Introduction to High-Performance Computing, Winter 2021/22

Exercise 5: MPI

The Message Passing Interface (MPI) provides an API for portable message passing. It is implemented as a library that can be linked against applications using it. For convenience, MPI implementations provide so-called compiler wrappers that automatically pass all command-line arguments (e.g., location of header and library files) to the underlying compiler to use MPI.
The names of those wrappers are not standarized and may vary across MPI implementations. Often, they have a prefix of '`mpi`' followed by the compiler name. On the CLAIX cluster of RWTH Aachen University, Intel compiler and Intel MPI are recommended, although other compilers and Open MPI are available as alternatives.
The compiler wrapper names to use Intel Compilers with Intel MPI are as follows:

`mpiicc` Intel MPI compiler wrapper using the Intel C compiler

`mpiicpc` Intel MPI compiler wrapper using the Intel C++ compiler

`mpiifort` Intel MPI compiler wrapper using the Intel Fortran compiler

To compile and link MPI programs with this combination simply replace the call to the underlying compiler with one using the corresponding wrapper scripts (e.g., replace `icc` by `mpiicc` on your compile line).

```
% mpiicc myapp.c -o myapp
```

Furthermore, MPI provides special startup mechanisms, to ensure that the corresponding number of processes are spawned on an appropriate number of compute nodes and that these processes can communicate with each other. Often, these startup scripts are called '`mpirun`' or '`mpiexec`', but sometimes these mechanisms are also integrated in a batch scheduling system like SLURM, that also handles reservation of nodes and efficient scheduling of batch jobs. The CLAIX cluster at RWTH Aachen University is using such a setup, where the startup script is named `'srun'`. However, for the purposes of this exercise, it should be sufficient to use the convenience variable `${MPIEXEC}` with the parameter `-np` to automatically schedule your application on a shared node as follows:

```
% ${MPIEXEC} -np <numberOfProcesses> <executable>
```

Problem 1 (Message passing in a ring)**.** A simple way to implement the reduce-to-all collective communication operation is by passing the partial results as messages in a logical ring of processes. Each process sends its partial result to its right neighbor, then receives the partial result of its neighbor on the left and combines it with the local data in order to get an updated partial result. This is repeated as many times as is the number of the processes in the logical ring. After the last step each process will hold the result of the reduction operation.

The following C code has been written as a sample implementation of the algorithm described above:

```c
#define LEN 4

int local[LEN];
int result[LEN] = { 0 };
int i, rank, size, step;
MPI_Status status;

/* Put some rank-specific values in local[] */

MPI_Comm_size(MPI_COMM_WORLD, &size);

for (step = 0; step < size; step++)
{
   /* Send our partial result to the next rank */
   MPI_Send(result, LEN, MPI_INT, next, 0, MPI_COMM_WORLD);

   /* Receive the result from the previous rank */
   MPI_Recv(result, LEN, MPI_INT, prev, 0, MPI_COMM_WORLD, &status);

   /* Combine the local values with the partial result */
   for (i = 0; i < LEN; i++)
      result[i] += local[i];
}

/* result now contains the vector of reduced values */
/* Should have equal content in all ranks - check that */
```

The program appears to be working correctly for small values of LEN on certain machines but otherwise deadlocks on other machines or when the value of LEN is large. What is the reason for such behavior? Suggest at least two different ways to remove the deadlock and write sample code to illustrate each solution.

Modify the program segment so that it performs product reduction instead of summation.

Problem 2 (Tree-based collective broadcast)**.** To improve the scalability of collective operations, tree-based algorithms are often used as they usually provide good general scalability with $\mathcal{O}(\log\ P)$ communication rounds (with P processes), as opposed to the $\mathcal{O}(P)$ communication rounds needed in the naïve ring approach of the Problem 1.
An alternative algorithm for broadcasting data is the binomial-tree algorithm as discussed in Exercise 3. It's based on $\mathcal{O}(\log\ P)$ rounds, where for each next round the number of communicating processes doubles. An intuitive approach for choosing communication partners is the following (where we assume the `root` to be rank 0):

1. The size of the interval considered starts with the size of the communicator.
2. With each round, the interval size is halved.
3. A process that has data to be broadcast (initially `root`, then all processes that received data so far) choose the recipient as the rank determined by

$$\text{receiving rank} = \text{sending rank} + \frac{\text{interval size}}{2}$$

4. Exchange data until the receiving process is the process itself.

Problem 2.1. Visualize for 8 processes of ranks 0 to 7 for each round the message transfers between the ranks.

Problem 2.2. How would one need to adapt the equation for choosing the communication partner to allow for arbitrary root ranks?

Problem 2.3. Implement a broadcast function using MPI point-to-point communication for an array of integer values and arbitrary root processes using the binomial-tree algorithm outlined above.

Problem 3 (Dissemination pattern)**.** An algorithm for reduction and barrier synchronization is the dissemination pattern. It is also round-based with a $\log p$ rounds, just like the binomial-tree algorithm. Here, we will use it to implement a barrier synchronization.
For each round the distance to the next process is doubled. The initial distance is 1. The receiving rank is chosen via:

$$\text{receiving rank} = (\text{sending rank} + \text{distance}) \mod (\text{total number of processes})$$

Problem 3.1. Visualize for 8 processes of ranks 0 to 7 for each round the message transfers between the ranks.

Problem 3.2. Implement a barrier function using MPI point-to-point communication for and arbitrary number of processes using the dissemination algorithm outlined above.

Problem 4 (Derived Datatypes)**.** MPI provides several routines to define derived datatypes. The two type handle creation functions `MPI_Type_contiguous` and `MPI_Type_vector` were shown to provide a very intuitive interface to rows and columns in two dimensional data.

Problem 4.1. Implement the data exchange of square integer matrices of size 10 between two processes using derived datatypes, where the initial matrix of the sender is transposed after receiving the matrix on the receiver process using MPI derived datatypes.

Problem 4.2. What is the *size* and *extent* of the derived datatypes used?

Discussion of this exercise on January 14, 2021.