Collatz problema

1. Formuluotė

Igiausios Collatz'o iteracijos paieška duotajame skaičių intervale. Kiekvienam i = 1,2,3,... 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 baigtinio 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, kuris iteruoja į 1-ą "ilgiausiai".

2.Algoritmas:

Šis algoritmas efektyviai paskirsto Collatz problemos skaičiavimus tarp kelių procesorių naudojant „master-slave“ modelį.

Master procesas:
Paskirsto darbo intervalus slave procesams.
Pats atlieka dalį skaičiavimų.
Surenka rezultatus iš slave procesų.
Išveda bendrus rezultatus ir laiką.

Slave procesai:
Gauna darbo intervalus iš master proceso.
Atlieka skaičiavimus savo intervale.
Siunčia rezultatus atgal master procesui.

3.Įrangos aprašymas:

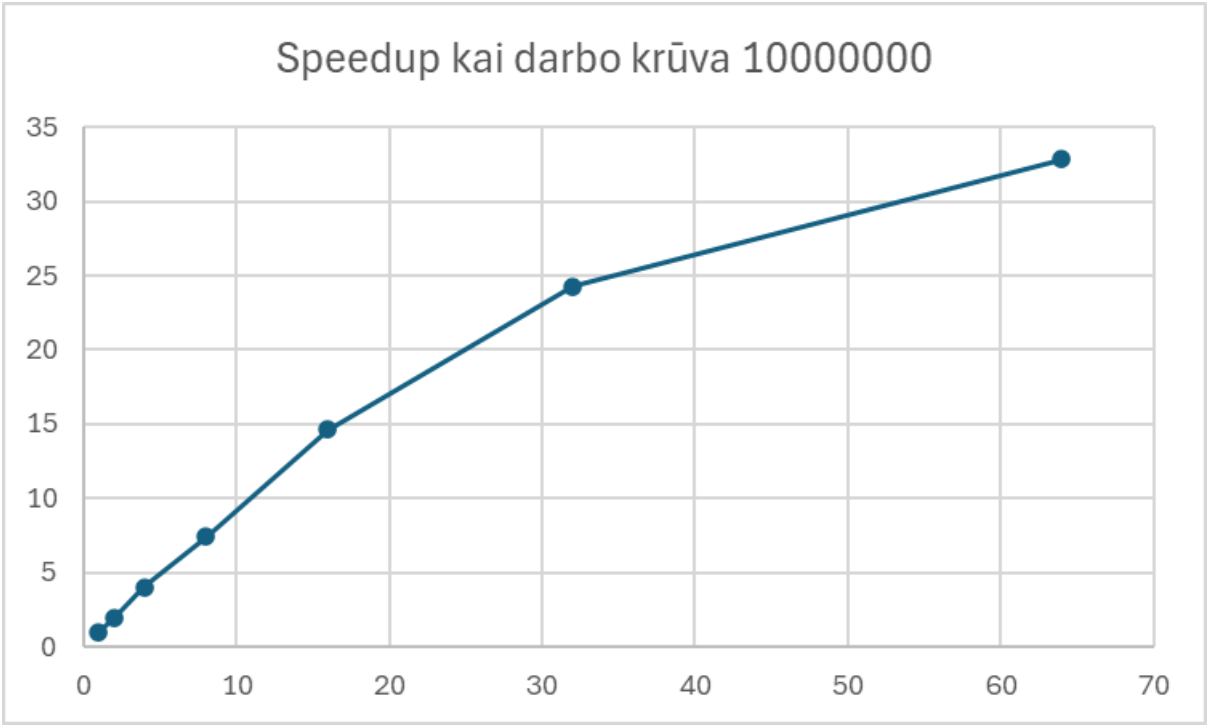
Hpc cluster

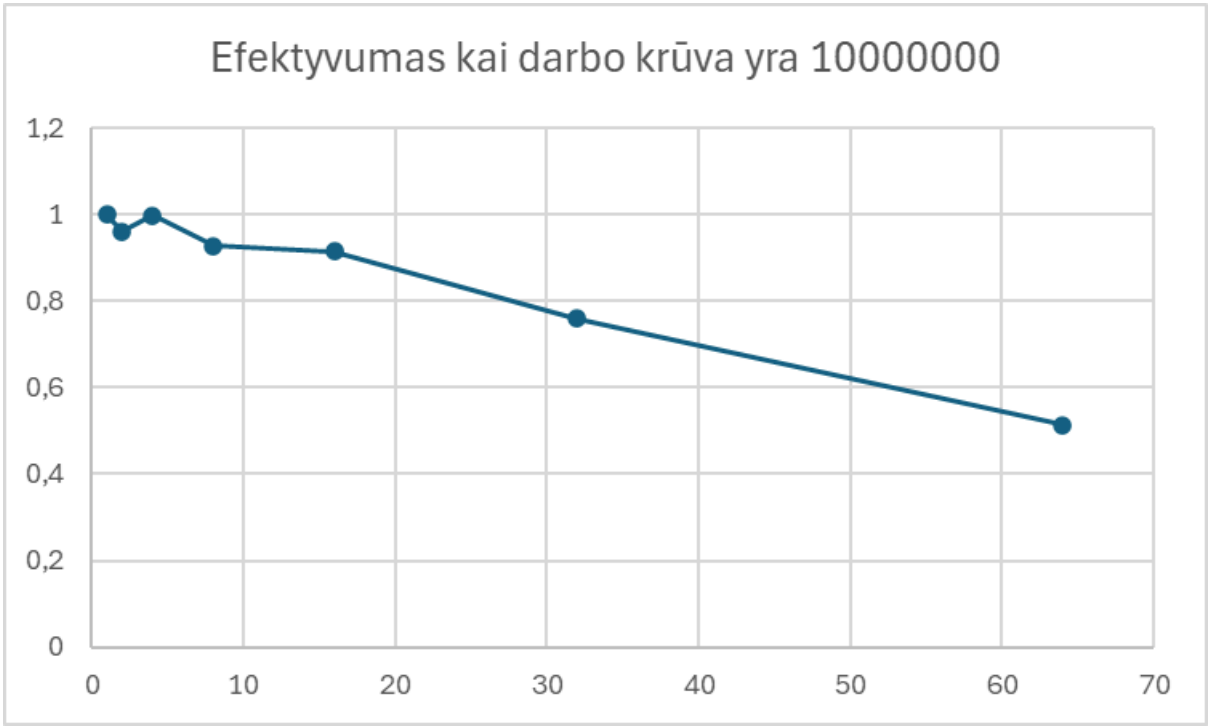
Procesorius - Intel® Xeon® Gold 6252

