Politechnika Śląska w Gliwicach
Instytut Informatyki
Zakład Oprogramowania

Laboratorium
Programowania Współbieżnego

Message Passing Interface

MATERIAŁY POMOCNICZE

WSTĘP

Przekazywanie komunikatów (ang. message passing) jest modelem odniesienia przy formułowaniu zadań i projektowaniu programów równoległych. W modelu tym zakłada się, że równocześnie wykonywane zadania komunikują się wysyłając i otrzymując komunikaty. Nie licząc komunikacji zadania są niezależne, tzn. przekazywanie komunikatów służy realizacji programowania wg zasady „wiele programów – wiele danych" (MIMD).
W rzeczywistych implementacjach tych wiele programów zakodowanych jest w postaci jednego. Niemniej jednak, w czasie wykonywania każdy proces wyróżniony jest identyfikatorem, który ustala kontekst dla procesu, a więc pozwala na przeprowadzanie różnych zadań, nawet jeśli zakodowane są one w pojedynczym programie.
Message Passing Interface (MPI) jest de facto standardem programowania z przekazywaniem komunikatów zarówno w języku C jak i w Fortranie. Jest to sprzętowo niezależna biblioteka ponad 100 funkcji, umożliwiających:

- Komunikację jeden do jednego
- Komunikację zbiorowa
- Modularyzację
- Definiowanie topologii procesów

Mnogość funkcji umożliwia budowanie wyrafinowanych programów równoległych, niemniej jednak MPI wymaga od programisty znajomości nie więcej niż 6 funkcji, aby zakodować wiele (jeśli nie wszystkie) algorytmów równoległych dostosowanych do modelu przekazywania komunikatów. Wspomniane funkcje wymienione są na rys.1 zaczerpniętym z [1].

ĆWICZENIA DO OPANOWANIA MPI

Zamieszczone poniżej przykłady zostały napisane przez Williama Groppa [2]. Oprócz sześciu podstawowych funkcji wykorzystują one także inne, głównie do komunikacji zbiorowej.

Pierwszy program: „Hello World"

Napisz program w MPI, w którym każdy proces MPI wyświetla

```
Hello world from process i of n
```

wykorzystując identyfikator w MPI_COMM_WORLD jako `i` oraz rozmiar MPI_COMM_WORLD jako `n`. Można założyć, że wszystkie procesy mogą wyświetlać informacje na ekranie.
Zwróć uwagę na kolejność wyświetlania. W zależności od zastosowanej implementacji MPI znaki z różnych wierszy mogą być wymieszane.

W rozwiązaniu należy zastosować następujące funkcje MPI:
`MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Finalize`

MPI_INIT(int *argc, char ***argv)
Initiate a computation.
argc, argv are required only in the C language binding,
where they are the main program's arguments.

MPI_FINALIZE()
Shut down a computation.

MPI_COMM_SIZE(comm, size)
Determine the number of processes in a computation.

IN	comm	communicator (handle)
OUT	size	number of processes in the group of comm (integer)

MPI_COMM_RANK(comm, pid)
Determine the identifier of the current process.

IN	comm	communicator (handle)
OUT	pid	process id in the group of comm (integer)

MPI_SEND(buf, count, datatype, dest, tag, comm)
Send a message.

IN	buf	address of send buffer (choice)
IN	count	number of elements to send (integer ≥ 0)
IN	datatype	datatype of send buffer elements (handle)
IN	dest	process id of destination process (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

MPI_RECV(buf, count, datatype, source, tag, comm, status)
Receive a message.

OUT	buf	address of receive buffer (choice)
IN	count	size of receive buffer, in elements (integer ≥ 0)
IN	datatype	datatype of receive buffer elements (handle)
IN	source	process id of source process, or MPI_ANY_SOURCE (integer)
IN	tag	message tag, or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (status)

Rys 1: Elementarne MPI. Wymienione funkcje wystarczą aby napisać szeroką klasę programów równoległych. Dla parametrów podano czy są to argumenty (IN), czy też wartości zwracane (OUT) oraz ich typ.

