Sprawozdanie – programowanie równoległe i rozproszone

Ćwiczenie 2 – wątki w Java oraz Python

Adrian Nowosielski, Cezary Skorupski

1. Inicjalizacja i argumenty wiersza poleceń

Program inicjuje MPI i pobiera numer oraz rozmiar komunikatora MPI.

Analizowane są argumenty wiersza poleceń w celu uzyskania przedziału całkowania [begin, end] oraz całkowitej liczby punktów (num_points), dla których ma zostać obliczona funkcja.

2. Funkcja całkowania

Całka jest obliczana dla danej funkcji func na przedziale [begin, end] przy użyciu określonej liczby punktów (num_points). Funkcja ta wykorzystuje prostą zasadę prostokątów do przybliżenia całki.

3. Podział pracy

Przedział całkowania [begin, end] jest dzielony pomiędzy procesy. Każdy proces oblicza swoją część całki na podstawie swojej rangi oraz całkowitej liczby procesów.

4. Obliczenia całkowania

Każdy proces oblicza swoją część całki przy użyciu funkcji integrate.

5. Komunikacja i redukcja

Proces główny (o randze 0) zbiera obliczone częściowe wyniki od innych procesów przy użyciu komunikacji punkt-punkt (MPI_Send i MPI_Recv). Każdy proces wysyła swoje obliczony wynik do głównego procesu.

Proces główny agreguje wszystkie częściowe wyniki przy użyciu operacji redukcji (MPI_Reduce), sumując wszystkie częściowe całki w celu uzyskania ostatecznego wyniku.

6. Wyjście wyniku

Po zebraniu wszystkich wyników i obliczeniu ostatecznej całki, główny proces wypisuje wynik.

7. Warianty implementacji MPI

mpi-sending-main.c: Ta implementacja używa jawnie MPI_Send i MPI_Recv do komunikacji pomiędzy procesami. Proces główny wysyła pracę i zbiera wyniki od procesów roboczych.

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

double function(double x) {
    return x * x;
}

double integrate(double (*f)(double), double begin, double ending, int number_of_points) {
    double h = (ending - begin) / number_of_points;
    double sum = 0;
    for (int i = 1; i <= number_of_points; ++i) {
        sum += f(begin + i * h);
    }

    return sum * h;
}

int main(int argc, char **argv) {
    if (argc < 4) {
        fprintf(stderr, "There was a problem with the input arguments: [%s]\n", argv[0]);
        fprintf(stderr, "Program should get: [begin_of_integration_range]: double, [end_of_integration_range]: double, [number_of_points]: int");
        exit(EXIT_FAILURE);
    }

    MPI_Init(&argc, &argv);

    int process_rank, size_of_cluster;
    const int RANK_OF_ROOT_PROCESS = 0;
    const int NUMBER_OF_ELEMENTS_IN_BUFFER = 1;

    MPI_Comm_size(MPI_COMM_WORLD, &size_of_cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

    double begin_of_integration_range = atof(argv[1]);
    double end_of_integration_range = atof(argv[2]);
    int number_of_points_to_integrate = atoi(argv[3]);
    double global_dx = (end_of_integration_range - begin_of_integration_range) / number_of_points_to_integrate;

    double global_integration_result = 0.0;

    int points_for_each_process = number_of_points_to_integrate / size_of_cluster;
    double begin_of_range_for_process = begin_of_integration_range + process_rank * points_for_each_process * global_dx;
    double end_of_range_for_process = begin_of_range_for_process + points_for_each_process * global_dx;
    double single_process_integration_result = integrate(function, begin_of_range_for_process, end_of_range_for_process, points_for_each_process);
```