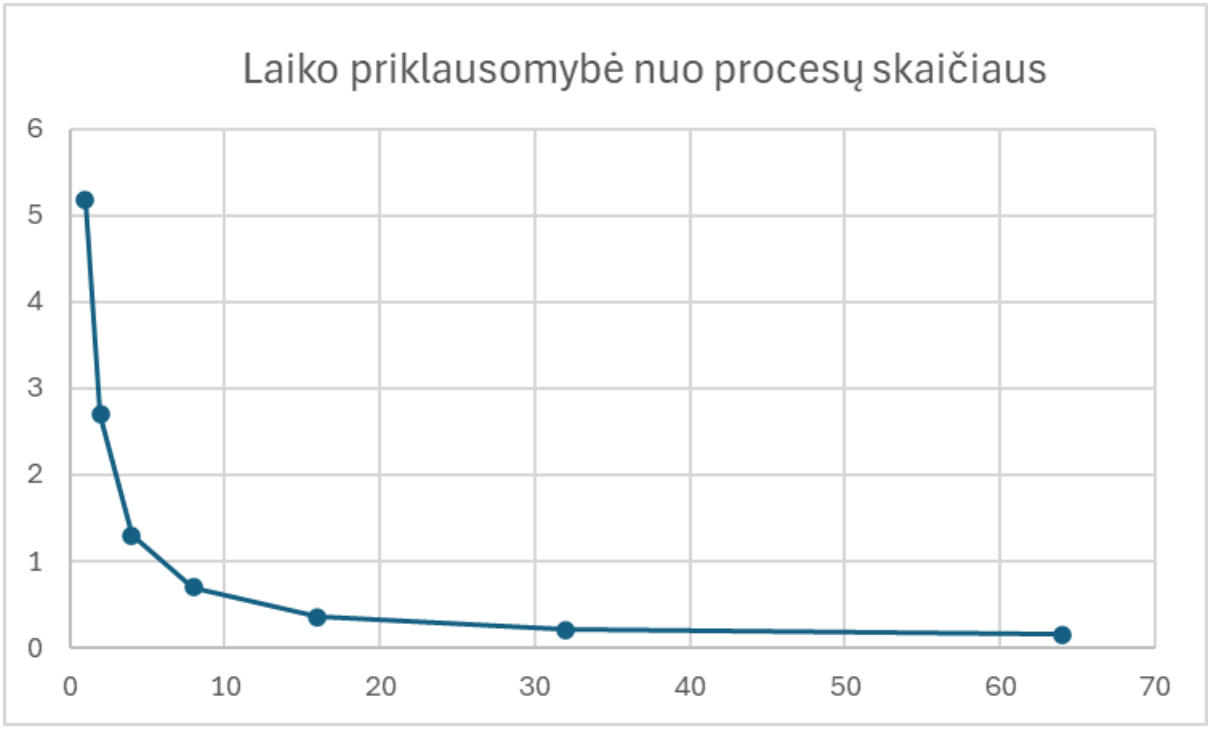
24 cores 48 threads

4.Tyrimo rezultatai

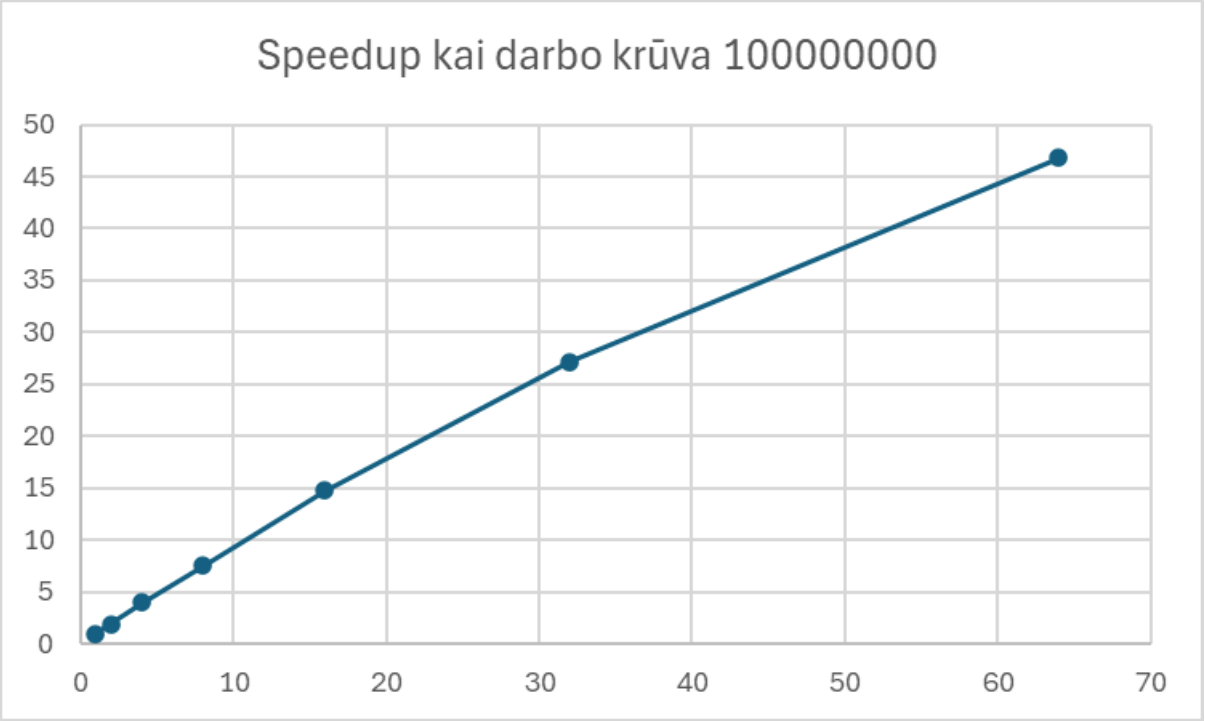
	Vykdymo laikas	Greitėjimas	Efektyvumas	Krūvis
1	5,172723	1	1	10000000
2	2,693365	1,92054289	0,960271445	10000000
4	1,296868	3,98862722	0,997156804	10000000
8	0,696639	7,42525612	0,928157015	10000000
16	0,353561	14,6303552	0,914397197	10000000
32	0,213054	24,2789293	0,75871654	10000000
64	0,157298	32,8848619	0,513825966	10000000

