Michał Dybaś

**Temat projektu:**

Program, który oblicza iloczyn skalarny n-elementowych wektorów rzeczywistych równolegle w środowisku MPI. Program wykorzystuje dwie architektury środowiska MPI.

* Dla środowiska gdzie nie została ustalona relacja pomiędzy procesami
* Dla środowiska w którym relacja pomiędzy procesami (procesorami) została ustalona przy pomocy funkcji MPI\_Graph\_Create, a zdefiniowana struktura ma postać pierścienia (architektura Hypercube d=3)

Spis treści

[1. Kod źródłowy środowiska MPI projekt\_1.c, w którym nie została ustalona relacja między procesami. 3](#_Toc99453749)

[2. Kod źródłowy środowiska MPI projekt\_2.c, w którym została ustalona relacja między procesami 7](#_Toc99453750)

[3. Działanie aplikacji i wykonanie eksperymentów 11](#_Toc99453751)

[4. Podsumowanie wyników 17](#_Toc99453752)

[5. Wnioski 17](#_Toc99453753)

# Kod źródłowy środowiska MPI projekt\_1.c, w którym nie została ustalona relacja między procesami.

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <time.h>

//  • Dla środowiska gdzie nie została ustalona relacja pomiędzy procesami

//  kompilacja   mpicc -o projekt\_1 projekt\_1.c

//  uruchamianie mpirun --host localhost:liczba\_procesorów/wątków ./projekt\_1

int main(int argc, char \*\* argv)

{

    //Deklaracja i incjalizacja zmienych

    double \*vt\_1, \*vt\_1\_part, \*vt\_2, \*vt\_2\_part, \*scores, score = 0; // Wskaźniki do tablic liczb typu double (nasze wektory i pod\_wektory) i tablicy na wyniki oraz sumy dla każdego procesu z osobna

    int vt\_normal\_size; // Zmienna przechowująca ilość elementów dla każdego wektora, jest ona inicjalizowana przez użytkownika

    int vt\_extend\_size; // Zmienna przechowująca ilość elementów (równych 0), o które został rozszerzony wektor

    int vt\_real\_size; // Zmienna przechowująca realną ilość elementów dla każdego wektora

    int vt\_part\_size; // Zmienna przechowująca ilość elementów wektorów jaki przypada na każdy z procesów

    int rank; // Numer mojego procesu

    int size; // Liczba procesów

    struct timespec start, end; // Zmienna przechowująca czas rozpoczęcia i końca obliczeń

    MPI\_Init(&argc, &argv); // Rozpoczęcie obliczeń MPI

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // Pobranie aktualnego numeru procesu

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); // Pobranie liczby procesów

    if(rank == 0) // Instrukcje wykonywane dla głównego procesu programu (proces root)

    {

        fprintf(stderr, "Podaj długośc wektorów (liczba całkowita większa od 0): "); // Wyświetlenie komunikatu z prośbą o podanie długości wektora

        scanf("%d", &vt\_normal\_size); // Pobranie długości wektów od użytkownika

        if(vt\_normal\_size < 0) // Wyjście z programu dla wektora mniejszego niż 1

        {

            printf("Podany rozmiar wektorów jest zbyt mały, ilość elementów jest mniejsza niż ilość procesów: %d.\n", size);

            MPI\_Finalize(); // Koniec obliczeń MPI

            exit(1); // Wyjście z programu z kodem błędu

        }

        vt\_extend\_size = size - (vt\_normal\_size % size); // Obliczenie o ile elementów należy rozszerzyć wektor, aby można było podzielić je równo dla każdego procesu

        vt\_real\_size = vt\_normal\_size + vt\_extend\_size; // Obliczenie realnej długości wektorów

        vt\_part\_size = vt\_real\_size / size; // Obliczenie ilości elementów przypadających na każdy proces

        vt\_1 = malloc(vt\_real\_size \* sizeof(double)); // Zaalokowanie pamięci na pierwszy wektor

        vt\_2 = malloc(vt\_real\_size \* sizeof(double)); // Zaalokowanie pamięci na drugi wektor

        vt\_1\_part = malloc(vt\_part\_size \* sizeof(double)); // Zaalokowanie pamięci na pierwszy wektor częsciowy

        vt\_2\_part = malloc(vt\_part\_size \* sizeof(double)); // Zaalokowanie pamięci na drugi wektor częściowy

        scores = malloc(size \* sizeof(double)); // Zaalokowanie pamięci na tablice wyników

        for(int i = 0; i < vt\_real\_size; i++) // Inicjalizacja wektorów liczbami losowymi

