Grupa lab.
5

Data ćwiczenia
18.12.2023

Labratorium 11

Imie i nazwisko
Łukasz Sawina

Cel

Celem ćwiczenia było poznanie pracy równoległej z przesyłaniem komunikatów MPI.

Przesyłanie nazwy hosta

W pierwszym programie musieliśmy uruchomić nasz kod przy pomocy mpiexe, który uruchamia równolegle kilka programów, w których pierwszy znich przyjmuje komunikaty z pozostałych i odczytuje ich nazwy hostów.

Na początku w kodzie inicjalizowany jest mechanizm komunikacji MPI.

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
```

MPI_Init odpowiada za inicjalizację komunikacji, MPI_Comm_rank jest funkcją pobierającą wartość rank – numer procesu, MPI_Comm_size pobiera liczbę procesów biorących udział w komunikacji.

Następnie w kodzie sprawdzamy, czy size jest większe od 1, ponieważ wykonanie tgo zadania jest bez sensu przy tylko jednym procesie, w rzeczywistości nie będzie miał od kogo odebrać komunikatu. Jeśli jednak w komunikacji jest więcej procesów, to każdy z procesów zależnie od tego jaką ma wartość rank dostaje inne zadanie.

```
if( rank != 0 ){ dest=0; tag=0;
    gethostname(hostname, sizeof(hostname));
    MPI_Send( &hostname, 256, MPI_CHAR, dest, tag, MPI_COMM_WORLD );

  } else {

    for( i=1; i<size; i++ ) {

      MPI_Recv( &hostname, 256, MPI_CHAR, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &status );
      printf("Dane od procesu o randze (status.MPI_SOURCE ->) %d: Nazwa hosta: %s (i=%d)\n",
          status.MPI_SOURCE, hostname, i );
    }
  }
```

W tym przypadku, jeśli numer rank jest różny od 0, to proces wysyła komunikat o swojej nazwie hosta pobranej przy pomocy funkcji *gethostname()*, jako parametry przyjmuje tablicę charów oraz jej rozmiar, w kodzie ustawiony rozmiar jest na 256 znaków. Następnie przy pomocy MPI_Send() wysyłamy komunikat. Pierwszym parametrem jest zmienna którą chcemywysłać, w tym przypadku tablica, następnie mamy ilość danych, tutaj 256, ponieważ taki jest rozmiar tablicy, trzecim parametrem jest typ zmiennej jaką wysyłamy. W MPI nie wykorzystujemy zwykłych typów danych, tylko konkretne poprzedzone „MPI_”. Kolejne trzy parametry to numer odbiorcy, tag, oraz MPI_COMM_WORLD. MPI_COMM_WORLD to tak jakby rodzina procesów które są uruchomione, przy pomocy jego możemy informować, że w tej rodzinie wysyłamy z tego procesu lub wysyłamy do tego procesu.

Jeśli numer rank procesu ma wartość 0 to odpowiada za odebranie wszystkich komunikatów oraz wyświetlenie ich wartości. Przy pomocy MPI_Recv() proces odbiera komunikaty od konkretnych procesów, wartości parametrów są identyczne jak dla MPI_Send().

Wynikiem programu jest:

```
Dane od procesu o randze (status.MPI_SOURCE ->) 2: Nazwa hosta: lukasz-ThinkPad (i=1)
Dane od procesu o randze (status.MPI_SOURCE ->) 1: Nazwa hosta: lukasz-ThinkPad (i=2)
Dane od procesu o randze (status.MPI_SOURCE ->) 3: Nazwa hosta: lukasz-ThinkPad (i=3)
Dane od procesu o randze (status.MPI_SOURCE ->) 4: Nazwa hosta: lukasz-ThinkPad (i=4)
```

Program został uruchomiony z 5 procesami, daltego w wyniku widać tylko 4, ponieważ jeden z nich jest odpowiedzialny za odbieranie oraz wyświetlanie.

Sztafeta

Kolejnym programem jest sztafeta, w tym programie pierwszy proces ma wysłać komunikat do drugiego, drugi do trzeciego i tak do końca, w zadaniu było powiedziane, aby sztafeta zrobiłą jedno okrążenie, jednak w mojej wersji całośc umieszczona jest w nieskończonej pętli i wykonuje się cały czas.

Początek programu jest identyczny jak poprzednio, inicjalizowana jest komunikacja MPI, pobierane są wartości rank oraz size. W tym programie musimy wyróżnić trzy rodzaje procesów, pierwszy/środkowy/ostani.

Pierwszy proces odpowiedzialny jest za wysłanie pierwszego komunikatu, w mojej wersji dodatkowo za odebranie ostatniego oraz rozpoczęcie od nowa sztafety.

