m} + x_{i+m}}{4}$$

bei sudaryti pilną lygčių sistemą.

Nuoseklusis kodas, kuriame iteracijų skaičius fiksuotas:

```
for (iteration = 0; iteration < limit; iteration++) {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
    for (i = 1; i < n; i++) /* atnaujinti taškus */
        for (j = 1; j < n; j++)
            h[i][j] = g[i][j];
}
```

Čia duomenys saugomi, naudojant $n*n$ masyvą.

Skaičiuojant norimu tikslumu, naudosime kodą:

```
do {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);

    for (i = 1; i < n; i++) /* atnaujinti taškus */
        for (j = 1; j < n; j++)
            h[i][j] = g[i][j];
```

```

continue = FALSE; /* testi ar ne */
for (i = 1; i < n; i++) /* patikrinti kiekv. tašką, ar konverguoja*/
    for (j = 1; j < n; j++)
        if (!converged(i, j) /*jei rastas nekonverguojantis*/
            continue = TRUE;
            break;
    }
} while (continue == TRUE);

```

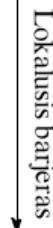
Lygiagretusis kodas fiksuotu iteracijų skaičiumi.

```

for (iteration = 0; iteration < limit; iteration++) {
    g = 0.25 * (w + x + y + z);
    send(&g, Pi-1,j); /* nesiblokuojantis siuntimas */
    send(&g, Pi+1,j);
    send(&g, Pi,j-1);
    send(&g, Pi,j+1);
    recv(&w, Pi-1,j); /* synchroninis gavimas */ 
    recv(&x, Pi+1,j);
    recv(&y, Pi,j-1);
    recv(&z, Pi,j+1);
}

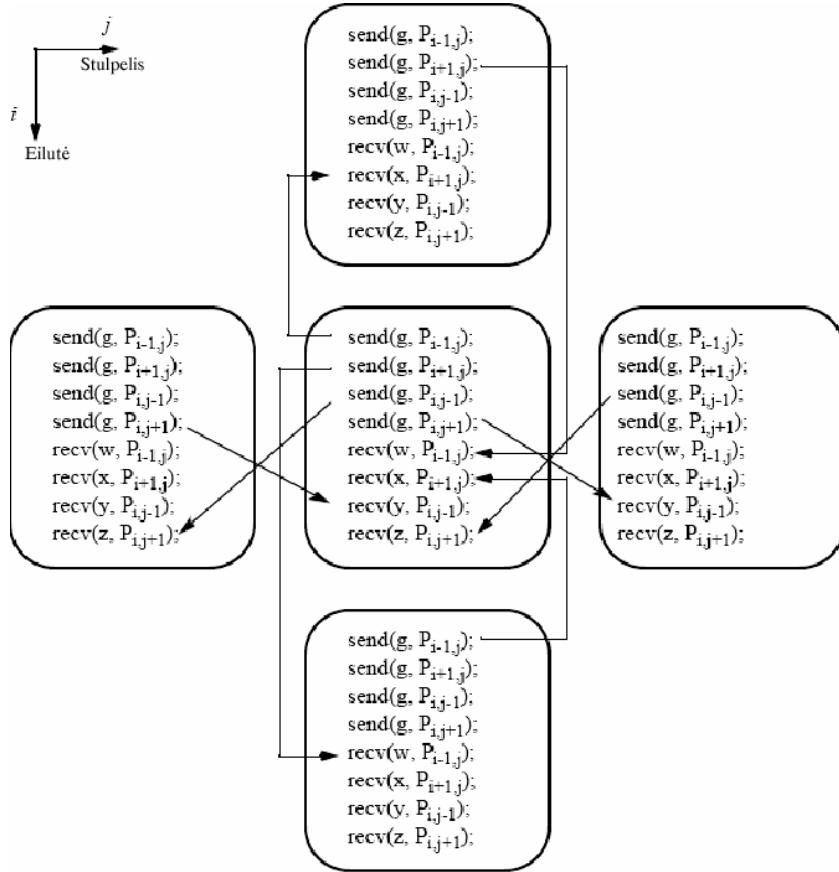
```

Lokalusis barjeras



Svarbu naudoti neblokuojantįjį `send()`, priesingu atveju procesai paklius į aklavietę. Kiekvienas `recv()` turi laukti blokuodamasis atitinkamo `send()`;

Pranešimų perdavimo schema, sprendžiant šilumos uždavinį.



8.14 pav. Šilumos uždavinio sprendimo pranešimų schema [5]

Jeigu naudosime iteracinių kodą, kuris vykdo procesus iki bus pasiekta reikiama tikslumas, kodas galėtų būti sekantis [5].

```

iteration = 0;
do {
    iteration++;
    g = 0.25 * (w + x + y + z);
    send(&g, P_{i-1,j}); /* lokalai blokuojantieji siuntimai */
    send(&g, P_{i+1,j});
    send(&g, P_{i,j-1});
    send(&g, P_{i,j+1});
    recv(&w, P_{i-1,j}); /* lokalai blokuojantieji gavimai */
    recv(&x, P_{i+1,j});
    recv(&y, P_{i,j-1});
    recv(&z, P_{i,j+1});
} while(!converged(i, j)) || (iteration < limit));
send(&g, &i, &j, &iteration, Master);

```

Jeigu įskaityti ir kampus, gausime sekantių kodą [5]:

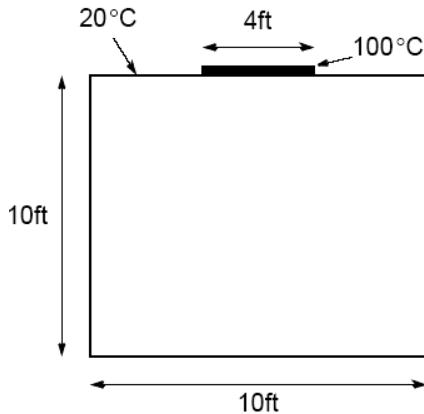
```

if (last_row) w = bottom_value;
if (first_row) x = top_value;
if (first_column) y = left_value;
if (last_column) z = right_value;
iteration = 0;
do {
    iteration++;
    g = 0.25 * (w + x + y + z);
    if !(first_row) send(&g, R-1,j);
    if !(last_row) send(&g, R+1,j);
    if !(first_column) send(&g, R,j-1);
    if !(last_column) send(&g, R,j+1);
    if !(last_row) recv(&w, R-1,j);
    if !(first_row) recv(&x, R+1,j);
    if !(first_column) recv(&y, R,j-1);
    if !(last_column) recv(&z, R,j+1);
} while ((!converged) || (iteration < limit));
send(&g, &i, &j, iteration, master);

```

Konkretus pavyzdys.

Kambarys turi keturias sienas ir židinį. Sienų temperatūra yra 20°C , o židinio 100°C . Reikia rasti Jacobi'o iteracijomis temperatūrą kambario viduje.



8.15 pav. Konkretizuotas modelis [5]

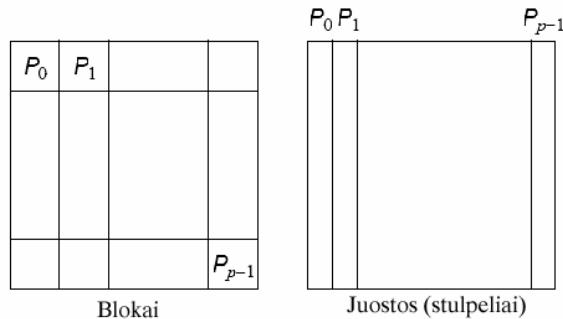
Gautas rezultatas, išvestas izolinijomis galėtų atrodyti taip:



8.16 pav. Sprendinys: temperatūros pasiskirstymas objekte [5]

Duomenų paskirstymas procesoriams.

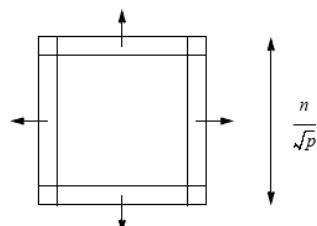
Paprastai taškų daugiau negu procesorių, taigi reikia taškus paskirstyti. Paprasčiausiai tai atlikti naudojant stačiakampes sritis – išskirstyti blokais arba stulpeliais.



8.17 pav. Srities skaidymas [5]

Išskirstant blokais, turime keturis kraštus, kuriuose bus keičiamasi duomenimis. Komunikacijos laiką galime nusakyti:

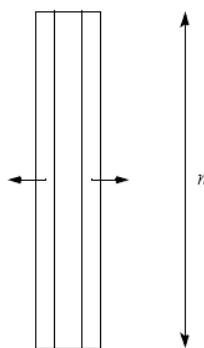
$$t_{\text{commsq}} = 8 \left(t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}} \right)$$



8.18 pav. Komunikacija, skirtant kvadratiniais blokais

Dalijant stulpeliais, pasikeitimąs vyksta tik dviejuose šonuose. Taigi:

$$t_{\text{commcol}} = 4(t_{\text{startup}} + nt_{\text{data}})$$



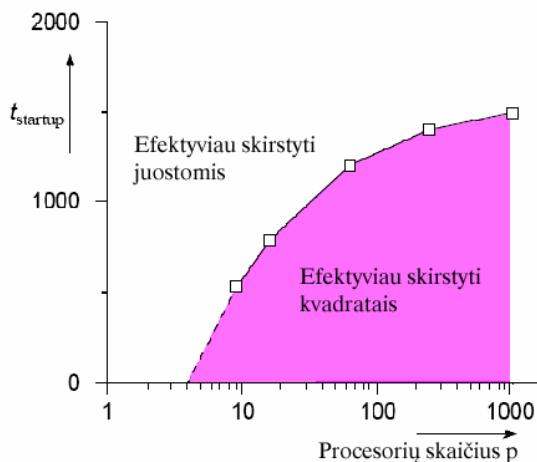
8.19 pav. Komunikacija, skirtant stulpeliais [5]

Bendru atveju paskirstymas stulpeliais (juostomis) geresnis, esant didesniams pranešimo iniciavimo (*startup*) laikui, o blokinis – esant mažesniams *startup* laikui. Atsižvelgiant į aukščiau pateiktasias išraiškas, blokinis dalijimas turi didesnį komunikacijos laiką negu juostinis, jeigu

$$t_{\text{startup}} > n \left(1 - \frac{2}{\sqrt{p}}\right) t_{\text{data}}$$

kai $p \geq 9$.

Pranešimo pasiruošimo laikas, esant blokiniam ir juostiniam paskirstymui.



8.20 Pasiruošimo (*startup*) laikai blokinės ir juostinės komunikacijos atveju [5]

Taškai – vaiduokliai (*ghost*).

Taip vadinamos papildomos taškų eilutės, skirtos saugoti gretimos briaunos taškus. Kiekvienas taškų masyvas lokaliame procese papildomas eilutėmis/stulpeliais, kad tilptų taškai-vaiduokliai. Tuo būdu supaprastinamas programos kodas.

Saugumas ir aklavietės.

Situacija, kai visi procesai pirmiai išsiunčia pranešimus, o tik po to juos priima nėra *saugi*, kadangi ji remiasi `send()`’u buferizavimu. Buferizavimo apimtis nėra specifikuota MPI standarte. Todėl, jeigu buferizavimui sistemoje pritruks vietos, pranešimo išsiuntimas gali būti atidėtas, o tai gresia aklaviete.

Tam, kad kodas taptų saugu, sukeiskite `send()` ir `recv()` kreipinius gretimuose procesuose taip, kad tik vienas procesas kviečia pirmą `first()`. Tada netgi sinchroninis `send()` neišsauks

aklavietės. Taip pat neblogas variantas yra naudoti MPI-saugias apsikeitimo pranešimais funkcijas:

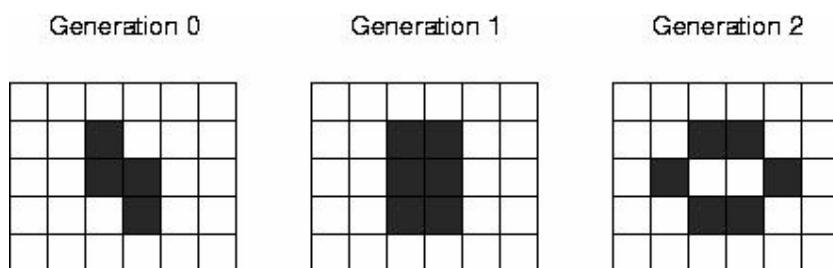
- MPI_Sendrecv() – saugus kombinuotas send() ir recv() variantas;
- Buferizuotas send: MPI_Bsend(), kuriame buferis nurodomas išreikštiniu būdu;
- Neblokuojančios send/recv: MPI_ISend(), MPI_Irecv(), kurios iš kartoj grįžta kartu su MPI_Wait(), MPI_Waitall(), MPI_Waitany(), MPI_Test(), MPI_Testall(), MPI_Testany();

Kitos visiškai sinchronizuotos problemos.

Ląsteliniai (cellular) automatai.

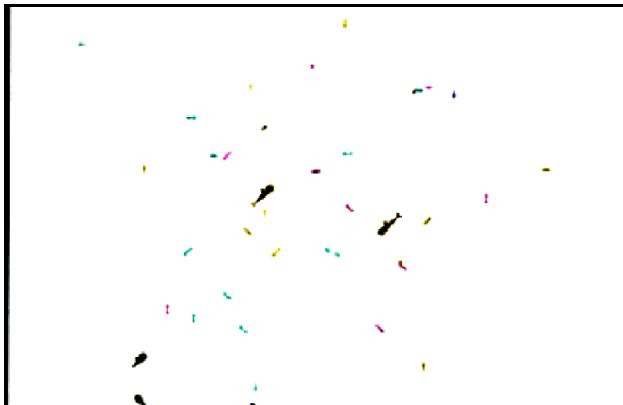
- Problemos erdvė sudalyta į ląsteles.
- Kiekviena ląstelė turi baigtinį būsenų skaičių.
- Lastelės sekanti būsena priklauso nuo ankstesnės pastarosios bei aplinkinių ląstelių būsenos pagal tam tikras apibrėžtas taisykles.
- Nauja būsena apskaičiuojama visoms naujos kartos ląstelėms vienu metu.

Labiausiai žinomas konkretus automatas yra “*Game of Life*” kurį aprašė *John Horton Conway*, Kembridžo matematikas. Kadangi taisyklės yra visuotinai žinomos, žemiau pateiki tik kelių „kartų“ generacijų pavyzdžiai.



8.21 Konkretaus žaidimo "life" kartos [Wikipedia]

Kitas įdomus trimatis ląstelinis automatas, modeliuojantis ekosistemą yra rykliai-žuvys (*sharks and fishes*). Po kelių kartų vaizdas gali būti sekantis.



8.22 pav. Ekosistemos modelio vaizdas [5]

“Rimti” lastelinių automatu taikymai.

- skysčių/dujų dinamika;
- skysčių/dujų aptekėjimas aplink objektus;
- dujų difuzija;
- biologinių objektų augimas;
- jūros ar upės kranto erozija/padėties kitimas.

Iš dalies synchroniniai skaičiavimai.

Tai tokis skaičiavimo procesas, kai individualūs procesai nesiekia būti synchronizuoti su kitaip procesais po kiekvienos iteracijos.

Idėja reikšminga, kadangi procesų synchronizacija yra brangi operacija, lėtinanti procesų darbą taigi ir visos lygiagrečiosios sistemos darbą.

Panagrinėsime, kaip būtų galima išspręsti šilumos pasiskirstymo uždavinį, nenaudojant globalių barjerų.

Sritij, kurioje ieškomas sprendimas pateikjame kaip dvimatį taškų masyvą. Kiekvieno taško reikšmė skaičiuojama kaip keturių aplinkinių taškų vidurkis tol, kol gautoji reikšmė nesukonverguos reikiamu tikslumu. Laukimo laikas gali būti sumažintas, nereikaujant synchronizacijos po kiekvienos iteracijos.

Nuoseklusis kodas:

```

do {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);

    for (i = 1; i < n; i++)          /*      Atnaujinti reikšmes rasti paklaidą      */
        for (j = 1; j < n; j++) {
            dif = h[i][j] - g[i][j];
            if (dif < 0) dif = -dif;
            if (dif < max_dif) max_dif = dif;
            h[i][j] = g[i][j];
        }
} while (max_dif > tolerance); /* Konvergavimo sąlyga */

```

Pirmaoji kodo sekcija yra tradicinis Jacobi iteracijų metodas, kai sekančios iteracijos reikšmės skaičiuojamos, panaudojant ankstesnės iteracijos reikšmes.

Jeigu leistume naujos iteracijos skaičiavimą, neradę visų ankstesnės iteracijos reikšmių, tokiu atveju būtų panaudotos reikšmės iš dar ankstesnių iteracijų.

