PPCES 2013: MPI Lab

12 March 2013
https://sharepoint.campus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/mpilab2013.tar
Hristo Iliev, iliev@rz.rwth-aachen.de
Christian Iwainsky, iwainsky@rz.rwth-aachen.de
Sandra Wienke, wienke@rz.rwth-aachen.de

Abstract

The purpose of this Lab is to make you feel comfortable with the basic concepts of MPI. In the **morning session** you will deal with the principles of point-to-point communication while accomplishing tasks 1-3. In the **afternoon**, you will put hands on tasks practicing the usage of collectives and MPI in general (tasks 4-6). Furthermore you will get to know Vampir which is a performance analysis tool but is also very well suited for visualization of message passing, which we will be using for this workshop.

Before you start

Before you start, log onto one of our Linux cluster frontends (cluster-linux, cluster-linux-nehalem) using X-Win32. Then, download the MPI Lab from the PPCES webpage. Extract it to a suitable location (e.g. create directory named MPILab, go there and extract the downloaded archive using “tar -xf ../mpilab2013.tar”).
The lab archive contains the stubs for the exercises described below. Intermediate solution stubs are provided where appropriate. Please don’t cheat as this will deprive you of a lot of learning experience.
The following make targets will be available for all exercises.

```
make [release | debug]                  # build the program
make run [NPROCS=<#processes>]          # run the program
make runParallel [NPROCS=<#processes>]  # run the program in parallel (ex. 6 only)
make clean                              # clean directory
make vampir                             # trace the program and start Vampir
```

1. Ping Pong

One basic MPI program using point-to-point communication is a “ping pong” between two MPI processes. A ping-pong program skeleton can be found in directory ***1_pingPong***. Complete the source code parts marked by “TODO” of this program. Note: You can use Vampir to visualize the behavior of your code.

a) Modify the ping-pong stub, such that the first process of the MPI program transmits its input to the second process. The second process should then print the received value and send its negative value back to the first process, which should again print the received value.
b) Modify the ping-pong program such that each rank sends an individual random number of elements. Hint: Note that you may have to transmit the number of elements to the second process before sending them in an additional message.
c) What is the behavior of the program for NPROCS=1 and NPROCS=3? Modify the code to display an error message for too few processes and to execute properly for larger number of processes.

d) Implement part b) of the assignment without explicitly sending the number of elements.

e) Bonus task: Implement a loop to send/receive messages with different sizes. How does the message size influence the time being spent in MPI functions? You may use MPI_Wtime() to measure process local time and set the maximum array size to 2^{26} to make the impact of the data size clearer.

2. Sending and Receiving

One basic usage of MPI is to send and receive data. This however can lead to unexpected situations if not done in the correct way. A send-receive skeleton can be found in directory ***2_sendReceive***.

a) Look at the given program and execute it with 2 processes. What is happening?
Note: You can abort the program execution by hitting Ctrl-c.

b) Modify the given program using MPI_Send() and MPI_Recv() such that it becomes a correct MPI-program and completes execution.

c) Can the send and receive operation be replaced with a single MPI call? Use the correct operation to replace the send and receive.

d) Modify your code to utilize non-blocking communication primitives.

e) Change the code to work with more than 2 MPI processes. In this case the messages should be sent to and received from the next higher rank.
Hint: Will a special treatment be necessary for the last rank?

3. Count-down Ring

Using send and receive operations, implement a round-robin communication that passes along an integer value, starting with the stub given in directory ***3_countdownRing***. Each time the value is received it should be decremented by a random number (use function **random_dec**). Once this value reaches zero or less, the process that is currently updating it should notify all other processes of its rank. Every process then should display the rank of the process which decremented the counter to zero. You can supply the initial countdown value like that:

```
make run N=<countdown>
```