        {

            if(i < vt\_normal\_size) // Wypełnienie elementów wektorów liczbami typu rzeczywistego z przedziału (-25, 25)

            {

                srand(time(NULL) + rand());

                vt\_1[i] = (double) rand() / RAND\_MAX \* 50 - 25;

                vt\_2[i] = (double) rand() / RAND\_MAX \* 50 - 25;

            }

            else // Dla dodatkowo wygenerowanych elementów wartość wynosi 0, aby wynik obliczeń nie został zakłamany

            {

                vt\_1[i] = 0;

                vt\_2[i] = 0;

            }

        }

        clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start); // Pobranie czasu rozpoczęcia obliczeń

        MPI\_Bcast(&vt\_real\_size, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); // Wysłanie rozmiaru wektora

        MPI\_Bcast(&vt\_part\_size, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); // Wysłanie rozmiaru części wektora

        MPI\_Scatter(vt\_1, vt\_part\_size, MPI\_DOUBLE, vt\_1\_part, vt\_part\_size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD); // Wysłanie podzielonego wektora 1 na kawałki do procesów w grupie

        MPI\_Scatter(vt\_2, vt\_part\_size, MPI\_DOUBLE, vt\_2\_part, vt\_part\_size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD); // Wysłanie podzielonego wektora 1 na kawałki do procesów w grupie

        MPI\_Barrier(MPI\_COMM\_WORLD); // Bariera synchronizacyjna

        for (int i = 0; i < vt\_part\_size; i++) // Obliczenie iloczynu częsciowych wektorów

        {

            score += (vt\_1[i] \* vt\_2[i]);

        }

        fprintf(stderr, "Iloczyn skalarny procesora %d wynosi: %f\n", rank, score); // Wyświetlenie informacji o obliczonym częściowym iloczynie skalarnym

        MPI\_Barrier(MPI\_COMM\_WORLD); // Bariera synchronizacyjna

        MPI\_Gather(&score, 1, MPI\_DOUBLE, scores, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD); // Zebranie wyników z wszystkich procesów grupie

        score = 0; // Wyzerowanie iloczynu skalarnego

        for (int i = 0; i < size; i++) // Połączenie wyników częściowych

        {

            score += scores[i];

        }

        fprintf(stderr, "Iloczyn skalarny wektorów %d-elementowych dla %d procesorów wynosi: %f\n", vt\_normal\_size, size, score); // Wyświetlenie informacji o obliczonym iloczynie skalarnym

        free(vt\_1); // Zwolnienie obszaru pamięci wektora 1

        free(vt\_2); // Zwolnienie obszaru pamięci wektora 2

        free(vt\_1\_part); // Zwolnienie obszaru pamięci wektora częściowego 1

        free(vt\_2\_part); // Zwolnienie obszaru pamięci wektora częściowego 2

        free(scores); // Zwolnienie obszaru pamięci tablicy wyników

    }

    else // Instrukcje wykonywane przez procesy podległe

    {

        MPI\_Barrier(MPI\_COMM\_WORLD); // Bariera synchronizacyjna

        MPI\_Bcast(&vt\_real\_size, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); // Odebranie rozmiaru wektora

        MPI\_Bcast(&vt\_part\_size, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); // Odebranie rozmiaru części wektora

        vt\_1 = malloc(vt\_real\_size \* sizeof(double)); // Zaalokowanie pamięci na pierwszy wektor

        vt\_2 = malloc(vt\_real\_size \* sizeof(double)); // Zaalokowanie pamięci na drugi wektor

        vt\_1\_part = malloc(vt\_part\_size \* sizeof(double)); // Zaalokowanie pamięci na pierwszy wektor częsciowy

        vt\_2\_part = malloc(vt\_part\_size \* sizeof(double)); // Zaalokowanie pamięci na drugi wektor częściowy