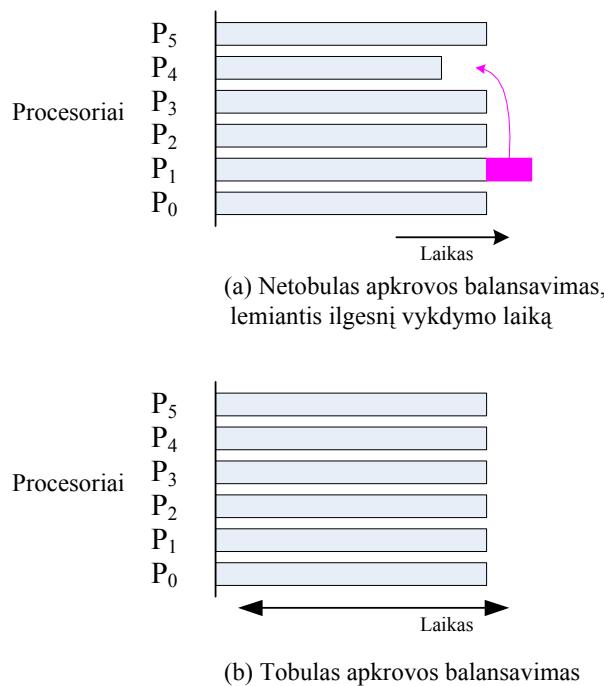
Tokį skaičiavimo būdą vadinsime *asynchroniškai iteratyviuoju metodu*.

Reikia pastebėti, kad šio metodo konvergavimo matematinės sąlygos gali būti griežtesnės negu iprastojo metodo. Gali būti neleistina naudoti *bet kurios* iteracijos reikšmes, norint kad asynchroninis metodas konverguotų.

9 Apkrovos balansavimas ir baigmės aptikimas

Apkrovos balansavimas (*load balancing*) naudojamas tam, kad išskirstytoje programoje tolygiai apkrautu veikiančius procesorius ir tuo pačiu pasiekti kuo didesnį programos spartinimą.* Skaičiavimų baigmės (*termination*) išskirstytas aptikimas yra taip pat komplikuotas.

Žemiau patektas paveikslėlis demonstruoja, kaip galime sumažinti lygigrečiosios sistemos darbo laiką, perkeldami dalį darbo iš pirmojo procesoriaus (labiau apkrauto) į ketvirtajį (mažiau apkrautą).



9.1 pav. Apkrovos balansavimas

Iki šiol nagrinėjome ribotus apkrovos balansavimo atvejus, kai darbas procesoriams buvo paskirstomas iš anksto, arba (Mandelbroto'o skaičiavimo atveju) darbiniai procesai tarpusavyje nekomunikavo. Šiame skyriuje aptarsime atvejį, kai procesai tarpusavyje komunikuoją, tarpusavyje skirstydamai darbus.

Apkrovos balansavimas gali būti vykdomas statiskai – prieš vykdant „darbinius“ procesus arba dinamiškai – darbinių procesų veikimo metu.

* Skyriuje naudojama [5] medžiaga

9.1 Statinis balansavimas

Statinio balansavimo (kitaip vadinamo – išdėstymo - *mapping*) metodai:

- *Round robin* – darbų priskyrimas procesams nuosekliai – ratu. Kai visi procesai gauna darbą grįztama prie pirmojo proceso ir skirstymo algoritmas kartojamas.
- Atsitiktiniai (*Randomized*) algoritmai, kurie išrenkā procesus atsitiktine tvarka.
- Rekursyvaus skaidymo pusiau (*recursive bisection*) metodas – rekursyviai dalija problemą į dalines problemas turinčias vienodą skaičiuojamą vertę, tuo pačiu sumažinant komunikacijos kaštus.
- *Simulated annealing* — tam tikra optimizacijos technika.
- Genetiniai (*genetic*) algoritmai – kita optimizacijos technika.

Iš esmės statiniai balansavimo metodai sprendžia taip vadinamą objektų pakavimo į kuprines (*bin packing*) uždavinį. Kartu yra ir sava specifika: siekiant minimizuoti komunikacijos kaštus, turi būti atsižvelgta į komunikacinio tinklo struktūrą. Kaip žinome iš matematikos, kuprinės uždavinys priklauso NP-pilnų uždavinių klasei, t.y., polinominis sprendimo algoritmas nėra žinomas. Todėl, esant didesniams skirstomų darbų ir procesų skaičiui, tikslus skirstymo uždavinio sprendimas yra per brangus. Šiuo atveju gali būti naudojami apytiksliai metodai, grindžiami euristikomis.

Statiniai metodai sunkiai įgyvendinami praktiškai, kadangi

- daugelio užduočių sudėtingumas dažnai nėra žinomas iš anksto ir gali būti nustatomas tik jas vykdant;
- kartais iš anksto nežinomas užduočių skaičius (pavyzdžiui, sprendžiant paieškos uždavinius);
- dažnai nepavyksta tikslai įvertinti komunikacijos greičio, dėl kompiuterinio tinklo darbo netolygumų.

Dinaminis apkrovos balansavimas leidžia iškaityti anksčiau minėtus faktorius, tačiau apsikeitimas apkrovos informacija tarp atskirų procesų gali turėti papildomų išlaidų. Daugeliu atveju dinaminis balansavimas yra žymiai efektyvesnis negu statinis, ypač heterogeninėse išskirstytose aplinkose.

9.2 Dinaminis apkrovos balansavimas

Kaip ir anksčiau, laikysime, jog turime procesų (procesorių) aibę bei užduočių, kurias reikia atlikti, aibę. Uždavinys – paskirstyti užduotis procesams, kad procesai būtų kuo labiau apkrauti.

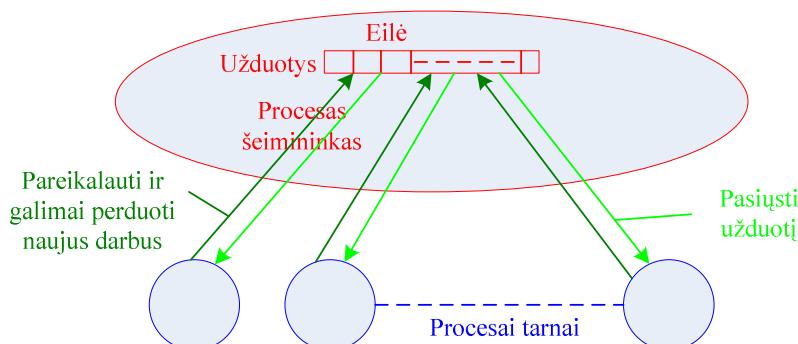
Dinaminį apkrovos balansavimą galime klasifikuoti kaip

- centralizuotą;
- decentralizuotą.

Centralizuotas balansavimas remiasi šeimininko-tarno struktūra, kai vienas procesas skirsto užduotis likusiems. Decentralizuoto balansavimo atveju procesai tarnai gali apsikeitinėti užduotimis tarpusavyje.

9.2.1 Centralizuotas dinaminis apkrovos balansavimas

Centralizuotu atveju – procesorius šeimininkas saugo užduotis, kurias reikia įvykdyti. Užduotys persiunčiamos tarnams. Tarnui užbaigus užduotį, jis pareikalauja naujos užduoties iš proceso šeimininko. Sąveikos schema pateikta piešinelyje.



9.2 pav. Centralizuota darbų krūva [5]

Kartais naudojami terminai darbų krūva, replikuoti darbininkai, procesorių ferma (*work pool, replicated worker, processor farm*).

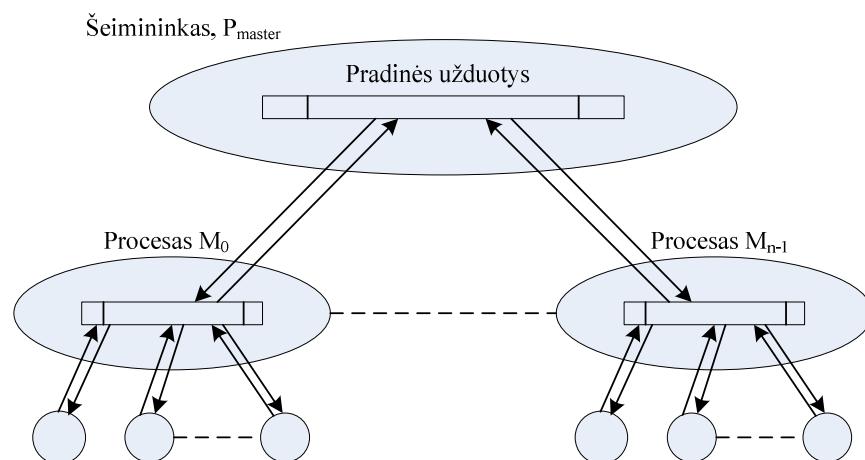
Programos baigmė nusakoma sąlygomis:

- darbų krūva yra tuščia ir
- kiekvienas procesas kreipėsi su prašymu naujam darbui, nesant naujai sukurtų darbų.

Nėra pakankama tuščios eilės sąlyga, esant vienam ar daugiau vykdomų procesų, jeigu vykdomieji procesai gali sukurti naujų darbų.

9.2.2 Decentralizuotas dinaminis apkrovos balansavimas

Nors centralizuota darbų krūvos schema yra plačiai naudojama, tačiau ji turi esminį trūkumą – šeimininkas vienu metu gali aptarnauti tik vieną procesą – tarną. Esant didesniams procesui skaičiui tai gali tapti pagrindiniu trukdžiu, efektyviai skirtant darbus. Viena iš darbų krūvos atmaina – decentralizuota darbų krūva, sudaranti medžio pavidalo struktūrą, bando išspręsti “siauro kaklelio” problemą.

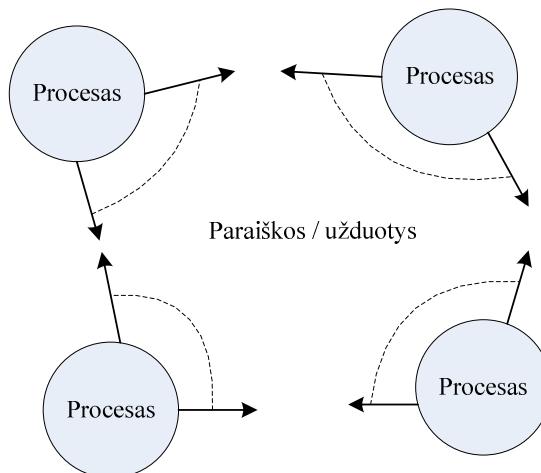


9.3 pav. Išskirstyta darbų krūva

Čia šeimininkas išskirsto darbus mini-šeimininkams, kurie aptarnauja savo tarnų grupę. Jeigu turime optimizavimo uždavinį, tai mini-šeimininkai tampa atsakingi už lokalių minimumų paiešką, kurie perduodami šakniniam procesui, kad apskaičiuoti globalų optimumą. Akivaizdu, kad ši struktūra gali būti toliau rekursyviai skaldoma, kad pasiekti tinkamą aptarnaujamą tarnų skaičių.

9.2.2.1 Visiškai išskirstyta darbų krūva.

Joje procesai prašo darbų vienas iš kito.



9.4 Visiškai išskirstyta darbų krūva

Užduočių perdavimą gali inicijuoti

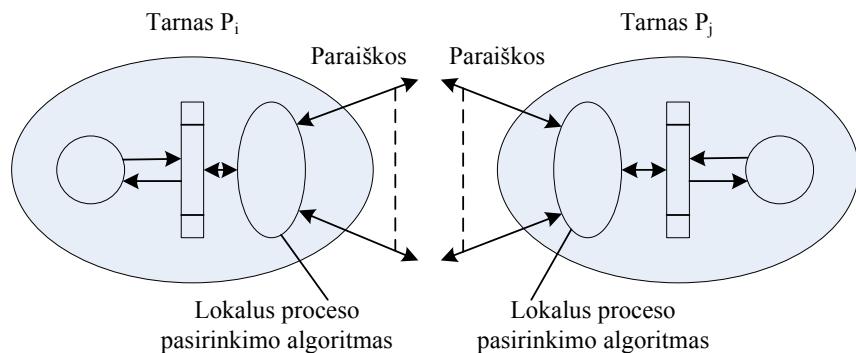
- procesai – siuntėjai arba
- procesai gavėjai.

Proceso – gavėjo inicijuota komunikacija reiškia, jog procesas prašo darbo iš kito pasirinkto proceso. Tipiskai, prašymas pateikiamas tada, kai procesas nebeturi užduočių arba numato, kad jų greitai pritrucks. Yra parodyta, jog šis metodas gerai veikia, esant didelei procesų apkrovai. Dejam procesų apkrovą yra sunku numatyti iš anksto.

Proceso-siuntėjo inicijuota komunikacija reiškia, jog procesas siunčia užduotis pasirinktajam procesui. Tipiskai sunkiai apkrauti procesai siunčia savo užduotis kitiems procesams, norintiems juos priimti. Irodyta, jog metoda gerai veikia, esant lengvai visos procesų sistemos apkrovai.

Dar viena galimybė – panaudoti abu minėtuosius metodus kartu. Deja, kaip jau buvo minėta, sunku numatyti sistemos apkrovą. Be to, esant dideliai sistemos apkrovai, sunku pasiekti balansą, dėl procesų trūkumo.

Decentralizuotas procesų pasirinkimo mechanizmas parodytas paveikslėlyje.



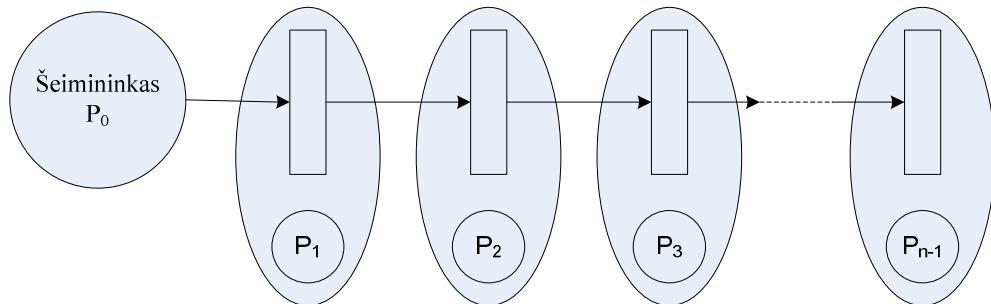
9.5 pav. Decentralizuotas procesų pasirinkimo mechanizmas

Procesų pasirinkimas turi remti tam tikru lokaliu proceso išsirinkimo algoritmu, kuris užtikrintu kuo tolygesnę procesų apkrovą. Plačiai naudojami šie algoritmai.

- Ciklinis (*round robin*) algoritmas – procesas P_i reikalauja užduoties iš proceso P_x , kur x - skaitliuko reikšmė moduliu n , padidinama po kiekvieno kreipinio, čia n – procesų skaičius, o x nesutampa su i .
- Atsitiktinės apklausos (*random polling*) algoritmas – procesas P_i kreipiasi užduoties į proceso P_x , kur x – atsitiktinis skaičius tarp 0 ir $n - 1$ (išskyrus i).

9.2.2.2 Apkrovos balansavimas, naudojant tiesinę struktūrą.

Šio skyrelio metodai gali būti pritaikyti ir kitokios konfigūracijos tinklui. Pagreindinė idėja – turėti užduočių eilę, kurios individualias vietas gali prieiti procesoriai.

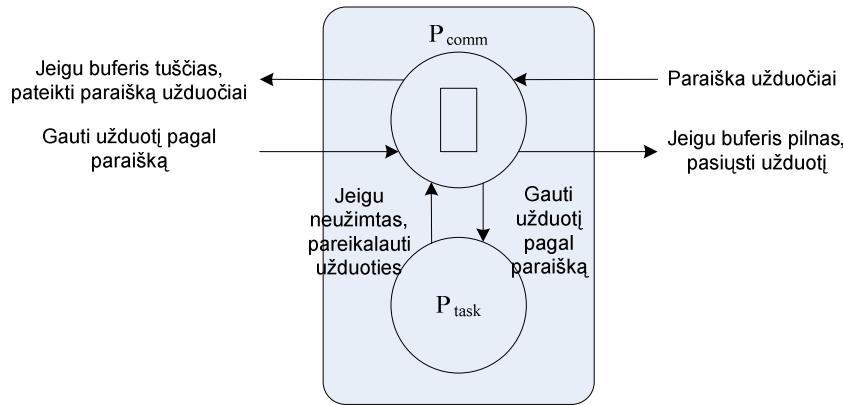


9.6 pav. Apkrovos balansavimas konvejeryje

Procesas – šeimininkas (P_0) užpildo užduočių eilę užduotimis, ir darbai slenka eile pirmyn.

