Programming of parallel computers
Computer Lab no. 2: Communications in MPI

DCAMM, DTU and IT, Uppsala University

January, 2008

Some of the suggested exercises utilize a number of available test codes, which can be downloaded from
`http://www2.imm.dtu.dk/courses/FortranMPI/MPI/Labs/Lab2.`

Exercise 1 Read through the following programs, which illustrate different types of communications in MPI. Compile and run the codes for varying number of PEs.

1. The test code `alltoall` is an example of using nonblocking communications.
2. The test code `IO_gather` illustrates gathering messages from all processes.
3. The test code `ring`, as the name suggests, simulates a ring architecture and sends a short message from one to the next PE, until the message comes back to the processor which initiated the send-loop.

Exercise 2 Compare the parallel performance of two vector operations: a scalar product ($s = \mathbf{v}^T\mathbf{w}$) and a vector update operation ($\mathbf{r} = \mathbf{v} + \alpha\mathbf{w}$), where $\mathbf{v}, \mathbf{w}, \mathbf{r}$ are vectors (one-dimensional arrays) of size pN, α, s are scalars and p is the number of the PEs involved in the program run. We will assume that the arrays are already distributed among all PEs.
Observe that the second operation is fully local, while the first one requires both local operations and a global communication.
Proceed as follows:

- Initialise the local array size N. This can be done in two ways:
  - Assign a value to N explicitly in the source code. Then one needs to edit and recompile every time when one wants to vary N.
  - Read the current value of N from processor P0 and sent it to all others.
- Allocate three double precision arrays of size N on each PE, call them `V, W, R`, say, and initialize their entries to be equal to 1, 2 and 0, correspondingly.

  Assign some value to a scalar variable `alfa`.

- Perform the operation `R = V + alfa*W` and time it (using `MPI_WTIME`).
- Perform the operation `s = sum(V(i)*W(i), i=1,...,p*N`. It requires two steps: (a) a local array multiplication and (b) a global communication, say `ALL_REDUCE`. Time the whole operation.

Do the following test runs:

1. Keep the number of PEs equal and increase the value of N, say 100, 1000, 10000, 100000.
2. Keep the size of N constant, but large enough, say 100000, and increase the number of processors.

How much communication overhead does the global communication routine add to the overall computing time, compared to local computations only?
On a piece of paper write down briefly your observations and comments, and deliver it to Maya Neytcheva before you finish the lab.

Exercise 3 (Ping-pong) This is the classical "ping-pong" communications test, where process 0 sends a message to process 1, which then sends it back to process 0.

1. Consider the code `pingpong`. Compile and execute it on two processors.
2. Study the implemented algorithm and the input parameters message length, number of hops. Can we draw some conclusion about the underlying interconnection network by varying those parameters?

Exercise 4 (round-robin) Test MPI's basic blocking send/receive timings by sending a message on a "round-robin" trip through the processors.
Basic outline of the test:

- Allocate an array msg(1:n) in Fortran (or msg[n] in C) for a message of n double precision numbers.
- Initialize MPI
- Get p - the number of processes and $myrank$ - this process's rank.
- Call `MPI_BARRIER` to let everybody synchronize.
- Process 0 initializes msg[n] to have all zeros in it.
- Process 0 calls `MPI_WTIME()` to get start time tstart of the messaging circle.
- Send the message msg around the ring; each time it arrives at a process k, it will have all of its entries incremented by 1.0

- When the message gets back to process 0, it gets the finish time and then checks the array msg to make sure it has the correct values in it.

In the special case where the number of processes is $p = 2$, this round-robin algorithm acts similarly to the ping-pong algorithm from (Exercise 3).
Doing with it p being larger and sending the message all the way around all processes could help to average out any differences in the underlying communications hardware/software topology and implementation and we could see an average of messaging performance. The total number of messages required for this is p, so it should be like sending a single message p times.

1. Create a program code which implements the above algorithm. Use blocking send and receive communications. Possibly you can modify `pingpong`.

2. When you have done your own implementation and it works correctly, compare with the given code `ring`. What is the difference? You may also change `ring` to send not integers but long messages as in your code and check whether the different type of send commands make difference in the overall time, and for how long messages.

3. Additional task: building and testing a parallel performance model

   For the implementation with blocking send-receive, try to analyse the underlying computer network using the following simple linear model

   $$T(n) = T_{lat} + \frac{n}{R_{bw}},$$

   where $T(n)$ is the total execution time, n is the length of the message sent (in double words), T_{lat} is the latency time to get a message started, R_{bw} is the bandwidth rate in double words per minute.

   Initially try message sizes that are powers of two: $8, 16, ..., 63556$. Finding T_{lat} and R_{bw} is then a matter of solving a linear least squares fit, which one can do easily with Matlab (or Excel, or a hand calculator, or any other valid way). Then one could test how well the model predicts the communication time for messages of other lengths: 1000, 10000, 100000.

   Warnings: However, beware that it is not bulletproof. For instance, the least squares fit may produce a value of T_{lat} which is negative. Or it could happen that the average time for a message of length 32 is smaller than that for a message of length 8. What interpretation can one have in such a case?