Exercise 5

hand out: 2021-01-06, hand in: 2021-01-20 14:00

Introduction: Message Passing Interface

Overview

The Message Passing Interface (MPI) is a communication protocol that was designed to supply programmers with a standard for distributed-memory parallel programming that is portable and usable on a variety of platforms. MPI was designed to handle distributed-memory systems, i.e. clusters.

Since MPI provides a means to enable communication between different processes. It does not depend on shared-memory architectures. As is the case for multi-threading systems such as OpenMP. It can, however, make use of shared memory for fast and improved communication.

MPI is a language-independent communication protocol. It supports both point-to-point and collective communication. Besides communication primitives, MPI also provides topology, synchronization, I/O, and other facilities.

Concepts

The following concepts provide context for all MPI facilities.

Communicator objects connect a group of processes in the MPI session. Each communicator object gives a unique identifier and arranges the contained processes in an ordered topology. Communicators can be further partitioned using MPI calls.

Point-to-point is the simplest communication that occurs between just two processes in MPI session. Most popular use case is a pair of `MPI_Send/MPI_Recv` calls, but a number of other important MPI functions exist.

Collective functions involve communications among all processes in a process group. A typical example is `MPI_Bcast`, which broadcasts data from local process to all processes in a group. Another example is the `MPI_Reduce` function, which takes data from all processes in a group, performs a reduction operation, such as summing and stores the result on just one node. Other operations perform more sophisticated operations.

Derived Types are a MPI facility for defining messages sent between processes. It is necessary to specify the message content because MPI is intended to operate in heterogeneous environments. Many MPI functions require type information, because they perform operations on data. Examples for basic MPI types are `MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE`.

Setup And Usage

To compile and run MPI programs, you will need to install an implementation of the MPI standard. MPICH is a commonly used one. On Ubuntu 20.04 MPICH can be installed with the following command:

```
sudo apt install mpich
```

Alongside the MPI library and headers, it ships a compiler wrapper (`mpicxx.mpich`) that sets the necessary flags to link against MPI and the command `mpirun.mpich` that is used to start MPI programs. Example usage:

```
mpirun.mpich -N 4 ./foo
```

The flag `-N` is used to indicate the number of processes that should be started.

Functions

MPI offers a wide array of functions for communication and synchronization between processes. However, you will only need the most common ones in this exercise:

MPI_Init initializes the MPI runtime. Must be called before any other MPI functions.

MPI_Finalize shuts down the MPI runtime. Should be called right before the process exits.

MPI_Comm_rank is used to retrieve the rank of the calling process.

MPI_Comm_size is used to retrieve the number of active processes.

MPI_Send is used to send messages to other processes. Usage:

```
MPI_Send(
  const void *buf,          // send buffer
  int count,                // number of elements in buf
  MPI_Datatype datatype,    // datatype of each element in buf
  int dest,                 // rank of the destination process
  int tag,                  // message tag
  MPI_Comm comm             // communicator
);
```

MPI_Recv is the counterpart to `MPI_Send` and used to receive messages. Usage:

```
MPI_Recv(
  void *buf,              // buffer where the received message
                          // is written to
  int count,              // number of received elements
  MPI_Datatype datatype,  // datatype of received elements
  int source,             // rank of the sending process
  int tag,                // message tag
  MPI_Comm comm,          // communicator
  MPI_Status *status      // status object
);
```

Note: It is also possible to pass `MPI_ANY_SOURCE` as `source` to receive from any process.

MPI_Bcast broadcasts a message to all processes. Usage:

```
MPI_Bcast(
  void *buf,              // target buffer
  int count,              // number of elements
  MPI_Datatype datatype,  // type elements in buf
  int root,               // rank of process that performs the broadcast
  MPI_Comm comm           // communicator
);
```

MPI_Reduce performs a reduction operation across all processes. Usage:

```
MPI_Reduce(
  const void *sendbuf,     // input buffer
  void *recvbuf,           // output buffer, only significant for root
  int count,               // number of elements
  MPI_Datatype datatype,   // datatype of each element
  MPI_Op op,               // reduction operation
  int root,                // root process, receives the result
  MPI_Comm comm            // communicator
);
```

Some of the predefined operations to pass to `op` are

- `MPI_MIN`: calculate the minimum across all values
- `MPI_PROD`: calculate the product of all values
- `MPI_BOR`: calculate the bitwise or of all values
- `MPI_BAND`: calculate the bitwise and of all values