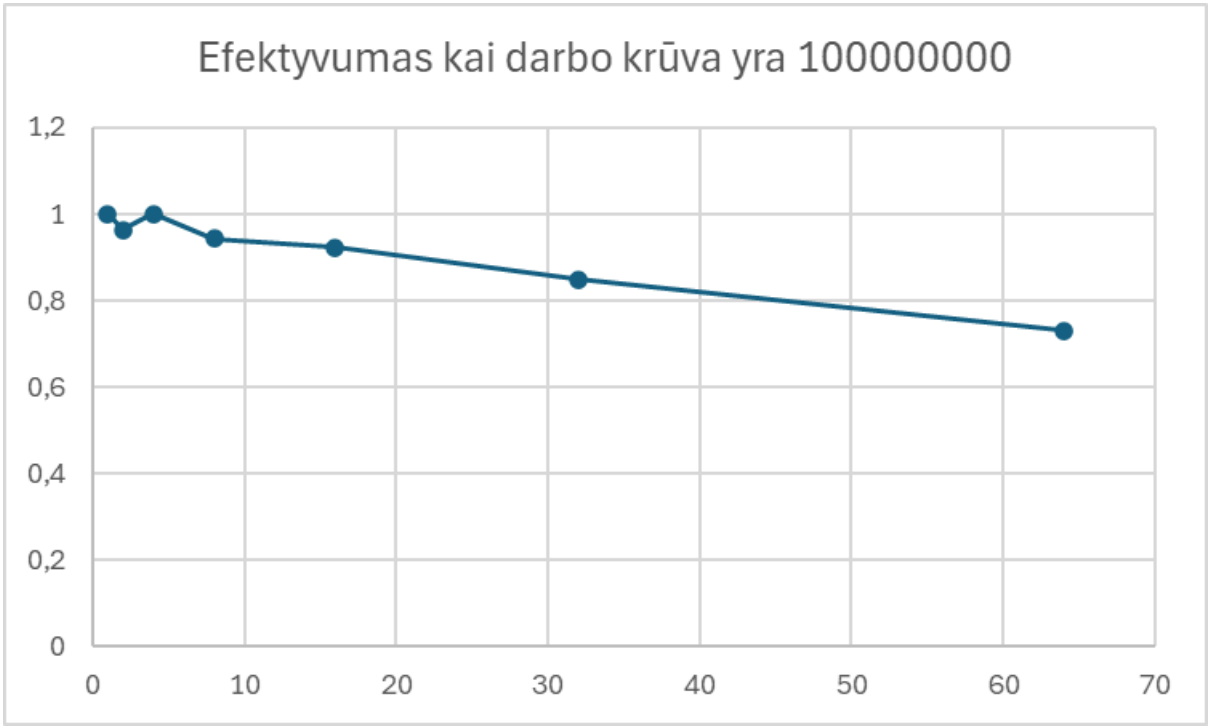


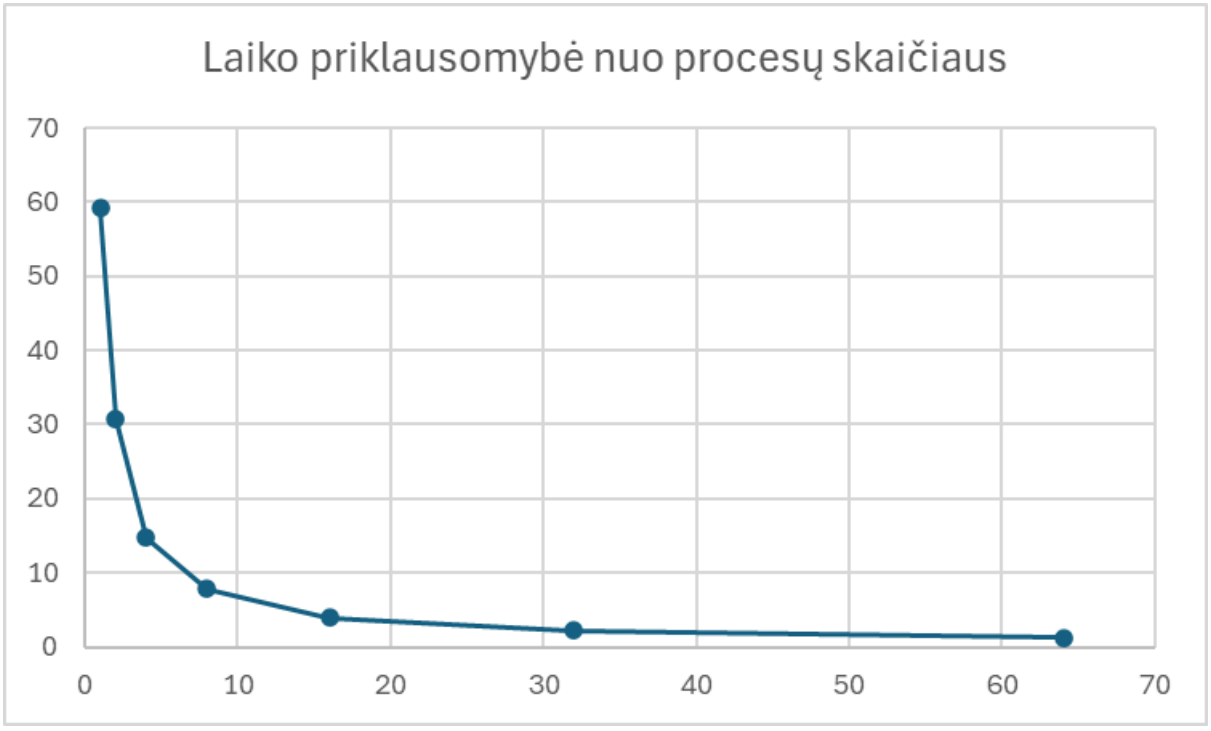

Efektyvumas kai darbo krūva yra 10000000
1,2
1
0,8
0,6
0,4
0,2
0
0
10
20
30
40
50
60
70


Laiko priklausomybė nuo procesų skaičiaus
6
5
4
3
2
1
0
0
10
20
30
40
50
60
70

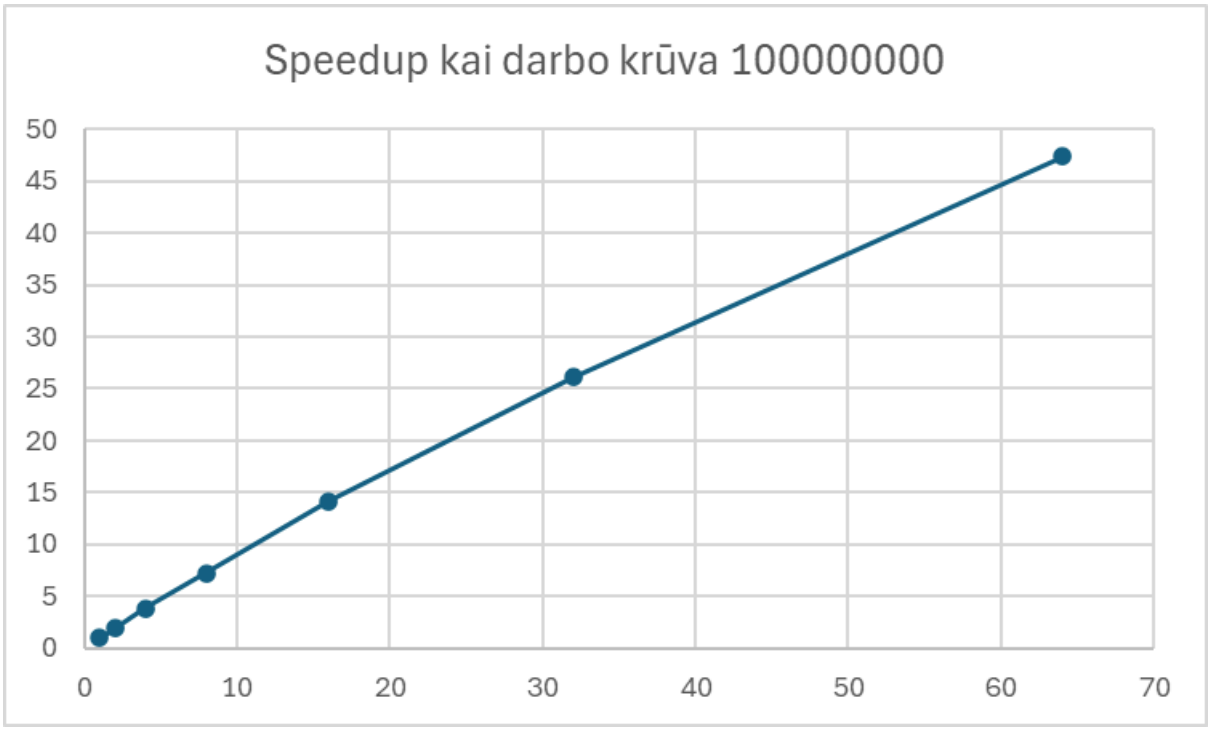
	Vykdymo laikas	Greitėjimas	Efektyvumas	Krūvis
1	59,192536	1	1	100000000
2	30,697562	1,92824876	0,964124382	100000000
4	14,772244	4,00701044	1,001752611	100000000
8	7,840784	7,54931344	0,94366418	100000000
16	4,005838	14,7765676	0,923535475	100000000
32	2,177041	27,1894448	0,849670149	100000000
64	1,264575	46,8082447	0,731378823	100000000

