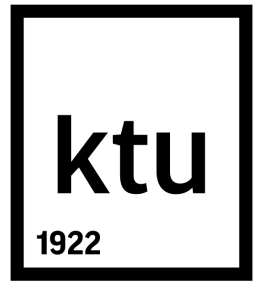
# Lygiagretusis programavimas MPI

Karolis Ryselis

Kauno Technologijos Universitetas




*A beginner programmer is someone who deadlocks by accident. An expert is someone who can deadlock for sure.*

# Paskaitos turinys



ktu 1922

# MPI kolektyvinė komunikacija

- Tiesioginė komunikacija (angl. *point-to-point communication*) — tokia komunikacija tarp procesų, kurioje dalyvauja **du** procesai: siuntėjas ir gavėjas.
- Kolektyvinė komunikacija (angl. *collective communication*) — tokia komunikacija tarp procesų, kurioje dalyvauja **visi** komunikatoriaus procesai.
- Kolektyvinė komunikacija sukuria sinchronizacijos tašką tarp procesų, t.y., visi procesai turi pasiekti tam tikrą tašką kode prieš tęsdami darbą.

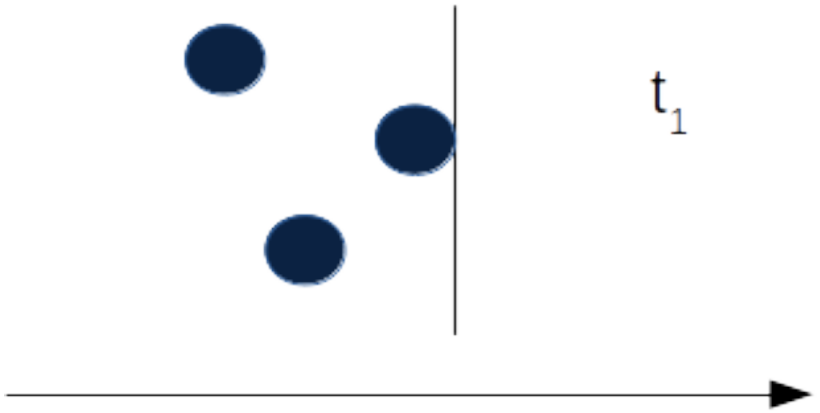
# MPI barjeras

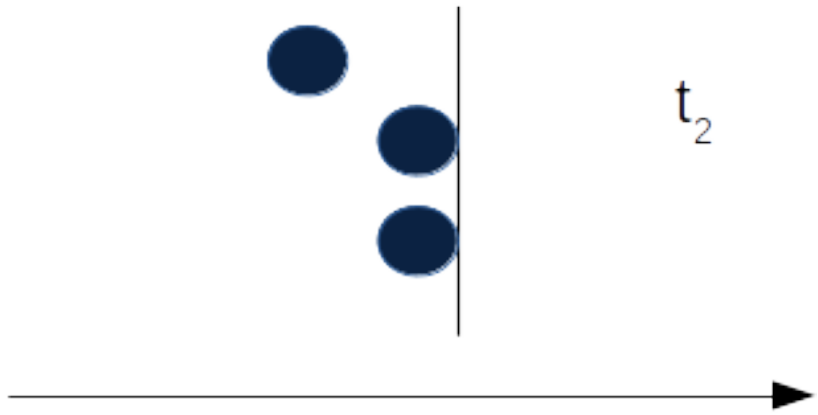
- Barjeras — vienas paprasčiausių kolektyvinės komunikacijos variantų.
- Barjeras skirtas procesų sinchronizacijai — kai procesas pasiekia barjerą, jis laukia, kol visi kiti procesai taip pat pasieks barjerą. Kai visi procesai pasiekia barjerą, galima tęsti darbą.
- MPI barjeras realizuotas siunčiant žetoną ratu — visą ratą galima apsukti tik tada, kai visi procesai jau pasiekė barjerą.