Kai procesas P_i ($1 \leq i < n$), aptinka, jog jo įėjime pateikta nauja užduotis, ir procesas neužimtas darbu, procesas nuskaito užduotį iš eilės ir apdoroja. Užduotys pasislenka eile pirmyn, užpildydamos neužimtus procesus. Tuo būdu visi procesai gaus užduotis, ir eilė bus užpildyta užduotimis. Aukštesni prioriteto užduotys turėtų būti pateikiamos eilės priekyje.

Eilės slinkimo veiksmus galėtų aptarnauti po du procesus. Vienas būtų atsakingas už užduočių perdavimą tarp gretimų procesų. Antrasis – už užduočių vykdymą, kaip parodyta paveikslėlyje.



9.7 pav. Atskiro proceso naudojimas konvejerinės balansavimo schemas grandyje

Nurodytąją komunikaciją galėtų įgyvendinti žemiau pateiktas kodas.

Procesas – šeimininkas (P0).

```
for (i = 0; i < no_tasks; i++) {
    recv(P1, request_tag); /* Pareikalauti užduoties */
    send(&task, Pi, task_tag); /* Siųsti užduotis į eilę */
}
recv(P1, request_tag); /* Pareikalauti užduoties */
send(&task, Pi, task_tag); /* Užduočių pabaiga */
```

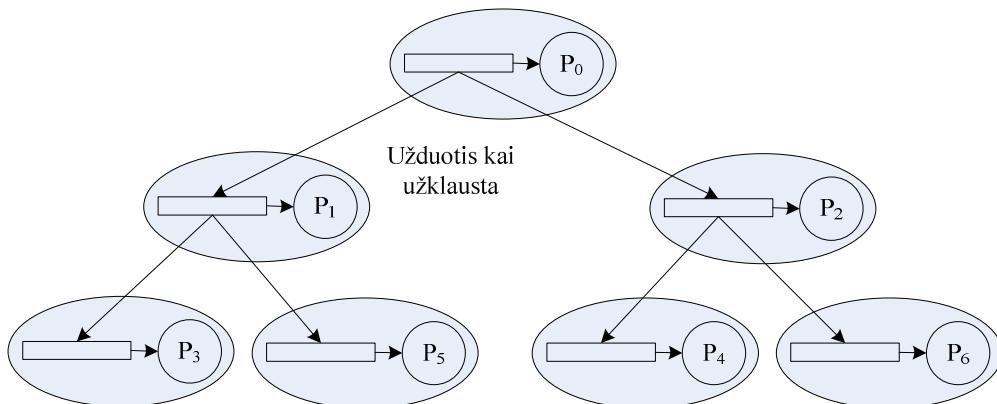
Procesas - tarnas P_i ($1 < i < n$):

```
if (buffer == empty){
    send(Pi-1, request_tag); /* Pareikalauti užduoties */
    recv(&buffer, Pi-1, task_tag); /* Užduotis iš kairiojo proceso */
}
if (buffer == full && !busy) { /* Gauti naują užduotį */
    task = buffer; /* Gauti užduotį */
    buffer = empty; /* Nustatyti buferį tuščią */
    busy = TRUE; /* Nustatyti, kad procesas užimtas */
}
nrecv(Pi+1, request_tag, request); /* Patikrinti pranešimą iš dešinės */
if (request && buffer == full){
    send(&buffer, Pi+1); /* Pasiųsti užduotį pirmyn */
    buffer = empty
}
if (busy) { /* Tęsti dabartinę užduotį */
    ... Atlikti darbo dalį ...
    if (...darbas atliktas ...) busy = FALSE
}
```

Čia kreipinys **nrecv()** žymi nesiblokuojantį skaitymą, kurio prieikia patikrinant, ar iš dešinės negautas prašymas užduotims.

MPI nesiblokuojanti pranešimo gavimo operacija įgyvendinama funkcija **MPI_Irecv()**, kuri iškar gražina deskriptorių, kurį vėliau galima panaudoti tolimesnėse operacijose tam, kad laukti, kol pranešimas bus atsiųstas arba patikrinti ar pranešimas jau atsiųstas (atitinkamai, **MPI_Wait()** and **MPI_Test()** funkcijos).

Tiesinė pranešimų tinklo topologijos metodai, naudojami ankstesniame pavyzdje, gali būti nesunkiai perkelti ir į sudėtingesnius tinklus, pvz., medži.



9.8 pav. Apkrovos balansavimas medyje

Šiame tinkle perteklinės užduotys gali būti persiunčiamos grafu žemyn.

9.3 Išskirstytos programos baigmės aptikimo sąlygos

Kai skaičiavimai išskaidyti, gana sunku aptikti skaičiavimų baigmę, išskyrus tą atvejį, kai pakanka, jog vienas procesas ras reikiama sprendinį.

Bendru atveju, kad programa būtų baigusi užduočių atlikimą laiko momentu t , turi būti tenkinamos šios sąlygos:

- kiekvienam procesui laiko momentu t būtų tenkinamos nuo uždavinio priklausančios lokalios baigmės sąlygos
- bei laiko momentu t visi anksčiau išsiuisti pranešimai turi būti pasiekę adresatą.

Egzistuoja subtilūs skirtumai tarp centralizuotų ir išskirstytų apkrovos balansavimo sistemų, susiję su “tranzitinių” pranešimų perdavimu. Tranzitinio pranešimo egzistavimas gali reikšti, jog tam tikras procesas, kuris, laikoma, jau baigė savo darbą, gali pratesti užduočių vykdymą. Tokių tranzitinių užduočių egzistavimą sunkiau aptikti išskirstytose sistemose. Reikia atsiminti, jog pranešimo perdavimo laiką sunku įvertinti iš anksto.

Žemiau pateikiamas vienas labai bendras išskirstytos baigmės aptikimo algoritmas, kurį pateikė Bertsekas ir Tsitsiklis (1989).

Kiekvienas procesas yra vienoje iš dviejų būsenų:

- neaktyvus – užbaigęs visus savo turimus darbus;
- aktyvus.

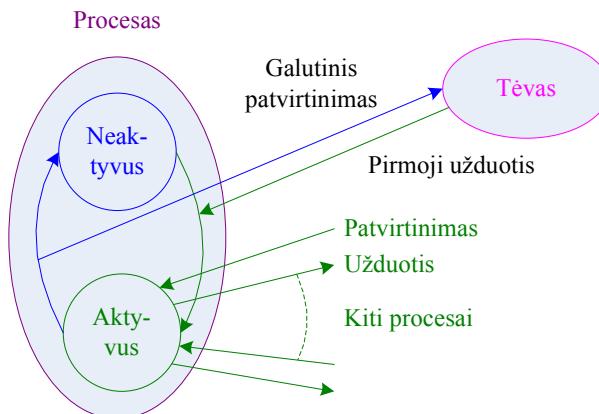
Procesas, kuris nusiunčia darbą neaktyviam procesui tampa jo “tėvu”. Kai procesas gauna užduotį, procesas tuoju pat turi pasiūsti atgal patvirtinimo pranešimą, išskyrus tą atvejį, kai užduotį atsiuntė jo tévas.

Patvirtinantis pranešimas tėvui nusiunčiamas tik tuo atveju, jeigu procesas tampa pasiruošęs tapti neaktyviu, t.y., kai

- egzistuoja lokalios proceso baigmės sąlygos (visos užduotys baigtos) ir
- procesas išsiuntė visus patvirtinimus gautoms užduotims ir
- procesas gavo visus patvirtinimus užduotims, kurias pats išsiuntė kitiems.

Procesas turi tapti neaktyviu, prieš tėviniam procesui tampant neaktyviu. Kai pirmasis procesas tampa neaktyviu, skaičiavimai gali būti laikomi globaliai baigtais.

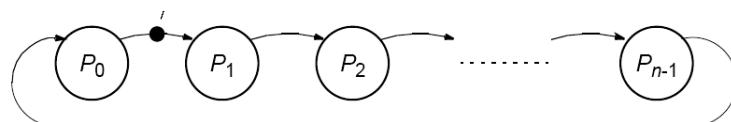
Žemiau pateikta lokalios baigmės schema.



9.9 pav. Prosesų baigmės nustatymas, naudojant patvirtinančiuosius pranešimus [5]

Baigmės aptikimo algoritmai gali būti pritaikyti konkrečiai tinklo struktūrai.

Baigmės aptikimas žiede gali remis žymės (*token*) perdavimu kaip parodyta paveikslėlyje.



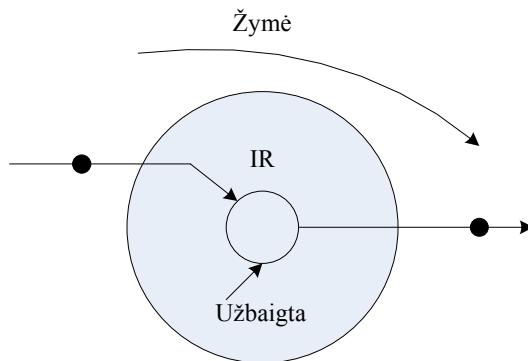
9.10 pav. Baigmės žiede aptikimas, naudojant žymę

Algoritmo žingsniai.

1. Kai procesas P0 baigiasi, jis sukuria žymę, kuria persiunčia procesui P1.

2. Kai procesas P_i ($1 \leq i < n$) yra gavęs žymę ir pasibaigia, jis persiunčia žymę dešiniajam procesui P_{i+1} . Procesas P_{n-1} grąžina žymę pradiniam procesui P_0 .
3. Kai procesas P_0 gauna žymę, tai reiškia, jog visi procesoriai žiede užbaigė savo užduotis. Jeigu reikia, likusiems procesams gali būti nusiuistas pranešimas apie globalią baigmę.
4. Algoritmas remiasi prielaida, jog joks procesas negali būti reaktyvuojamas po jo lokaliosios baigmės pasiekimo. Pavyzdžiui, algoritmas negali būti taikomas darbų krūvai, kur procesas gali patalpinti į krūvą naujus darbus.

Algoritmą vieno proceso požiūriu nusako pateiktoji schema.



9.11 pav. Algoritmas lokalios baigmės aptikimui

Tuo atveju, kai procesai gali būti pakartotinai aktyvuojami, tinkta **dvigubo apėjimo žiedu baigmės aptikimo algoritmas**, kuris susideda iš dviejų fazų. Tuo atveju, jeigu procesas P_i persiuntė naują darbą procesui P_j , po to, kai perduavė žymę j-jam procesui, žymė turi cirkuliuoti pakartotinai.

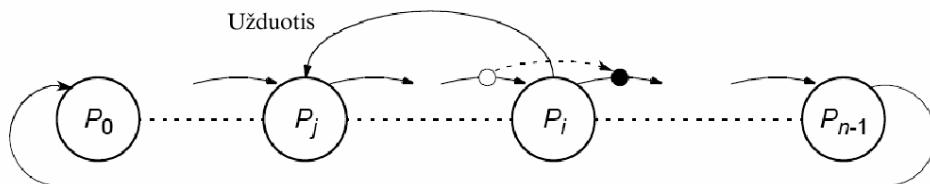
Tam, kad iškaityti šią situaciją, nuspalvinkime procesus ir perduodamą žymę balta arba juoda spalva.. Jeigu priimama juoda žymė, vadinas globalios baigmės salyga buvo neišpildyta ir žymė turi prabėgti žiedą dar kartą.

Algoritmo aprašymas, pradedant procesu P_0 .

1. Procesas P_0 , pasibaigęs ir išsiuntęs sugeneruotą baltą žymę procesui P_1 , tampa baltu..
2. Žymė yra perduodama iš vieno proceso į sekantį, kai procesas užbaigia (lokaliai) savo darbą, tačiau žymės spalva gali būti pakeista. Jeigu procesas P_i perduoda užduotį procesui P_j , kur $j < i$ (t.y., žiedo pradžios kryptimi) procesas P_i tampa *juodu* procesu, priešingu atveju procesas - *baltas*. Juodas procesas nuspalvina siunčiamą tollyn žymę juodai. Baltas procesas perduoda žymę, nekeisdamas žymės spalvos. Po to, kai procesas perduoda žymę, jis tampa baltu procesu.

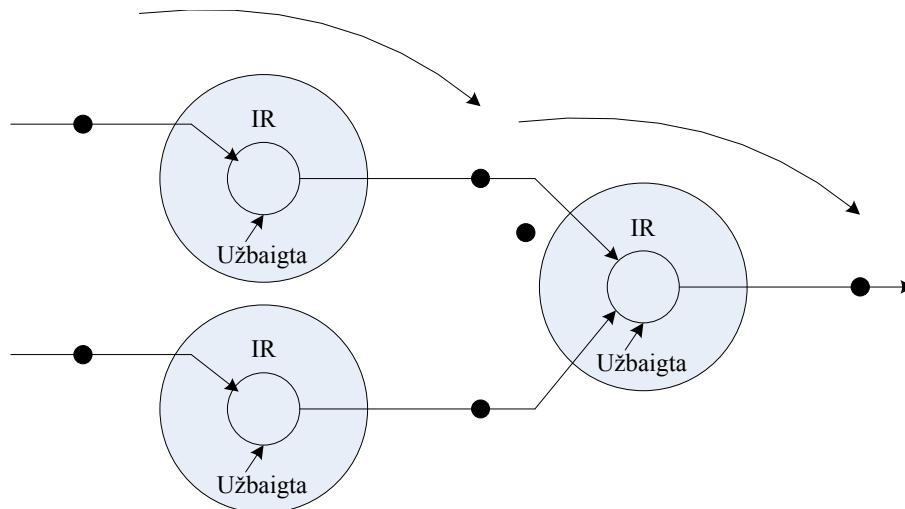
3. Jeigu procesas P_0 gauna iš paskutinio proceso juodą žymę – jis perduoda tollyn kaip baltają. Jeigu gauna baltą žymę – tai požymis, kad visi procesai baigė darbą.

Abejuose žiediniuose algoritmuose P_0 trampa globaliosios baigties centriniu tašku. Laikoma, kad kiekvienai užklausai generuojamas patvirtinimo signalas. Žemiau pateiktas paveikslėlis vaizduoja užduoties perdavimas kairiau esančiam procesui.



9.12 pav. Užduoties perdavimas ankstesniajam procesui [5]

Panašaus pobūdžio algoritmai, kurie remiasi lokaliosios baigmės sąlygomis, gali būti naudojami ir medžio pavidalo struktūrose.



9.13 Baigmės nustatymas medyje

Pastovios energijos išskirstytos baigmės algoritmas.

Naudojasi kiekybinėmis sistemos charakteristikomis, vaizdžiai vadinant “energija”. Energijos tékmė primena žymės tékmę, tik turi kiekybinę išraišką.

Programa startuoja, turēdama tam tikrą energijos kiekį, kuri sukoncentruota šakniniame procese. Programos vykdymo metu šakninis procesas paskirsto energiją kitiems procesams. Jeigu pastarieji gauną prašymą pasidalinti užduotimis iš kitų procesų, atitinkamai dalijamas ir turimu energijos kiekiu.

Kai procesas tampa neveiksniu, jis sugražina gautajį energijos atgal, prieš prašydamas naujų užduočių. Energijos gražinimas yra atidedamas tol, kol procesui nebus sugražintas jo išdalytas energijos kiekis.

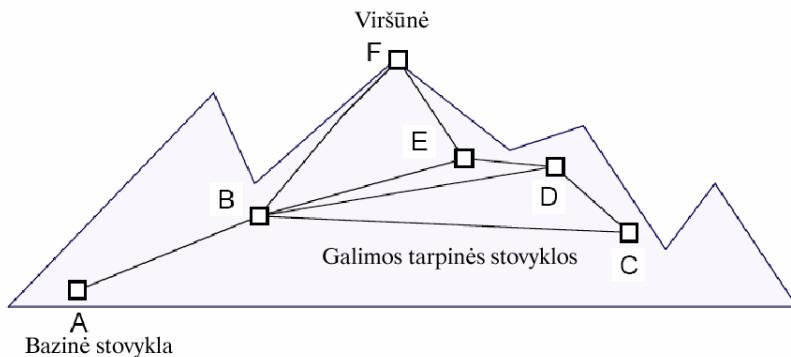
Kai visa energija grąžinama atgal į šakninį procesą ir pastarasis tampa nedirbančiu, skaičiavimai gali baigtis.

Duotasis algoritmas turi trūkumą – dalijant energiją trupmeniniai vienetais, gautujų dalių suma dėl skaičiavimo tikslumo trūkumo gali būti lygi pradinei reikšmei.

9.4 Apkrovos balansavimo ir baigmės aptikimo pavyzdys: trumpiausio kelio grafe paieškos uždavinys

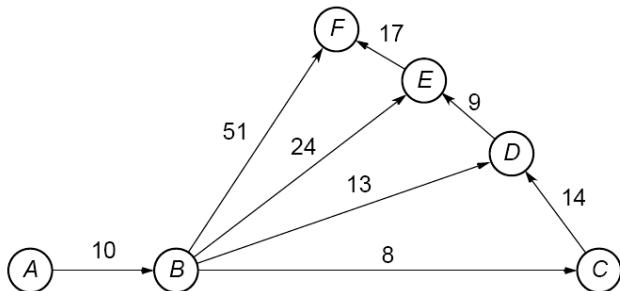
Nagrinėjame grafą su užduotais briaunų svoriais. Reikia rasti trumpiausią kelią tarp dviejų duotųjų viršūnių, t.y., kelią, kurio briaunų svorių suma būtų mažiausia. Galimos įvairios šios problemos variacijos.