        MPI\_Scatter(vt\_1, vt\_part\_size, MPI\_DOUBLE, vt\_1\_part, vt\_part\_size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD); // Odebranie podzielonego wektora 1 na kawałki

        MPI\_Scatter(vt\_2, vt\_part\_size, MPI\_DOUBLE, vt\_2\_part, vt\_part\_size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD); // Odebranie podzielonego wektora 1 na kawałki

        for (int i = 0; i < vt\_part\_size; i++) // Obliczenie iloczynu częsciowych wektorów

        {

            score += (vt\_1\_part[i] \* vt\_2\_part[i]);

        }

        fprintf(stderr, "Iloczyn skalarny procesora %d wynosi: %f\n", rank, score); // Wyświetlenie informacji o obliczonym częściowym iloczynie skalarnym

        MPI\_Barrier(MPI\_COMM\_WORLD); // Bariera synchronizacyjna

        MPI\_Gather(&score, 1, MPI\_DOUBLE, NULL, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD); // Wysłanie wyników do procesu zbierającego wyniki

        free(vt\_1); // Zwolnienie obszaru pamięci wektora 1

        free(vt\_2); // Zwolnienie obszaru pamięci wektora 2

        free(vt\_1\_part); // Zwolnienie obszaru pamięci wektora częściowego 1

        free(vt\_2\_part); // Zwolnienie obszaru pamięci wektora częściowego 2

    }

    clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end); // Pobranie informacji o czasie zakończenia obliczeń

    MPI\_Finalize(); // Koniec obliczeń MPI

    if(rank == 0)

    {

        u\_int64\_t delta\_us = (end.tv\_sec - start.tv\_sec) \* 1000000 + (end.tv\_nsec - start.tv\_nsec) / 1000; // Obliczenie różnicy między czasem końca obliczeń, a rozpoczęciem w mikrosekundach

        fprintf(stderr, "Obliczenia trwały %6.2f milisekund\n", (double) delta\_us / 1000); // Wyświetlenie informacji na temat czasu obliczeń

    }

    return 0; // Wyjście z programu

}

# Kod źródłowy środowiska MPI projekt\_2.c, w którym została ustalona relacja między procesami

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <time.h>

//  • Dla środowiska gdzie nie została ustalona relacja pomiędzy procesami

//  kompilacja   mpicc -o projekt\_1 projekt\_1.c

//  uruchamianie mpirun --host localhost:liczba\_procesorów/wątków ./projekt\_1

int main(int argc, char \*\* argv)

{

    //Deklaracja i incjalizacja zmienych

    double \*vt\_1, \*vt\_1\_part, \*vt\_2, \*vt\_2\_part, \*scores, score = 0; // Wskaźniki do tablic liczb typu double (nasze wektory i pod\_wektory) i tablicy na wyniki oraz sumy dla każdego procesu z osobna

    int vt\_normal\_size; // Zmienna przechowująca ilość elementów dla każdego wektora, jest ona inicjalizowana przez użytkownika

    int vt\_extend\_size; // Zmienna przechowująca ilość elementów (równych 0), o które został rozszerzony wektor

    int vt\_real\_size; // Zmienna przechowująca realną ilość elementów dla każdego wektora

    int vt\_part\_size; // Zmienna przechowująca ilość elementów wektorów jaki przypada na każdy z procesów

    int rank; // Numer mojego procesu

    int size; // Liczba procesów

    struct timespec start, end; // Zmienna przechowująca czas rozpoczęcia i końca obliczeń

    MPI\_Comm ring\_hyper\_d3; // Zmienna przechowująca moją grupę procesów

    /\* Definicja struktury grafu!!! \*/

    int nnodes = 8; // Liczba węzłów

    int index[8] = {0, 1, 2, 3, 4, 5, 6, 7}; // Definicja indeksów

    int edges[16] = {0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 0}; // Definicja krawędzi

    //int edges[16] = {1, 0, 2, 1, 3, 2, 4, 3, 5, 4, 6, 5, 7, 6, 0, 7}; // Definicja krawędzi

    int reorder = 1; // Zezwolenie na zmianę kolejności procesów w celu poprawy efektywności