Compare the following example output for this exercise:

```
> make run NPROCS=4 N=55
Counting down from 55
Process 1 has received the bomb (54 on the clock) and is still alive!
Process 2 has received the bomb (53 on the clock) and is still alive!
Process 3 has received the bomb (52 on the clock) and is still alive!
Process 0 has received the bomb (51 on the clock) and is still alive!
Process 1 has received the bomb (50 on the clock) and is still alive!
Process 2 has received the bomb (49 on the clock) and is still alive!
Process 3 has received the bomb (48 on the clock) and is still alive!
Process 3 has received the bomb (29 on the clock) and is still alive!
Process 0 has received the bomb (47 on the clock) and is still alive!
Process 0 has received the bomb (23 on the clock) and is still alive!
Process 1 has received the bomb (41 on the clock) and is still alive!
Process 1 has received the bomb (15 on the clock) and is still alive!
Process 2 has received the bomb (35 on the clock) and is still alive!
Process 2 has received the bomb (7 on the clock) and is still alive!
Process 3 lost
I am process 0 and 3 is the looser
I am process 2 and 3 is the looser
I am process 1 and 3 is the looser
I am process 3 and 3 is the looser
```

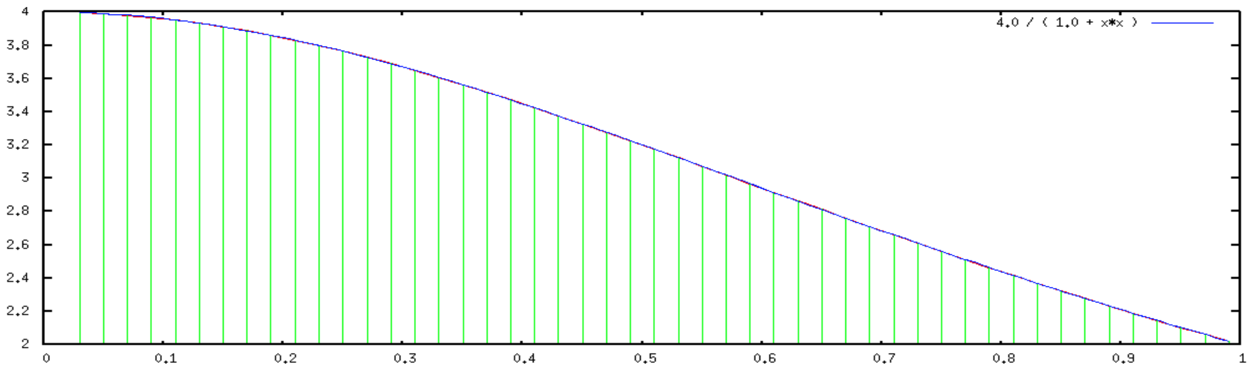
4. Controller-Worker

The controller-worker concept is an often used technique in MPI applications. There one “controller” process distributes work to the other “worker” processes. In this exercise you should implement this concept using (again) point-to-point communication routines.
The trapezoid numeric function integration rule serves as a basis for this exercise. The following formula is parallelized:

$$\int_{x_0}^{x_1} f(x)\,dx \approx \frac{x_1 - x_0}{2N} \sum_{i=0}^{N-1} \left(f(x_i) + f(x_{i+1})\right); \quad x_i = x_0 + i * \frac{x_1 - x_0}{N}$$

As a test for this integration we use an integral representation of π.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$



Code for this assignment is located in directory ***4_integration***.

a) Compile and then execute the given example with different process counts, e.g. NPROCS=2, NPROCS=4. Why is no speedup observed? You can use Vampir to get an insight.

b) Modify the given example so that the work is correctly done in parallel.

c) Refactor (rewrite) your program so that rank 0 calls a function called **controller** and every other process calls a function **worker**. The program should still compute the integral in parallel, with the work distribution occurring in the **controller** function and the work processing in the **worker** function.

d) Calling MPI_Send()/MPI_Recv() for each function evaluation is not very efficient. Modify your code to transmit ranges to be computed instead of single values.
What effect does this have on the work balancing?