1. Rasti trumpiausią kelią žemėlapyje tarp dviejų duotųjų miestų. Čia svoriai reiškia atstumus tarp miestų.
2. Greičiausia maršrutą tarp dviejų vietovių, kai svoriai reiškia kelionės laiką. (Pavyzdžiui, kai galima rinktis skirtingus kelionės būdus – lėktuvu, traukiniu, ar autobusu).
3. Pigiausią maršrutą, kai svoriai – atskirų atkarpuų kelionės kaina.
4. Geriausią kelią užkopti į kalną, kai duotas kopimo maršrutų žemėlapis (šis atvejis ir bus nagrinėjamas).
5. Geriausią maršrutą kompiuteriniame tinkle, kai užduoti minimalūs pranešimų tarp atskirų mazgų kelionės laikai.



9.14 pav. Kopimo į viršūnę maršrutai [5]

Tegu duotas kalno kopimo maršrutų žemėlapis. Mus dominačią informaciją galime pateikti grafu.



9.15 pav. Kopimo į viršūnę grafas [5]

Grafo briaunos nusako išlaidas. Taigi uždavinys – su mažiausiomis išlaidomis pasiekti viršūnę F. Mūsų nagrinėjamu atveju grafo briaunas vienakryptės, bet, jeigu nagrinėtume pakilimo į viršūnę ir nusileidimo uždavinį, briaunų įvertis nusileidimo kryptimi galėtų būti skirtinės negu pakilimo.

Grafa programoje galime aprašyti kaip:

- jungčių matricą – dvimatių masyvą a, kuriamė $a[i][j]$ - svoris priskirtas briaunai iš viršūnės i į j, jeigu tokia briauna egzistuoja, arba „begalybė“ – jeigu briaunos nėra.
- „jungčių“ sąrašą – kiekvienai viršūnei, nurodant iš jos išeinančių viršūnių sąrašą kartu su briaunų svoriais.

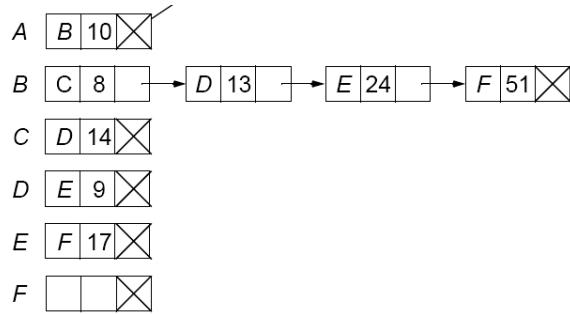
Sąrašas – tinkama struktūra, kai grafas yra retas. Tačiau paieška sąraše yra lėtesnė.

Jungčių matrica atrodys taip:

A	•	10	•	•	•	•
B	•	•	8	13	24	51
C	•	•	•	14	•	•
D	•	•	•	•	9	•
E	•	•	•	•	•	17
F	•	•	•	•	•	•

9.16 pav. Jungčių matrica [5]

o sąrašas:



9.17 pav. Jungčių vaizdavimas sąrašinėmis struktūromis

9.4.1 Paieška grafe

Yra du gerai žinomi paieškos grafe algoritmai:

- Moore'o vienos išeities viršūnės trumpiausiojo kelio algoritmas (1957 m.)
- Dijkstra'os algoritmas (1959 m.),

kurie yra labai panašūs.

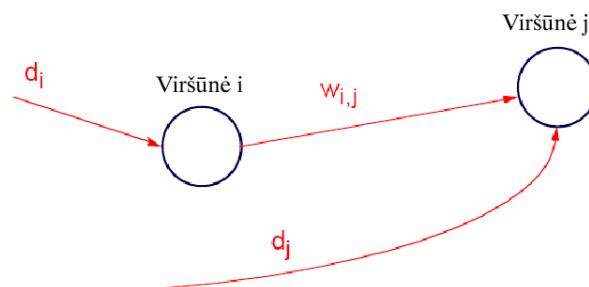
Žemiau naudojamas Moore'o algoritmo variantas, kuris geriau tinka išlygiagretinimui, tačiau gali būti mažiau efektyvus negu Dikstra'os.

9.4.2 Moore'o algoritmas.

Startuoja iš pradinės viršūnės. Tegu prijome i-ją viršūnę. Surandame atstumą į viršūnę j per viršūnę i ir sulyginame su anksčiau nustatytu minimaliu atstumu iki viršūnės j . Reikia pakeisti minimalų atstumą iki viršūnės j į mažesnį, jeigu atstumas per viršūnę i yra trumpesnis. Jeigu d_i yra dabartinis minimums tarp išeities viršūnės į viršūnę i ir $w_{i,j}$ yra briaunos tarp viršūnių i ir j svoris, tai :

$$d_j = \min(d_j, d_i + w_{i,j}).$$

Arba grafiškai:



9.18 pav. Grafo reprezentacija

Galutinį sprendimą gauname, kartodami šią formulę iteratyviai.

Duomenų struktūros. Duomenis galima saugoti kaip “pirmas įėjės – pirmas išeina” viršūnių eilę. Pradžioje į eilę patalpinama pradinė viršūnė. Esamą trumpiausią atstumą nuo pradinės viršūnės iki i -sios galime saugoti kaip masyvo $dist$ i -ji elementą. Pradžioje, nė vienas iš atstumų nėra žinomas, taigi masyvo elementai inicijuojami begalybe.

Programos kodas. Tarkime masyvo elementas $w[i][j]$ saugo briaunos iš i į j svorį (arba begalybę, jeigu briaunos nėra). Tada taikoma iteratyvi formulė:

```
newdist_j = dist[i] + w[i][j];
if (newdist_j < dist[j]) dist[j] = newdist_j;
```

Jeigu rastas trumpesnis atstumas iki viršūnės j , viršūnė j yra įtraukiamā į eilę, jeigu dar nėra eilėje, kas iššauks atstumą iki viršūnės j perskaičiavimą. Tai svarbi šio algoritmo ypatybė, kuria nepasižymi Dijkstra’os algoritmas.

Paieškos grafe žingsniai.

Pradiniu laiko momentu turime tokius duomenis

Nagrinėjamos viršūnės							Dabartiniai trumpiausi atstumai						
A							0	•	•	•	•	•	•
							viršūnės	A	B	C	D	E	F
vertex_queue							dist []						

Čia atstumas iki viršūnės A įvertinas 0. Kadangi iš A galime pereiti tik į viršūnę B, pastarajā įtraukiame į eilę, kartu nustatydami atstumą iki B lygiu 10.

Nagrinėjamos viršūnės							Dabartiniai trumpiausi atstumai						
B							0	10	•	•	•	•	•
							viršūnės	A	B	C	D	E	F
vertex_queue							dist []						

Iš B galime pereiti į F , E , D ir C . Pastebėsime, kad šiam algoritmui nėra svarbus naujų viršūnių peržiūros eilišumas. Tegu peržiūrėsime viršūnes nurodytaja tvarka. Nustatome naujus atstumus iki C, D, E, F kaip parodyta brėžinyje. Kadangi atstumai iki visų viršūnių masyve $dist$ yra atnaujinti, visas viršūnes (išskyrus F – galutinę viršūnę, iš kurios neišeina jokia briauna) įtraukiame į viršūnių eilę.

Nagrinėjamos viršūnės						Dabartiniai trumpiausi atstumai					
E	D	C				0	10	18	23	34	61
vertex_queue						viršūnės					
dist []											

Nagrinėjame briauną, kuri veda iš E to F , turinčią svorį 17. Kadangi nauja atstumo iki taško F vertė $|AE| + 17 = 34+17 = 51 < 61$ (mažesnė už anksčiau apskaičiuotą), atnaujiname masyvą `dist`. Gausime:

Nagrinėjamos viršūnės						Dabartiniai trumpiausi atstumai					
D	C					0	10	18	23	34	50
vertex_queue						viršūnės					
dist []											

Išnagrinėjus briauną iš D į E , gausime pagerintą atstumo iki E reikšmę lygią 32, todėl iš naujo įtraukiame viršūnę E į nagrinėjamų viršūnių eilę.

Nagrinėjamos viršūnės						Dabartiniai trumpiausi atstumai					
C	E					0	10	18	23	32	50
vertex_queue						viršūnės					
dist []											

Toliau nagrinėjame briauną CD - jokių pasikeitimų. Taigi liko išnagrinėti briauną EF , kuri pagerina atstumo iki F reikšmę.

Nagrinėjamos viršūnės						Dabartiniai trumpiausi atstumai					
						0	10	18	23	32	49
vertex_queue						viršūnės					
dist []											

Kadangi naujų viršūnių neaptikta – nustatėme minimalius atstumus nuo A iki bet kurios kitos viršūnės, išskaitant ir tikslo viršūnę F .

Paprastai prireikia gauti ir kelią, todėl kartu su viršūnių atstumų atnaujinimais įsimename ir kelius. Taigi, mūsų atveju, optimalus kelias $A \rightarrow B \rightarrow D \rightarrow E \rightarrow F$.

Nuoseklusis kodas. Tegu funkcija `next_vertex()` gražina eilinę viršūnę iš viršūnių sąrašo arba `no_vertex`, jeigu tokios nėra. Sakykime jungčių matrica yra $w[][]$. Tada paiešką galēsime įvykdyti pateiktąjai programa.

```
while ((i = next_vertex()) != no_vertex) /*Kol neperžūrėjome visas viršūnes*/
    for (j = 1; j < n; j++)           /*Gauti briauną*/
```

```

        if (w[i][j] != infinity) { /*Jeigu briauna yra*/
            newdist_j = dist[i] + w[i][j];
            if (newdist_j < dist[j]){
                dist[j] = newdist_j;
                append_queue(j); /*Itraukti į eilę*/
            }
        }
    }
}

```

Lygiagrečios realizacijos.

Realizacija, kuri remiasi centralizuota darbų krūva.

Centralizuota darbų krūva saugo viršūnių eilę `vertex_queue[]` kaip užduotis. Kiekvienas procesas - tarnas ima viršūnes iš viršūnių sąrašo ir grąžina naujas viršūnes, kurias reikės apdoroti. Grafo incidentiškumo matrica nekintanti, taigi šeimininkas ir tarnai darbo metu turės apsikeitinėti minimalių atstumų matricos `dist[]` reikšmėmis.

Šeimininkas.

```

while (vertex_queue() != empty){
    recv(PANY, source = Pi); /* Prašyti užduoties iš tarto */
    v = get_vertex_queue();
    send(&v, Pi);           /* Nusiusti sekancia virsune */
    send(&dist, &n, Pi);    /* Einamasis dist masyvas */
    .....
    recv(&j, &dist[j], PANY, source=Pi); /* Naujas atstumas */
    append_queue(j, dist[j]);           /* Itraukti viršūnę į eilę */
                                         /* ir atnaujinti atstumų masyvą */
}
recv(PANY, source = Pi); /* Gauti užduotį iš tarto */
send(Pi, termination_tag); /* Baigmės pranešimas */

```

Tarno kodas.

```

send(Pmaster); /* Pateikti paraišką užduočiai */
recv(&v, Pmaster, tag); /* Gauti viršūnių skaičių */
if (tag != termination_tag){
    recv(&dist, &n, Pmaster); /* Gauti dist masyvą */
    for (j = 1; j < n; j++) /* Gauti sekančią briauną */
        if (w[v][j] != infinity){ /* Jeigu briauna yra */
            newdist_j = dist[v] + w[v][j];
            if (newdist_j < dist[j]){
                dist[j] = newdist_j;
                send(&j, &dist[j], Pmaster); /* Itraukti viršūnę į eilę ir pasiūsti
                                         atnaujintą atstumą */
            }
        }
}

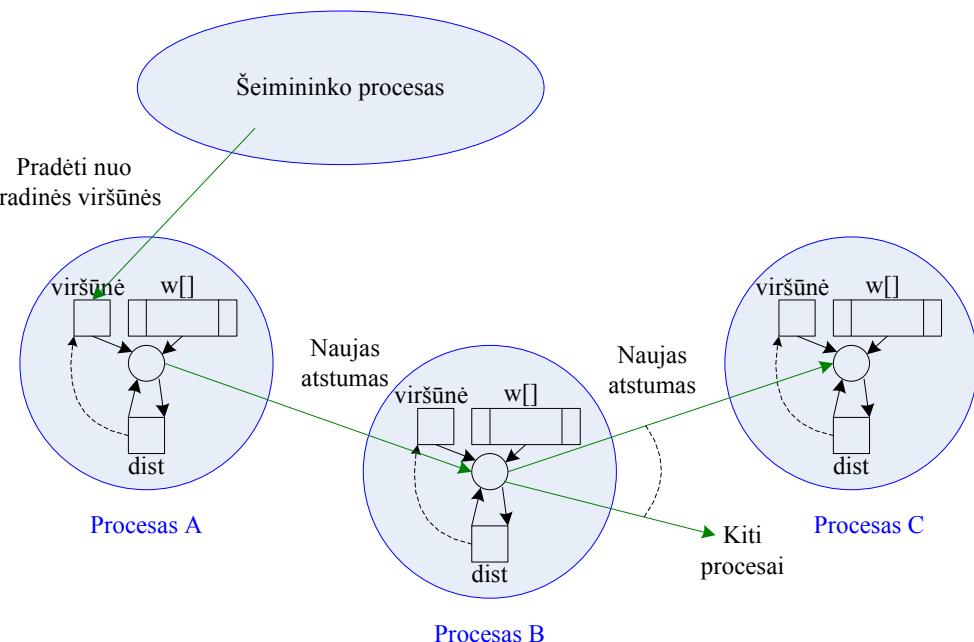
```

Decentralizuota darbų krūvą galima realizuoti, išskirstant eilę `vertex_queue[]` taip, kad už kiekvieną (*i*-ją) viršūnę būtų atsakingas atskiras procesas, taip pat šis procesas saugotų ir

minimalų atstumą nuo pradinės iki i-sios viršūnės. Aišku, nekintanti svorių matrica (arba sąrašas) taip pat turėtų būti paskleista kiekvienam procesui.

Darbas pradedamas nuo pradinės viršūnės A. Aktyvuojamas procesas, atsakingas už A. Šis procesas įvykdys aplinkinių taškų paiešką, tam, kad nustatytu atstumus iki kaimyninių viršūnių.

Atstumas iki viršūnės j bus persiųstas atitinkamam procesui, kad sulyginti jį su einamaja reikšme.



9.19 pav. Išskirstyta paieška grafe [5]

Jeigu perduotas atstumas mažesnis, procesas j turi naujai gautajį atstumą išsisaugoti kaip mažiausią. Tuo būdu, visi mažiausi atstumai iki kaimyninių viršūnių bus atnaujinti. Jeigu atstumas pasikeitė, i-sis procesas bus iš naujo aktyvuotas, kad toliau testų paiešką.

Proceso torno kodas galėtų būti užrašomas taip.

```

recv(newdist, PANY);
if (newdist < dist){
    dist = newdist;
    vertex_queue = TRUE; /* Itraukti į eilę */
} else vertex_queue = FALSE;
if (vertex_queue == TRUE) /* Pradėti paiešką aplink viršūnę */
    for (j = 1; j < n; j++) { /* Gauti sekančią viršūnę */
        if (w[j] != infinity){
            d = dist + w[j];
            send(&d, Pj); /* Pasiusti atstumą procesui j*/
        }
    }
}

```

Programa galėtų būti supaprastinta, taigi supaprastintas kodas tarnui atrodytu:

```
recv(newdist, PANY);
if (newdist < dist)
    dist = newdist; /* Pradėti paiešką aplink viršūnę */
for (j = 1; j < n; j++) /* Gauti sekančią briauną */
    if (w[j] != infinity){
        d = dist + w[j];
        send(&d, Pj); /* Pasiųsti atstumą procedūrai j */
    }
```

Aišku, reikalingas mechanizmas, kuris kartotų iteracijas ir aptiktų globalią baigmę, kreipiant dėmesį į tranzitinius pranešimus.

Paprasčiausiu atveju, galima naudoti synchroninius pranešimus, kai procesas negali testi darbo, kol gavėjas nepatvirtino pranešimo gavimo. Tuo atveju apdorojamas tik aktyvusis viršūnių eilės elementas. Šiuo atveju sprendinys gali būti ne ypač efektyvus, kadangi galimos prastovos.

Jeigu viršūnių skaičius didelis, praktiškas sprendimas turėtų paskirti vienam procesui viršūnių grupę.

