Tomasz Ligęza

Programowanie równoległe. Przetwarzanie równoległe i rozproszone.

Sprawozdanie z laboratorium 11.

Cel zajęć:

Opanowanie podstaw programowania z przesyłaniem komunikatów MPI.

W ramach zajęć zrealizowałem następujące kroki:

1. Utworzyłem katalog roboczy lab_11.
2. Uzupełniłem kod pliku MPI_simple.c tak, aby wysyłana była tablica znaków zawierająca adres internetowego węzła nadawcy:

```
if (size > 1) {
   if (rank != 0) {
       dest = 0;
       tag = 0;
       gethostname(sender, 256);
       MPI_Send(&sender, 256, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   } else {
       for (i = 1; i < size; i++) {
           MPI_Recv(&receive_sender, 256, MPI_CHAR, MPI_ANY_SOURCE,
                    MPI_ANY_TAG, MPI_COMM_WORLD, &status);
           printf("Dane od procesu o randze: %d (status.MPI_SOURCE -> %d)
                  (i=%d) (sender=%s)\n",
                  ranksent, status.MPI_SOURCE, i, receive_sender);
       }
   }
} else {
   printf("Pojedynczy proces o randze: %d (brak komunikatów)\n", rank);
}
```

```
[ligeza_tomasz@ESTERA lab_11]$ make
/usr/lib64/openmpi/bin/mpiexec -np 4 ./MPI_simple
Dane od procesu o randze: -2063445776 (status.MPI_SOURCE -> 2) (i=1) (sender=ESTERA)
Dane od procesu o randze: -2063445776 (status.MPI_SOURCE -> 3) (i=2) (sender=ESTERA)
Dane od procesu o randze: -2063445776 (status.MPI_SOURCE -> 1) (i=3) (sender=ESTERA)
[ligeza_tomasz@ESTERA lab_11]$
```

3. Utworzyłem kolejny podkatalog roboczy sztafeta.

4. Na podstawie kodu z pliku MPI_simple.c zaimplementowałem system sztafety - najpierw działającej na zasadzie “otwartego klucza”:

```
int number = 12;
int number_recv;
if (size > 0) {
   if (rank == 0) {
       int next = rank + 1;
       MPI_Send(&number, 1, MPI_INT, next, tag, MPI_COMM_WORLD);
       printf("Proces %d wyslal liczbe %d do procesu %d\n", rank, number,
              next);
   } else if (rank < size - 1) {
       int prev = rank - 1;
       int next = rank + 1;
       MPI_Recv(&number_recv, 1, MPI_INT, prev, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
       printf("Proces %d otrzymal liczbe %d od procesu %d\n", rank,
              number_recv, prev);
       number = number_recv + 1;
       MPI_Send(&number, 1, MPI_INT, next, tag, MPI_COMM_WORLD);
       printf("Proces %d wyslal liczbe %d do procesu %d\n", rank, number,
              next);
   } else if (rank == size - 1) {
       int prev = rank - 1;
       MPI_Recv(&number_recv, 1, MPI_INT, prev, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
       printf("Proces %d otrzymal liczbe %d od procesu %d\n", rank,
              number_recv, prev);
   }
}
```

```
/usr/lib64/openmpi/bin/mpiexec -np 4 ./sztafeta
Proces 0 wyslal liczbe 12 do procesu 1
Proces 1 otrzymal liczbe 12 od procesu 0
Proces 1 wyslal liczbe 13 do procesu 2
Proces 2 otrzymal liczbe 13 od procesu 1
Proces 2 wyslal liczbe 14 do procesu 3
Proces 3 otrzymal liczbe 14 od procesu 2
[ligeza_tomasz@ESTERA sztafeta]$
```

5. Następnie system sztafety na zasadzie zamkniętego klucza, gdzie ostatni proces podaje liczbę do pierwszego procesu:

```
while (size > 0) {
   int prev = rank - 1;
   int next = rank + 1;
   if (rank == 0) {
       prev = size - 1;
```

```
    } else if (rank == size - 1) {
        next = 0;
    }
    if (!(rank == 0 && number == 0)) {
        MPI_Recv(&number_recv, 1, MPI_INT, prev, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);
        printf("Proces %d otrzymal liczbe %d od procesu %d\n", rank,
               number_recv, prev);
    }
    number = number_recv + 1;
    MPI_Send(&number, 1, MPI_INT, next, tag, MPI_COMM_WORLD);
    printf("Proces %d wyslal liczbe %d do procesu %d\n", rank, number,
           next);
    sleep(1);
}
```

```
Proces 0 wyslal liczbe 1 do procesu 1
Proces 1 otrzymal liczbe 1 od procesu 0
Proces 1 wyslal liczbe 2 do procesu 2
Proces 2 otrzymal liczbe 2 od procesu 1
Proces 2 wyslal liczbe 3 do procesu 3
Proces 3 otrzymal liczbe 3 od procesu 2
Proces 3 wyslal liczbe 4 do procesu 0
Proces 0 otrzymal liczbe 4 od procesu 3
Proces 0 wyslal liczbe 5 do procesu 1
Proces 1 otrzymal liczbe 5 od procesu 0
Proces 1 wyslal liczbe 6 do procesu 2
Proces 2 otrzymal liczbe 6 od procesu 1
```