```
if( rank == 0 ){
        if(value != 0)
        {
           MPI_Recv( &value, 1, MPI_INT, size-1, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
           printf("Proces %d odebrał liczbę %d do procesu %d\n", rank, value, size-1);
        }

        value++;
        dest=1; tag=0;
        MPI_Send( &value, 1, MPI_INT, dest, tag, MPI_COMM_WORLD );
        printf("Proces %d wysłał liczbę %d do procesu %d\n", rank, value, dest);

    }
```

Najpierw sprawdzane jest czy wart przekazywana pomiędzy procesami ma zmienioną wartość, na tej podstawie możemy określić czy to pierwsza wiadomość czy kolejne okrążenie. Jeśli jest zmieniona to odbieramy z ostatniego procesu wiadomość i wyświetlamy ją.

Niezależnie od tego czy jest to pierwsza wiadomość, czy kolejna z rzędu, wartość przekazywana jest zwiększana (wartość będzie rosła od 1 do konkretnej wartości maksymalnej INT) i wysyłana jest do procesu z numerem 1.

Ostani proces odpowiedzialny jest za zakończenie sztafety, czli wyświetlenie informacji o końcu oraz wysłaniu wartości na początek sztafety.

```
else if(rank == size-1) {

        MPI_Recv( &value, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status );
        printf("Proces %d odebrał liczbę %d do procesu %d\n", rank, value, rank-1);

        printf("Jestem ostani więc zamykam pierścien i idziemy od nowa \n");
        value++;
        dest=0; tag=0;
        MPI_Send( &value, 1, MPI_INT, dest, tag, MPI_COMM_WORLD );
        printf("Proces %d wysłał liczbę %d do procesu %d\n", rank, value, dest);
    }
```

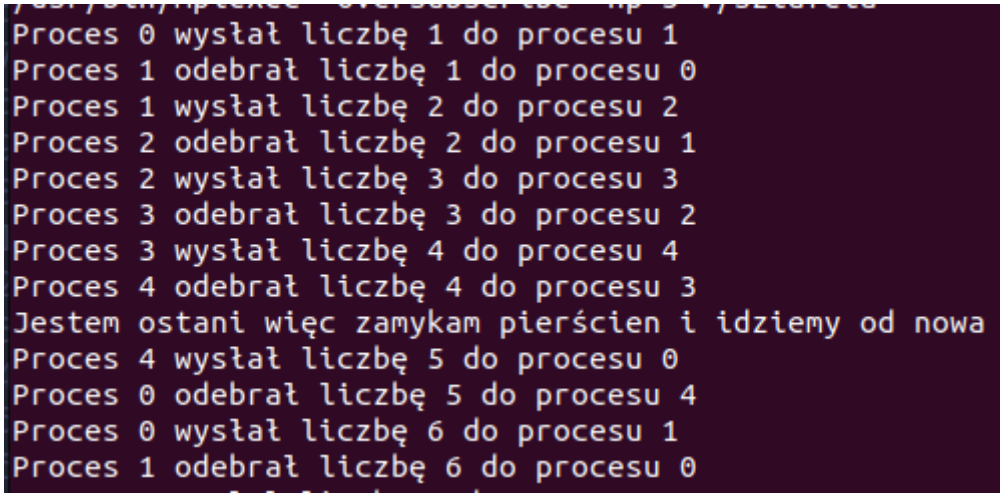
Komunikat jest odbierany od procesu o jeden wcześniej, dlatego można to określić jako rank-1, następnie wyświetlane są wiadomości i wysyłany jest nowy komunikat do procesu 0.

Środkowy komunikat odpowiedzialny jest za pobranie komunikatu z wcześniejszego procesu, wyświetlenei go oraz wysłanie do kolejnego.