10 Lygiagrečiųjų algoritmų taikymai – duomenų rikiavimas

Rikiavimas – tai duomenų (pavyzdžiui, skaičių) išdėstymas tam tikra tvarka (pavyzdžiui, didėjimo).^{*} Nuoseklieji rikiavimo algoritmai yra gerai išstudijuoti. Čia pateiksime lygiagrečias žinomų algoritmų versijas. Pastebėsime, kad „prasti“ nuoseklieji algoritmai gali gerai tiki lygiagrečiajam vykdymui. Be to buvo sukurti specialūs lygiagretieji rikiavimo algoritmai.

10.1 Potencialus spartinimas

Paprastumo dėlei rikiuosime skaičius, nors žemiau pateikti svarstymai tinkta ir kitiems rikiuojamiems duomenims su nustatytu tvarkos sąryšiu. Nenaudojant specialių skaičių savybių galime gauti $O(n \log n)$ sudėtingumo įvertį nuosekliems rūšiavimo algoritmams. Šis įvertis atitinka rūšiavimo sąlaja (*mergesort*) blogiausią atvejį, arba *quicksort* algoritmo – vidurkinį. Taigi, geriausią, ką galime išgauti naudojant n procesorių yra

$$\frac{O(n \log n)}{n} = O(\log n)$$

Teoriškai galima sukonstruoti lygiagretujį algoritmą su duotuoju įverčiu, tačiau įvertyme paslėpta konstanta yra labai didelė (t.y., algoritmas nepraktiškas). Taigi, įvertis - $O(\log n)$ – tikslas, kurį nelengva pasiekti. Be to gali būti, kad tam prireiks procesorių skaičiaus, didesnio negu n.

10.2 Palyginimo ir sukeitimo (Compare-and-Exchange) pavidalo algoritmai

Tai patys žinomiausi klasiniai rikiavimo algoritmai.

Tegu duoti du skaičiai A ir B. Jie sulyginami, ir jeigu A didesnis už B – sukeičiami vietomis.

```
if (A > B) {  
    temp = A;  
    A = B;  
    B = temp;  
}
```

Žemiau nagrinėsime palyginimo ir sukeitimo atvejį, realizuojamą pranešimų siuntimui.

* Skyriuje naudojama [5] medžiaga

1 – sis variantas.

Procesas P1 siunčia skaičių A procesui P2, kuris palygina A su B ir siunčia atgal į pirmajį procesą skaičių B, jeigu A didesnis negu B arba A priešingu atveju.

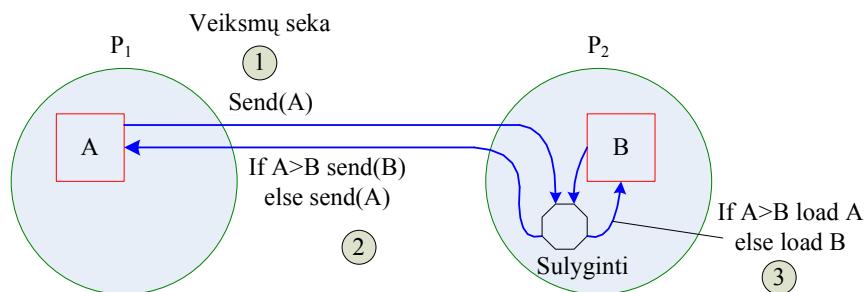
Procesas P1

```
send (&A, P2);
recv(&A, P2);
```

Procesas P2

```
recv(&A, P1);
if (A > B) {
    send (&B, P1);
    B = A;
}
else
    send (&A, P1);
```

Grafiškai tai atrodytu:



10.1 pav. Palyginimas ir sukeitimasis, naudojant pranešimus

2-sis variantas.

Procesas P1 siunčia skaičių A procesui P2, o P2 siunčia B procesui P1 and P2 to send B to P1. Tada abu procesai atlieka sulyginimo operacijas. P1 išsaugo didesnijį iš A ir B, o P2 išsaugo mažesnijį iš A ir B.

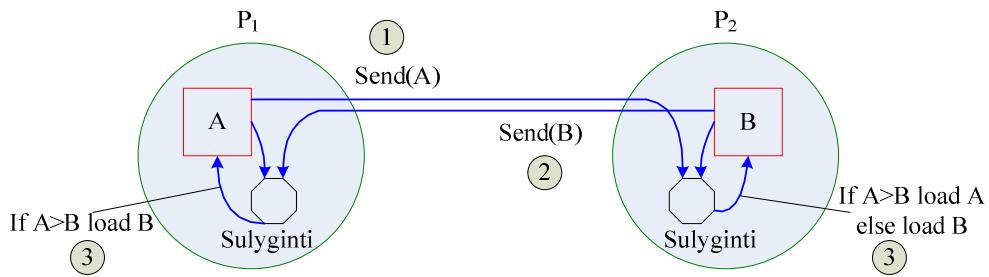
Procesas P1

```
send (&A, P2);
recv(&B, P2);
if (A > B) A = B
```

Procesas P2

```
recv(&A, P1);
send(&B, P1);
if (A > B) B = A;
```

Vaizdžiai:



10.2 pav. Simetriškas palyginimas ir sukeitimasis

Pastebėsime, jog antrajame variante panaudoti pertekliniai skaičiavimai, sutaupant komunikacijos kaštus. Tačiau reikia atsižvelgti, jog remiamasi prielaida, jog $A > B$ reikšmė bus ta pati, skaičiuojant skirtingais procesoriais. Tačiau, kaip žinome, jeigu procesoriai skaičiuoja skirtingu tikslumu, duotoji sąlyga gali būti netenkinama.

Ši situacija atsiranda daugelyje skaičiavimų, kuriuos atliekame, norëdami sutaupyti pranešimų perdavimą arba realizuodami SPMD modelį.

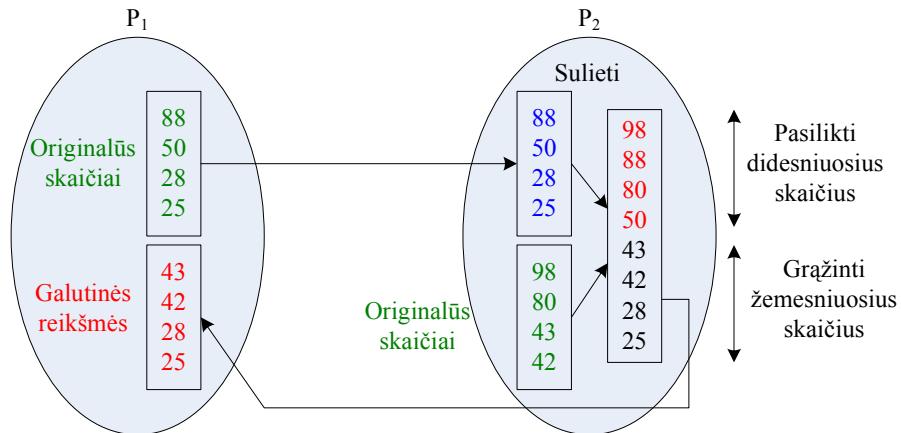
Duomenų skirstymas (partitioning)

Iki šiol nagrinėjome atvejus, kai kiekvienam skaičiui yra priskirtas vienas procesorius. Bendru atveju, procesorių bus mažiau negu skaičių, todėl vienam procesoriui teks apdoroti daugiau skaičių, nors algoritmo struktūra išlieka panaši.

Tegu turime p procesorių ir n skaičių. Kiekvienam procesoriui yra priskirta n/p skaičių.

1-ji versija.

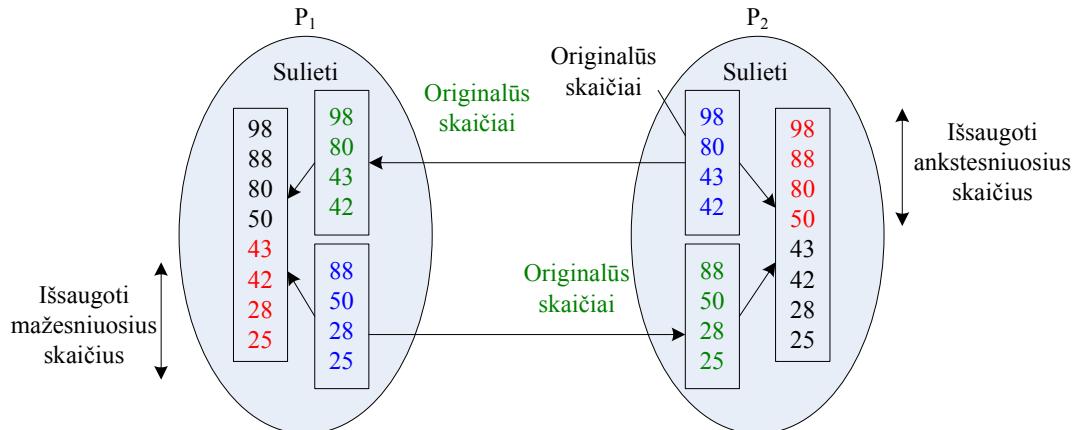
Pirmasis procesas siunčia savo skaičius antrajam, kuris sulieja juos į savo turimą sarašą, išrenka didžiausius, o mažiausius nusiunčia pirmajam procesui.



10.3 pav. Dviejų sąrašų suliejimas - pirmoji versija

Antroji dviejų sąrašų suliejimo versija.

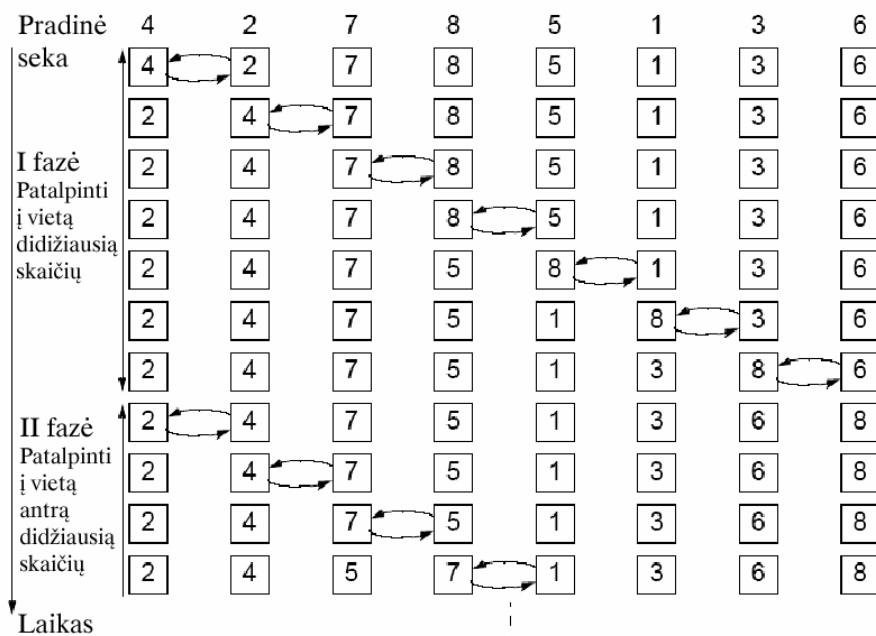
Procesai apsikeičia sąrašais ir nusiunčia vienas kitam atitinkamai didesnę ar mažesniją pusę.



10.4 pav. Dviejų sąrašų suliejimas -antroji versija

10.3 Rūšiavimas burbulais (Bubble Sort).

Rūšiuojant burbulais, atliekami sulyginimai ir sukeitimai tarp gretimų elementų. Pradėjus nuo kairiojo masyvo krašto, didžiausias skaičius yra perstumiamas (burbuliuojamas) į kraštinę dešiniąjā pozicija. Šie veiksmai yra kartojami ($n-1$) kartą, kol visi skaičiai neatsidurs savo vietose.



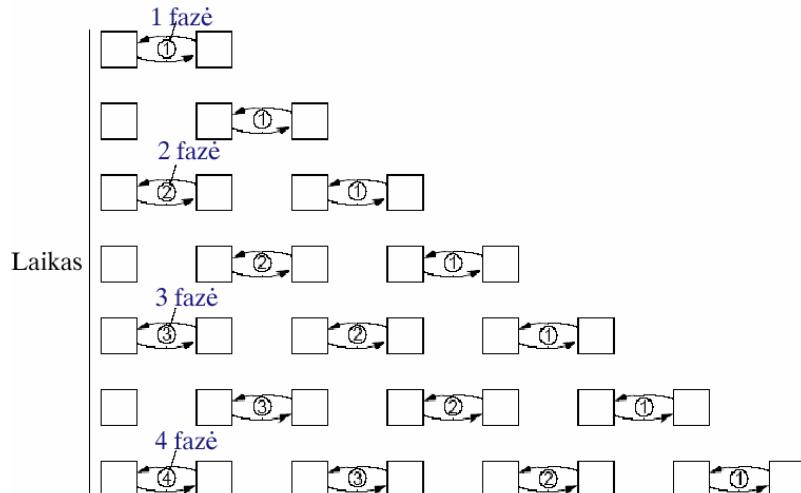
10.5 pav. Rūšiavimo burbulais žingsniai

Skaičiavimų sudėtingumas (sulyginimo operacijų skaičius) yra

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Kitaip tariant, sudėtingumas yra n^2 eilės, laikant vienos operacijos sudėtingumas yra konstantinis O(1).

Lygiagretusis rūšiavimas burbulais. Lygiagretaus darbo sėlyga, - nauja iteracija gali startuoti ankstesniajai dar nesibaigus, tik tuo atveju, jeigu nauja iteracija nepralenkia ankstesniosios.



10.6 Rūšiavimo burbulais persiklojantys veiksmai

Taigi, rūšiuojant galėtų būti panaudojama konvejerio tipo komunikacinė struktūra.

Viena iš efektyviai išlygiagretinamų rūšiavimo burbulais variacijų yra taip vadinamas lyginiai-nelyginiai (*Odd-Even*) rūšiavimo būdas.

Nustatomos dvi besikaitaliojančios rūšiavimo fazės: lyginė ir nelyginė. Lyginėje fazėje lyginiai numerius turintys procesai apsikeičia skaičiumi su dešiniaisiais kaimynais, nelyginėje fazėje – nelyginiai procesai su dešiniaisiais kaimynais. Nuosekliajame programavime, šis rūšiavimo būdas nenaudojamas, nes neturi jokių pranašumų su įprastu rūšiavimu, tačiau lygiagreti realizacija sumažina laiko sąnaudas iki O(n) eilės. Efektyviam algoritmo darbui pakanka tiesinio (žiedinio tinklo), kuris yra optimalus šiam algoritmui.

Žemiau pateikiama 8-nių skaičių rūšiavimo schema.

Žingsnis	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Laikas	0	4 ←→ 2	7 ←→ 8	5 ←→ 1	3 ←→ 6			
	1	2	4 ←→ 7	8 ←→ 1	5 ←→ 3	6		
	2	2 ←→ 4	7 ←→ 1	8 ←→ 3	5 ←→ 6			
	3	2	4 ←→ 1	7 ←→ 3	8 ←→ 5	6		
	4	2 ←→ 1	4 ←→ 3	7 ←→ 5	8 ←→ 6			
	5	1	2 ←→ 3	4 ←→ 5	7 ←→ 6	8		
	6	1 ←→ 2	3 ←→ 4	5 ←→ 6	7 ←→ 8			
	7	1	2 ←→ 3	4 ←→ 5	6 ←→ 7	8		

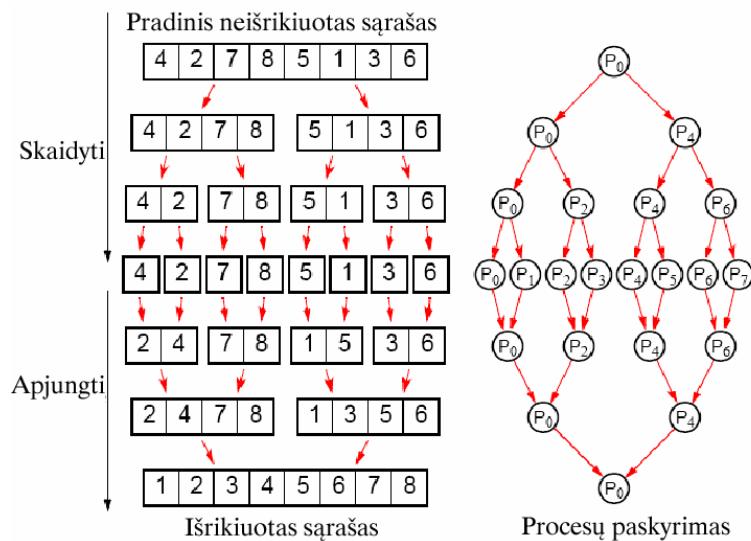
10.7 pav. Rikiavimas odd-even metodu

10.4 Rikiavimas sąlaja (mergesort)

Tai klasikinis nuoseklusis algoritmas, naudojantis skaldyk ir valdyk principą. Pradinis sąrašas skaidomas į dvi dalis. Gautos dalys skaidomos toliau, kol kiekviena dalis susideda tik iš vieno skaičiaus.

