

Sprawozdanie z przedmiotu Programowanie Równoległe i Rozproszone

MPI (Message Passing Interface)

Wykonał:

Imię i nazwisko

Magdalena Paszko

Nr indeksu

76024

Grupa

L2

Prowadzący : dr inż. Krzysztof Szerszeń

1. Wprowadzanie

MPI to nazwa standardu biblioteki przesyłania komunikatów dla potrzeb programowania równoległego. Pod skrótem MPI kryje się tylko formalna specyfikacja interfejsu, nie jest to nazwa żadnego konkretnego pakietu oprogramowania. Aktualnie obowiązująca wersja standardu MPI to 1.2, obecnie na ukończeniu są jednak prace nad wersją 2.0.

MPI realizuje model przetwarzania współbieżnego zwany **MIMD** (Multiple Instruction Multiple Data), a dokładniej **SPMD** (Single Program Multiple Data). Zakłada on, że ten sam kod źródłowy wykonuje się jednocześnie na kilku maszynach i procesy mogą przetwarzać równocześnie różne fragmenty danych, wymieniając informacje przy użyciu **komunikatów**

Takie podejście ma wiele zalet, z których najbardziej spektakularną jest chyba możliwość współbieżnych obliczeń wykonywanych na maszynach o zupełnie różnych architekturach (np. Linux-x86 oraz Solaris-Sparc). Zaletą (chyba:) jest również rezygnacja z koncepcji pamięci dzielonej i wynikające z tego ogólne uproszczenie programowania.

Realizując ten model, MPI umożliwia:

- Wymianę komunikatów między procesami
(Główny nacisk jest położony na wymianę danych, ale możliwe jest również wysyłanie komunikatów kontrolnych, czy synchronizacja procesów)
- Uzyskiwanie informacji o środowisku
(Typowy przykład to ilość aktywnych proces[ów/-orów], czy numer aktualnego procesu)
- Kontrolę nad systemem
(Inicjalizacja/kończenie programu, kontrola poprawności przesyłanych komunikatów itp.)

Wszystkie te rzeczy są realizowane przy minimalnym stopniu skomplikowania kodu źródłowego - nie mając pojęcia o **MPI** i znając podstawy programowania równoległego byłam w stanie zrozumieć programy przykładowe i próbować pisać własne.

2. Definicja

Całkowanie numeryczne

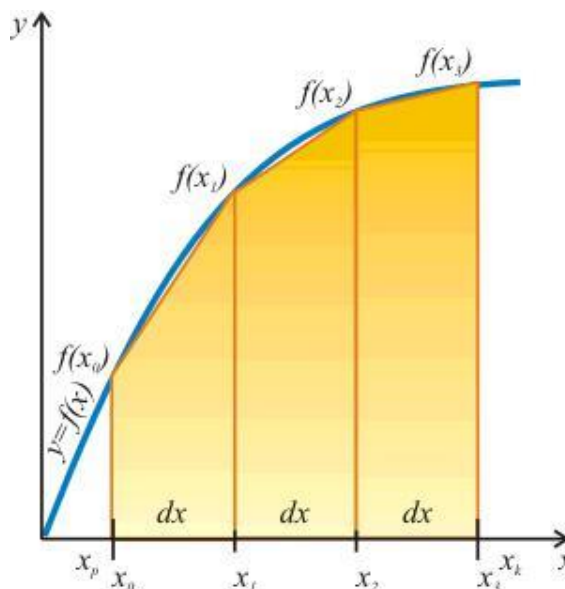
Metoda polegająca na przybliżonym obliczaniu całek oznaczonych.

Proste metody całkowania numerycznego polegają na przybliżeniu całki za pomocą odpowiedniej sumy ważonej wartości całkowanej funkcji w kilku punktach.

Do uzyskania dokładniejszego przybliżenia dzieli się przedział całkowania na niewielkie fragmenty.

Ostateczny wynik to suma oszacowań całek w poszczególnych podprzedziałach. Przedział zwykle dzieli się na równe podprzedziały.