```
else {
        dest=rank+1; tag=0;

        MPI_Recv( &value, 1, MPI_INT, rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        printf("Proces %d odebrał liczbę %d do procesu %d\n", rank, value, rank-1);

        value++;
        MPI_Send( &value, 1, MPI_INT, dest, tag, MPI_COMM_WORLD );

        printf("Proces %d wysłał liczbę %d do procesu %d\n", rank, value, dest);
    }
```

Tutaj podobnie jak wcześniej komunikat pobierany jest z rank-1, czyli poprzedniego procesu, wyświetlana oraz wysyłana do kolejnego procesu czyli rank+1.

Wynikiem jest:



Jak widać wartość jest przesyłana pomiędzy kolejnymi procersami w odpowiedniej kojelności oraz po dotarciu do ostaniego wyświetlony jest komunikat o zamknięciu pierścienia i wysłania ponownie do pierwszego, który dalej wysyła do kolejnych komunikat.

Struktura

W tym zadaniu musimy przesłać pomiędzy procesami więcej niż jedną zmienną, w moim przypadku będzie to struktura ponizej:

```
struct {
        int next;
        double val;
        char name[30];
    } dane, odebrane_dane;
```

Do wykonania tego musimy użyć MPI_PACKED, do utworzonej zmiennej ustawiamy początkowe wartości, następnie musimy je spakować przy pomocy MPI_Pack.

```
MPI_Pack(&dane.next, 1, MPI_INT, buffer, buffer_size, &position, MPI_COMM_WORLD);
MPI_Pack(&dane.val, 1, MPI_DOUBLE, buffer, buffer_size, &position, MPI_COMM_WORLD);
MPI_Pack(dane.name, 20, MPI_CHAR, buffer, buffer_size, &position, MPI_COMM_WORLD);
```

Kolejno pakujemy każdą zmienną ze struktury, określając ich ilość oraz typ zmiennej. Dodatkowo musimy dodać pewien buffor jego rozmiar oraz pozycję.

Program, który wykorzystuje MPI_PACKED to ten sam co poprzednio, tylko zamiast jednej zmiennej wysyłana jest struktura, daltego opiszę jedynie odbieranie oraz wysyłanie danych, ponieważ reszta jest identyczna.

W sytuacji, gdy nasz proces musi wysłać wcześniej spakowane dane wywołuje MPI_Send(), tylko tym razem zamiast przesłania zmiennej, przesyłamy buffor, jego pozycję oraz jako typ zmiennej ustawiamy MPI_PACKED.

MPI_Send(buffer, position, MPI_PACKED, dane.next, **0**, MPI_COMM_WORLD);

Tak wysłane dane mogą zostać odebrane przy pomocy MPI_Reciv(), który przyjmuje identyczne parametry jak MPI_Send().

MPI_Recv(buffer, buffer_size, MPI_PACKED, rank**-1**, **0**, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

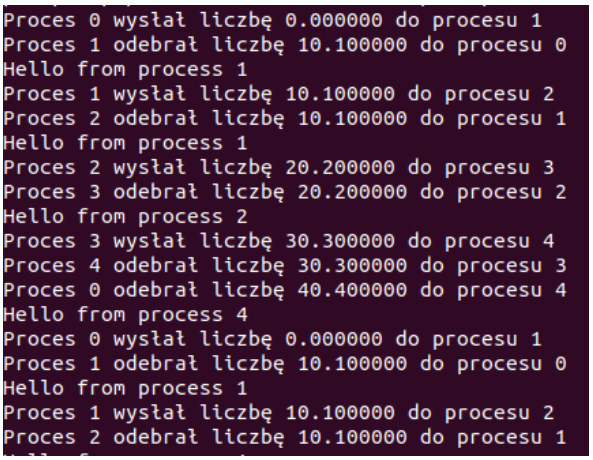
Jednak po odebraniu naszych danych są one spakowane, dlatego aby je wyświetlić musimy je najpierw rozpakować przy pomocy MPI_Unpack.

MPI_Unpack(buffer, buffer_size, &position, &odebrane_dane.next, **1**, MPI_INT, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &position, &odebrane_dane.val, **1**, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &position, odebrane_dane.name, **20**, MPI_CHAR, MPI_COMM_WORLD);

Jako parametry przyjmujemy buffor, który odebraliśmy, rozmiar bufforu, pozycję oraz konkretne miejsca do których ma nam wypakować daną zmienną, ich typ oraz MPI_COMM_WORLD.

Po rozpakowaniu danych możemy bez problemu je wyświetlić, edytować oraz ponownie zapakować i wysłać.

Wynikiem programu jest:



Jak widać, tym razem przesyłamy oraz wyświetlamy więcej zmiennych oraz o różnych typach. I tak oto każdy proces wyświetla numer, od kogo oraz do kogo odbiera/wysyła, wartość tablicy charów, którą otrzymał oraz zmienną double jako wartość, kktóra zostaje zwiększana.

Potok

W tym zadaniu musimy wykorzystać nasze procesy jako potok, każdy z nich musi wykonać inną operację, zależną od swojej wartości rank. W moim programie działam na 5 procesach, w których

pierwszy inicjalizuje dane i wysyła je dalej, środkowe wykonują operacje na otrzymanych danych, a ostatni wyświetla wynikowe dane.

Działanie programu ponownie oparte jest na zasadzie jak sztafeta. W tym przypadku główną różnicą jest zadanie jakie wykonują śrokowe procesy. Jako dane przesyłana jest struktura:

```
struct {
    int next;
    double val;
    char name[30];
    int liczba_duzych;
    int liczba_malych;
    int liczba_spacji;

}
```

Wartość name ustawiona jest na „Hello, World!”, środkowe procesy mają za zadanie obliczyć ilość dużych/małych znaków oraz ilość spacji.

```
switch (rank)
        {
        case 1:
          {
            for (int i = 0; odebrane_dane.name[i] != '\0'; ++i) {

              if (isupper(odebrane_dane.name[i])) {
                odebrane_dane.liczba_duzych++;
              }
            }
          }
          break;
        case 2:
          {
            for (int i = 0; odebrane_dane.name[i] != '\0'; ++i) {

              if (!isupper(odebrane_dane.name[i])) {
                odebrane_dane.liczba_malych++;
              }
            }
          }
          break;
        case 3:
          {
            for (int i = 0; odebrane_dane.name[i] != '\0'; ++i) {

              if (odebrane_dane.name[i] == ' ') {
                odebrane_dane.liczba_spacji++;
              }
            }
          }
          break;
        default:
          break;
        }
```

Zawartość pierwszego oraz ostatniego procesu nie zmienia się za bardzo, dane w nich są pakowane oraz rozpakowywane i wysyłąne oraz odbierane, dlatego je pomijam w analizie kodu.

Środkowe procesy jednak mają konkretne zadania przypisane dla konkretnej rangi. Dzięki instrukcji switch() łatwo można rozpisać te odpowiedzialności. Każdy środkowy proces będzię miał odpowiedni blok instrukcji przypisany do swojego numeru.

Wynikiem programu jest:

```
Proces 0 wysłał wiadomość do procesu 1
Proces 1 odebrał wiadomość do procesu 0
Proces 1 wysłał wiadomość do procesu 2
Proces 2 odebrał wiadomość do procesu 1
Proces 2 wysłał wiadomość do procesu 3
Proces 3 odebrał wiadomość do procesu 2
Proces 3 wysłał wiadomość do procesu 4
Proces 4 odebrał wiadomość do procesu 3
Małych liter: 8, Dużych liter: 2, spacji: 1
```

Jak widać komunikacja odbyła się na zasadzie sztafety oraz dane zostały policzone poprawnie. Dodając do programu więcej procesów można im bez problemu dodać kolejne odpowiedzialności, jak np. liczenie ilości przecinków kropek itp.

Własny typ

Ostanim programem do napisania było utworzenie własnego typu, który będzie mógł być przekazywany jako parametr do MPI_Send/MPI_Reciv, tak aby nie bawić się w pakowanie oraz rozpakowywanie danych. Jako nowy typ danych wykorzystuję strukturę z 3 zadania.

Do utworzenia nowego typu musimy utworzyć zmienną, której nazwa będzie naszym nowym typem, określić długości kolejnych elementów w strukturze, ich typy oraz przypisać poszczególne adresy do adresów zmiennych.