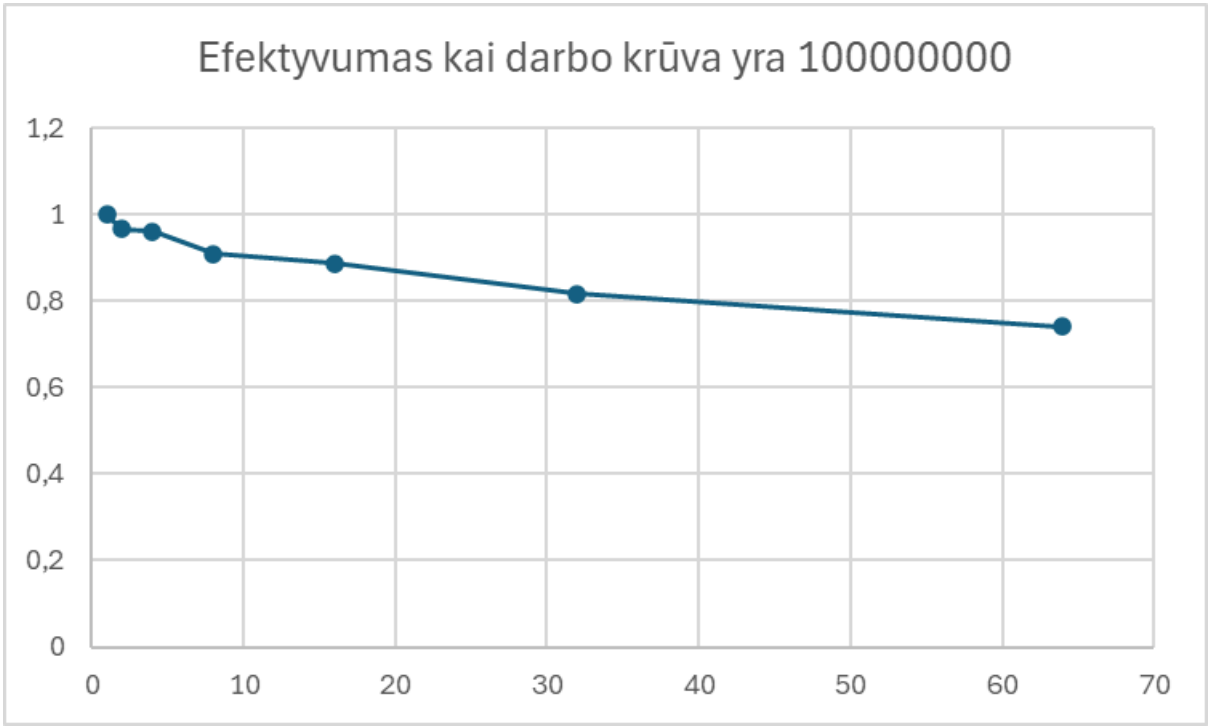


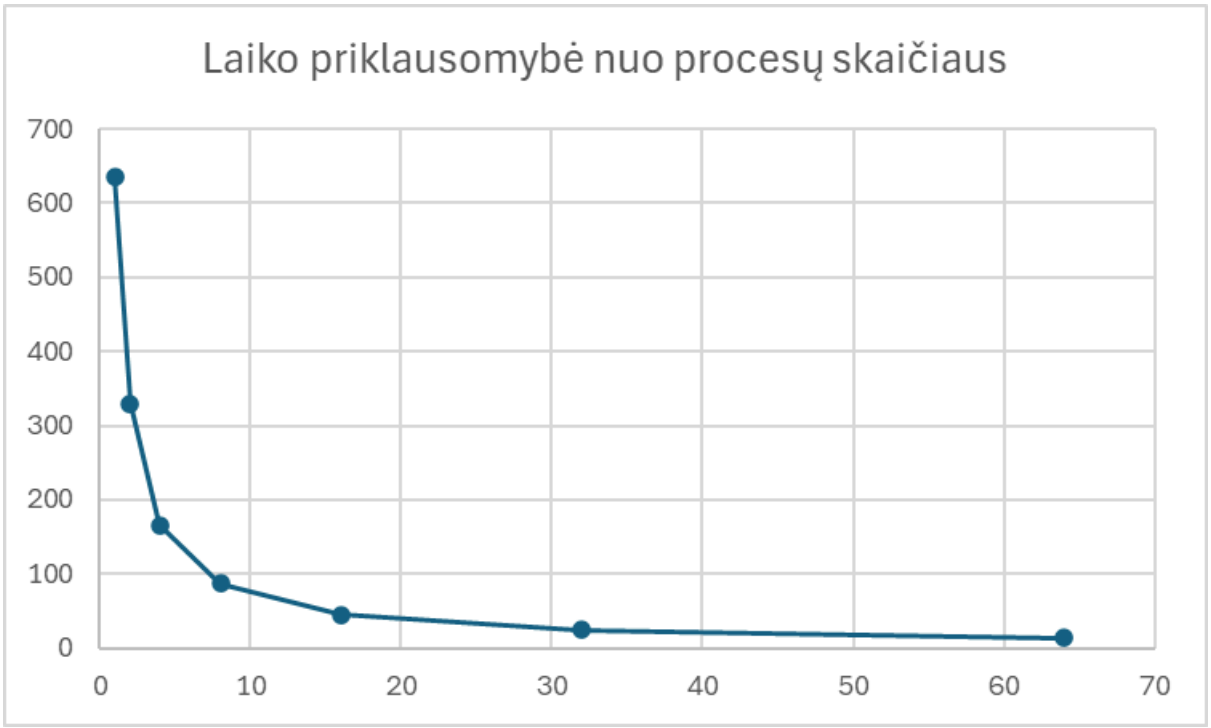

Efektyvumas kai darbo krūva yra 100000000
1,2
1
0,8
0,6
0,4
0,2
0
0
10
20
30
40
50
60
70


Laiko priklausomybė nuo procesų skaičiaus
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
70

	Vykdymo laikas	Greitėjimas	Efektyvumas	Krūvis
1	635,994891	1	1	1000000000
2	329,297988	1,93136586	0,965682929	1000000000
4	165,484384	3,84323207	0,960808016	1000000000
8	87,406383	7,27629801	0,909537252	1000000000
16	44,811872	14,1925535	0,887034594	1000000000
32	24,324485	26,1462839	0,817071372	1000000000
64	13,428258	47,3624271	0,740037924	1000000000




Efektyvumas kai darbo krūva yra 100000000
1,2
1
0,8
0,6
0,4
0,2
0
0
10
20
30
40
50
60
70


Laiko priklausomybė nuo procesų skaičiaus
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70

5.Išvados ir pastėbėjimai:

Tyrimo duomenys atskleidžia, kaip keičiasi vykdymo laikas, greitėjimas ir efektyvumas, didėjant procesorių skaičiui (nuo 1 iki 64), esant trims skirtingiems krūviams: 10,000,000; 100,000,000; ir 1,000,000,000. Analizuojant šiuos duomenis, galima padaryti keletą išvadų:

Padidinus procesorių skaičių, vykdymo laikas pastebimai mažėja visais atvejais. Tai natūralu, nes padidėjus procesorių skaičiui, darbas paskirstomas tarp daugiau procesorių, dėl ko bendra užduoties atlikimo trukmė mažėja.