5. A stepping stone to global communication

Often data needs to be distributed from a single process to all processes or collected from all the processes into a single rank of the MPI program. MPI provides dedicated functions to do that. In this exercise you will implement your own global operations like broadcast, scatter, gather and all-to-all using only point-to-point MPI calls. The code stubs are located in directory ***5_myGlobals***. Note that you can compile and execute the code at any time to observe data distribution. This example is simple toy that will be used to practice message passing by transmitting process-specific random numbers to the neighboring processes. You may specify a process rank as an argument to the executable and that process will plot its data structures so you can investigate the communication behavior.

a) Implement the **bcast_Int** function. It should distribute an integer value from the process with rank equal to *root* to all other processes in the communicator *comm*. After the operation completes all processes should have a copy of the data from process with rank *root*.

b) Implement the **scatter_Int** function. This function should distribute the contest of the send buffer in process *root* to the participating processes. The first *sendcnt* integers of *sendbuffer* should be available in the receive buffer of rank 0, the next *sendcnt* integers at rank 1 and so forth.

c) Implement the **gather_Int** function which is the reverse operation of **scatter_Int**. It should collect *recvcnt* integers from each process in the communicator and store them in the receive buffer of the process with rank *root*.

d) Implement the **alltoall_Int** function. Its operation is a combined scatter-gather. Rank 0 scatters its data to ranks 1, …, nProcs-1, then rank 1 scatters to 0, 2, …, nProcs-1 and so forth.

e) Implement the **sum_Int** function. It should collect at process *root* a single integer value from each process in the communicator *comm* and sum all collected values. Use it to sum over all processes' ranks.
Note: sum {0, 1, …, numProcs-1} = numProcs*(numProcs-1)/2

f) Now use the corresponding collective MPI calls. What is the difference?

g) You may have observed that the output may have been shuffled. Consider the potential reasons for this. Fix it.

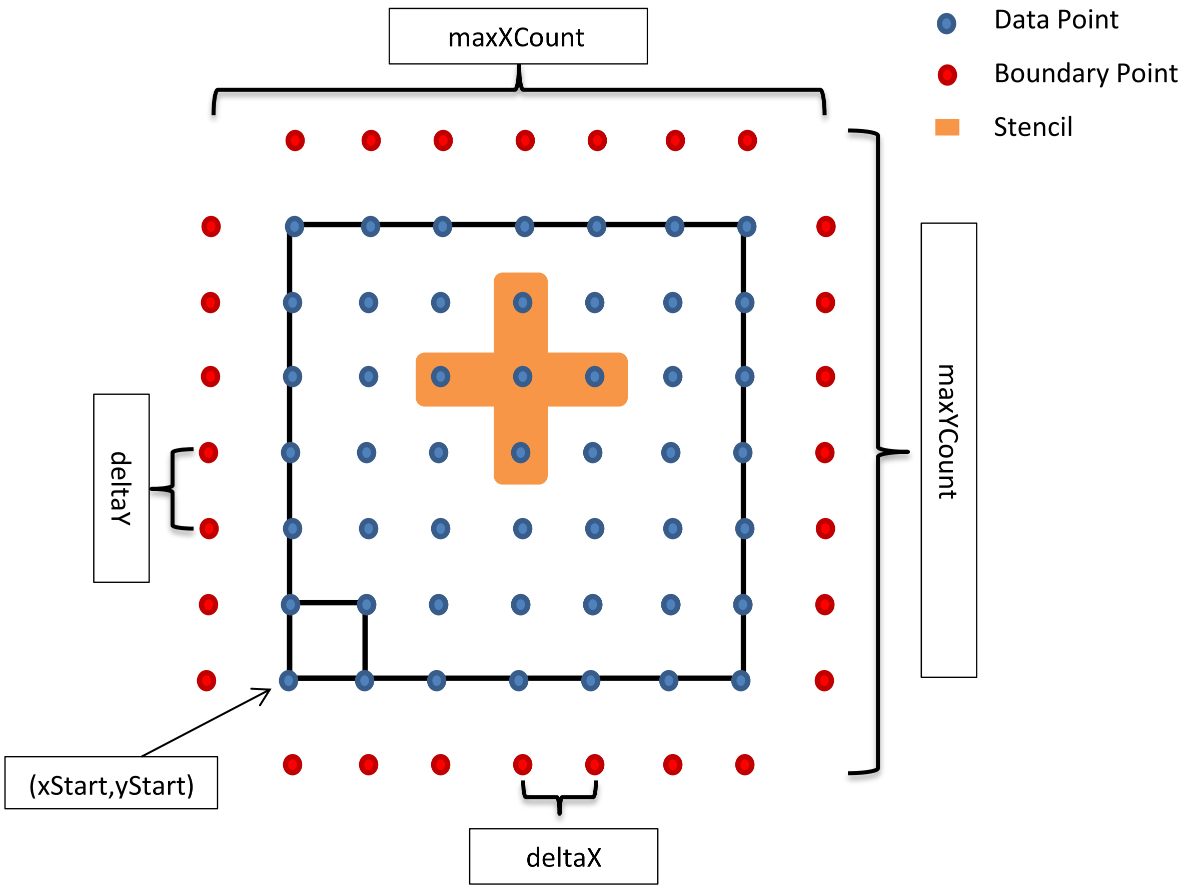
Note: Don't forget to test your code with different process counts.