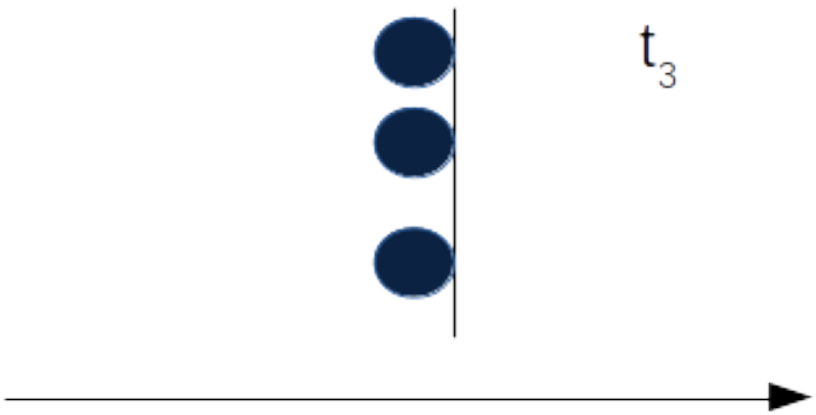
`Comm::Barrier()`

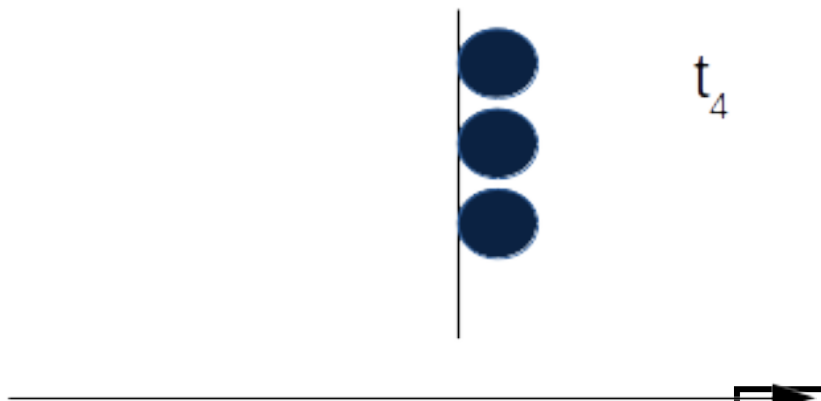
Funkcija, kurios vykdymas baigiamas, kai visi procesai įeina į barjerą.

# MPI barjeras



# MPI barjeras

```
Init();
auto rank = COMM_WORLD.Get_rank();
auto process_count = COMM_WORLD.Get_size();
auto count = 0;
auto messages_recvd = (process_count - 1) * MESSAGES_SENT_PER_PROCESS;
// initialize random number generator
for (auto iteration = 0; iteration <= MAX_ITERATIONS; iteration++) {
    if (rank == ROOT_PROCESS) {
        for (auto i = 0; i < messages_recvd; i++) {
            int increment_by;
            COMM_WORLD.Recv(&increment_by, 1, INT, ANY_SOURCE, ANY_TAG);
            count += increment_by;
        }
    } else {
        for (auto i = 0; i < MESSAGES_SENT_PER_PROCESS; i++) {
            auto increment_by = dist(rng);
            COMM_WORLD.Send(&increment_by, 1, INT, ROOT_PROCESS, 0);
        }
    }
    if (rank == 0) {
        cout << count << endl;
    }
    COMM_WORLD.Barrier();
}
```

# MPI barjeras

Programos rezultatai:

```
/usr/bin/mpirun -np --oversubscribe 101 barrier
209214075
422912846
639033059
851525632
1062726110
1276675074
```

- Barjeras neveiks, jei ne visi komunikatoriaus procesai iškvies `Barrier` funkciją — susidarys aklavietė.
- `--oversubscribe` parinktis reikalinga, jeigu paleidžiame daugiau procesų, nei turime branduolių.

# MPI transliacija (*broadcast*)

- Transliacija yra kolektyvinės komunikacijos rūšis.
- Vienas procesas visiems komunikatoriaus procesams siunčia tą pačią žinutę.
- MPI transliacija dažnai naudojama perduoti naudotojo įvestus duomenis ar konfigūracijos parametrus visiems procesams.

# MPI transliacija

`Comm::Bcast(const void* buf, int count, const Datatype& datatype, int root)`

Persiunčia žinutę `buf` iš proceso `root` visiems kitiems procesams.

| | |
|---|---|
| `buf` | Siunčiamos žinutės adresas |
| `count` | Siunčiamos žinutės dydis |
| `datatype` | Siunčiamos žinutės tipas |
| `root` | Žinutę siunčiantis procesas |

- Žinutę siunčia procesas `root`, visi kiti žinutę priima.
- Tiek siunčiantys, tiek priimantys procesai kviečia tą pačia funkciją, proceso rolė nustatoma pagal parametro `root` reikšmę.