Pavyzdžiui, kai krūvis yra 10,000,000, vykdymo laikas sumažėja nuo 5,172723 sekundžių (1 procesorius) iki 0,157298 sekundžių (64 procesoriai).

Greitėjimas auga, didinant procesorių skaičių, tačiau ne visada linijiniu santykiu. Tai reiškia, kad pridėjus daugiau procesorių, greitėjimas nėra tiesiogiai proporcingas jų skaičiui.
Pavyzdžiui, kai krūvis yra 100,000,000, greitėjimas padidėja nuo 1 (1 procesorius) iki 46,80824467 (64 procesoriai).

Efektyvumas mažėja, didėjant procesorių skaičiui. Tai natūralu, nes padidinus procesorių skaičių, didėja koordinavimo ir sinchronizavimo kaštai, dėl kurių dalis procesorių gali likti nevisiškai išnaudota. Efektyvumas yra artimas 1, kai procesorių skaičius yra mažas, tačiau jis mažėja, kai procesorių skaičius auga. Pavyzdžiui, kai krūvis yra 1,000,000,000, efektyvumas sumažėja nuo 1 (1 procesorius) iki 0,740037924 (64 procesoriai).

Esant mažesniems krūviams, efektyvumo mažėjimas didinant procesorių skaičių yra ryškesnis. Pavyzdžiui, kai krūvis yra 10,000,000, efektyvumas sumažėja iki 0,513825966 (64 procesoriai), o kai krūvis yra 1,000,000,000, efektyvumas sumažėja iki 0,740037924 (64 procesoriai).