```c
    MPI_Request request;
    double single_result;
    MPI_Status status;

    if (process_rank != RANK_OF_ROOT_PROCESS) {
        MPI_Isend(&single_process_integration_result, NUMBER_OF_ELEMENTS_IN_BUFFER, MPI_DOUBLE, RANK_OF_ROOT_PROCESS, 0, MPI_COMM_WORLD, &request);
        MPI_Wait(&request, &status);
    } else {
        double final_integration_result = single_process_integration_result;

        for (int i = 1; i < size_of_cluster; i++) {
            MPI_Irecv(&single_result, NUMBER_OF_ELEMENTS_IN_BUFFER, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &request);
            MPI_Wait(&request, &status);
            final_integration_result += single_result;
        }

        printf("Final result of integrating function from range [%g] to [%g] is: %g\n", begin_of_integration_range, end_of_integration_range, final_integration_result);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

mpi-reduction-main.c: Ta implementacja wykorzystuje MPI_Reduce do agregacji wyników, co efektywnie obsługuje redukcję (np. sumowanie w tym przypadku) częściowych wyników od wszystkich procesów.

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

double function(double x) {
    return x * x;
}

double integrate(double (*f)(double), double begin, double end, int number_of_points) {
    double h = (end - begin) / number_of_points;
    double sum = 0;
    for (int i = 1; i <= number_of_points; ++i) {
        sum += f(begin + i * h);
    }

    return sum * h;
}


int main(int argc, char **argv) {
    if (argc < 4) {
        fprintf(stderr, "There was a problem with the input arguments: [%s]\n", argv[0]);
        fprintf(stderr, "Program should get: [begin_of_integration_range]: double, [end_of_integration_range]: double, [number_of_points]: int");
        exit(EXIT_FAILURE);
    }

    MPI_Init(&argc, &argv);

    int process_rank, size_of_cluster;
    const int RANK_OF_ROOT_PROCESS = 0;
    const int NUMBER_OF_ELEMENTS_IN_BUFFER = 1;

    MPI_Comm_size(MPI_COMM_WORLD, &size_of_cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

    double begin_of_integration_range = atof(argv[1]);
    double end_of_integration_range = atof(argv[2]);
    int number_of_points_to_integrate = atoi(argv[3]);
    double global_dx = (end_of_integration_range - begin_of_integration_range) / number_of_points_to_integrate;

    double global_integration_result = 0.0;

    int points_for_each_process = number_of_points_to_integrate / size_of_cluster;
    double begin_of_range_for_process = begin_of_integration_range + process_rank * points_for_each_process * global_dx;
    double end_of_range_for_process = begin_of_range_for_process + points_for_each_process * global_dx;
    double single_process_integration_result = integrate(function, begin_of_range_for_process, end_of_range_for_process, points_for_each_process);

    MPI_Reduce(&single_process_integration_result, &global_integration_result, NUMBER_OF_ELEMENTS_IN_BUFFER, MPI_DOUBLE, MPI_SUM, RANK_OF_ROOT_PROCESS, MPI_COMM_WORLD);

    if (process_rank == RANK_OF_ROOT_PROCESS) {
        printf("Final result of integrating function from range [%g] to [%g] is: %g", begin_of_integration_range, end_of_integration_range, global_integration_result);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

mpi-non-blocking.c: Ta wersja demonstruje komunikację nieblokującą (MPI_Isend i MPI_Irecv) do wysyłania i odbierania częściowych wyników asynchronicznie. Może to nakładać się na obliczenia, co potencjalnie poprawia wydajność.

