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

DCAMM, DTU and IT, Uppsala University

January, 2008

A number of test programs are to be studied. Most of them are provided both in Fortran and C. The programs to begin with are
`hello_0.f, deadlock.f, pi_0.f` (respectively `hello_0.c, deadlock.c, pi_0.c`) and can be downloaded from
`http://www2.imm.dtu.dk/courses/FortranMPI/MPI/Labs/Lab1`.
Make a copy of these files, together with the example makefile in some of your directories.

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 -f Makefile_F hello`

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

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

   `mprun -np 1 hello` (for one processor) or, say,
   `mprun -np 4 hello` (for four processors).

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

3. Check in the sub-directory `hello_pool`. There you find further examples of bringing some extra functionality in this MPI code.

Exercise 2 (Deadlock test) In the `deadlock` program two processors 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 up the executable `deadlock` by issuing `make -f Makefile_F 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.

   To help this, below you find information of the parameters of the MPI_SEND and MPI_RECV routines.

MPI_SEND (buf, count, datatype, dest, tag, comm, status)		
IN	buf	initial address of send buffer
IN	count	number of entries to send
IN	datatype	datatype of each entry
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator

MPI_RECV(buf, count, datatype, source, tag, comm, status)		
IN	buf	initial address of receive buffer
IN	count	number of entries to receive
IN	datatype	datatype of each entry
IN	source	rank of source
IN	tag	message tag
IN	comm	communicator
OUT	status	return status

   Hint: Try to understand the importance of the message tag parameter.

Exercise 3 (Computing the value of π in parallel) An example of a *Divide-and-conquer* technique, or the *domain decomposition* approach.
The program `pi` computes π in parallel using numerical integration.
The well-known way to compute π is the following:

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

If we use the midpoint rule, we can compute the above integral as follows (see Figure 1):

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

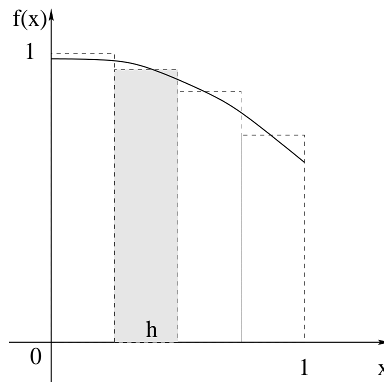


Figure 1:

The trivial parallel implementation of the latter formula is that if we have p processors 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 processor, 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 codes in Fortran and C and observe what MPI functions are utilized.

2. build the executable `pi` by
   `make -f Makefile_F pi`

   Run the program on various number of processors.

   The C-program is written in such a way that the interval $[0, 1]$ is subdivided into 10^8 subintervals and each processor takes $\frac{1}{p}10^8$ subintervals to work with.

   For the F-program the number of subintervals (n) per processor is an input parameter. Thus, the total number of subintervals is $n * p$.

   The way to compute the global sum in the C-version is by using variant (b).

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

```
...
double precision MPI_WTIME,t_begin,t_end
...
t_begin = MPI_WTIME()
do something
t_end=MPI_WTIME()
write(*,*) my_proc,' Time: ',t_end-t_begin
```

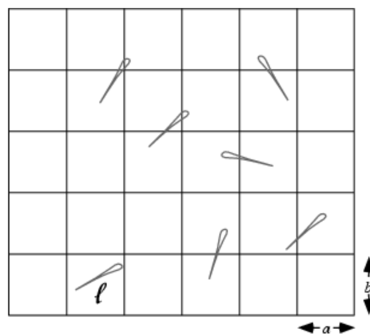
4. Check in the sub-directory `Num_int`. There are several other implementations of the same algorithm. The routine `trapets`, for example uses another numerical integration scheme (the trapets-rule) which has a higher numerical accuracy. Also, one cam see the serial implementation of the algorithm. Pay particular attention to the implementation of the communications in `pi_send` and `pi_reduce`.

<u>Exercise</u> 4 (Computing the value of π using a Monte Carlo method)

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).
`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.



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).

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 $\frac{N(2\ell(a+b) - \ell^2)}{Cab}$ 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.f`).
2. Compile and run the program for different numbers of processors (1, 2, 4,...).
3. How does the accuracy of the computed value of π behaves on one and many PEs? It is better to have more processors 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?
4. Amend the code in order to be able to time the execution. Do you observe that it scales? For instance, change the code so that you can have the same total number of trials, distributed evenly over a varying number of processors. What happens with the execution time when you increase the number of processors? Does the time on one PE reduce twice when two processors are used, compared to one PE? Or four times when four PEs are used?

Exercise 5 (If you have some time and you are curious...)

There are more programs in the sub-directory `More_ex`.
You can study them, see what are they doing compile and run them. Are they correct, for example, `simple_mp`?