    MPI\_Init(&argc, &argv); // Rozpoczęcie obliczeń MPI

    MPI\_Graph\_create(MPI\_COMM\_WORLD, nnodes, index, edges,reorder, &ring\_hyper\_d3); // Utworzenie struktury grafu, według powyższej definicji sturktury

    MPI\_Comm\_rank(ring\_hyper\_d3, &rank); // Pobranie aktualnego numeru procesu

    MPI\_Comm\_size(ring\_hyper\_d3, &size); // Pobranie liczby procesów

    if(rank == 0) // Instrukcje wykonywane dla głównego procesu programu (proces root)

    {

        fprintf(stderr, "Podaj długośc wektorów (liczba całkowita większa od 0): "); // Wyświetlenie komunikatu z prośbą o podanie długości wektora

        scanf("%d", &vt\_normal\_size); // Pobranie długości wektów od użytkownika

        if(vt\_normal\_size < 1) // Wyjście z programu dla wektora mniejszego niż 1

        {

            printf("Podany rozmiar wektorów jest zbyt mały, ilość elementów jest mniejsza niż ilość procesów: %d.\n", size);

            MPI\_Finalize(); // Koniec obliczeń MPI

            exit(1); // Wyjście z programu z kodem błędu

        }

        vt\_extend\_size = size - (vt\_normal\_size % size); // Obliczenie o ile elementów należy rozszerzyć wektor, aby można było podzielić je równo dla każdego procesu

        vt\_real\_size = vt\_normal\_size + vt\_extend\_size; // Obliczenie realnej długości wektorów

        vt\_part\_size = vt\_real\_size / size; // Obliczenie ilości elementów przypadających na każdy proces

        vt\_1 = malloc(vt\_real\_size \* sizeof(double)); // Zaalokowanie pamięci na pierwszy wektor

        vt\_2 = malloc(vt\_real\_size \* sizeof(double)); // Zaalokowanie pamięci na drugi wektor

        vt\_1\_part = malloc(vt\_part\_size \* sizeof(double)); // Zaalokowanie pamięci na pierwszy wektor częsciowy

        vt\_2\_part = malloc(vt\_part\_size \* sizeof(double)); // Zaalokowanie pamięci na drugi wektor częściowy

        scores = malloc(size \* sizeof(double)); // Zaalokowanie pamięci na tablice wyników

        for(int i = 0; i < vt\_real\_size; i++) // Inicjalizacja wektorów liczbami losowymi

        {

            if(i < vt\_normal\_size) // Wypełnienie elementów wektorów liczbami typu rzeczywistego z przedziału (-25, 25)

            {

                srand(time(NULL) + rand());

                vt\_1[i] = (double) rand() / RAND\_MAX \* 50 - 25;

                vt\_2[i] = (double) rand() / RAND\_MAX \* 50 - 25;

            }

            else // Dla dodatkowo wygenerowanych elementów wartość wynosi 0, aby wynik obliczeń nie został zakłamany

            {

                vt\_1[i] = 0;

                vt\_2[i] = 0;

            }

        }

        clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start); // Pobranie czasu rozpoczęcia obliczeń

        MPI\_Bcast(&vt\_real\_size, 1, MPI\_INT, 0, ring\_hyper\_d3); // Wysłanie rozmiaru wektora

        MPI\_Bcast(&vt\_part\_size, 1, MPI\_INT, 0, ring\_hyper\_d3); // Wysłanie rozmiaru części wektora

        MPI\_Scatter(vt\_1, vt\_part\_size, MPI\_DOUBLE, vt\_1\_part, vt\_part\_size, MPI\_DOUBLE, 0, ring\_hyper\_d3); // Wysłanie podzielonego wektora 1 na kawałki do procesów w grupie

        MPI\_Scatter(vt\_2, vt\_part\_size, MPI\_DOUBLE, vt\_2\_part, vt\_part\_size, MPI\_DOUBLE, 0, ring\_hyper\_d3); // Wysłanie podzielonego wektora 1 na kawałki do procesów w grupie

        MPI\_Barrier(ring\_hyper\_d3); // Bariera synchronizacyjna

        for (int i = 0; i < vt\_part\_size; i++) // Obliczenie iloczynu częsciowych wektorów

