Uniwersytet Bielsko-Bialski

LABORATORIUM

Obliczeń Równoległych i Systemów Rozproszonych

Sprawozdanie nr 11

interface programowania OPENMPI

GRUPA: 2B / SEMESTR: 5 / ROK: 3

Igor Gawłowicz / 59096

Cel ćwiczenia

Cel ćwiczenia: Celem ćwiczenia było zapoznanie się z podstawowymi funkcjami i operacjami związanymi z programowaniem współbieżnym w MPI (Message Passing Interface). Ćwiczenie obejmowało instalację i konfigurację OPEN MPI, a następnie praktyczne wykorzystanie funkcji takich jak `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Barrier`, `MPI_Send`, `MPI_Recv`, `MPI_Finalize`, `MPI_Get_processor_name`, `MPI_Reduce` do realizacji prostych programów współbieżnych.

Przebieg ćwiczenia

MPI czyli Message Passing Interface stanowi specyfikację pewnego standardu wysokopoziomowego protokołu komunikacyjnego i powiązanego z nim interface'u programowania, służącego programowaniu współbieżnemu. Reprezentuje on znacznie wyższy poziom abstrakcji niż standardowe UNIX System V IPC czy POSIX Threads.

Po instalacji i konfiguracji OPEN MPI możemy sprawdzić za pomocą mpicc (opcje zwyczajowe, jak i w gcc i pozostałych)

$ mpicc hello_c.c -o hello

zaś uruchomienie

$ ./hello

Hello, world, I am 0 of 1

... program po prostu wyświetla informację na iloma procesami (wątkami) zarządza komunikator (tu: 1) i jaki jest numer bieżącogo (tu: 0).

Ten sposób uruchomiania i wykorzystania komunikatora, jest w gruncie rzeczy bardzo nietypowy. Oczywiście można w ten sposób uruchamiać programy wykorzystujące API OpenMPI niemniej jest to sposób bardzo nietypowy i nie służący przetwarzaniu współbieżnemu.

Poprawnie powinniśmy użyć w tym celu wrapper'a mpirun podając ile procesów (-np lub -n) ma być uruchomionych

```
$ mpirun -np 4 ./hello
Hello, world, I am 0 of 4
Hello, world, I am 1 of 4
Hello, world, I am 2 of 4
Hello, world, I am 3 of 4
```

W trakcie uruchomiania programu, w linii komend mpirun można podać explicite plik konfiguracyjny określający ile zadań i na którym węźle klastra ma być uruchomionych. Służy temu opcje (równoważnie)

```
$ mpirun --hostfile hosts ./hello
Hello, world, I am 0 of 4
Hello, world, I am 1 of 4
```

```
Hello, world, I am 2 of 4
Hello, world, I am 3 of 4
```

a plik hosts poszukiwany jest domyślnie w bieżącym katalogu. W tym przypadku jego zawartość była następująca

10.60.20.76 slots=4

Identyczny efekt przyniesie użycie opcji --host

$ mpirun --host 10.60.20.76,10.60.20.76,10.60.20.76,10.60.20.76 ./hello

bezpośrednio po nazwie może być podana ilość procesorów, a właściwie procesów jaka może być uruchamiana na danych węźle

10.60.20.76 slots=2 max_slots=4
10.20.24.51 slots=2 max_slots=2

przy czym szeregowanie zadań na węzłach może się odbywać wg slotu (--byslot, domyślnie) albo wg węzła (--bynode). Przyjrzyjmy się różnicą, zakładając że plik hosts zawiera linie jak wyżej. Dla --bynode

```
$ mpirun --hostfile hosts -np 6 --byslot | sort
Hello World I am rank 0 of 6 running on 10.60.20.76
Hello World I am rank 1 of 6 running on 10.60.20.76
Hello World I am rank 2 of 6 running on 10.20.24.51
Hello World I am rank 3 of 6 running on 10.20.24.51
Hello World I am rank 4 of 6 running on 10.60.20.76
Hello World I am rank 5 of 6 running on 10.60.20.76
```

```
$ mpirun --hostfile hosts -np 6 --bynode | sort
Hello World I am rank 0 of 6 running on 10.60.20.76
Hello World I am rank 1 of 6 running on 10.20.24.51
Hello World I am rank 2 of 6 running on 10.60.20.76
Hello World I am rank 3 of 6 running on 10.20.24.51
Hello World I am rank 4 of 6 running on 10.60.20.76
Hello World I am rank 5 of 6 running on 10.60.20.76
```