Tada prasideda sekanti algoritmo fazė – apjungimas. Kiekviena skaičių pora apjungiama į sutvarkytą porą. Poros apjungiamos į surikiuotą keturių elementų sąrašą ir t.t. Galiausiai gaunas pilnas surikiuotas sąrašas.

Žemiau pateikta aštuonių elementų masyvo sąrašo rikiavimas sąlaja.



10.8 pav. Rikiavimas sąlaja, naudojant medžio struktūros procesus

Pastebėsime, kad visi procesai užimti tik viename iš šešių žingsnių.

Algoritmo darbo laikas nuosekliu atveju yra $O(n \log n)$.

Lygiagrečiu atveju, reikės atlikti $2 \log n$ žingsnių, tačiau kiekviename žingsnyje gali tekti atlikti daugiau negu vieną primityvią operaciją. Komunikacijos kaštai sudarys $O(\log p + n)$, o skaičiavimų (kurie vykdomi tik suliejimo fazėje) – $O(p)$. Čia atsižvelgiant į tai, jog vienam procesoriui sulieti dvi n ilgio sekas gali kainuoti (blogiausiu atveju) $2n+1$ operacijų. Taigi bendras ivertinimas – $O(p+n)$ operacijų.

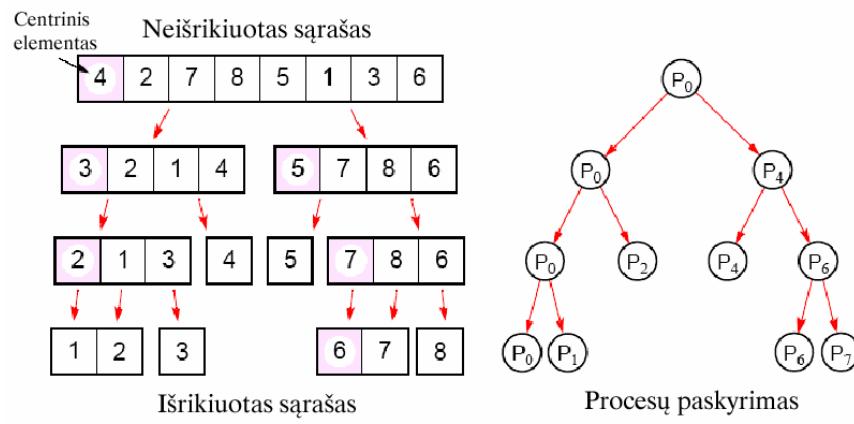
10.5 Quicksort rikiavimas

Tai labai populiarus nuoseklusis algoritmas turintis labai gerą vidurkinį sudėtingumo ivertį – $O(n \log n)$. Klausimas, ar gali lygiagrečioji versija pasiekti $O(\log n)$ ivertį. Deja, *quicksort* algoritmas, kaip ir sėlajos algoritmas nepasižymi šia savybe.

Prisiminkime quicksort algoritmą. Pradžioje pradinis sąrašas skirstomas į du posarašius taip, kad visi pirmojo sąrašo elementai būtų mažesni negu antrojo sąrašo. Tai pasiekiamas, pradžioje išrinkus

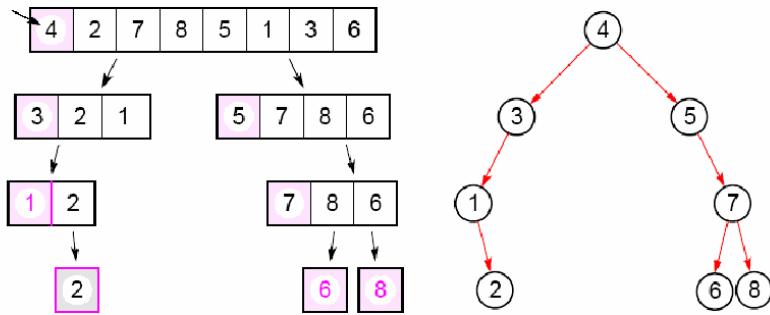
centrinį elementą (*pivot*), su kuriuo lyginami pradinio sąrašo elementai. Visi mažesni eina į pirmąjį sąrašą, didesni į antrąjį. Centriniu elementu, gali būti bet kuris pradinio sąrašo elementas, bet dažnai išrenkamas pirmasis. Sąrašų skirstymo į sutvarkytus posarašius kartojama tol, kol posarašiai susideda tik iš vieno elemento. Galiausiai, visi posarašiai susideda tik iš vieno elemento – taigi, išsaugant posarašių tvarką vienas kito atžvilgiu, gausime galutinį išdėstymą. Nuoseklusis kodas paprastai vykdo sąrašų skaidymą rekursyviai.

Lygiagretusis algoritmas naudoja medžio pavidalo struktūrą, kaip parodyta paveikslėlyje.



10.9 pav. Quicksort, naudojantis medžio pavidalo procesų struktūrą

Naudoti centriniai elementai pavaizduoti žemiau.



10.10 pav. Quicksort rikiavimas išskiriant centrinius elementus

Galutinis išrūšiuotas sąrašas gaunamas apeinant centrinius elementus reikiama tvarka.

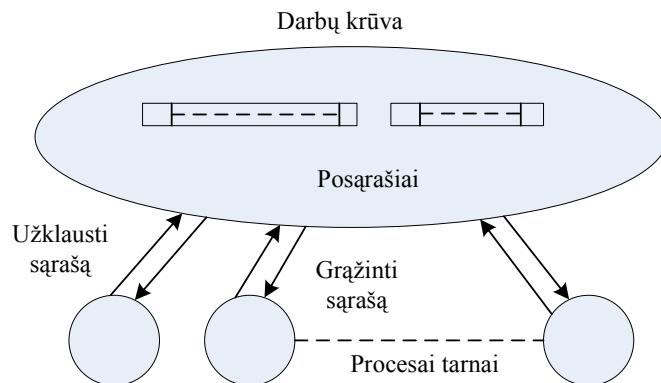
Fundamentali problema, su kuria susiduriame konstruodami pateiktąjį medį yra pradinis skirstymas, kurį atlieka vienintelis procesas, kas labai riboja spartinimą.

Skaičiavimų laikas: $n + n/2 + n/4 + \dots + 1$. Taigi, skaičiavimų sąnaudos vertinamos $O(n)$.

Bendru atveju, medis gali būti nesubalsuotas, taigi rezultatai gali tik pablogėti.

Quicksort realizacija darbų krūva.

Geresnį balansuotumą būtų galima pasiekti naudojant darbų krūvos principą. Pradinis sąrašas patalpinamas į darbų krūvą. Procesas, suskirstęs pradinį sąrašą į dvi dalis, vieną grąžina į krūvą, kitą – apdoroja pats. Darbų krūvos panaudojimas neleidžia visiškai išvengti procesorių prastovų, tačiau gali akomoduoti situaciją, kai vieni darbai yra atliekami ilgiau negu kiti.



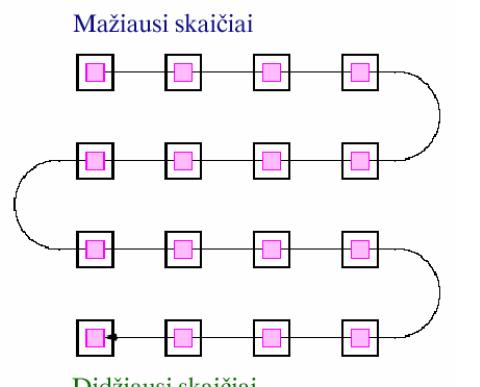
10.11 pav. Quicksort realizacija darbų krūva

10.6 Rikiavimas specializuotuose tinklų topologijose

Algoritmai gali pasinaudoti tinklo topologijų teikiama nauda. Atkreipsime dėmesį į du tinklus – tinklą (mesh) ir hiper-kubą. Nors tinklo architektūros paprastai yra paslėptos nuo programų kūrėjų, tačiau MPI pateikia priemones leidžiančias atvaizduoti turimą tinklą į tinklą ar hiperkubą, taigi naudoti žemiau aptariamus algoritmus.

10.6.3 Rikiavimas dviejų dimensijų tinkle.

Galutinis elementų išsidėstymas tokiam tinkle gali būti eilutėmis, ar „gyvatėlės“ pavidalu, kaip pateikta žemaiu.

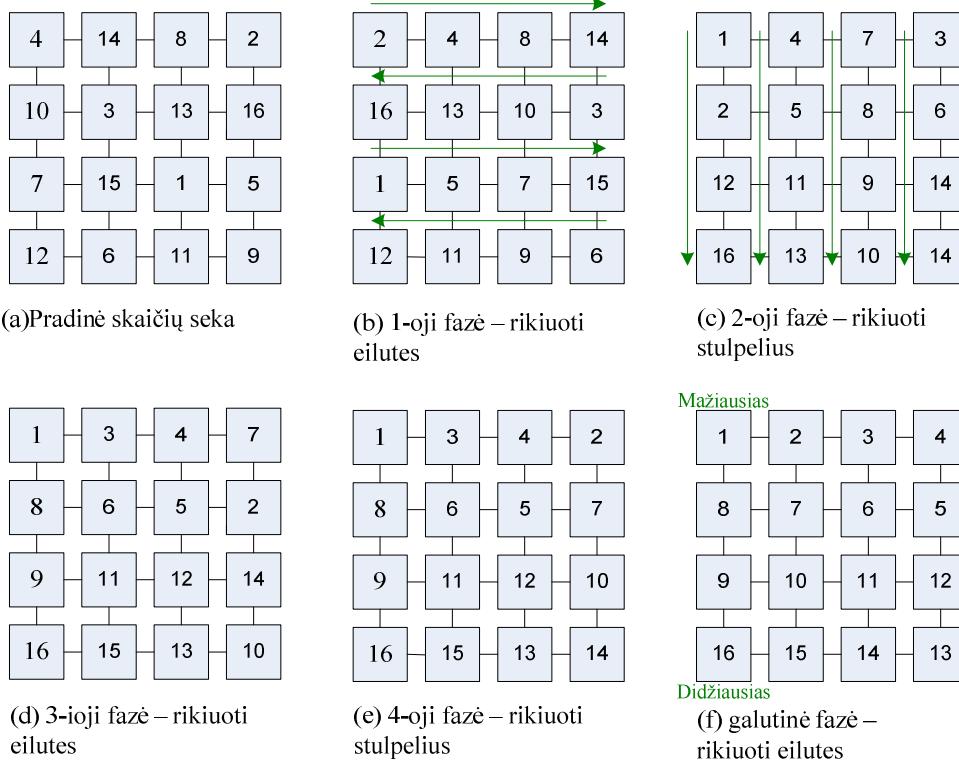


10.12 pav. Galutinis elementų išdėstymas tinkle

10.6.4 Shearsort – rikiavimas.

Pakaitomis vykdomas eilučių ir stulpelių rūšiavimas, kol nebus gautas išrūšiuotas sąrašas.

Eilučių rikiavimas vykdomas kaitaliojant kryptis (gyvatėlės forma).



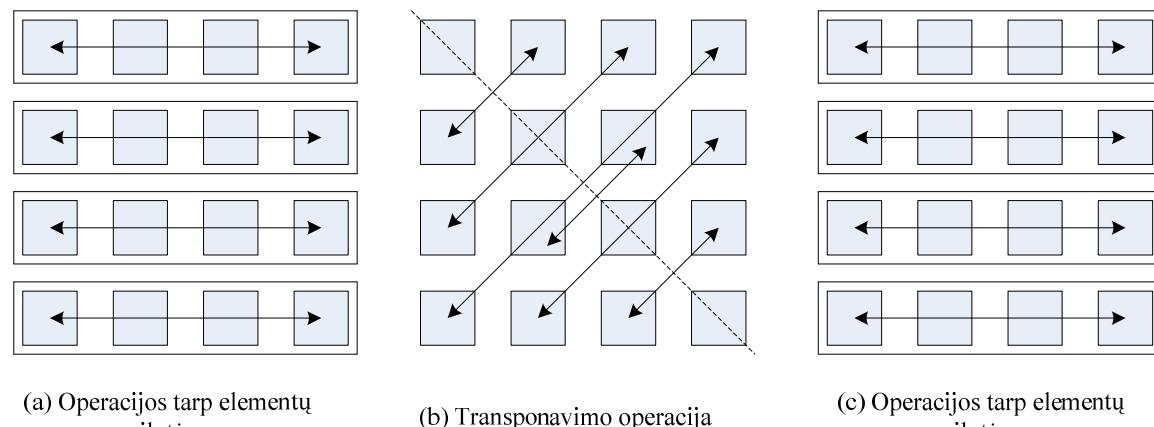
10.13 pav. Shearsort – rikiavimas

Turint n skaičių prireiks $\sqrt{n}(\log n + 1)$ žingsnių, kad atlikti rikiavimą

$\sqrt{n} \times \sqrt{n}$ dydžio tinkle.

Lygiagrečiojoje realizacijoje prireiks atlikti transpozicijas (eilučių sukeitimą su stulpeliais).

Transpozicijas galima įgyvendinti pateiktaja schema.



10.14 pav. Sukeitimų atlikimo schema

Transpozicijos gali būti įgyvendintos, naudojant o(n) komunikacijų. Ši skaičių galima dar labiau sumažinti, naudojant visi-su-visais (all-to-all) MPI duomenų apsikeitimo operaciją.

10.7 Algoritmai hiperkubo tinkle:quicksort

Hiperkubas suteikia struktūrines charakteristikas, leidžiančias efektyviai realizuoti skaldyk ir valdyk pavidalo rikiavimo algoritmus, tokius kaip *quicksort*.

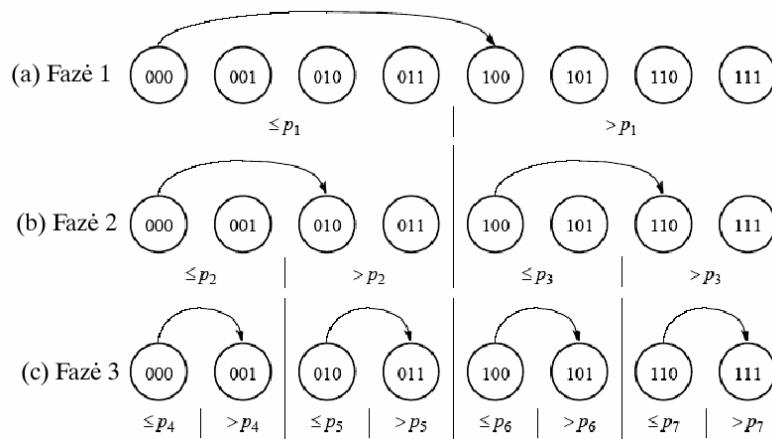
Tarkime, turime n skaičių sąrašą, kuris patalpintas d -dimensijų hiperkubo viršūnėje. Sąrašas gali būti suskirstytas į dvi dalis, naudojant quicksort algoritmą. O viena iš dalių gali būti pasiūsta gretimai viršūnei aukščiausio matavimo kryptimi.

Vėliau dvi viršūnės gali testi šį procesą.

Pavyzdys. Tegu pradinis sąrašas yra patalpintas 3-d kubo viršūnėje 000.

- 1-sis žingsnis 000 -> 001 (skaičiai didesni negu centrinis elementas, pvz., c1)
- 2-sis žingsnis 000 -> 010 (skaičiai didesni negu centrinis elementas, pvz., c2)
 - 001 -> 011 (skaičiai didesni negu centrinis elementas, pvz., c3)
- 3-sis žingsnis 000 -> 100 (skaičiai didesni negu centrinis elementas, pvz., c4)
 - 001 -> 101 (skaičiai didesni negu centrinis elementas, pvz., c5)
 - 010 -> 110 (skaičiai didesni negu centrinis elementas, pvz., c6)
 - 011 -> 111 (skaičiai didesni negu centrinis elementas, pvz., c7)

Galiausiai, atskirose dalys gali būti surūšiuotos, naudojant nuoseklujį algoritmą, visos vienu metu. Jeigu reikia, išrūšiuotos dalis gali būti apjungtos į galutinį surikiuotą sąrašą.



10.15 pav. Rikiavimas hiperkube

10.8 Kiti rūšiavimo algoritmai

Kaip žinome, apatinis nuoseklaus rikiavimo, kuris remiasi elementų sulyginimu, rėžis yra $O(n \log n)$.

Atitinkamai – lygiagretaus algoritmo apatinio rėžio įvertinimas - $O((\log n)/p)$, turint p procesorių arba $O(\log n)$, naudojant n procesorių.