Rozwiązanie

```
#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
int  argc;
char **argv;
```

```
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    printf("Hello world from process %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

Współdzielenie danych

Często istnieje potrzeba, aby jeden proces pobrał dane od użytkownika, wczytując je z terminala lub jako argumenty linii poleceń, a następnie rozesłał tę informację do wszystkich pozostałych procesów.

Napisz program, który wczytuje wartość całkowitą i rozsyła tę wartość do wszystkich procesów MPI . Każdy proces powinien wyświetlić swój numer (rank) oraz wartość, którą otrzymał. Wartości powinny być wczytywane aż do podania liczby ujemnej.

W rozwiązaniu należy zastosować następujące funkcje MPI:
`MPI_Init, MPI_Comm_rank, MPI_Bcast, MPI_Finalize`

Solution

```
#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value;
    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    do {
       if (rank == 0)
           scanf( "%d", &value );

       MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD );

       printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);

    MPI_Finalize( );
    return 0;
}
```

Przesyłanie w pierścieniu

Napisz program, który pobiera dane od procesu 0 i wysyła je do wszystkich innych procesów wykorzystując przesyłanie w pierścieniu. Oznacza to, że proces `i` powinien odbierać dane i wysyłać je do procesu `i+1`, aż do momentu, kiedy dane dotrą do ostatniego procesu.

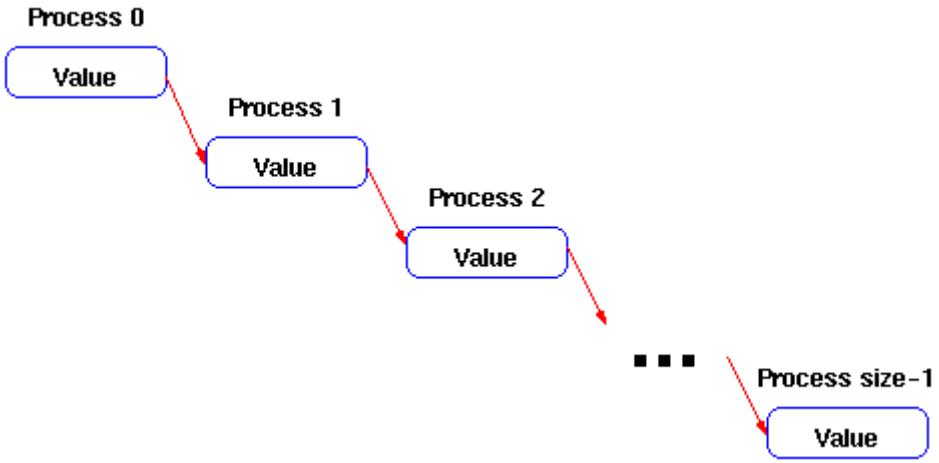

Figure 2: Schemat rozsyłania w pierścieniu.

Można założyć, że dane składają się z pojedynczej wartości całkowitej. Proces 0 pobiera dane od użytkownika.

W rozwiązaniu należy zastosować następujące funkcje MPI:
`MPI_Send, MPI_Recv`

Solution

```
#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value, size;
    MPI_Status status;

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    do {
      if (rank == 0) {
          scanf( "%d", &value );
          MPI_Send( &value, 1, MPI_INT, rank + 1, 0,
     MPI_COMM_WORLD );
      }
      else {
```

```
            MPI_Recv( &value, 1, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD,
                      &status );
            if (rank < size - 1)
              MPI_Send( &value, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD );
        }
        printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);

    MPI_Finalize( );
    return 0;
}
```

Obliczanie liczby PI z wykorzystaniem funkcji komunikacji zbiorowej

W ćwiczeniu przedstawiony jest prosty program do obliczania wartości `pi`. Wykorzystany algorytm został wybrany ze względu na swoją prostotę. W metodzie tej obliczana jest całka funkcji `4/(1 + x*x)` w przedziale pomiędzy 0 i ½. Całka jest przybliżana przez sumę `n` interwałów, przy czym przybliżenie wartości całki w każdym interwale wynosi `(1/n)*4/(1+x*x)`.
Procesor nadrzędny (o numerze 0) pobiera od użytkownika liczbę interwałów i rozsyła tę wartość do pozostałych procesorów. Następnie każdy proces dodaje wartość wyliczoną dla co `n`-tego interwału `(x = -1/2+rank/n, -1/2+rank/n+size/n,...)`.Na zakończenie, z wykorzystaniem operacji redukcji, obliczana jest globalna suma wartości obliczonych przez poszczególne procesory.

W rozwiązaniu należy zastosować następujące funkcje MPI:
`MPI_Bcast`, `MPI_Reduce`

```
#include "mpi.h"
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done)
    {
      if (myid == 0) {
          printf("Enter the number of intervals: (0 quits) ");
          scanf("%d",&n);
```

```
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;

        h   = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += 4.0 / (1.0 + x*x);
        }
        mypi = h * sum;

        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                   MPI_COMM_WORLD);

        if (myid == 0)
            printf("pi is approximately %.16f, Error is
%.16f\n",
                   pi, fabs(pi - PI25DT));
    }
    MPI_Finalize();
    return 0;
}
```

References

[1] Foster I. (1995) *Designing and Building Parallel Programs. Concepts and Tools for Parallel Software Engineering*, Addison Wesley

[2] http://www-unix.mcs.anl.gov/mpi/tutorial