        {

            score += (vt\_1[i] \* vt\_2[i]);

        }

        fprintf(stderr, "Iloczyn skalarny procesora %d wynosi: %f\n", rank, score); // Wyświetlenie informacji o obliczonym częściowym iloczynie skalarnym

        MPI\_Barrier(ring\_hyper\_d3); // Bariera synchronizacyjna

        MPI\_Gather(&score, 1, MPI\_DOUBLE, scores, 1, MPI\_DOUBLE, 0, ring\_hyper\_d3); // Zebranie wyników z wszystkich procesów grupie

        score = 0; // Wyzerowanie iloczynu skalarnego

        for (int i = 0; i < size; i++) // Połączenie wyników częściowych

        {

            score += scores[i];

        }

        fprintf(stderr, "Iloczyn skalarny wektorów %d-elementowych dla %d procesorów wynosi: %f\n", vt\_normal\_size, size, score); // Wyświetlenie informacji o obliczonym iloczynie skalarnym

        clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end); // Pobranie informacji o czasie zakończenia obliczeń

        u\_int64\_t delta\_us = (end.tv\_sec - start.tv\_sec) \* 1000000 + (end.tv\_nsec - start.tv\_nsec) / 1000; // Obliczenie różnicy między czasem końca obliczeń, a rozpoczęciem w mikrosekundach

        fprintf(stderr, "Obliczenia trwały %6.2f milisekund\n", (double) delta\_us / 1000); // Wyświetlenie informacji na temat czasu obliczeń

        free(vt\_1); // Zwolnienie obszaru pamięci wektora 1

        free(vt\_2); // Zwolnienie obszaru pamięci wektora 2

        free(vt\_1\_part); // Zwolnienie obszaru pamięci wektora częściowego 1

        free(vt\_2\_part); // Zwolnienie obszaru pamięci wektora częściowego 2

        free(scores); // Zwolnienie obszaru pamięci tablicy wyników

    }

    else // Instrukcje wykonywane przez procesy podległe

    {

        MPI\_Barrier(ring\_hyper\_d3); // Bariera synchronizacyjna

        MPI\_Bcast(&vt\_real\_size, 1, MPI\_INT, 0, ring\_hyper\_d3); // Odebranie rozmiaru wektora

        MPI\_Bcast(&vt\_part\_size, 1, MPI\_INT, 0, ring\_hyper\_d3); // Odebranie rozmiaru części wektora

        vt\_1 = malloc(vt\_real\_size \* sizeof(double)); // Zaalokowanie pamięci na pierwszy wektor

        vt\_2 = malloc(vt\_real\_size \* sizeof(double)); // Zaalokowanie pamięci na drugi wektor

        vt\_1\_part = malloc(vt\_part\_size \* sizeof(double)); // Zaalokowanie pamięci na pierwszy wektor częsciowy

        vt\_2\_part = malloc(vt\_part\_size \* sizeof(double)); // Zaalokowanie pamięci na drugi wektor częściowy

        MPI\_Scatter(vt\_1, vt\_part\_size, MPI\_DOUBLE, vt\_1\_part, vt\_part\_size, MPI\_DOUBLE, 0, ring\_hyper\_d3); // Odebranie podzielonego wektora 1 na kawałki

        MPI\_Scatter(vt\_2, vt\_part\_size, MPI\_DOUBLE, vt\_2\_part, vt\_part\_size, MPI\_DOUBLE, 0, ring\_hyper\_d3); // Odebranie podzielonego wektora 1 na kawałki

        for (int i = 0; i < vt\_part\_size; i++) // Obliczenie iloczynu częsciowych wektorów

        {

            score += (vt\_1\_part[i] \* vt\_2\_part[i]);

        }

        fprintf(stderr, "Iloczyn skalarny procesora %d wynosi: %f\n", rank, score); // Wyświetlenie informacji o obliczonym częściowym iloczynie skalarnym