Egzistuoja rikiavimo algoritmai, kurie įgalina geresni negu $O(n \log n)$ įvertį, tačiau pastarieji remiasi specialiomis rikiuojamų skaičių savybėmis.

Pradžioje išnagrinėkime vieną rūšiavimo rangais algoritmą (*rank sort*), kuris nepasiekia optimalaus įverčio $O(n \log n)$, tačiau gali būti lengvai išlygiagretinamas ir sėkmingai taikomas kompiuterių klasteriuose.

10.8.1 Rūšiavimas rangais

Esmė, kiekvienam pateiktajam skaičiui yra nustatomas už jį mažesnių skaičių kiekis. Šis kiekis nusako kiekvieno skaičiaus poziciją galutiniame sąraše – rangas.

Taigi, nuskaitome skaičių $a[0]$ ir sulyginame su visais kitais skaičiais $a[1], a[2], \dots, a[n-1]$. Tegul mažesniųjų negu $a[0]$ kiekis yra x. Vadinas x – skaičiaus $a[0]$ galutinė pozicija išrikiuotame sąraše. Nukopijuojame $a[0]$ į x-ają poziciją sąraše $b[0], b[1], \dots, b[n-1]$. Analogiški veiksmai pakartojami su likusiais elementais $a[1], a[2], \dots, a[n-1]$.

Bendras nuoseklaus rūšiavimo laikas yra $O(n^2)$ operacijų (prastas rezultatas).

Nuoseklusis kodas:

```
for (i = 0; i < n; i++) /* kiekvienam skaičiui */
    x = 0;
    for (j = 0; j < n; j++) /* gauti mažesnių kiekį */
        if (a[i] > a[j]) x++;
    b[x] = a[i]; /* nukopijuoti į reikiama poziciją */
}
```

Pateiktasis kodas neveiks, jeigu sekoje bus pasikartojantys skaičiai, tačiau kodą nesunkiai galima pritaikyti ir šiam atvejui.

Lygiagretusis kodas, naudojantis n procesorių.

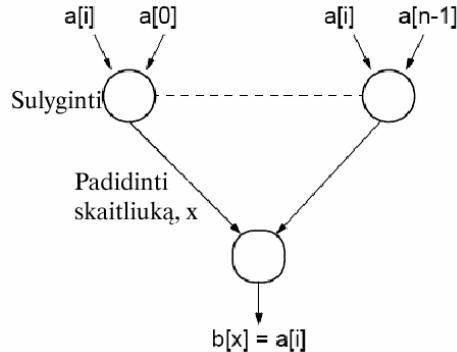
Kiekvienam skaičiui priskirtą po vieną procesorių Galutinį indeksą kiekvienas procesorius randa per $O(n)$ žingsnių, taigi dirbant isiemis lygiagrečiais sudėtingumas taip pat lygus $O(n)$.

Naudojant *forall* konstrukciją kodas galėtų atrodyti taip:

```
forall (i = 0; i < n; i++) /* kiekvienam skaičiui lygiagrečiai*/
    x = 0;
    for (j = 0; j < n; j++) /* suskaičiuoti mažesnių skaičių kiekį */
        if (a[i] > a[j]) x++;
    b[x] = a[i]; /* nukopijuoti į reikiama vietą */
}
```

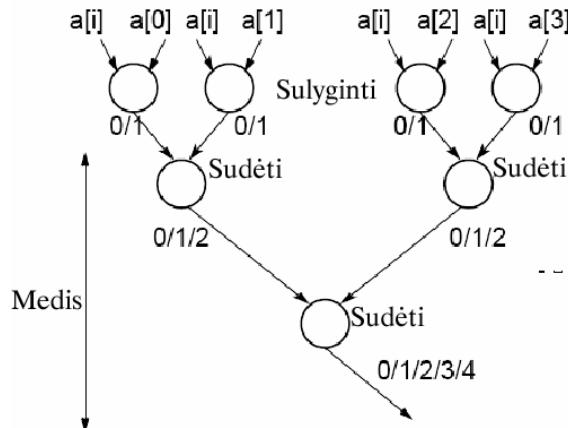
Gautasis įvertis $O(n)$ yra toks pat geras kaip ir ankstesniu išnagrinėtu algoritmu. Ši įverti galime dar labiau pagerinti, jeigu turime daugiau procesorių.

Tegu turime $n*n$ procesorių. Kiekvienam skaičiui lyginti su kitais galime panaudoti $(n-1)$ procesorių:



10.16 pav. Skaičių sulyginimas, naudojant po procesorių

Taigi, turint n skaičių prireiks $(n-1)n$ procesorių. Skaitliuko didinimas gali būti atliktas tiesiskai per n žingsnių, arba log n žingsnių, naudojant medžio pavidalo struktūrą:



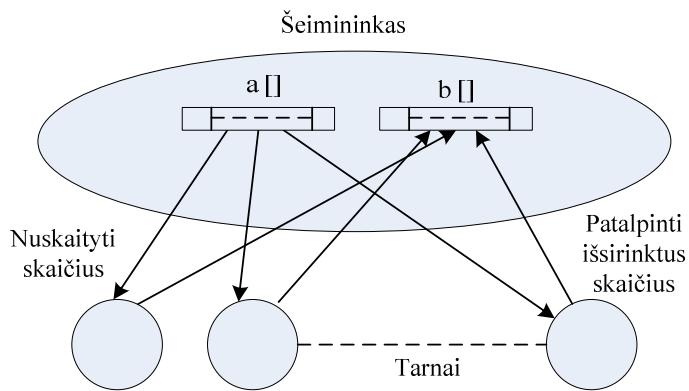
10.17 pav. Skaičiaus rango nustatymas medžio pavidalo struktūra

Vadinasi, galutinis algoritmo įvertis yra $O(\log n)$, tačiau procesorių panaudojimo efektyvumas yra menkas.

Teoriškai galima sumažinti algoritmo sudetingumą iki $O(n)$ traktuojant, kad visos skaitliuko didinimo operacijos vyksta nepriklausomai.

Rūšiavimo rangais įgyvendinimas, naudojant pranešimų sistemas.

Šeimininko tarno metodas:



10.18 pav. Rūšiavimas rangais, naudojant šeiminingo-tarno schemą

Reikalauja, kad kiekvienas procesas pasiektų skaičių sąrašą. Geriau tinkamas įgyvendinti bendrosios atminties sistemoje.

10.8.2 Rikiavimas išskaičiuojant (*counting sort*)

Jeigu rikiuojami elementai yra skaičiai, egzistuoja rikiavimo rangais metodo patobulinimas, susijęs su skaičių kodavimu, leidžiantis sumažinti algoritmo sudėtingumą nuo $O(n^2)$ iki $O(n)$. Naujasis metodas vadinamas *counting sort*. Skaičiavimų rikiavimas yra stabilus rikiavimo algoritmas, t.y., vienodi skaičiai išlaiko tą pačią tvarką galutiniame sąraše kaip ir pradiniame.

Sakykime, kad skaičiai, kuriuos reikia rikiuoti, yra masyve a , o rezultatą reikia patalpinti i masyvą b . Algoritmas naudoja papildomą masyvą c , kurio ilgis – galimų pradinių skirtingu elementų masyve a skaičius. Pavyzdžiui, jeigu pradiniai skaičiai priklauso intervalui $1..m$, tai masyvo c ilgis lygus m . Algoritmas turi tokius etapus.

1 etapas. Pradžioje, masyvas c saugo pradinio skaičių sąrašo pasikartojimų dažnį, t.y., elementų histogramą. Ją galima suskaičiuoti per $O(m)$ žingsnių, naudojant kodą:

```
for (i = 1; i <= m; i++)
    c[i] = 0;
for (i = 1; i <= m; i++)
    c[a[i]]++;
```

2 etapas. Skaičių kiekis, mažesnis nei kiekvienas duotasis skaičius, yra randamas naudojant prefiks-sumų operaciją masyvui c . Kitaip tariant, skaičiuojamos visos galimos sumos pavidalo: $c[0]$, $c[0]+c[1]$, $c[0]+c[1]+c[2]$, ... Histogramai, saugomai masyve c , sumos gaunamos per $O(m)$ žingsnių, naudojant procedūrą:

```
for (i = 2; i <= m; i++)
```

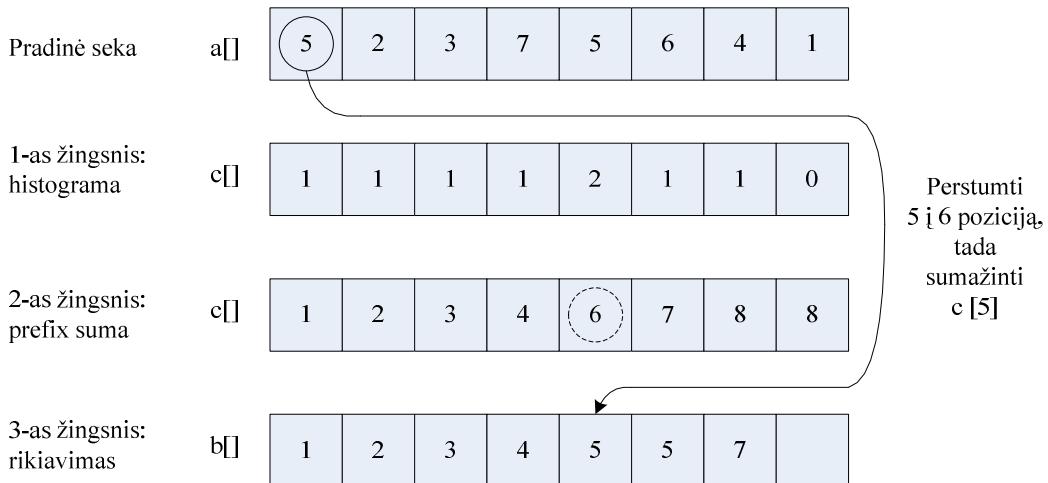
```
c[i] = c[i] + c[i-1];
```

Galutinis etapas. Masyvas b užpildomas surikiuotomis reikšmėmis iš masyvo a, naudojant procedūrą:

```
for (i = n; i >= 1; i--) {
    b[c[a[i]]] = a[i]
    c[a[i]]--; /* kad užtikrinti stabilų rikiavimą */
}
```

Visos procedūros sudėtingumas yra eilės $O(n+m)$. Jeigu m yra tiesiskai susijęs su n, vadinasi galutinis nuosekliosio procedūros sudėtingumas $O(n)$.

Grafiškai procedūrą būtų galima pavaizduoti:



10.19 pav. Rikiavimo procedūra

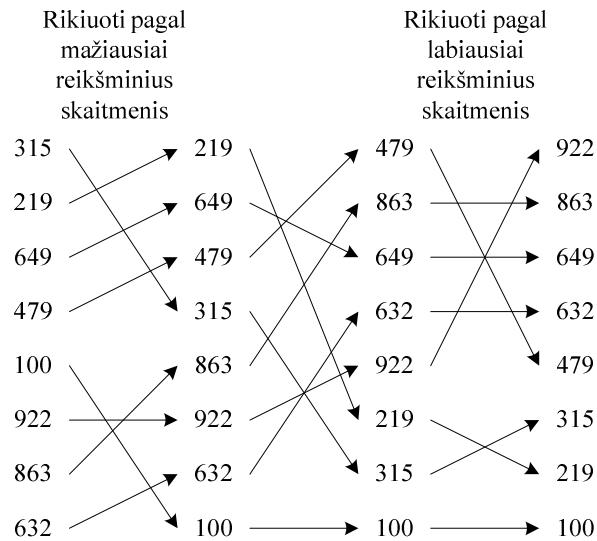
Lygiagrečioji realizacija gali remti lygiagrečiuoju prefiks sumų skaičiavimo algoritmu, kurio sudėtingumas – $O(\log n)$ žingsnių su $(n-1)$ procesorių. Galutinė rūšiavimo fazė gali būti pasiekta per $O(n/p)$ laiką su p procesorių arba $O(1)$ laiką su n procesorių, kai aukščiau pateikto ciklo kūną vykdo skirtini procesoriai vienu metu.

10.8.3 Pozicinė rikiavimas (radix sort)

Šiame metode numatoma, jog rikiuojami skaičiai pateikti skaitmenimis poziciniu skaičiavimo pavidalu, pavyzdžiui, dešimtainiu ar dvejetainiu. Skaitmens pozicija reiškia jo svarbą rikiavime.

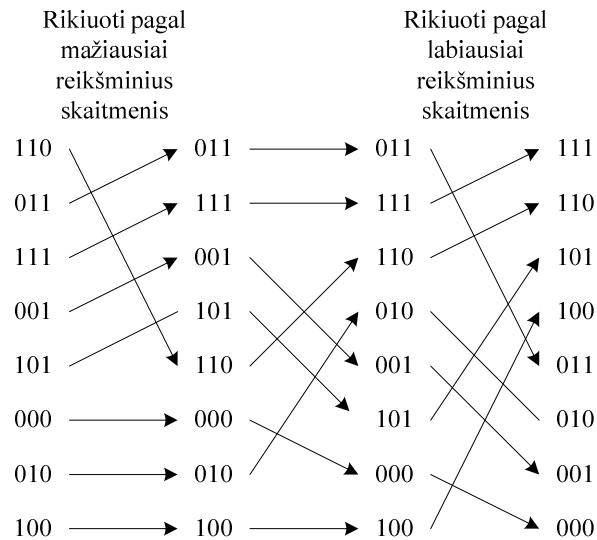
Rikiujant poziciniu būdu, pradžioje išrikuojami skaičiai pagal mažiausiai reikšminį skaitmenį, po to gauta seka rūšiuojami pagal antrosios pozicijos reikšmę. Galiausiai surikiujama pagal vyriausiąjį, labiausiai reikšminį skaitmenį. Rikiuojanr svarbu išlaikyti pradinę skaičių tvarką visiems skaičiams su lygiu skaitmeniu toje pačioje pozicijoje, t.y., algoritmas turi būti stabilus.

Naudojant dešimtainius skaičius algoritmas gali atrodyti taip:



10.20 Pozicinis (dešimtainis rikiavimas)

Dvejetainio rikiavimo pavyzdys:



10.21 pav. Dvejetainio rikiavimo pavyzdys

Pozicinis lygiagretėsis rikiavimas gali būti įgyvendintas, naudojant lygiagrečiuosius rikiavimo algoritmus kiekvienoje fazėje, rikiujant skaičius pagal tam tikrą bitą ar bitų grupę. Pavyzdžiu, galėtume panaudoti *counting sort* algoritmą su prefiksiniu sumų skaičiavimu.

11 Literatūra

1. H.M.Deitel. Operating systems. Addison-Wesley, 1990.
2. Alwyn Barry. Concurrent Programming. <http://www.cs.bath.ac.uk/~amb/UQC011H2/concprog.html> (2007.05.01)
3. Ruediger R. Asche. Detecting Deadlocks in Multithreaded Win32 Applications. MSDN library, 1994.
4. POSIX Threads Programming. <http://www.llnl.gov/computing/tutorials/pthreads/> (2007.05. 01)
5. B.Wilkinson, M.Allen. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (Prentice Hall 2004 2nd Edition).
6. Gregory R. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming Addison-Wesley, 2000.
7. I.Foster. Designing and Building Parallel Programs. Addison-Wesley, 1995 (Prieinama ir *online*)
8. B. Nichols et al. "Pthreads Programming". O'Reilly and Associates, 1996
9. Bill Lewis (Author), Daniel J. Berg. Threads Primer: A Guide to Multithreaded Programming. 1996.
10. R.Čiegeis. Lygiagretieji algoritmai ir tinklinės technologijos. Technika, Vilnius, 2005
11. Claudia Leopold .Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches. 2000.
12. Ian Foster, Carl Kesselman. The Grid 2: Blueprint for a New Computing Infrastructure.2003
13. Hagit Attiya, Jennifer Welch. Distributed Computing (Second Edition). Wiley & Sons, 2004
14. Cameron Hughes, Tracey Hughes . Parallel and Distributed Programming Using C++. Addison Wesley , 2003.
15. George Coulouris, Jean Dollimore, Tim Kindberg. Distributed Systems : Concepts and Design (4th Edition). Addison Wesley ,2005
16. http://www.accelerating.org/acc2003/moores_law.htm (2006.05.01)
17. <http://archive.computerhistory.org/resources/> (2006.05.01)

12 Priedas. Laboratorinių darbų pavyzdys

12.1 Kritinės sekcijos problemos raiška

Darbo tikslas - suvokti kritinės sekcijos (KS) problemą, sugebėti iššaukti negatyvius padarinius ir išspręsti KS problemą konkrečiam modeliniam taikymui.

Trukmė: 6 valandos.

Priemonės: Java programavimo aplinka arba aplinka, leidžianti operacijų sistemų gijų posistemui (WIN32, POSIX) funkcijų kvietinius. Pageidautina naudoti kelių procesorių sistemas.

Užduoties formulavimas.

