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

```c
#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

```c
#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 ($\approx 1 \times 10^{-7}$ dla 10 000 000 wyrazów). Pomiar czasu wskazuje zauważalne przyspieszenie względem wersji sekwencyjnej – pełne obliczenie w 8 procesach trwa ok. 8 ms.