Metoda trapezów



Przedział całkowania $\langle x_p, x_k \rangle$ dzielimy na $n+1$ równo odległych punktów $x_0, x_1, x_2, \dots, x_n$. Punkty te wyznaczamy w prosty sposób wg wzoru:

dla $i = 0, 1, 2, \dots, n$

$$x_i = x_p + \frac{i}{n}(x_k - x_p)$$

Obliczamy odległość między dwoma sąsiednimi punktami - będzie to wysokość każdego trapezu:

$$dx = \frac{x_k - x_p}{n}$$

Dla każdego wyznaczonego w ten sposób punktu obliczamy wartość funkcji $f(x)$ w tym punkcie:

$$f_i = f(x_i), \text{ dla } i = 1, 2, \dots, n$$

Pole pod wykresem funkcji przybliżane jest polami n trapezów. Pole i -tego trapezu obliczamy wg wzoru:

dla $i=1, 2, \dots, n$

$$f_i = f(x_i), \text{ dla } i = 1, 2, \dots, n$$

Przybliżona wartość całki jest sumą pól wszystkich otrzymanych w ten sposób trapezów:

$$s = P_1 + P_2 + \dots + P_n$$

czyli:

$$s = \frac{f_0 + f_1}{2} dx + \frac{f_1 + f_2}{2} dx + \frac{f_2 + f_3}{2} dx + \dots + \frac{f_{n-2} + f_{n-1}}{2} dx + \frac{f_{n-1} + f_n}{2} dx$$

$$s = \frac{dx}{2} (f_0 + f_1 + f_1 + f_2 + f_2 + f_3 + \dots + f_{n-2} + f_{n-1} + f_{n-1} + f_n)$$

$$s = \frac{dx}{2} (f_0 + 2f_1 + 2f_2 + \dots + 2f_{n-1} + f_n)$$

$$s = dx (f_1 + f_2 + \dots + f_{n-1} + \frac{f_0 + f_n}{2})$$

Wyprowadzony na końcu wzór jest podstawą przybliżonego wyliczania całki w metodzie trapezów.

$$\int_{x_p}^{x_k} f(x) dx \approx \frac{x_k - x_p}{n} \left(\sum_{i=1}^{n-1} f(x_p + i \frac{x_k - x_p}{n}) + \frac{f(x_p) + f(x_k)}{2} \right)$$