        MPI\_Barrier(ring\_hyper\_d3); // Bariera synchronizacyjna

        MPI\_Gather(&score, 1, MPI\_DOUBLE, NULL, 1, MPI\_DOUBLE, 0, ring\_hyper\_d3); // Wysłanie wyników do procesu zbierającego wyniki

        free(vt\_1); // Zwolnienie obszaru pamięci wektora 1

        free(vt\_2); // Zwolnienie obszaru pamięci wektora 2

        free(vt\_1\_part); // Zwolnienie obszaru pamięci wektora częściowego 1

        free(vt\_2\_part); // Zwolnienie obszaru pamięci wektora częściowego 2

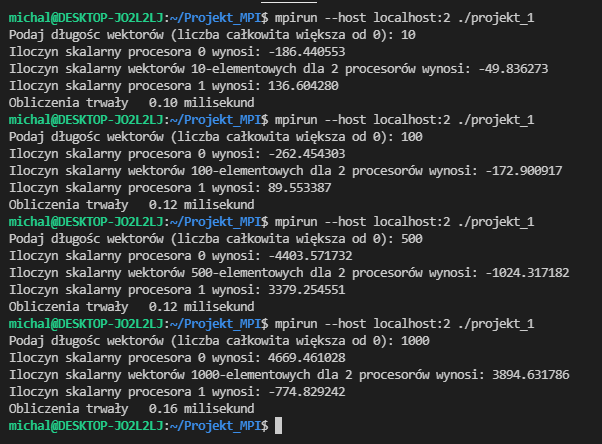
    }

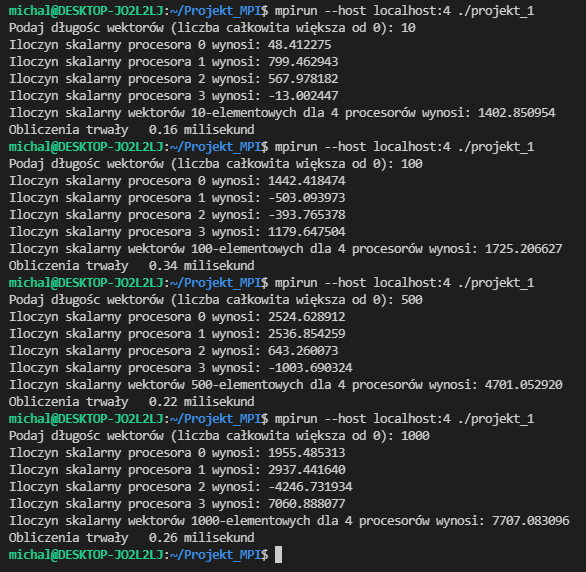
    MPI\_Finalize(); // Koniec obliczeń MPI