# MPI transliacija

```
int main() {
    Init();
    auto rank = COMM_WORLD.Get_rank();
    int data[DATA_SIZE];
    if (rank == MAIN_PROCESS) {
        // fill data array
    }
    COMM_WORLD.Bcast(data, DATA_SIZE, INT, MAIN_PROCESS);
```

# MPI transliacija

```
    switch(rank) {
        case MAIN_PROCESS: {
            auto min = 0;
            auto max = 0;
            COMM_WORLD.Recv(&min, 1, INT, MIN_CALCULATOR, 0);
            COMM_WORLD.Recv(&max, 2, INT, MAX_CALCULATOR, 0);
            cout << "minimum value " << min << endl;
            cout << "maximum value " << max << endl;
            break;
        }
        case MIN_CALCULATOR: {
            auto *min_value = min_element(data, data + DATA_SIZE);
            COMM_WORLD.Send(&min_value, 1, INT, MAIN_PROCESS, 0);
            break;
        }
        case MAX_CALCULATOR: {
            auto *max_value = max_element(data, data + DATA_SIZE);
            COMM_WORLD.Send(&max_value, 1, INT, MAIN_PROCESS, 0);
            break;
        }
    }
    Finalize();
    return 0;
}
```

# MPI transliacija

Programos rezultatai:

```
/usr/bin/mpirun -np 3 bcast
minimum value 65
maximum value 999901
```

# MPI transliacija

- MPI transliaciją galima realizuoti pasinaudojant `Send` ir `Recv`, tačiau tai nebus taip efektyvu.
- Jei nulinis procesas siųs visiems kitiems procesams savo duomenis, o likę procesai juos priims, bus išnaudotas tik vienas komunikacijos kanalas vienu metu.
- MPI `Bcast` realizacija naudojasi dvejetainiu medžiu: pagrindinis procesas persiunčia savo duomenis kitiems dviems procesams, kiekvienas sekantis — dar dviems kitiems, kol visi procesai turi tuos pačius duomenis.

# Scatter ir Gather

`Scatter` duomenis, saugomus viename procese, persiunčia dalimis visiems komunikatoriaus procesams.

`Gather` duomenis, saugomus skirtinguose procesuose, surenka viename procese į bendrą duomenų rinkinį.

# Scatter

- `Scatter` veikia panašiai, kaip `Bcast`, bet siunčia ne visiems procesams tuos pačius duomenis, o visiems skirtingą jų dalį.
- `Scatter` priima duomenų masyvą ir jo elementus paskirsto visiems procesams jų numerio didėjimo tvarka.
- Procesas, iš kurio siunčiama, taip pat gaus dalį duomenų, nepaisant to, kad jame yra visi duomenys.

# Scatter

```
Comm::Scatter(void* send_data, int send_count, Datatype
datatype, void* recv_data, int recv_count, Datatype
recv_datatype, int root)
```

`send_data` duomenų masyvas, esantis `root` procese

`send_count` kiek elementų siunčiama kiekvienam procesui

`send_datatype` kokio tipo elementai siunčiami

`recv_data` duomenų masyvas, į kurį bus gaunami duomenys

`recv_count` duomenų masyvo, į kurį gaunami duomenys, dydis

`recv_datatype` priimamų duomenų dydis

`root` proceso, iš kurio siunčiami duomenys, numeris

# Scatter

```
int main() {
    Init();
    auto total_processes = COMM_WORLD.Get_size();
    auto data_size = total_processes * NUM_PER_PROCESS;
    int* full_data = nullptr;
    auto rank = COMM_WORLD.Get_rank();
    if (rank == MAIN_PROCESS) {
        full_data = new int[data_size];
        iota(full_data, full_data + data_size, 0);
    }
    int chunk[NUM_PER_PROCESS];
    COMM_WORLD.Scatter(full_data, NUM_PER_PROCESS, INT, chunk,
            NUM_PER_PROCESS, INT, MAIN_PROCESS);
    Finalize();
    cout << "Process " << rank << " received ";
    for_each(chunk, chunk + NUM_PER_PROCESS, [](auto num)
            {cout << num << " ";});
    cout << endl;
    delete[] full_data;
    return 0;
}
```