Wzór Leibniza

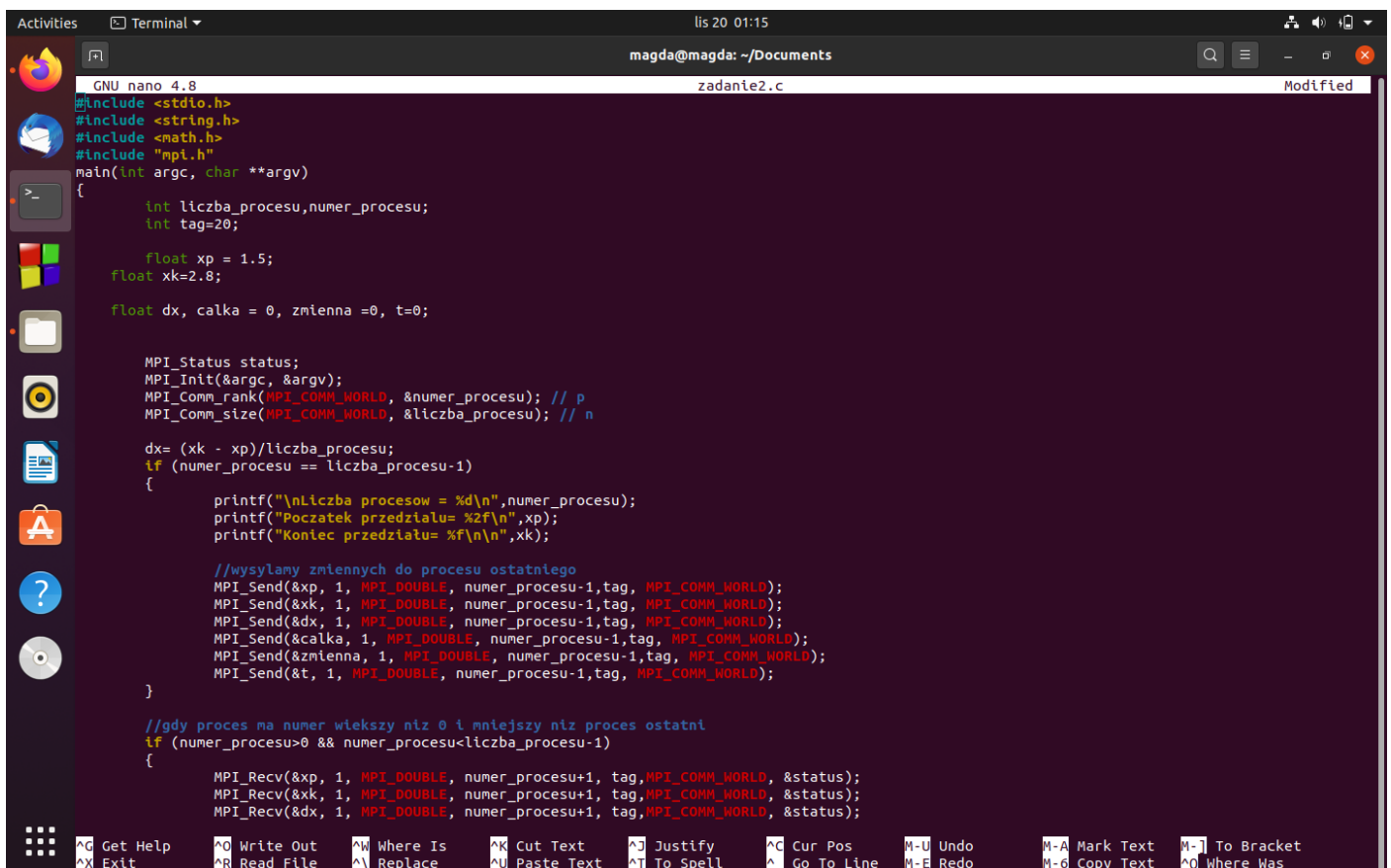
Wzór pozwalający obliczyć n-tą pochodną iloczynu funkcji.

Został wprowadzony przez niemieckiego matematyka Gottfrieda Leibniza.

$$\pi = 4 \cdot \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{2 \cdot n - 1} = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots \right)$$