Chcąc wprowadzić w kodzie programu przetwarzanie współbieżne z wykorzystaniem MPI, należy w części początkowej zawrzeć instrukcje inicjujące a końcowej zamykające komunikator. Nie jest to trudne bowiem służą temu wywołania zaledwie dwóch funkcji:

int MPI_Init(int *argc, char ***argv);
int MPI_Finalize(void);

zwracające 0 w przypadku sukcesu lub kod błędu w przypadku niepowodzenia. Deklaracja nagłówkowa tej i wszystkich pozostałych funkcji MPI znajduje się w

#include <mpi.h>

W najprostszym przypadku kod programu korzystającego z MPI, będzie miał postać jak w listingu.

```
#include <stdio.h>
#include <mpi.h>
int main( int argc,char **argv )
{
    MPI_Init( &argc,&argv );
    printf( "...programowanie z MPI jest proste\n" );
    MPI_Finalize();
    return 0;
}
```

Jeżeli zapiszemy pod nazwą proste.c, to kompilacja

$ mpicc proste.c -o proste

a wykonanie na dwóch procesach

```
$ mpirun -np 2 ./proste
...programowanie z MPI jest proste
...programowanie z MPI jest proste
```

Już na tym przykładzie, że na poziomie procesu konieczna jest możliwość jego identyfikacji. Służą temu dwie dalsze funkcje

**int MPI_Comm_size(MPI_Comm comm, int size); *int MPI_Comm_rank(MPI_Comm comm, int rank);*

przy czym parametr formalny comm określa komunikator, którego odwołania dotyczy – bieżący MPI_COMM_WORLD Pierwsza określa ilość procesów współbieżnych obsługiwanych przez dany komunikator, zaś druga numer identyfikujący bieżący proces w danej grupie (de facto jest to indeks, czyli od 0 do size-1).

W przykładzie pobierzemy informację o ilości procesów, a następnie każdy poda informację o sobie.

```
#include <stdio.h>
#include <mpi.h>
int main( int argc, char *argv[] )
{
    int world,current;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &current );
    MPI_Comm_size( MPI_COMM_WORLD, &world );
    printf("proces %d z %d\n",current+1,world );
    MPI_Finalize();
    return 0;
}
```

```
```

```
$ mpirun -np 3 ./kolejne
proces 1 z 3
proces 2 z 3
proces 3 z 3
```

Warto pamiętać, że - tradycyjnie – jeżeli proces (wątek) będzie wykonywał jakiekolwiek zadania i może wystąpić niezrównoważenie obciążeń, to trzeba zapewnić przynajmniej synchronizację w pewnych punktach

int MPI_Barrier(MPI_Comm comm)

Naturalnym i często występującym elementem w kodzie współbieżnym jest jego różnicowanie między procesami. Równocześnie dość praktyczną możliwością jest pobranie nazwy węzła na którym znajduje się dany proces, czemu służy funkcja

**int MPI_Get_processor_name(char *name,int resultlen);*

Pierwszy parametr określa nazwę symboliczna węzła w postaci łańcucha znakowego a drugi jego długość. Deklarując zmienną name wygodnie jest tu skorzystać ze stałej MPI_MAX_PROCESSOR_NAME zdefiniowanej w mpi.h aktualnie jako

#define MPI_MAX_PROCESSOR_NAME 256

Kolejny przykład prezentuje sposób różnicowania kodu między węzeł nadrzędny identyfikowany stałą symboliczną MASTER i pozostałe

```
#include<stdio.h>
#include<mpi.h>
#define MASTER 0 //...wyróżniony węzeł MASTER o indeksie 0
int main( int argc, char *argv[] )
{
    int world,current,len;
    char name[MPI_MAX_PROCESSOR_NAME+1];
    MPI_Init( &argc,&argv ); //...inicowanie komunikatora
    MPI_Comm_size( MPI_COMM_WORLD,&world ); //...ilość procesów skojarzonych
    MPI_Comm_rank( MPI_COMM_WORLD,&current ); //...indeks bieżącego procesu
    MPI_Get_processor_name( name,&n );
    if( current==MASTER )
    {
        printf( "MASTER" );
    } //...zadania dla MASTERa
    else
    {
        printf( "SLAVE");
    } //...zadania dla SLAVE'ów
    //...o tego miejsca kod wspólny, dla wszystkich procesów
    printf( "\t%d.%d [%lu->%lu] | węzeł: %s\n",current+1,world,
    (unsigned long)getppid(),(unsigned long)getpid(),name );
    MPI_Finalize(); //...zamykamy komunikator
    return 0;
}
```

