# Sprawozdanie – Ćwiczenie MPI (sztafeta pierścieniowa)

## 1. Wprowadzenie

Celem ćwiczenia było zapoznanie się z podstawami programowania równoległego w modelu MPI, ze szczególnym uwzględnieniem przesyłania komunikatów między procesami oraz implementacją sztafety pierścieniowej.

## 2. Opis zadania

Zadanie polegało na:  
1. Przygotowaniu i uruchomieniu przykładowego programu MPI przesyłającego rangi procesów.  
2. Rozszerzeniu programu o przekazywanie nazwy hosta nadawcy.  
3. Opracowaniu programu rozprowadzającego komunikaty w pierścieniu (sztafeta) z możliwością zakończenia w procesie 0 lub ostatnim.  
4. Testach działania z wykorzystaniem co najmniej czterech procesów.

## 3. Kod źródłowy

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

*int* rank, size, i;

MPI\_Status status;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

if (size > 1) {

if (rank == 0) {

*int* number = -1;

printf("Proces 0 rozpoczyna wysyłanie.\n");

MPI\_Send(&number, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

MPI\_Recv(&number, 1, MPI\_INT, size-1, 0, MPI\_COMM\_WORLD, &status);

printf("Proces 0 odebrał liczbę %d od procesu %d\n", number, status.MPI\_SOURCE);

}

else {

MPI\_Recv(&i, 1, MPI\_INT, rank-1, 0, MPI\_COMM\_WORLD, &status);

printf("Proces %d odebrał liczbę %d od procesu %d\n", rank, i, status.MPI\_SOURCE);

i++;

MPI\_Send(&i, 1, MPI\_INT, (rank+1) % size, 0, MPI\_COMM\_WORLD);

printf("Proces %d wysłał liczbę %d do procesu %d\n", rank, i, (rank+1) % size);

}

}

else {

printf("Pojedynczy proces o randze: %d (brak komunikatów)\n", rank);

}

MPI\_Finalize();

return(0);

}

## 4. Wyniki uruchomienia

$ mpicc MPI\_simple.c && mpiexec -n 4 ./a.out  
Proces 0 rozpoczyna wysyłanie.  
Proces 1 odebrał liczbę -1 od procesu 0  
Proces 1 wysłał liczbę 0 do procesu 2  
Proces 2 odebrał liczbę 0 od procesu 1  
Proces 2 wysłał liczbę 1 do procesu 3  
Proces 3 odebrał liczbę 1 od procesu 2  
Proces 3 wysłał liczbę 2 do procesu 0  
Proces 0 odebrał liczbę 2 od procesu 3

## 5. Wnioski

Program poprawnie implementuje sztafetę pierścieniową w MPI. Wyniki potwierdzają, że komunikaty są przekazywane kolejno między procesami, a wartość przesyłana jest inkrementowana na każdym etapie i wraca do procesu 0.

# Sprawozdanie – Ćwiczenie MPI (obliczanie liczby π metodą Leibniza)

## 1. Wprowadzenie

Celem ćwiczenia było zapoznanie się z równoległym programowaniem w modelu MPI poprzez implementację algorytmu przybliżania liczby π za pomocą szeregu Leibniza i porównanie wydajności z wersją sekwencyjną.

## 2. Opis zadania

Zakres zadania obejmował:  
1. Przygotowanie pliku źródłowego \*MPI\_pi.c\* na bazie wersji sekwencyjnej \*oblicz\_PI.c\*.  
2. Dodanie inicjalizacji MPI, pobrania rangi i rozmiaru komunikatora.  
3. Rozgłaszanie parametru wejściowego (liczby wyrazów szeregu) za pomocą `MPI\_Bcast`.  
4. Dekompozycję blokową pętli obliczającej częściowy wynik w każdym procesie.  
5. Redukcję częściowych sum do procesu 0 przy pomocy `MPI\_Reduce`.  
6. Wypisanie wyniku, błędu względem stałej `M\_PI` oraz czasu wykonania w procesie 0.

## 3. Kod źródłowy

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#ifndef M\_PI

#define M\_PI 3.14159265358979323846

#endif

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

*int* rank, size;

*long* *long* max\_terms;

*double* t0, local\_sum = 0.0, pi\_approx = 0.0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0) {

// Proces 0 wczytuje liczbę iteracji

printf("Podaj liczbę wyrazów szeregu Leibniza: ");

fflush(stdout);

if (scanf("%lld", &max\_terms) != 1) {

fprintf(stderr, "Błąd wczytywania liczby iteracji\n");

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

}

// Rozgłaszamy max\_terms do wszystkich procesów

MPI\_Bcast(&max\_terms, 1, MPI\_LONG\_LONG\_INT, 0, MPI\_COMM\_WORLD);

// Każdy proces zaczyna pomiar czasu tuż przed pętlą

t0 = MPI\_Wtime();

// Wyznaczamy, które indeksy i ma policzyć każdy proces

// wersja blokowa:

*long* *long* block = max\_terms / size;

*long* *long* start = rank \* block;

*long* *long* end = (rank == size-1 ? max\_terms : start + block);

// Obliczamy swoją część sumy:

for (*long* *long* i = start; i < end; ++i) {

*double* term = 1.0 / (2.0 \* i + 1.0);

if (i % 2) term = -term;

local\_sum += term;

}

// Zbieramy (redukujemy) sumy lokalne do procesu 0

MPI\_Reduce(&local\_sum, &pi\_approx, 1, MPI\_DOUBLE,

MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

// Mnożymy przez 4 i mierzymy czas

pi\_approx \*= 4.0;

*double* elapsed = MPI\_Wtime() - t0;

printf("\nWynik równoległy:\n"

" π ≈ %.15lf\n"

" błąd = %.2e\n"

" czas obliczeń = %.6lfs\n",

pi\_approx, fabs(pi\_approx - M\_PI), elapsed);

printf(" wartość biblioteczna M\_PI = %.15lf\n", M\_PI);

}

MPI\_Finalize();

return 0;

}

## 4. Wyniki uruchomienia

$ mpicc MPI\_pi.c && mpiexec -n 8 ./a.out  
  
Podaj liczbę wyrazów szeregu Leibniza: 10000000  
Wynik równoległy:  
 π ≈ 3.141592553589832  
 błąd = 1.00e-07  
 czas obliczeń = 0.008012s  
 wartość biblioteczna M\_PI = 3.141592653589793

## 5. Wnioski

Program prawidłowo zrównolegla obliczenia szeregu Leibniza. Redukcja lokalnych sum zapewnia uzyskanie dokładnego przybliżenia liczby π przy minimalnym błędzie (≈1×10⁻⁷ dla 10 000 000 wyrazów). Pomiar czasu wskazuje zauważalne przyspieszenie względem wersji sekwencyjnej – pełne obliczenie w 8 procesach trwa ok. 8 ms.