3. Implementacja algorytmów

a) Metoda trapezów



```
GNU nano 4.8
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"
main(int argc, char **argv)
{
    int liczba_procesu, numer_procesu;
    int tag=20;

    float xp = 1.5;
    float xk=2.8;

    float dx, calka = 0, zmienna =0, t=0;

    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &numer_procesu); // p
    MPI_Comm_size(MPI_COMM_WORLD, &liczba_procesu); // n

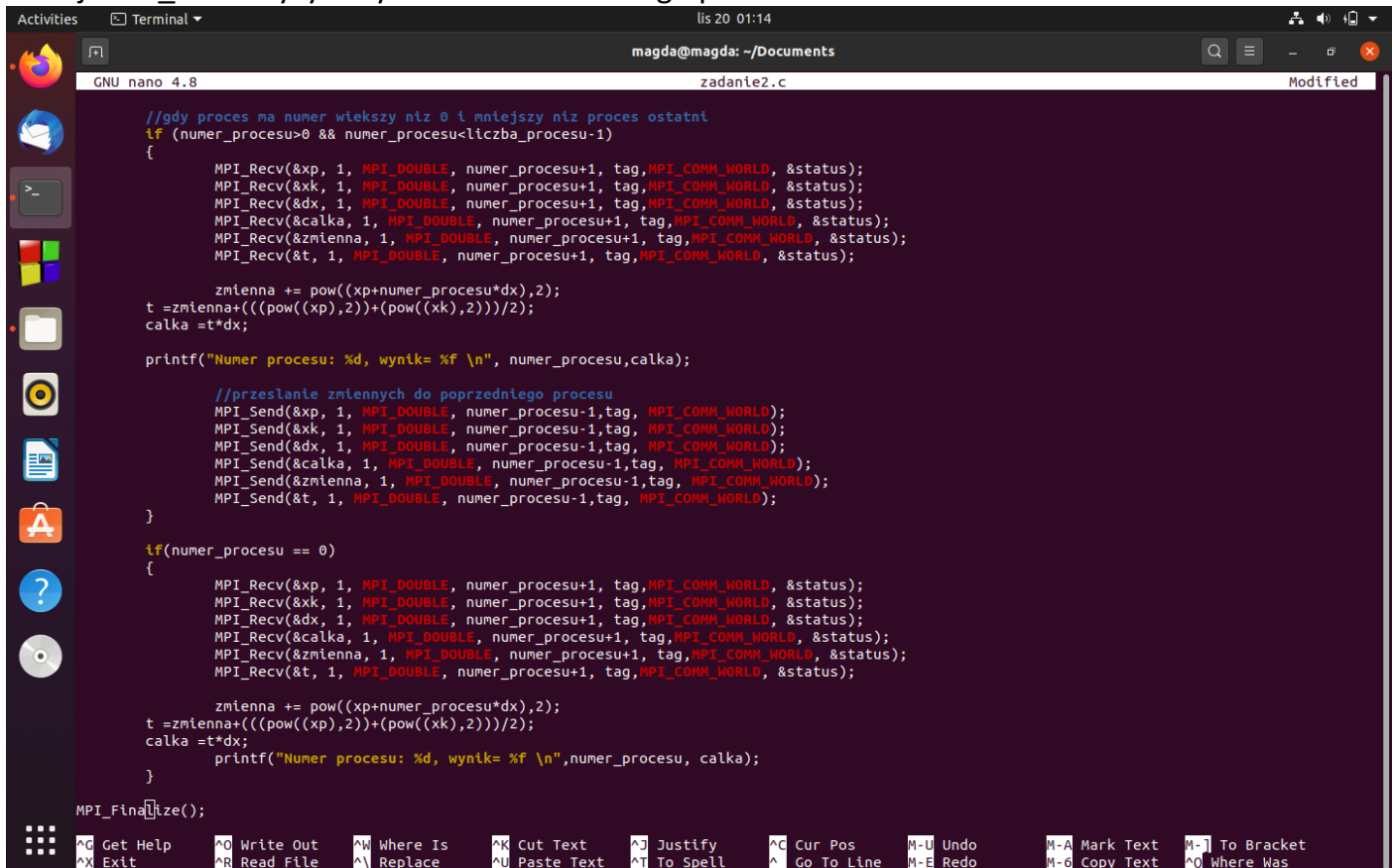
    dx = (xk - xp)/liczba_procesu;
    if (numer_procesu == liczba_procesu-1)
    {
        printf("\nliczba procesow = %d\n", numer_procesu);
        printf("Poczatek przedzialu= %2f\n", xp);
        printf("Koniec przedzialu= %f\n", xk);

        //wysylamy zmiennych do procesu ostatniego
        MPI_Send(&xp, 1, MPI_DOUBLE, numer_procesu-1, tag, MPI_COMM_WORLD);
        MPI_Send(&xk, 1, MPI_DOUBLE, numer_procesu-1, tag, MPI_COMM_WORLD);
        MPI_Send(&dx, 1, MPI_DOUBLE, numer_procesu-1, tag, MPI_COMM_WORLD);
        MPI_Send(&calka, 1, MPI_DOUBLE, numer_procesu-1, tag, MPI_COMM_WORLD);
        MPI_Send(&zmienna, 1, MPI_DOUBLE, numer_procesu-1, tag, MPI_COMM_WORLD);
        MPI_Send(&t, 1, MPI_DOUBLE, numer_procesu-1, tag, MPI_COMM_WORLD);
    }

    //gdy proces ma numer wiekszy niz 0 i mniejszy niz proces ostatni
    if (numer_procesu > 0 && numer_procesu < liczba_procesu-1)
    {
        MPI_Recv(&xp, 1, MPI_DOUBLE, numer_procesu+1, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&xk, 1, MPI_DOUBLE, numer_procesu+1, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&dx, 1, MPI_DOUBLE, numer_procesu+1, tag, MPI_COMM_WORLD, &status);
    }
}
```