Sprawdzimy najpierw **-o cmd,ppid,pid**

```
CMD                          PPID     PID
-bash                         400     401
ps -o cmd,ppid,pid            401    2597
```

```
$ mpirun -np 6 ./slaves
SLAVE   4.6 [2738->2742] | węzeł: cs-87756155163-default
SLAVE   3.6 [2738->2741] | węzeł: cs-87756155163-default
SLAVE   5.6 [2738->2743] | węzeł: cs-87756155163-default
SLAVE   6.6 [2738->2744] | węzeł: cs-87756155163-default
MASTER  1.6 [2738->2739] | węzeł: cs-87756155163-default
SLAVE   2.6 [2738->2740] | węzeł: cs-87756155163-default
```

Jako przykład użycia komunikacji blokującej przygotujemy program w którym węzły slave prześlą pojedynczą liczbą do węzła (procesu) master.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#define MASTER 0
#define TAG 'R'+'i'+'R'
int main(int argc, char *argv[])
{
    int current,world,counter;
    double x,Sx;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    //...pobieramy informacje o ilości procesów i indeksie do bieżącego
    MPI_Comm_size( MPI_COMM_WORLD,&world );
    MPI_Comm_rank( MPI_COMM_WORLD,&current );
    //...jeszcze tylko wartość startowa dla generatora zmiennych pseudolosowych
    srand( (unsigned)time( NULL ) );
    //...i możemy zaczynać
    if( current!=MASTER ) //...czyli dla SLAVE'ów
    {
        x = (double)rand()/RAND_MAX;
        MPI_Send( (void*)(&x),1,MPI_DOUBLE,MASTER,TAG,MPI_COMM_WORLD );
    }
    else //... tym razem dla MASTER'a
    {
        for( counter=MASTER+1,Sx=0.0;counter<world;counter++ )
        {
            MPI_Recv( (void*)(&x),1,MPI_DOUBLE,counter,TAG,MPI_COMM_WORLD,&status
);
            Sx += x;
```

```
        }
        printf( "\nProces %d odebrał od %d do %d, sumę
Sx=%f\n\n",MASTER,MASTER+1,world-1,Sx );
    }
    //...od tego miejsca już kod wspólny
    MPI_Finalize();
    return 0;
}
```

Niestety środowisko w chmurze nie pozwala na taką ilość procesów więc musimy improwizować

```
$ mpirun -np 100 ./block


===================================================================================
=
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 2931 RUNNING AT cs-87756155163-default
=   EXIT CODE: 9
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================================================
=
YOUR APPLICATION TERMINATED WITH THE EXIT STRING: Terminated (signal 15)
This typically refers to a problem with your application.
Please see the FAQ page for debugging suggestions
```

```
$ mpirun -np 50 ./block

Proces 0 odebrał od 1 do 49, sumę Sx=29.416904
```

Nie zawsze koordynacja czasowa wymiany danych ma znaczenie krytyczne dla procesu, tak że niekoniecznie muszą one wiązać się z koniecznością wprowadzenia blokowania. W przypadku operacji nieblokujących sterowanie zwracane jest natychmiastowo do procesu, tak że może on podjąć dalsze działania.

Kolejny przykład użycia **MPI_Send()** i **MPI_Receive()**, tym razem celem wyznaczenia wariancji. Zwróćmy tutaj uwagę na sposób różnicowania kodu pomiędzy procesy

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define MASTER 0
#define TAG 'K'+'M'
#define MEGA 100000LU
int main( int argc, char *argv[] )
{
    unsigned long N,counter;
```

```
    double x,Sxx,Var;
    int world,current,len;
    char name[MPI_MAX_PROCESSOR_NAME+1];
    MPI_Status status;
    MPI_Init(&argc, &argv); //...standardowe inicjacja MPI
    MPI_Comm_size( MPI_COMM_WORLD, &world );
    if( argc>1 )
    {
        sscanf( argv[1],"%lu" ,&N );
    }
    else
    {
        N = 0;
    }
    N = ( N<1 )?( 1*MEGA ):( N*MEGA ); //...ustalmy ilość składników
    MPI_Comm_rank( MPI_COMM_WORLD,&current );
    MPI_Get_processor_name( name,&len );
    printf( "(proces%2d)@%s",current,name );
    if( current==MASTER )
    {
        printf( "...ilość składników %lu*%lu\n",N/MEGA,MEGA );
        for( current=MASTER+1,Sxx=0.0;current<world;current++ )
        {
            MPI_Recv( (void*)&x,1,MPI_DOUBLE,current,TAG,MPI_COMM_WORLD,&status );
            Sxx += x;
        }
        Var = Sxx/(N-1);
        printf( "...MASTER zyskał wariancję Var=%g\n",Var );
        printf( " dla %lu*%lu składników sumy\n",N/MEGA,MEGA );
    }
    else
    {
        for( counter=current,Sxx=0.0;counter<=N;counter+=(world-1) )
        {
            x = (double)rand()/RAND_MAX; Sxx += (x-0.5)*(x-0.5);
        }
        printf( "...SLAVE wysyła sumę %g\n",Sxx );
        MPI_Send( (void*)&Sxx,1,MPI_DOUBLE,MASTER,TAG,MPI_COMM_WORLD );
    }
    MPI_Finalize();
    return 0;
}
```