6. Your first own parallelisation

The program in directory ***6_jacobi*** solves numerically a finite difference version of the screened Poisson equation:

$$(\nabla^2 - \alpha)u = \frac{d^2}{dx^2}u + \frac{d^2}{dy^2}u - \alpha u = f$$

using the iterative Jacobi method with over-relaxation.



In this exercise you must apply everything you have learnt today in order to parallelise the given serial version of the Jacobi solver, as this is what you will usually have to do on your own after this course. Change to directory ***6_jacobi***.

In general the algorithm works as follows:

a. Initialise *u_old* to zero
b. Compute an approximation at every point of *u* (a discretization point of your solution vector) based on the former approximation *u_old* (implemented in function **one_jacobi_iteration**)
c. Swap *u* and *u_old*
d. Repeat from b) until there is no significant change anymore or the maximum number of iterations is reached

The figure above shows some of the important variables in the implementation of the algorithm.
Note: Even though the number of data points in each direction is 7 one needs 9 points to include the boundary.

Depending on how confident you feel about programming with MPI you can either start with the already partitioned **jacobi.c** / **jacobi.f90** and complete the TODOs in it or you can try to do the full parallelisation from scratch. We have provided the original sequential version of the solver as **jacobi_serial.c** / **jacobi_serial.f90**. You may find the following guiding questions helpful:

What are the dependencies for the computation of a single iteration?
What communication method is suitable for computing the error?
How would you partition the domain? Is there any difference between FORTRAN and C?
Can you handle the case when domain size is not integer divisible by the number of MPI processes?

7. Using the Batch System

Long-running computations and computations that must run on an empty machine to minimize side effects (e.g. benchmarks) should be submitted to the batch system (Platform LSF). A batch script must be submitted using the **bsub** command. A batch script is basically a shell script containing all commands which must be executed together with a list of (recommended) options for the batch system. It is also possible to provide these options as command line arguments to **bsub**. In directory ***7_batch*** you will find an example script ***submit.sh*** together with the sample MPI program ***sendRecv.c***. Note that the batch script can (and should!) be tested interactively before you submit it. The interactive test lets you more easily spot eventual errors without the implied waiting time of the batch system. To run the batch script interactively first make it executable with the command:

```
chmod 755 submit.sh
```

and then execute it:

```
./submit.sh
```

Note: MPI_EXEC line will execute with two processes only as the value of the environment variable FLAGS_MPI_BATCH depends on the execution environment. For interactive testing you have to set this variable:

```
FLAGS_MPI_BATCH="-np 4" ./submit.sh
```

After you have tested your script you can submit it to the batch system:

```
bsub < submit.sh
```

Notice the I/O redirection symbol. It is very important: if you omit it the job will be queued but none of the LSF options specified inside it will be respected.
The batch system responds to the submission with the job ID of the queued job:

Job <xxxxxx> is submitted to default queue <qqqqqq>.

The status of all your jobs can be monitored by the **bjobs** command. Only pending (PEND) and running (RUN) jobs as well as jobs in error state are shown; finished jobs are not listed.
You can cancel a job with the **bkill** command:

```
bkill xxxxxx
```

where *xxxxxx* is the job ID from the output of the **bsub** command.

Compile and run this program to check your batch script.
Submit your script to the batch system and wait it to complete.

Note: More information about batch jobs can be found in the HPC Primer at http://www.rz.rwth-aachen.de/hpc/primer.