```
MPI_Datatype Wlasny_typ;
int blocklengths[3] = {1, 1, 30};
MPI_Aint displacements[3];
MPI_Datatype types[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};

MPI_Get_address(&wyslane.next, &displacements[0]);
MPI_Get_address(&wyslane.val, &displacements[1]);
MPI_Get_address(wyslane.name, &displacements[2]);

displacements[2] -= displacements[0];
displacements[1] -= displacements[0];
displacements[0] = 0;

MPI_Type_create_struct(3, blocklengths, displacements, types, &Wlasny_typ);
MPI_Type_commit(&Wlasny_typ);
```

Na końcu wywoływana jest funkcja MPI_Type_create_struct(), która przyjmuje ilość elementów, tablice długości poszczególnych zmiennych, przesunięcie względem początku struktury, tablicę

typów danych zmiennych struktury oraz adres do naszego własnego typu. Na końcu całość jest zapisywana przy pomocy MPI_Type_commit().

Dzięki temu teraz zamist pakować wszystkie dane oraz je rozpakowywać możemy wykonać wysłanie oraz odebranie w następujący sposób.

```
MPI_Send(&wyslane, 1, Wlasny_typ, wyslane.next, 0, MPI_COMM_WORLD);
MPI_Recv(&odebrane, 1, Wlasny_typ, rank - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

Jak widać jako trzeci parametr przesyłamy Wlasny_typ, dzięki czemu możemy bezpośrednio przekazywać oraz odbierać struktury i od razu je wyświetlać oraz modyfikować.

Program ponownie jest kopią programu ze sztafetą, tylko wykorzystuje Wlasny_typ, dlatego wynik programu jest podobny jak w zadaniu 3.

```
Proces 0 wysłał liczbę 0.000000 do procesu 1
Proces 1 odebrał liczbę 0.000000 od procesu 0
Hello from process 0
Proces 1 wysłał liczbę 10.100000 do procesu 2
Proces 2 odebrał liczbę 10.100000 od procesu 1
Hello from process 1
Proces 2 wysłał liczbę 20.200000 do procesu 3
Proces 3 odebrał liczbę 20.200000 od procesu 2
Hello from process 2
Proces 3 wysłał liczbę 30.300000 do procesu 4
Proces 4 odebrał liczbę 30.300000 od procesu 3
Hello from process 3
Proces 0 odebrał liczbę 40.400000 od procesu 4
Hello from process 4
Proces 0 wysłał liczbę 0.000000 do procesu 1
Proces 1 odebrał liczbę 0.000000 od procesu 0
Hello from process 0
Proces 1 wysłał liczbę 10.100000 do procesu 2
Proces 2 odebrał liczbę 10.100000 od procesu 1
```

Wnioski

Mechanizm komunikacji MPI działa na innej zasadzie niż poprzednio poznawane mechanizmy. Tutaj równolegle uruchamiane jest kilka procesów, wykonujących ten sam kod, a nie proces główny, który tworzy kolejne procesy i zarządza ich wykonaniem. Dzięki temu też poznaliśmy lepiej mechanizm SPMD oraz komunikację Punkt-do-Punktu.

Cała komunikacja opiera się na dwóch prostych funkcjach MPI_Send() oraz MPI_Reciv(), które są funkcjami blokującychmi, czyli MPI_Reciv() zablokuje nam proces aż do momentu, gdy otrzyma komunikat od konkretnego procesu. Istnieją wersje nieblokujące MPI_Isend() oraz MPI_Irecive(), które niwelują to zachowanie.

Do uruchomienia kodu wykorzystujacy MPI ważne jest aby zamiast klasycznego gcc wykorzystać kompialtor np. mpiexec/mpicc, a ilość procesów określana jest jako parametr -np.

Podstawowe wykorzystanie MPI do przesyłania prostych typów jest łatwą kwestią, utrudnienia oraz dodatkowa praca pojawia się w sytuacji gdy chcemy wysłać więcej danych, skonfigurowanie pakowania oraz tworzenia własnych typów może być kwestią problematyczną, ale na pewno

ułatwiającą życie, ponieważ nie musimy wysyłać każdej wartości z osobna, a wysłać całość jeden raz.

Dodatkowym ważnym aspektem pisania programu z wykrzystaniem mpi jest, aby na początku zainicjalizować komunikację przy pomocy MPI_Init(), a na końcu programu ją zakończyć MPI_Finalize().