Programming of parallel computers
Computer Lab no. 1: Simple MPI programs

IT, Uppsala University

March-May, 2019

Getting started

It is assumed that you have some basic experience with the Unix/Linix command line. If you need to refresh your skills, you can review the first section of the first lab of the High Performance Programming course. Also, remember that the man command is always your friend. You can either use the Thinlinc-server or log in with `ssh -X username@servername.it.uu.se` or to one of the Solaris x86_64 servers listed at `http://www.it.uu.se/datordrift/faq/unixinloggning`. The Linux servers available for students are listed at `http://www.it.uu.se/datordrift/maskinpark/linux`.

In the lab, a number of test programs are to be studied. These should be downloaded from the course webside at the student portal, under 'Course Material/Labs'. The files are `hello.c`, `deadlock.c`, `pi.c`, `onetoall.c`, `exchange.c`, `pingpong.c`, `buffon-laplace.c`, `buffon-laplace.h` and `Makefile`.
Download the file `lab1-code.zip` into your directory and unpack it. In case you want to move files between your laptop and the Unix servers you can use the scp command, e.g.,
`scp file username@servername.it.uu.se:` or
`scp username@servername.it.uu.se:catalog/file .` moving a file to your computer. For editing files you can use `emacs` or some other editor that you are used to on the Linux system.
Compile and run MPI programs Have a look at `Makefile` to see how to compile MPI programs. Notice that we use `mpicc`. The runs are then handled with the command `mpirun -np N a.out` requesting N processes to run program `a.out`.

Report

The lab is a part of the examination. To pass you need to either attend the lab and work actively on the tasks (no need to finish all tasks), or write a short informal report summarizing your findings together with your source code for the different tasks, where some programming is required.

Exercises

Exercise 1 (Start and stop MPI)

1. Try the program `hello`, which is the very first step in using MPI. Compile and link the program issuing the command
   `make hello`

   This will use the the makefile to build an executable file named `hello`.

   Run the program on one and more processes by issuing the command

   `mprun -np 1 hello` (for one process) or, say,
   `mprun -np x hello` (for x processes)

2. Modify the program so that each process prints its number (rank) in addition to the greetings. Also make the process with rank zero to print the total number of processes that take part in the current execution of the program.

Exercise 2 (Deadlock test) In the `deadlock` program two processes try to exchange the variable a between each other, and save it as b using synchronous communications.
This is one of the simplest examples of such - the pair `SEND -- RECEIVE`, which performs to so-called *point-to-point* communication.

1. Build the executable `deadlock` by issuing `make deadlock`.

2. Run the program:
   `mprun -np 2 deadlock`. As the name indicates, the program will hang forever in a deadlock and you have to use `<Ctrl/C>` to break out the deadlock.

3. Modify the code so that it will not deadlock.

Exercise 3 (Point-to-Point communication) In the program `exchange` two processes exchange the variable a between each other, and save it as b using point-to-point communication.

- Compile the program via `make exchange` and run with two processes, `mpirun -np 2 ./exchange`. Study the source file and compare with the output of the program.

- Modify the program to include a third process. The three processes should now exchange data in a circular fashion, with p0 sending to p1, p1 to p2 , and p2 to p0. You will need to add a branch case for the third process, modify the source and destination ranks in the communication calls, and a distinct message tag for the new communication. Which order of the send and receive calls is correct for the third process? What happens if the wrong order is used?

- Modify the program to handle arbitrary number of processes exchanging data in a circular fashion.

- Notice that the program gets rather complicated already for this simple example with potential deadlocks if things are not right. Also, note that all the transfers are performed in succession without any overlap. Modify your program to instead use non-blocking communication with `MPI_Isend` and `MPI_Irecv` (see the documentation `http://www.open-mpi.org/doc/current`). You will need to supply `MPI_Request` handles to the calls, and add appropriate `MPI_Wait` commands to wait for communication to finish. Make sure that your program is correct by consulting the instructor. What is the correct order of the calls now? What are the advantages of using non-blocking communication?

Exercise 4 (Ping-pong) Study the program file `pingpong.c`. The code uses a synchronous send and a blocking receive to send a message between two processes back and forth, for different lengths of the message. Similarly to the other cases, build the executable with `make pingpong`. Run the code on two processes and look at the time and transfer rate. The minimal time is an estimate of the latency (start-up time), and the maximal transfer rate is an estimate of the bandwidth of the interface. You can also copy-paste the table to Matlab and plot the time and/or rate as a function of message size, preferably in a log-log scale.

Exercise 5 (Communication 'one-to-all') Study the case where we distribute data from one process to all others. In the program `onetoall` we solve this by sending data from process 0 to the others one at a time.

- Compile and run the program `onetoall`. Study the source file and compare with the output.
- One problem with the solution above is that process 0 is more loaded than the others. Modify the program to a pass-on sequence as in Figure 1 to get a more fair load between the processes.

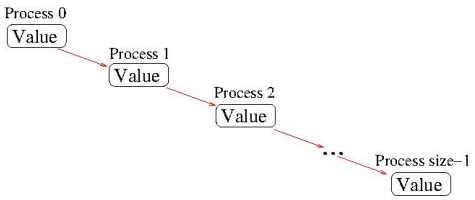


Figure 1: 'Pass-on' communication pattern

- Challenge (optional): A more efficient way to globally share data is to use a *fan-out* sequence as in Figure 2, i.e., in the first step process 0 sends to process 1, in the second step process 0 sends to process 2 while process 1 sends to process 3, in the third step all four processes send to processes 5-7, etc. Modify your pass-on program to implement the fan-out sequence with point-to-point communication. Assume that the number of processes is 2^k, where k is an integer.