Funkcja inicjalizuje środowisko wykonywania programu, m.in. tworzy domyślny komunikator MPI_COMM_WORLD. Dopiero od momentu wywołania MPI_Init można używać pozostałych funkcji MPI. Funkcja pobiera numer aktualnego procesu (w obrębie komunikatora comm) i umieszcza go w zmiennej numer_procesu. Funkcja pobiera ilość procesów (w obrębie komunikatora comm i umieszcza ją w zmiennej liczba_procesu. Następnie przypisujemy

wartość do dx według wzoru na całkowanie met. trapezów i wypisujemy dane. Dalej za pomocą funkcji MPI_Send wysyłamy dane do określonego procesu.



```
//gdy proces ma numer wiekszy niz 0 i mniejszy niz proces ostatni
if (numer_procesu>0 && numer_procesu<liczba_procesu-1)
{
    MPI_Recv(&xp, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);
    MPI_Recv(&xk, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);
    MPI_Recv(&dx, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);
    MPI_Recv(&calka, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);
    MPI_Recv(&zmienna, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);

    zmienna += pow((xp+numer_procesu*dx),2);
    t = zmienna+(((pow((xp),2))+pow((xk),2)))/2);
    calka = t*dx;

    printf("Numer procesu: %d, wynik= %f \n", numer_procesu,calka);

    //przeslanie zmiennych do poprzedniego procesu
    MPI_Send(&xp, 1, MPI_DOUBLE, numer_procesu-1,tag, MPI_COMM_WORLD);
    MPI_Send(&xk, 1, MPI_DOUBLE, numer_procesu-1,tag, MPI_COMM_WORLD);
    MPI_Send(&dx, 1, MPI_DOUBLE, numer_procesu-1,tag, MPI_COMM_WORLD);
    MPI_Send(&calka, 1, MPI_DOUBLE, numer_procesu-1,tag, MPI_COMM_WORLD);
    MPI_Send(&zmienna, 1, MPI_DOUBLE, numer_procesu-1,tag, MPI_COMM_WORLD);
    MPI_Send(&t, 1, MPI_DOUBLE, numer_procesu-1,tag, MPI_COMM_WORLD);
}

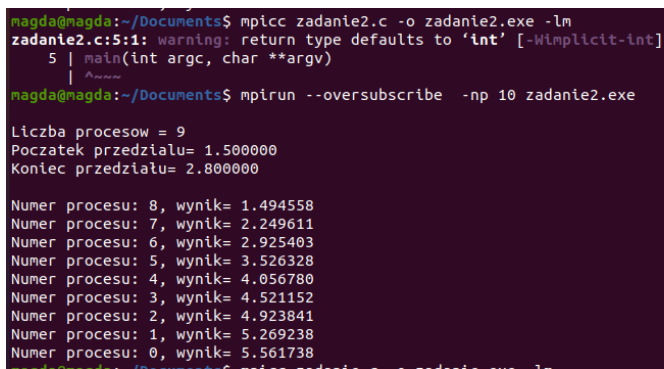
if(numer_procesu == 0)
{
    MPI_Recv(&xp, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);
    MPI_Recv(&xk, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);
    MPI_Recv(&dx, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);
    MPI_Recv(&calka, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);
    MPI_Recv(&zmienna, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);
    MPI_Recv(&t, 1, MPI_DOUBLE, numer_procesu+1, tag,MPI_COMM_WORLD, &status);

    zmienna += pow((xp+numer_procesu*dx),2);
    t = zmienna+(((pow((xp),2))+pow((xk),2)))/2);
    calka = t*dx;
    printf("Numer procesu: %d, wynik= %f \n",numer_procesu, calka);
}

MPI_Finalize();
```

Funkcja MPI_Recv odczytuje z kolejki komunikatora comm (z ewentualnym blokowaniem do czasu nadejścia) pierwszy komunikat od procesu source oznaczony znacznikiem tag typu int. Wynik umieszczany jest w buforze xp, xk, dx, calka, zmienna, t. Dalej według wzoru na całkę metodą trapezów obliczamy kolejne pola trapezów z całki, które są zapoczątkowane w ostatnim procesie n-1 i są przekazywane do kolejnych procesów w kolejności malejącej aż do procesu o numerze 0. Na koniec funkcja MPI_Finalize zwalnia zasoby używane przez MPI i przygotowuje system do zamknięcia.