# Scatter

Programos rezultatai:

```
/usr/bin/mpirun -np 4 scatter
Process 3 received 18 19 20 21 22 23
Process 1 received 6 7 8 9 10 11
Process 0 received 0 1 2 3 4 5
Process 2 received 12 13 14 15 16 17
```

# Gather

- `Gather` yra funkcija, atvirkščia `Scatter`.
- `Gather` priima kiekviename procese esantį masyvą ir jų elementus surašo į nurodytą nurodyto proceso masyvą.
- Duomenys bus paimami ir iš surenkančio proceso.

# Gather

```
Comm::Gather(void* send_data, int send_count, Datatype
datatype, void* recv_data, int recv_count, Datatype
recv_datatype, int root)
```

`send_data` duomenų masyvas, esantis `root` procese

`send_count` kiek elementų siunčiama kiekvienam procesui

`send_datatype` kokio tipo elementai siunčiami

`recv_data` duomenų masyvas, į kurį bus gaunami duomenys

`recv_count` duomenų masyvo, į kurį gaunami duomenys, dydis

`recv_datatype` priimamų duomenų dydis

`root` proceso, į kurį surenkami duomenys, numeris

# Gather

- Pakanka, kad tik surenkantis procesas turėtų inicializuotą buferį `recv_data`, kiti procesai gali perduoti `nullptr`.
- `recv_count` nurodomas vieno proceso duomenų kiekis.

# Gather

```
int main() {
    int chunk[NUM_PER_PROCESS];
    int *full_data = nullptr;
    int data_size;
    Init();
    auto rank = COMM_WORLD.Get_rank();
    auto start_number = rank * NUM_PER_PROCESS;
    iota(chunk, chunk + NUM_PER_PROCESS, start_number);
    if (rank == MAIN_PROCESS) {
        data_size = COMM_WORLD.Get_size() * NUM_PER_PROCESS;
        full_data = new int[data_size];
    }
    COMM_WORLD.Gather(chunk, NUM_PER_PROCESS, INT, full_data,
            NUM_PER_PROCESS, INT, MAIN_PROCESS);
    Finalize();
    if (rank == MAIN_PROCESS) {
        cout << "Main process received values: ";
        for_each(full_data, full_data + data_size,
                 [](auto num) { cout << num << " "; });
        cout << endl;
    }
```

# Gather

Programos rezultatai:

```
/usr/bin/mpirun -np 4 gather
Main process received values: 0 1 2 3 4 5 6 7 8 9 10 11 12
13 14 15 16 17 18 19 20 21 22 23
```

# Lygiagretusis vidurkis

```
int main() {
    double* random_numbers = nullptr;
    Init();
    auto rank = COMM_WORLD.Get_rank();
    auto total_processes = COMM_WORLD.Get_size();
    if (rank == 0) {
        auto total_elements = ELEMENTS_PER_PROCESS * total_processes;
        random_numbers = new double[total_elements];
        generate_random_numbers(random_numbers, total_elements);
    }
    double random_numbers_chunk[ELEMENTS_PER_PROCESS];
    COMM_WORLD.Scatter(random_numbers, ELEMENTS_PER_PROCESS, DOUBLE,
        random_numbers_chunk, ELEMENTS_PER_PROCESS, DOUBLE, 0);
```

# Lygiagretusis vidurkis

```
    double average = get_average(random_numbers_chunk,
        ELEMENTS_PER_PROCESS);
    double* averages_of_chunks = nullptr;
    if (rank == 0) {
        delete[] random_numbers;
        averages_of_chunks = new double[total_processes];
    }
    COMM_WORLD.Gather(&average, 1, DOUBLE,
        averages_of_chunks, 1, DOUBLE, 0);
    Finalize();
    if (rank == 0) {
        auto total_average = get_average(averages_of_chunks,
            total_processes);
        cout << "average: " << total_average << endl;
    }
    return 0;
}
```

# *Reduce* operacija MPI

- *Reduce* operacijos skirtos iš masyvo gauti skaliarą.
- `MPI_Reduce` masyvo elementams su tuo pačiu indeksu skirtinguose procesuose gauna vieną elementą — kiekvienas procesas turi po tokio pat dydžio masyvą, rezultatas bus vienas naujas masyvas, kurio kiekvienas elementas bus gautas iš visų procesų masyvų elementų su tuo pačiu indeksu.
- MPI turi realizuotas funkcijas rasti sumai, sandaugai, minimumui, maksimumui, loginėms ir bitinėms operacijoms.
- Yra galimybė aprašyti savo operacijas.