Przykładowy efekt wykonania dla 5 procesów

```
$ mpirun -np 5 ./sr 10
(proces 0)@cs-87756155163-default...ilość składników 10*100000
(proces 1)@cs-87756155163-default
(proces 3)@cs-87756155163-default
(proces 4)@cs-87756155163-default
(proces 2)@cs-87756155163-default
...SLAVE wysyła sumę 20787.4
```

```
...SLAVE wysyła sumę 20787.4
...SLAVE wysyła sumę 20787.4
...SLAVE wysyła sumę 20787.4
...MASTER zyskał wariancję Var=0.0831498
 dla 10*100000 składników sumy
```

Istnieje także wersja tej funkcji, w efekcie wywołania której wszystkie procesy komunikatora otrzymują wynik operacji

**int MPI_Allreduce(void *send, void receivef, int count, MPI_Datatype datatype, MPI_Op op,MPI_Comm comm);*

W takim razie nieco inny sposób rozwiązania wcześniejszego zadania.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define MASTER 0
#define MEGA 100000LLU
int main( int argc, char *argv[] )
{
    unsigned long long N,i;
    double x,Sxx,Var;
    int world,current;
    MPI_Init(&argc, &argv);
    MPI_Comm_size( MPI_COMM_WORLD, &world );
    if( argc>1 )
    {
        sscanf( argv[1],"%llu" ,&N );
    }
    else
    {
        N = 0;
    }
    N = ( N<1 )?( 1*MEGA ):( N*MEGA );
    MPI_Comm_rank( MPI_COMM_WORLD,&current );
    for( i=current,Sxx=0.0;i<N;i+=world ) //...teraz wykonują wszyscy
    {
        x = (double)rand()/RAND_MAX; Sxx += (x-0.5)*(x-0.5);
    }
    MPI_Reduce( &Sxx, &Var, 1,MPI_DOUBLE,MPI_SUM,MASTER,MPI_COMM_WORLD );
    if( current==MASTER )
    {
        printf( "Ilość elementów N*M=%llu*%llu\n",N/MEGA,MEGA );
        printf( "Otrzymana wartość Var= %lf\n",Var/(N-1) );
    }
    MPI_Finalize();
    return 0;
}
```

```
$  mpirun -np 4 ./reduce 10
Ilość elementów N*M=10*100000
Otrzymana wartość Var= 0.083150
```

Wnioski

1. **Inicjalizacja MPI:** Programy korzystające z MPI powinny zaczynać od inicjalizacji MPI za pomocą funkcji `MPI_Init`.
2. **Równoległe wykonywanie kodu:** Dzięki MPI można uruchamiać programy równolegle na wielu procesach, co pozwala na przyspieszenie obliczeń.
3. **Komunikacja między procesami:** Komunikacja między procesami w MPI opiera się na funkcjach takich jak `MPI_Send` i `MPI_Recv`. Procesy mogą wymieniać dane w sposób blokujący lub nieblokujący.
4. **Synchronizacja:** Funkcja `MPI_Barrier` umożliwia synchronizację wszystkich procesów, co może być istotne w pewnych etapach programu.
5. **Identyfikacja procesów:** MPI udostępnia funkcje do identyfikacji numeru procesu (`MPI_Comm_rank`) oraz liczby wszystkich procesów (`MPI_Comm_size`).
6. **Zakończenie MPI:** Po zakończeniu korzystania z MPI należy użyć funkcji `MPI_Finalize` w celu zwolnienia zasobów.

W praktyce, programy MPI są bardziej skomplikowane niż przedstawione przykłady, ale podstawowe zasady korzystania z tej biblioteki pozostają podobne. Programy MPI mogą być używane do równoległego przetwarzania dużych zbiorów danych, rozwiązywania problemów numerycznych czy symulacji.