Wynik programu:

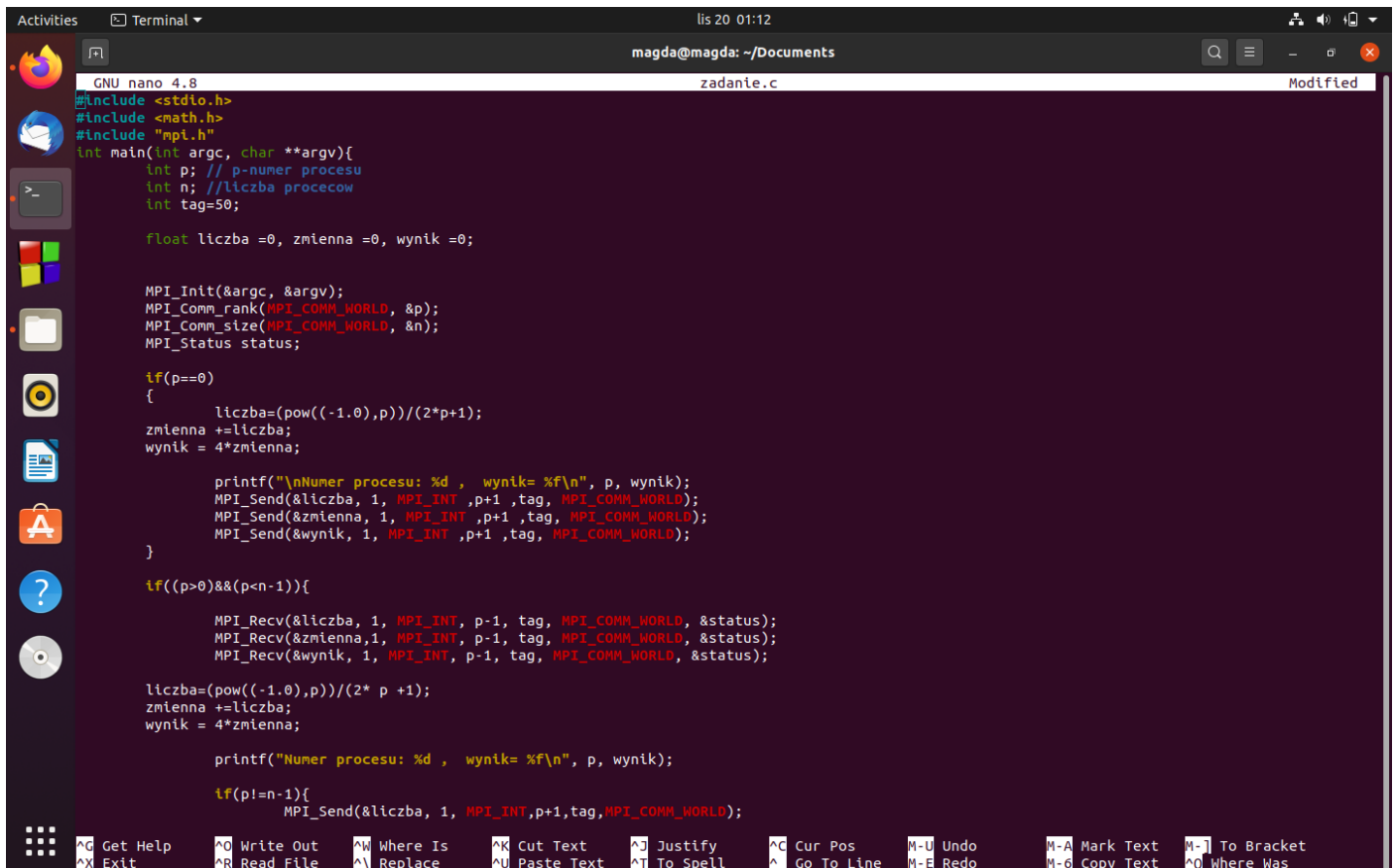


```
magda@magda:~/Documents$ mpicc zadanie2.c -o zadanie2.exe -lm
zadanie2.c:5:1: warning: return type defaults to 'int' [-Wimplicit-int]
5 | main(int argc, char **argv)
  | ^~~~~
magda@magda:~/Documents$ mpirun --oversubscribe -np 10 zadanie2.exe

Liczba procesow = 9
Początek przedziału= 1.500000
Koniec przedziału= 2.800000

Numer procesu: 8, wynik= 1.494558
Numer procesu: 7, wynik= 2.249611
Numer procesu: 6, wynik= 2.925403
Numer procesu: 5, wynik= 3.526328
Numer procesu: 4, wynik= 4.056780
Numer procesu: 3, wynik= 4.521152
Numer procesu: 2, wynik= 4.923841
Numer procesu: 1, wynik= 5.269238
Numer procesu: 0, wynik= 5.561738
```