# MPI_Reduce

```
Comm::Reduce(void* sendbuf, void* recvbuf, int count,
Datatype datatype, Op op, int root)
```

- `sendbuf` duomenų masyvas, esantis kiekviename procese
- `recvbuf` rezultatų masyvas, esantis `root` procese
- `count` duomenų ir rezultatų masyvų dydis (turėtų būti vienodas)
- `datatype` duomenų, su kuriais dirbama, tipas
- `op` operacija, kurią reikia atlikti (pvz, `MPI::MIN`)
- `root` proceso, į kurį surenkami duomenys, numeris

# Lygiagretusis maksimumas

```
int main() {
    Init();
    int items[ITEM_COUNT];
    // fill items with data
    int results[ITEM_COUNT];
    COMM_WORLD.Reduce(items, results, ITEM_COUNT, INT, MAX,
      ROOT_PROCESS);
    if (rank == ROOT_PROCESS) {
        for_each(results, results + ITEM_COUNT,
          [](auto item) { cout << item << endl; });
        auto global_maximum = max_element(results,
          results + ITEM_COUNT);
        cout << "Global maximum: " << *global_maximum
            << endl;
    }
    Finalize();
    return 0;
}
```

# Lygiagretusis maksimumas

Programos rezultatai su 30 procesų: 0

```
30
120
270
480
750
1080
1470
1920
2430
3000
3630
4320
5070
5880
Global maximum:  5880
```

# Duomenų perdavimas MPI

- MPI tarp procesų galima siųsti tik tam tikrus tipus.
- Problema kyla, kai reikia siųsti struktūrą ar objektą, MPI neturi tam skirto mechanizmo.
- Galima siųsti keletą žinučių kiekvieną kartą perduodant vis kitą lauką.
- Efektyviau struktūros kintamąjį ar objektą serializuoti ir persiųsti serializuotus duomenis.

# Universalūs serializavimo metodai

- Duomenis galima serializuoti ir deserializuoti naudojantis standartiniais serializavimo formatais, pvz., JSON, XML ar kt.
  1. Prieš siuntimą duomenys serializuojami į `char` masyvą (jei tai tekstinis formatas) ar baitų masyvą (jei tai dvejetainis formatas).
  2. Serializuoti duomenys siunčiami kaip masyvas.
  3. Serializuoti duomenys priimami kaip masyvas.
  4. Gauti duomenys deserializuojami į atitinkamo tipo objektą ar struktūrą.

# Serializavimas JSON

```
class Student {
private:
    string name;
    int study_year;
    double average_grade;
public:
    Student(string name, int year, double average_grade);
    string to_json();
    static Student from_json(string json_string);
    string get_name();
    int get_study_year();
    double get_average_grade();
};
```

# Serializavimas JSON

```
int main() {
    Init();
    auto rank = COMM_WORLD.Get_rank();
    if (rank == 0) {
        auto* student = get_student();
        string serialized = student->to_json();
        auto serialized_size = serialized.size();
        const char* serialized_chars = serialized.c_str();
        COMM_WORLD.Send(serialized_chars, (int) serialized_size, CHAR, 1, 1);
    } else {
        COMM_WORLD.Probe(0, 1, status);
        auto size = status.Get_count(CHAR);
        char serialized[size];
        COMM_WORLD.Recv(serialized, size, CHAR, 0, 1);
        auto student = Student::from_json(string(serialized, serialized + size));
        cout << student.to_string() << endl;
    }
    Finalize();
    return 0;
}
```

# Laiko matavimas MPI

- MPI laiką galima matuoti funkcija `Wtime()`.
- Tipinis panaudojimas:

```
auto start_time = Wtime();
// run computations
auto stop_time = Wtime();
auto elapsed = stop_time - start_time;
```