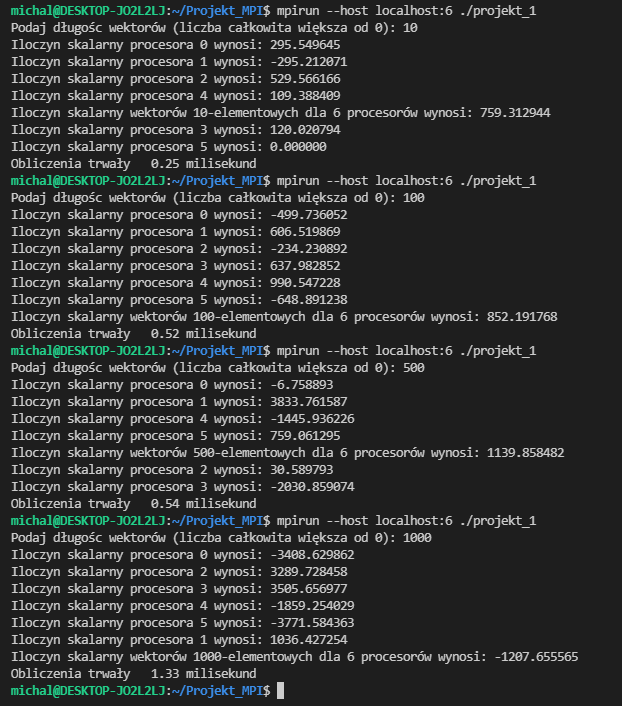
    return 0; // Wyjście z programu

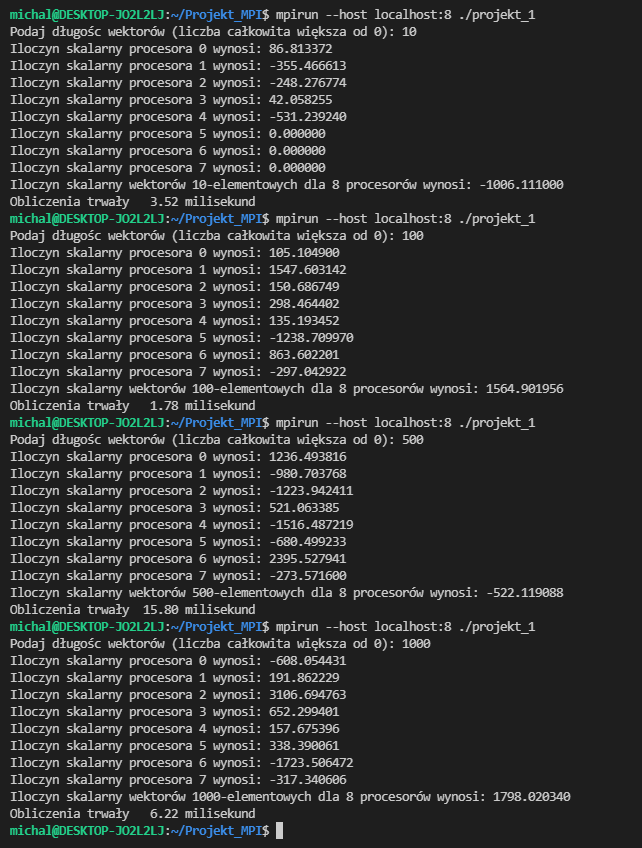
}

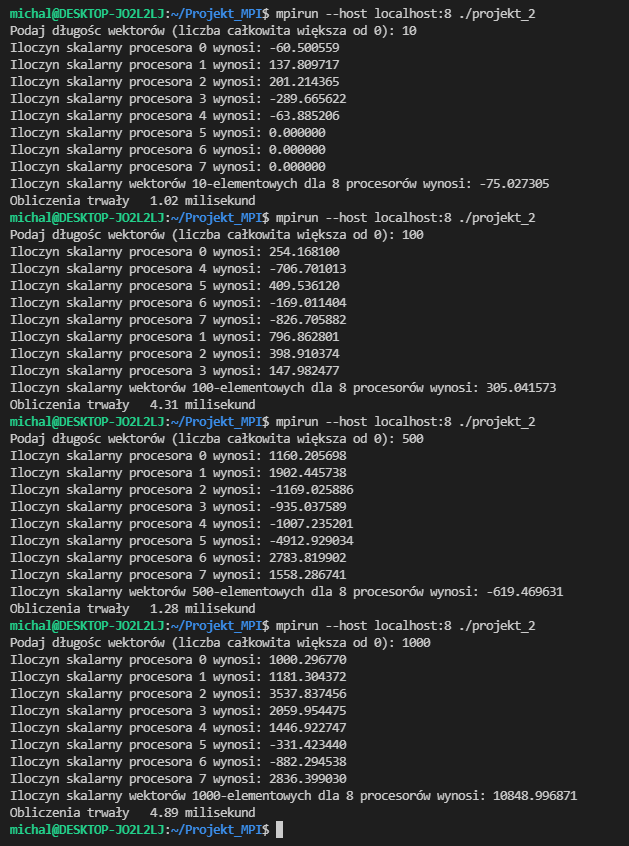
# Działanie aplikacji i wykonanie eksperymentów