6. W kolejnym zadaniu utworzyłem podkatalog struktura oraz skopiowałem do niego kod z pierścieniem otwartym. Utworzyłem strukturę zawierającą pola typu: double, int oraz char[8]. Wypełniłem ją oraz obliczyłem wielkość paczki zawierającej trzy zmienne:

```
struct record
{
    double scalar;
    int number;
    char name[8];
};
[...]

    struct record rekord;
    rekord.scalar = 5.0;
    rekord.number = 10;
    strncpy(rekord.name, "hello", 8);

    int aux = 0;
    MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD, &aux);
```

```
    int packet_size = aux;
    MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &aux);
    packet_size += aux;
    MPI_Pack_size(8, MPI_CHAR, MPI_COMM_WORLD, &aux);
    packet_size += aux;
```

7. Następnie dodałem kod odpowiedzialny za pakowanie, odpakowywanie oraz wysyłanie i odbieranie paczki. Procesy odbierające paczkę i wysyłające ją zmieniają również jej zawartość.

```
else if (rank < size - 1) {
    int prev = rank - 1;
    int next = rank + 1;

    void *buffor = (void *)malloc(packet_size);
    MPI_Recv(buffor, packet_size, MPI_INT, prev, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);

    //ODPAKOWANIE
    int pos = 0;
    struct record recv;
    MPI_Unpack(buffor, packet_size, &pos, &recv.scalar, 1, MPI_DOUBLE,
               MPI_COMM_WORLD);
    MPI_Unpack(buffor, packet_size, &pos, &recv.number, 1, MPI_INT,
               MPI_COMM_WORLD);
    MPI_Unpack(buffor, packet_size, &pos, &recv.name, 8, MPI_CHAR,
               MPI_COMM_WORLD);


    printf("Proces %d - odebral strukture: %lf, %d, %s\n", rank,
           recv.scalar, recv.number, recv.name);
    //EDYCJA
    recv.scalar += 0.1;
    recv.number += 1;
    //PAKOWANIE
    pos = 0;
    MPI_Pack(&recv.scalar, 1, MPI_DOUBLE, buffor, packet_size, &pos,
             MPI_COMM_WORLD);
    MPI_Pack(&recv.number, 1, MPI_INT, buffor, packet_size, &pos,
             MPI_COMM_WORLD);
    MPI_Pack(&recv.name, 8, MPI_CHAR, buffor, packet_size, &pos,
             MPI_COMM_WORLD);

    printf("Proces %d - wysyla strukture: %lf, %d, %s\n", rank,
           recv.scalar, recv.number, recv.name);
    MPI_Send(buffor, pos, MPI_PACKED, next, tag, MPI_COMM_WORLD);
}
```

Kod dla procesu rozpoczynającego i kończącego jest analogiczny do tego powyżej z tą różnicą, że rozpoczynający jedynie pakuje i wysyła wiadomość, a kończący jedynie paczkę odbiera i odpakowuje.

```
/usr/lib64/openmpi/bin/mpiexec -np 4 ./struktura
Proces 0 - wysyla strukture: 5.000000, 10, tomasz
Proces 1 - odebral strukture: 5.000000, 10, tomasz
Proces 1 - wysyla strukture: 5.100000, 11, tomasz
Proces 2 - odebral strukture: 5.100000, 11, tomasz
Proces 2 - wysyla strukture: 5.200000, 12, tomasz
Proces 3 - odebral strukture: 5.200000, 12, tomasz
[ligeza_tomasz@ESTERA struktura]$
```

Wnioski:

Aby zapętlić pierścień należało w jakiś sposób “wystartować” pracę pierścienia. W tym celu sprawdzam czy proces ma rangę 0 i czy wartość liczby wynosi 0 - wtedy jedyne co robię to wysyłam liczbę do następnego procesu. W przeciwnym przypadku oczekuję na wysłanie mi liczby z poprzedniego procesu. Dzięki temu pierwszy proces nie oczekuje w nieskończoność na wiadomość, która nigdy nie nadejdzie.

Ciekawe zjawisko występuje, gdy uruchomimy nasz zapętlony pierścień bez żadnego oczekiwania po działaniu jednego procesu:

```
Proces 3 wyslal liczbe 21588 do procesu 0
Proces 3 otrzymal liczbe 21591 od procesu 2
Proces 3 wyslal liczbe 21592 do procesu 0
Proces 3 otrzymal liczbe 21595 od procesu 2
Proces 3 wyslal liczbe 21596 do procesu 0
Proces 3 otrzymal liczbe 21599 od procesu 2
Proces 3 wyslal liczdo procesu 3
Proces 2 otrzymal liczbe 21414 od procesu 1
Proces 2 wyslal liczbe 21415 do procesu 3
Proces 2 otrzymal liczbe 21418 od procesu 1
Proces 2 wyslal liczbe 21419 do procesu 3
Proces 2 otrzymal liczbe 21422 od procesu 1
Proces 2 wyslal liczbe 21423 do procesu 3
Proces 2 otrzymal liczbe 21426 od procesu 1
Proces 2 wyslal liczbe 21427 do procesu 3
Proces 2 otrzymal liczbe 21430 od procesu 1
Proces 2 wyslal liczbe 21431 do procesu 3
Proces 2 otrzymal liczbe 21434 od procesu 1
Proces 2 wyslal liczbe 21435 do procesu 3
Proces 2 otrzymal liczbe 21438 od procesu 1
Proces 2 wyslal liczbe 21439 do procesu 3
Proces 2 otrzymal liczbe 21442 od procesu 1
```

Występuje tutaj race condition którego zasobem jest terminal (stdout). Jak widać cała komunikacja przebiega poprawnie, gdyż widzimy, że proces 3 wysyła liczbę 21588 od procesu 0 i otrzymuje liczbę 21591 od procesu 2.

[3] - 21588 -> [0], [0] - 21589 -> [1], [1] - 21590 -> [2], [2] - 21591 -> [3]