b) Wzór Leibniza



```
GNU nano 4.8                                zadanie.c                                Modified
#include <stdio.h>
#include <math.h>
#include "mpi.h"
int main(int argc, char **argv){
    int p; // p-numer procesu
    int n; //liczba procesow
    int tag=50;

    float liczba =0, zmienna =0, wynik =0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &p);
    MPI_Comm_size(MPI_COMM_WORLD, &n);
    MPI_Status status;

    if(p==0)
    {
        liczba=(pow((-1.0),p))/(2*p+1);
        zmienna +=liczba;
        wynik = 4*zmienna;

        printf("\nNumer procesu: %d ,   wynik= %f\n", p, wynik);
        MPI_Send(&liczba, 1, MPI_INT, p+1, tag, MPI_COMM_WORLD);
        MPI_Send(&zmienna, 1, MPI_INT, p+1, tag, MPI_COMM_WORLD);
        MPI_Send(&wynik, 1, MPI_INT, p+1, tag, MPI_COMM_WORLD);
    }

    if((p>0)&&(p<n-1)){

        MPI_Recv(&liczba, 1, MPI_INT, p-1, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&zmienna,1, MPI_INT, p-1, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&wynik, 1, MPI_INT, p-1, tag, MPI_COMM_WORLD, &status);

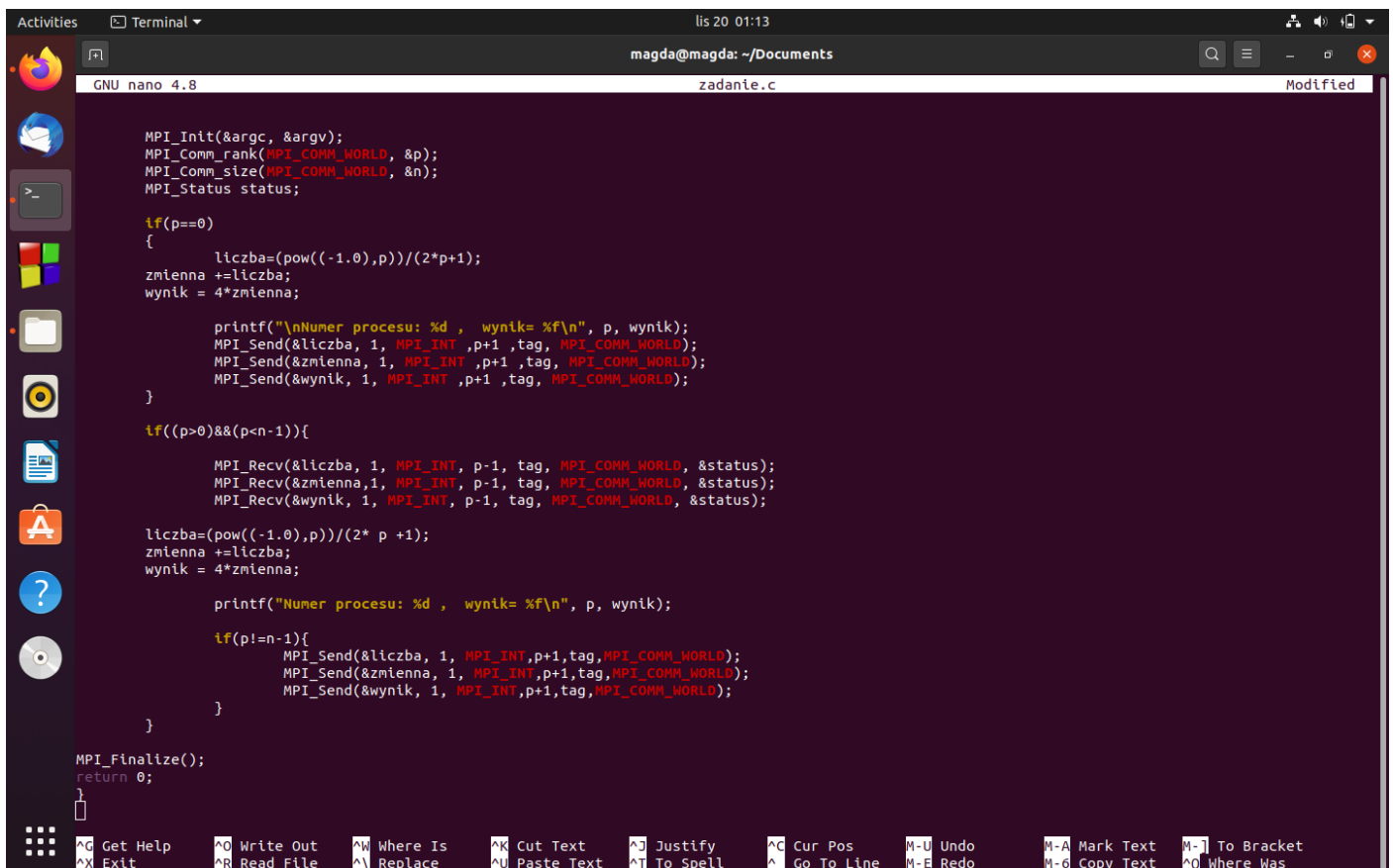
        liczba=(pow((-1.0),p))/(2* p +1);
        zmienna +=liczba;
        wynik = 4*zmienna;

        printf("Numer procesu: %d ,   wynik= %f\n", p, wynik);

        if(p!=n-1){
            MPI_Send(&liczba, 1, MPI_INT,p+1,tag,MPI_COMM_WORLD);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Funkcja inicjalizuje środowisko wykonywania programu, analogicznie jak w przykładzie wyżej. Pobiera numer aktualnego procesu i zapisuje w zmiennej p oraz pobiera ilość procesów i zapisuje w zmiennej n. Następnie obliczamy pierwszy wyraz z wzoru Leibniz-a. Dalej za pomocą funkcji MPI_Send wysyłamy dane do określonego procesu.