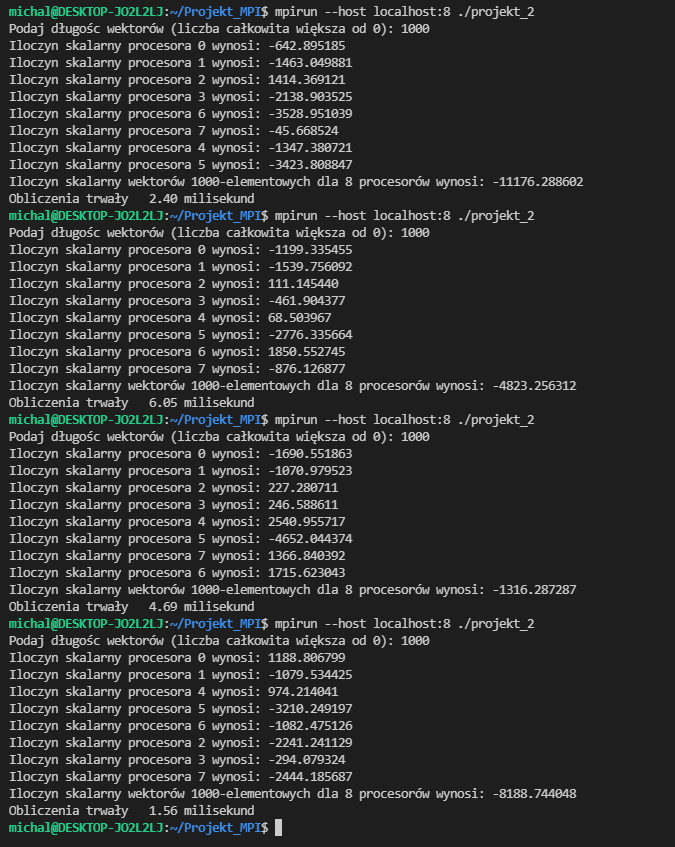












Rysunek Dowód na zakłamanie wyników

# Podsumowanie wyników

|  |  |  |  |
| --- | --- | --- | --- |
| Typ relacji | Liczba procesorów | Rozmiar wektorów | Czas w milisekundach |
| Brak | 2 | 10 | 0.10 |
| Brak | 4 | 10 | 0.16 |
| Brak | 6 | 10 | 0.25 |
| Brak | 8 | 10 | 3.52 |
| Pierścień ( hypercube d=3) | 8 | 10 | 1.02 |
|  |  |  |  |
| Brak | 2 | 100 | 0.12 |
| Brak | 4 | 100 | 0.34 |
| Brak | 6 | 100 | 0.52 |
| Brak | 8 | 100 | 1.78 |
| Pierścień ( hypercube d=3) | 8 | 100 | 4.31 |
|  |  |  |  |
| Brak | 2 | 500 | 0.12 |
| Brak | 4 | 500 | 0.22 |
| Brak | 6 | 500 | 0.54 |
| Brak | 8 | 500 | 15.8 |
| Pierścień ( hypercube d=3) | 8 | 500 | 1.28 |
|  |  |  |  |
| Brak | 2 | 1000 | 0.16 |
| Brak | 4 | 1000 | 0.26 |
| Brak | 6 | 1000 | 1.33 |
| Brak | 8 | 1000 | 6.22 |
| Pierścień ( hypercube d=3) | 8 | 1000 | 4.89 |

# Wnioski

Pierwsze rzecz na którą należy zwrócić uwagę jest częściowe zakłamanie wyników, ponieważ program uruchamiany jest na komputerze z zainstalowaną sporą ilością usług np. kilka baz danych, komunikatory internetowe itd. Z tego względu obciążenie procesora, może być różne. Wpływ na wyniki ma również to, iż wektory generowane są losowo, dlatego dane wejściowe nigdy nie są takie same. Jeśli porównamy wyniki wykonania programu z ustaloną relacją procesów i programu z brakiem takiej relacji dla takiej samej liczby procesów, to na pierwszy rzut oka widać, że średni czas obliczeń jest krótszy dla programu z ustaloną relacją procesów. Natomiast jeśli porównamy wyniki obliczeń dla różnej ilości procesów, dla nieustalonej relacji procesorów, to nie wygląda to już tak kolorowo. Większa liczba procesów nie powoduje przyśpieszenia obliczeń, jeśli same obliczenia nie są zbyt skomplikowane i dane wejściowe są małe. Wynika to prawdopodobnie z faktu, iż samo wygenerowanie struktury procesów i wysłanie tych informacji do innych procesów, także wymaga czasu. W połączeniu z prostymi algorytmami, może to spowodować sumarycznie dłuższy czas obliczeń, niż dla programu bez podziału na procesy. Dlatego zawsze należy się zastanowić, czy wdrożenie technologii systemów rozproszonych ma sens dla każdego danego przykładu z osobna.