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

double function(double x) {
    return x * x;
}

double integrate(double (*f)(double), double begin, double ending, int number_of_points) {
    double h = (ending - begin) / number_of_points;
    double sum = 0;
    for (int i = 1; i <= number_of_points; ++i) {
        sum += f(begin + i * h);
    }

    return sum * h;
}

int main(int argc, char **argv) {
    if (argc < 4) {
        fprintf(stderr, "There was a problem with the input arguments: [%s]\n", argv[0]);
        fprintf(stderr, "Program should get: [begin_of_integration_range]: double, [end_of_integration_range]: double, [number_of_points]: int");
        exit(EXIT_FAILURE);
    }

    MPI_Init(&argc, &argv);

    int process_rank, size_of_cluster;
    const int RANK_OF_ROOT_PROCESS = 0;
    const int NUMBER_OF_ELEMENTS_IN_BUFFER = 1;

    MPI_Comm_size(MPI_COMM_WORLD, &size_of_cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

    double begin_of_integration_range = atof(argv[1]);
    double end_of_integration_range = atof(argv[2]);
    int number_of_points_to_integrate = atoi(argv[3]);
    double global_dx = (end_of_integration_range - begin_of_integration_range) / number_of_points_to_integrate;

    double global_integration_result = 0.0;

    int points_for_each_process = number_of_points_to_integrate / size_of_cluster;
    double begin_of_range_for_process = begin_of_integration_range + process_rank * points_for_each_process * global_dx;
    double end_of_range_for_process = begin_of_range_for_process + points_for_each_process * global_dx;
    double single_process_integration_result = integrate(function, begin_of_range_for_process, end_of_range_for_process, points_for_each_process);
```

```c
    MPI_Request request;
    double single_result;
    MPI_Status status;

    if (process_rank != RANK_OF_ROOT_PROCESS) {
        MPI_Isend(&single_process_integration_result, NUMBER_OF_ELEMENTS_IN_BUFFER, MPI_DOUBLE, RANK_OF_ROOT_PROCESS, 0, MPI_COMM_WORLD, &request);
        MPI_Wait(&request, &status);
    } else {
        double final_integration_result = single_process_integration_result;

        for (int i = 1; i < size_of_cluster; i++) {
            MPI_Irecv(&single_result, NUMBER_OF_ELEMENTS_IN_BUFFER, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &request);
            MPI_Wait(&request, &status);
            final_integration_result += single_result;
        }

        printf("Final result of integrating function from range [%g] to [%g] is: %g\n", begin_of_integration_range, end_of_integration_range, final_integration_result);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

8. Porównanie czasów wykonania i dokładności rozwiązania

L.p.	Program	L. procesów	Początek przedziału całkowania	Koniec przedziału całkowania	Liczba punktów	czas [ms]	wynik
1	mpi-sending-main.c	4	0	10	100000	311.29	333.338
2	mpi-reduction-main.c	4	0	10	100000	442.44	333.338
3	mpi-non-blocking.c	4	0	10	100000	388.073	333.338
4	mpi-sending-main.c	4	0	1000	1000000	309.53	3.33334e+08

5	mpi-reduction-main.c	4	0	1000	1000000	414.90	3.3334e+08
6	mpi-non-blocking.c	4	0	1000	1000000	369.73	3.3334e+08
7	mpi-sending-main.c	2	0	1000	1000000	280.68	3.3334e+08
8	mpi-reduction-main.c	2	0	1000	1000000	298.59	3.3334e+08
9	mpi-non-blocking.c	2	0	1000	1000000	323.33	3.3334e+08

9. Wnioski:

Wybór Implementacji:

Jeśli zależy nam na prostocie i wydajności komunikacji, mpi-sending-main.c może być dobrym wyborem.

Jeśli zależy nam na wydajności obliczeń i redukcji, ale jesteśmy gotowi poświęcić nieco wydajności komunikacji, mpi-reduction-main.c jest odpowiedni.

mpi-non-blocking.c stanowi kompromis pomiędzy wydajnością obliczeń a komunikacją, co może być korzystne w przypadku równoważenia obciążenia i minimalizowania opóźnień.

Liczba Procesów:

W przypadku mniejszej liczby procesów (2), czas wykonania może być krótszy ze względu na mniejsze opóźnienia komunikacyjne.

Dokładność Rozwiązania:

Niezależnie od wybranej implementacji, dokładność rozwiązania (wartość całki) jest taka sama dla każdej konfiguracji.