Tai reiškia, kad didesni krūviai yra geriau pritaikyti paralelizavimui ir leidžia efektyviau išnaudoti didesnį procesorių skaičių.

Bendros Išvados
Optimalus procesorių skaičius priklauso nuo užduoties dydžio: Didėjant užduoties dydžiui (krūviui), daugiau procesorių gali būti efektyviau išnaudojami. Tačiau mažesnės užduotys neefektyviai išnaudoja didelį procesorių skaičių dėl koordinavimo ir sinchronizavimo kaštų.
Efektyvumas mažėja didėjant procesorių skaičiui: Tai yra neišvengiama dėl paralelinio apdorojimo apribojimų, tokių kaip duomenų priklausomybės ir sinchronizavimo kaštai.
Didelės užduotys yra geriau pritaikytos paralelizavimui: Kaip matyti iš duomenų, kai krūvis yra 1,000,000,000, efektyvumas su 64 procesoriais yra geresnis nei esant mažesniems krūviams.
Šios išvados padeda suprasti, kaip geriau optimizuoti užduočių paskirstymą tarp procesorių, siekiant maksimalaus našumo ir efektyvumo, atliekant didelius skaičiavimus naudojant paralelinį apdorojimą su MPI.

Priedai:

collatz.c:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

// Function to compute Collatz steps
int collatz_steps(unsigned long long n) {
    int steps = 0;
    while (n != 1) {
        if (n % 2 == 0) {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
        steps++;
```

```
    }
    return steps;
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc != 3) {
        if (rank == 0) {
            printf("Usage: %s <start> <end>\n", argv[0]);
        }
        MPI_Finalize();
        return 0;
    }

    unsigned long long start = strtoull(argv[1], NULL, 10);
    unsigned long long end = strtoull(argv[2], NULL, 10);
    unsigned long long range = end - start + 1;

    if (rank == 0) {
        unsigned long long local_start, local_end;
        int max_steps = 0;
        unsigned long long number_with_max_steps = 0;

        double start_time = MPI_Wtime();

        // Send tasks to slaves
        unsigned long long chunk_size = (range + size - 1) / size;
        for (int i = 1; i < size; i++) {
            local_start = start + i * chunk_size;
            local_end = (i == size - 1) ? end : (local_start + chunk_size - 1);
            MPI_Send(&local_start, 1, MPI_UNSIGNED_LONG_LONG, i, 0,
MPI_COMM_WORLD);
```

```
            MPI_Send(&local_end, 1, MPI_UNSIGNED_LONG_LONG, i, 0,
MPI_COMM_WORLD);
        }

        // Master process also does computation
        local_start = start;
        local_end = (size == 1) ? end : (start + chunk_size - 1);
        for (unsigned long long i = local_start; i <= local_end; i++) {
            int steps = collatz_steps(i);
            if (steps > max_steps) {
                max_steps = steps;
                number_with_max_steps = i;
            }
        }

        // Receive results from slaves
        for (int i = 1; i < size; i++) {
            int slave_max_steps;
            unsigned long long slave_number_with_max_steps;
            MPI_Recv(&slave_max_steps, 1, MPI_INT, i, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv(&slave_number_with_max_steps, 1,
MPI_UNSIGNED_LONG_LONG, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            if (slave_max_steps > max_steps) {
                max_steps = slave_max_steps;
                number_with_max_steps = slave_number_with_max_steps;
            }
        }

        double end_time = MPI_Wtime();
        printf("Number with max steps: %llu, Steps: %d\n",
number_with_max_steps, max_steps);
        printf("Total time taken: %f seconds\n", end_time - start_time);
    } else {
        unsigned long long local_start, local_end;
        int max_steps = 0;
```

```
        unsigned long long number_with_max_steps = 0;

        // Receive range from master
        MPI_Recv(&local_start, 1, MPI_UNSIGNED_LONG_LONG, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&local_end, 1, MPI_UNSIGNED_LONG_LONG, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        for (unsigned long long i = local_start; i <= local_end; i++) {
            int steps = collatz_steps(i);
            if (steps > max_steps) {
                max_steps = steps;
                number_with_max_steps = i;
            }
        }

        // Send results back to master
        MPI_Send(&max_steps, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&number_with_max_steps, 1,
MPI_UNSIGNED_LONG_LONG, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

collatz.sh

```
#!/bin/bash
#SBATCH -p main
#SBATCH -n64
module load openmpi
mpicc -o collatz collatz.c
mpirun -np 64 ./collatz 1 1000000000
```