```
MPI_Finalize();
return 0;
}
```

Analogicznie funkcja `MPI_Recv` odczytuje z kolejki komunikatora `comm` (z ewentualnym blokowaniem do czasu nadejścia) pierwszy komunikat od procesu `source` oznaczony znacznikiem `tag` typu `int`. Wynik umieszczany jest w buforze `liczba`, zmienna, `wynik`. Dalej według wzoru na przybliżenie π wzorem Leibniz-a obliczamy π , które są zapoczątkowane w procesie o indeksie 0 i są przekazywane do kolejnych procesów w kolejności rosnącej aż do procesu `n-1`. Procesy przekazują aktualną wartość przybliżenia π , każdy z procesu wypisuje jej wartość. Na koniec funkcja `MPI_Finalize` zwalnia zasoby używane przez MPI i przygotowuje system do zamknięcia.

Wynik programu:

```
magda@magda:~/Documents$ mpicc zadanie.c -o zadanie.exe -lm
magda@magda:~/Documents$ mpirun --oversubscribe -np 10 zadanie.exe

Numer procesu: 0 ,   wynik= 4.000000
Numer procesu: 1 ,   wynik= 2.666667
Numer procesu: 2 ,   wynik= 3.466666
Numer procesu: 3 ,   wynik= 2.895238
Numer procesu: 4 ,   wynik= 3.339682
Numer procesu: 5 ,   wynik= 2.976046
Numer procesu: 6 ,   wynik= 3.283738
Numer procesu: 7 ,   wynik= 3.017072
Numer procesu: 8 ,   wynik= 3.252366
```