Example

In following example each process generates a random integer and sends it to process 0. Process 0 collects the received values and prints them on the console:

```
#include <mpi.h>
#include <cstdlib>
#include <iostream>

int main(int argc, char **argv) {

  MPI_Init(&argc, &argv);

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (rank == 0) {
    std::cout << "Hi, I am process 0!" << std::endl;
    for (int i = 1; i < size; i++) {
      int msg;
      MPI_Recv(&msg, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      std::cout << "Process " << i << " sent: " << msg << std::endl;
    }
  } else {
    srand(rank);
    int msg = rand();
    MPI_Send(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

Possible output:

```
$ mpicxx.mpich mpi_example.cc && mpirun.mpich -N 4 ./a.out
Hi, I am process 0!
Process 1 sent: 1804289383
Process 2 sent: 1505335290
Process 3 sent: 1205554746
```

More information

More information can be found at:
`https://computing.llnl.gov/tutorials/mpi/`

https://www.mpich.org/documentation/guides/

Task 1: Message Passing Interface (MPI) 10 Punkte

Answer the following questions in the context of an MPI environment. For yes/no questions provide a short explanation.

a) (2 P.) What of the following is true for MPI collective calls?

(i) Processes can call collective functions in any order (in regard to the other processes), if no data is transferred.

(ii) Only processes that send or receive data have to participate.

(iii) Every process in the communicator needs to participate in the call.

b) (2 P.) Is the rank of a process always globally unique?

c) (2 P.) Can a process send a message to itself?

d) (2 P.) Will MPI schedule all issued sends and receives in a fair way?

e) (2 P.) What size is mandatory for the receive buffer?

(i) Can be any size, if too small MPI will keep the rest of the data available for an additional receive call.

(ii) Exactly the same size as the incoming message.

(iii) At least the size of the message but can be larger.

Task 2: Message Passing Mandelbrot 30 Punkte

Adapt the provided Mandelbrot set calculation to be used in an MPI environment.

a) (12 P.) Adapt the sequential Mandelbrot set calculation (mandelbrot_a.cc) by distributing the work over N processes (You can assume that the number of pixels is a multiple of N). Again each process is assigned its own color to identify which process calculated the pixel. You can use the provided `make_color` function or your own solution for this. While all processes (including rank 0) calculate their own horizontal stripe of the image, only the process with rank 0 collects and encodes the final image. For message passing, use the `MPI_Send/MPI_Recv` pair.

Analyze your results and discuss the resulting Mandelbrot image.

b) (14 P.) In this exercise process 0 will distribute the work on-the-fly to the remaining processes. (mandelbrot_b.cc)

Process 0 asks the user for a block size B. This value is then broadcasted to all other processes. Process $1, \dots, N$ ask process 0 for a line L and calculate a stripe of B lines starting at L. (You may assume that B is a multiple of the image height.) Upon completion they ask process 0 for the next position. Repeat until the entire image has been distributed among the workers. You can use $L = -1$ to indicate to a worker that no more work items are available. Finally reassemble the resulting image with `MPI_Reduce` in process 0 and encode it. Again, assign a unique color to each worker process.

c) (4 P.) Analyse your results from task **b)** for different block and worker pool sizes. What might have been the intention behind this approach and what problems do you see?

A simple MPI example can be found at `https://en.wikipedia.org/wiki/Message_Passing_Interface`. The template uses the MPICH MPI implementation. Under Ubuntu, you can install it with `apt install mpich`. To check if installed correctly you can compile and execute the provided template code as-is.

Task 3: Cache Miss Penalty

5 Punkte

We have three different applications A,B, and C. The following table gives the total number of instructions, the number of instructions which contain memory accesses, and the miss rate for several cache sizes. Please note that the total number of instructions already includes the number of instructions with memory access.

application	instructions	memory accesses instructions	miss rate at 64 kB	128 kB	256 kB
A	2 billion	1 billion	0.08	0.06	0.04
B	1.5 billion	900 million	0.07	0.06	0.05
C	2 billion	1.2 billion	0.09	0.07	0.05

Furthermore, we have the following 6 systems, characterized by their clock rate, cache size and miss penalty. Assume every instruction can be performed in one clock cycle (if no stall cycles occur).

system	clock rate	cache size	miss penalty
S1	1 GHz	64 kB	15 cycles
S2	1 GHz	128 kB	18 cycles
S3	1 GHz	256 kB	20 cycles
S4	2 GHz	64 kB	35 cycles
S5	2 GHz	128 kB	38 cycles
S6	2 GHz	256 kB	30 cycles

a) (5 P.) What is the execution time of all three applications on each of the 6 systems?