Sugalvoti probleminę situaciją, kurioje keli procesai naudoja resursą/resursus. Programa, imituojanti šių procesų darbą, turi demonstruoti (pirmasis darbo režimas), jog nesynchronizuota kelių gijų manipuliacija resursu (pavyzdžiu, kelioms gijoms prieinamu kintamuoju, kurio reikšmę gijos nuskaito / atnaujina) gali iššaukti nekorekтиšką programos darbo rezultatą.

Kitas, t.y., sinchronizuotas programos veikimo režimas (valdomas komandinės eilutės parametru bei naudojantis synchronized Java metodus arba semaforus/mutex'sus C,C++ aplinkose) turi užtikrinti teisingą programos veikimą. Programos komentare trumpai aprašyti dalykinę sritį bei problemą. Vizualiai (komentarais) išskirti kode kritines sekcijas. Siekiant padidinti programos nedeterminizmą leidžiama įtraukti atsitiktinius gijų vykdymo užlaikymus (Thread.sleep(...)), ar vykdomumo iniciatyvos atidavimą, tačiau, tik ištyrus, jog be užlaikymų programa nepakliūna į nepageidaujamas būsenas.

Problemu pavyzdžiai.

- 1) Bendras resursas - ekranas į kurį dvi gijos išveda žodžius paraidžiui. Nesant sinchronizacijos ekrane matome raidžių kratinę. Sinchronizuojant gijas išvedimo problemą išsprendžiame korekтиškai.

2) Bendras resursas - dvi sąskaitos. Keli procesai vykdo pervedimus iš vienos sąskaitos į kitą. Nekorektiškas programos darbo rezultatas - pakitęs suminis pinigų likutis sąskaitose.

1 pastaba. Nebus laikoma tinkama tokia programa, kuri iliustruoja gijų vykdymo nedeterminizmą, bet negalime nurodyti koks deterministinis vyksmas yra teisingas.

2 pastaba. Nerašyti sudėtingų programų, kuriose procesai bendrauja tarpusavyje, naudodami ivykių mechanizmą (`wait()`, `notify()`).

12.2 Gijas sinchronizuojančių objektų apibrėžtis

Tikslas. Išisavinti klasikines procesų / gijų synchronizavimo konstrukcijas, aptartas paskaitose, mokėti jas panaudoti modelinėje situacijoje, sugebėti konkrečiame taikyme atskirti procesų sąveikos aspektus nuo probleminių aspektų. Sugebėti kelių procesų sąveiką apibrėžti kaip java klasę (ar kitokį struktūrinį darinį) bei tinkamai ją panaudoti.

Trukmė: 8 valandos

Priemonės. Java aplinka arba C++ su POSIX gijomis.

Formulavimas. Išnagrinėti modelinę situaciją. Apibrėžti java klasę arba klases nusakančias esminę sąveikavimo tarp keletos (gijų) konstrukciją. Ivykių laukimus realizuoti `wait()` šeimos kreipiniai, o pranešimą apie ivykį - `notify/all()`. Neleistina panaudoti jau egzistuojančias (standartines) konstrukcijas (pvz., iš `util.concurrent` bibliotekos). Programa į ekraną turėtų išvesti darbo protokolą, kuris turėti pagrįsti teisingą programos veikimą.

Konkreči situacija pateikta žemiau išvardintuose variantuose.

Pastabos. Gijų saveikai užtikrinti nenaudoti išreikštino gijų veikos sustabdymo ir pratėsimo operacijų (`suspend()`, `release()`). Nesudaryti išreikštinių gijų eilių. Atkreipti dėmesį ne tik į programos teisingumą (programos būsenų terminais) bet ir gijų vykdomumo galimybę.

VARIANTAI

1. Semaforas kaip resursų skaitliukas.

Semaforo objektas inicijuojamas turimų resursų skaičiumi. Operacijos:

`request()` - laukia, kol resursas atsilaisvins,

<code>release()</code>	- grąžinamas resursas,
<code>numberAvailable()</code>	- grąžinamas laisvų resursų skaičius (nesiblokuojant).

2. Barjeras.

Objektas inicijuojamas "sąveikoje" dalyvaujančių gijų skaičiumi (N). Pagrindinė operacija - `waitBarier()`. Kiekviena ši metodą iškvietusi giją laukia, kol kvietėjų skaičius taps lygus N. Tada visos N gijos "paleidžiamos" o barjeras inicijuojamas iš naujo - paruošiamas sekančiam `waitBarier()` kvietimui.

3. Klasikinis "skaitytojų - rašytojų" veikos synchronizatorius.

Vienu metu gali būti tik vienas aktyvus rašytojas arba keli aktyvūs skaitytojai. Nustatomas protokolas: kiekvienas skaitytojas kreipiasi į synchronizatorių „pradēti skaityti“. Kai kreipinys grįžta,- laikome, jog turime aktyvų skaitytoją. Baigęs skaitymo veiksmus, skaitytojas įvykdo kreipinį „baigtis skaityti“ ir tampa neaktyviu. Analogiškos rašytojų operacijos – pradēti/baigtis rašyti. (*Pastaba* - nepainioti šio uždavinio su gamintojo-vartotojo problema).

4. Skaitytojų-rašytojų problemos variacija,

kai aktyvių "skaitytojų" skaičius negali viršyti N – programos parametras.

4. "Baltujų - Raudonujų" gijų veikos synchronizatorius.

Vienu metu leidžiama būti "aktyvioms" tik vienos spalvos gijoms (klasikinės "skaitytojų-rašytojų" problemos variacija).

5. "Rytų - Vakarų" gijų veikos synchronizatorius. -

klasikinės "skaitytojų-rašytojų" problemos variacija. Vienu metu leidžiama būti "aktyvioms" tik vienos krypties gijoms, bet ne daugiau negu N - programos parametras. Palyginimui situacija - vienos eiles automobiliu tiltas, kuriuo mašinos gali važiuoti viena kryptimi.

6. Žiedinis buferis, kaip duomenų perdavimo kanalas tarp dviejų gijų.

Objektas panaudojamas perduodant duomenis tarp dviejų gijų. Skaitanti gija turi blokuotis, jeigu kanalas tuščias. Rašančioji turi blokuotis – jeigu kanalo buferis pilnai užpildytas. Sinchronizacijos esmė – užtikrinti, kad duomenis, kuriuos siunčia gamintojas, ta pačia tvarka nuskaitytų vartotojas.

Tegu duomenys - sveikieji skaičiai. Buferio ilgis - parametras. Pademonstruoti,

kaip įėjimo duomenys, kurie nuskaitomi kaip įėjimo failo baitų reikšmės perduodami kitai gijai, kuri surašo duomenis į išėjimo failą.

7. Kanalas – skirstytuvas.

Žiedinis buferis, kaip pranešimo perdavimo kanalas tarp vienos gijos - siuntėjo, bei N (parametras) gijų gavėjų. Kiekvienas pranešimas laikomas nuskaitytu, jeigu jį nuskaitė VISOS gijos - gavėjos. Tegu realizacijoje pranešimo objektas būna String tipo.

8. Kanalas - surinktuvas.

Dalyvauja dvi gijos: siuntėjas ir gavėjas. Gija siuntėjas siunčiu numeruotus pranešimus. Kanalas turi surinkti pranešimus, kad gavėjas juos nuskaitytų nuoseklia (numeriu didėjimo) tvarka.

9. Klasikinis įvykių skaitiklis.

Inicijuojamas 0. Turi funkcijas

`advance()` - nedalomai padidinti skaitiklį vienetu;
`read ()` - nuskaityti skaitiklio reikšmę;
`await (value)` - laukti, kol skaitliuko reikšmė nesusilygins su `value`.

Pateikti skaitiklio prasmingo panaudojimo situaciją.

9. Masyvo "užraktas" turi operacijas

`lock (String name, int indexFrom, in indexTo),`
`unlock (String name).`

Metodas `lock()`, užrakina masyvo elementus nuo `indexFrom` iki `indexTo` imtinai. Jeigu bent vienas iš elementų jau "užrakintas" kitos gijos, kreipinys `lock()` blokuojasi. `Unlock()` - atlaisvina užrakintus elementus susietus su vardu "name". Realizuoti užraktą bei prasmingai panaudoti (pavyzdžiui, rikiuojant masyvo elementus)

10. Barjeras - su duomenų apsikeitimu

Objektas inicijuojamas "sąveikoje" dalyvaujančių gijų skaiciumi (N). Pagrindinė operacija -
`int waitBarier(int reikšmė).`

Kiekviena ši metodą iškvietusi giją laukia, kol kvietėjų skaičius taps lygus N. Tada visos N gijos "paleidžiamos" - metodas gražina asociatyvios operacijos (pvz. sumos) su visomis

"reikšmėmis" rezultatą - o barjeras inicijuojamas iš naujo, t.y., paruošiamas vėlesniams waitBarier() kvietiniui.

12.3 Lygiagretus sprendimas bendrosios atminties sistemių

Tikslias. Suprojektuoti ir realizuoti pateiktajai užduočiai (problemai) efektyvų lygiagretujį sprendimą bendrosios atminties architektūroje. Sugebėti identifikuoti programos „siauras vietas“, kuriose galimos procesorių prastovos.

Trukmė. 8 valandos.

Priemonės. Programavimo aplinka – bet kuri aplinka, įgalinanti paleisti ir sinchronizuoti gijas (java, C, C++ naudojant Win32/POSIX gijas). Techninė aplinka – daugelio procesorių (branduolių) sistemas.

Formulavimas. Suprojektuoti pateiktajai užduočiai (problemai) **efektyvų** lygiagretaus vykdymo algoritmą ir jį realizuoti išreikštinėmis JAVA/POSIX gijomis. "Darbinių" gijų skaičius - programos parametras. Jos "paleidžiamos" vykdymui programos pradžioje ir funkcionuoja iki pat programos baigmės (kol užduotyje specifikuotas darbas nebus atliktas). Sąveika tarp atskirų gijų, aišku, turi būti tokia, kad jos būtų maksimaliai "apkrautos". Be to neleistina tokia užduočių vykdymo disciplina, kai užduotys paskirstomos atskiriems "procesoriams" - gijoms iš anksto ir nekinta programos vykdymo eigoje. (Kitaip tariant, neleistinas synchronizacijos tarp atskirų gijų nebuvinimas).

Paruošti programos vykdymą dviem režimais - *derinimo režimu*, kuriame programos vykdymas gali būti dirbtinai sulėtintas (*sleep*) ir į ekraną išvedamas programos vykdymo protokolas, paaiškinantis veikimo principus, bei "*sparčiuoju*" režimu, kuriame neturi būti jokių dirbtinių stabdymų, o programa išvestų į ekraną vykdymo trukmę. Komandinė eilutė turi turėti papildomus parametrus - darbinių gijų skaičių bei "apkrovos" parametrą, nusakantį darbo apimtį (pvz., tikslumas, masyvo dydis ir t.t.). Sparčiuoju režimu programa turi demonstruoti vykdymo spartinimą, vykdant ją daugiaprocesorinėje sistemoje.

Eksperimentiškai nustatyti algoritmo spartinimo, plečiamumo (*scaling*) bei vykd. laiko priklausomybės nuo "darbų" dydžio (*grain size*) charakteristikas. Sugebėti pagrįsti gautuosius rezultatus.

Pastaba. Rašant programas objektinėse aplinkose (pvz., java) pageidautina išvengti pakartotinio objekto sukūrimo-atlaisvinimo, kadangi „šiuokšlių surinkėjas“ gali iškraipyti laikines programos vykdymo charakteristikas.

Variantai

1. Masyvo elementų rūšiavimas "quick sort" metodu.

2. Masyvo elementų rūšiavimas "burbuliuko" pavidalo metodu.

Metodo esmė - dviejų gretimų elementų sukeitimas vietomis. Gugliafrazė: "odd even sort"

3. Masyvo elementų rūšiavimas "suliejimo" pavidalo metodu.

Gugliafrazė: "merge sort"

4. Pirminių skaičių generavimas "Eratosthenes" rėčio metodu.

Priminimui: pirmasis pirmenis skaičius 2. Išbraukiami visi kartotiniai žinomam pirminiams skaičiui; tai kartojama su sekančiu neišbrauktu pirminiu ir t.t.) Likę neišbraukti skaičiai - pirminiai. Gugliafrazė: „Eratosthenes prime sieve“

5. Adaptyvusis integralo skaičiavimas.

Duota tolydi neneigiamą funkciją $f(X)$ ir du taškai a, b . Rasti integralą $I[a,b] = \int_a^b f(x) dx$, aproksimuojant jį figuros plotu, aprėžtos ašimi x , kreive $f(x)$ bei tiesėmis $y = a$, $y = b$.

Sprendimo būdas - dalyti sritį $[a,b]$ į daugelį mažesnių sričių vertikaliomis linijomis ir sumuoti pastarųjų plotus, kol gretimų skaidymo iteracijų skirtumas neviršys užduotą paklaidą. Gugliafrazė: "Adaptive Quadrature".

6. Paveikslėlio transformacija.

Duotas 2D paveikslėlis kaip pixel'ių masyvas. Atliki pastarojo iteratyvią transformaciją

$$V(i, j) = (V(i+1, j) + V(i, j+1) + V(i-1, j) + V(i, j-1)) / 4$$

kol "nesukonverguos" (kraštinių piksel'ių reikšmės fiksuotos). Gugliafrazė: Laplace's equation Jacobi iteration.

7. Ilgiausios Collatz'o iteracijos paieška duotajame skaičių intervale.

Kiekvienam $i = 1, 2, 3, \dots$

A(0) - pradinis iteracijos elementas, natūrinis.

$A(i+1) = 3 * A(i) + 1$, kai $A(i)$ - nelyginis

$A(i+1) = A(i) / 2$, kai $A(i)$ - lyginis.

Pagal Collatz'o hipotezę, kiekvienam natūriniam n , po baiginio skaičiaus žingsnių galima gauti 1.

Pvz.: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

Taigi, skaičių $[a,b]$ intervale reikia rasti skaičių n , kurio iteravimo į 1-ą seka - "ilgiausia".

Pastabos: skaičiams naudokite long tipą. Atkreipkite dėmesį, jog iteracijų skaičius labai skiriasi. Gugliafrazė: "Collatz problem"

8. Mandelbrot'o aibės skaičiavimas

Gugliafrazė: "Mandelbrot set"

9. N Valdovių išdėstymo NxN lentoje uždavinys.

Gugliafrazė: "N-queen"

10. Automato "Conway's Life Game" simuliacija.

11. Ilgiausio dviejų simbolių eilučių posekio radimas.

Gugliafrazė: "Longest common subsequence".

12. Susijusių sričių paveikslėlyje identifikavimas.

Susijusioje srityje visi pikselio kaimynai (dešinė, kairė, viršus, apačia) turi tą pačią pikselio vertę. Pradžioje visi pikseliai sunumeruoti nuo 1 iki N^2 (kitame masyve). Apdorojimo pabaigoje susijusios srities pikseliai turi įgauti didžiausią reikšmę. Gugliafrazė: „image segmentation“.

13. Lipschitz'o funkcijos maksimumo intervale radimas.

Vieno kintamojo Lipschitz'o funkcija $f(x)$ bet kuriuose apibrėžimo taškuose a ir b tenkina sąlyga: $|f(x)| < A^*|b-a|$, kur A nepriklauso nuo a ir b pasirinkimo. Taigi, galima ieškoti globalų maksimumą, pradinį intervalą rekursyviai skaidant į mažesnius tol, kol pasieksime reikiama tikslumą. Esmė, žinodami „einamają“ maksimumo reikšmę \max ir funkcijos reikšmę taške $f(x_0) < \max$, galime nustatyti x_0 aplinkos intervalą, kurio toliau smulkinti neverta. Gugliafrazė: „Lipschitz Function“

12.4 Lygiagretūs skaičiavimai pranešimų sistemose

Darbo tikslas. Ivaldyti į pranešimų perdavimus orientuotas sistemas (pvz., MPI). Mokėti pasinaudoti pagrindinėmis pranešimo perdavimo funkcijomis. Sugebėti suprojektuoti lygiagretujį algoritmą duotajai problemai pranešimų siuntimo terminais. Sugebėti nustatyti, kokios aplinkybės lemia kompiuterių klasteryje vykdomos programos efektyvumą. Mokėti nustatyti eksperimentiškai lygiagrečiosios programos spartinimo ir plečiamumo charakteristikas bei palyginti su teoriniais įverčiais.

Darbo trukmė. 10 valandų.

Techninės priemonės. Daugelio procesorių MPI sistema arba MPI-klasteris.

Darbo formulavimas. Duotajai problemai (problemų variantai pateikti 3-me darbe) suprojektuoti ir realizuoti MPI kreipinių efektyvų lygiagrečaus vykdymo algoritmą. Eksperimentiškai nustatyti algoritmo spartinimą ir plečiamumą. Pagrįsti gautus rezultatus.