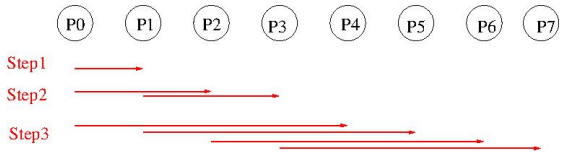
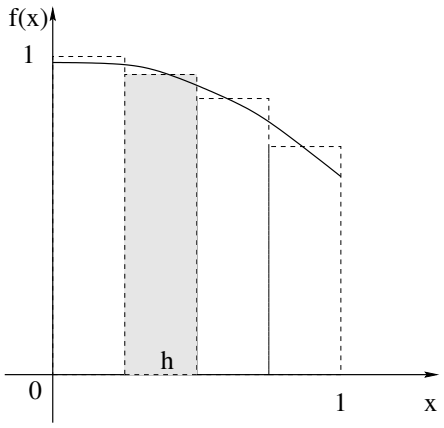
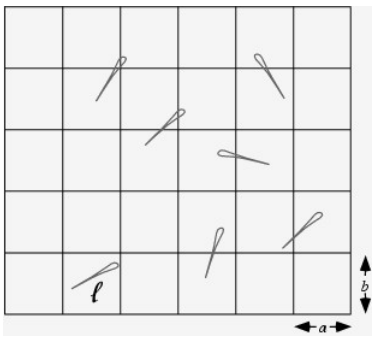


Figure 2: 'Fan-out' communication pattern



(a) π (numerical integration)



(b) π (Buffon-Laplace)

Figure 3: Two ways to compute π

Exercise 6 (Computing the value of π in parallel) An example of a *Divide-and-conquer* technique, or the *domain decomposition* approach.
The program `pi.c` computes π in parallel using numerical integration.
The well-known way to compute π is the following:

$$\int_0^1 \frac{4}{1+x^2} dx = 4 \int_0^1 d(arctg(x)) = 4\, arctg(x)|_0^1 = \pi$$

If we use the midpoint rule, we can compute the above integral as follows (see Figure 3a):

$$\int_0^1 \frac{4}{1+x^2} dx = h \sum_{i=1}^{n} \frac{4}{1+x_i^2} = h \sum_{i=1}^{n} \frac{4}{1+((i-1/2)h)^2}.$$

The trivial parallel implementation of the latter formula is that if we have p processes available, we slice the interval into p pieces, attach one interval to each of them to compute a partial sum, and then sum-up, say, in one process, which will know the answer.
Collecting the partial results can be done in two ways:
(a) using global reduction operation (a very simple example of another type of communication in MPI, the so-called *collective communication*).
(b) using point-to-point communications (send and receive).

1. Study both source code and observe what MPI functions are utilized.

2. Build the executable `pi` by `make pi`

   Run the program on various number of processes.

   The way to compute the global sum is by using variant (b). The interval $[0, 1]$ is subdivided into 10^8 subintervals and each process takes $\frac{1}{p}10^8$ subintervals to work with.

3. Change the code appropriately so that you can measure the execution time by adding `MPI_Wtime` in the code.

```
double t_begin, t_end;
...
t_begin = MPI_Wtime();
do something
t_end = MPI_Wtime();
printf("Elapsed time : %1.2f\n", t_end-t_begin);
...
```

4. You are welcome to try to use a collective communication to collect the sum.

Exercise 7 (Computing the value of π using a Monte Carlo method)

The Buffon-Laplace-needle problem

Problem Statement: More than 200 years before Metropolis coined the name 'Monte Carlo' method, George Louis Leclerc, Comte de Buffon, proposed the following problem.
'If a needle of length ℓ is dropped at random on the middle of a horizontal surface ruled with parallel lines a distance $d > \ell$ apart, what is the probability that the needle will cross one of the lines?' This problem was first solved by Buffon (1777, pp. 100-104), but his derivation contained an error. A correct solution was given by Laplace (1812, pp. 359-362; Laplace 1820, pp. 365-369, see the link below).
`http://mathworld.wolfram.com/Buffon-LaplaceNeedleProblem.html`

We reformulate somewhat the original problem in the following way. Imagine that a needle of length ℓ is dropped onto a floor with a grid of equally spaced parallel lines distances a and b apart, where ℓ is less than a and b (see Figure 3b). The probability that the needle will land on at least one line is given by

$$P(\ell, a, b) = \frac{2\ell(a+b) - \ell^2}{\pi\, ab}$$

(Uspensky 1937, p. 256; Solomon 1978, p. 4, see the link for full reference). The idea now is to keep dropping this needle over and over on the table, and to record the statistics. Namely, we want to keep track of both the total number of times that the needle is randomly dropped on the table (call this N), and the number of times that it crosses a line (call this C).

If you keep dropping the needle, eventually you will find that the number $\dfrac{N(2\ell(a+b) - \ell^2)}{Cab}$ approaches the value of π.

(Note that for large N the quantity C/N approaches the probability $P(\ell, a, b)$.)

In order to get a reasonably accurate approximation of π we need to perform a number of trials of order $10^6 - 10^8$. Since the separate trials are completely independent, we can perform those in parallel and sum up the result. This problem is an example of a trivial parallelism.

1. Study the implementation of the above algorithm (`pi_buffon-laplace.c`).
2. Compile and run the program for different numbers of processes (1, 2, 4,...).
3. How does the accuracy of the computed value of π behaves on one and many processing elements (PEs)? It is better to have more processes used, compared with one? How much more? What is the reason for the error behaviour you observe - the parallelization of the code, the algorithm or something else?