PPCES 2020: MPI Lab

December 3, 2020
Marc-André Hermanns, hermanns@itc.rwth-aachen.de

Portions thanks to:
Christian Iwainsky, Sandra Wienke, Hristo Iliev, Joachim Protze

Synopsis

The purpose of this hands-on lab is to make you familiar with the basic concepts of MPI. Tasks 0–4 will introduce you to the principles of basic point-to-point communication. Task 5 will practice the usage of collective communications and MPI in general. Furthermore, you will get to know ARM Performance Reports in task 6, which is a generic performance analysis tool. There is also an optional task 7 on the SLURM batch system.
Note: The flow of exercises does not strictly match the structure of the course. It is not required that you complete the exercises in the order in which they appear in this document.

Before You Start

Before you start, log into one of our Linux cluster frontends (**login18-1**, **login18-2**) using FastX. Then, download the MPI lab archive at https://tinyurl.com/PPCESonline-2020-MPI-Lab (if the link does not work, check out the PPCES web site) and extract it to a suitable location (e.g., in a directory named **PPCES2020/MPI**) using the following commands :

```
mkdir –P ~/PPCES2020/MPI
cd ~/PPCES2020/MPI
wget https://tinyurl.com/PPCESonline-2020-MPI-Lab
tar -xvf PPCESonline-2020-MPI-Lab
```

The lab archive contains skeleton code for the exercises described below. Intermediate solutions are provided where appropriate. Sample solutions to all problems are also provided in the **solutions** folder. We would advise you to not look at the solutions before you have tried your best to solve each exercise on your own.

Each problem comes with a Makefile with the following targets:

```
make [release | debug]                   # build the program
make run [NPROCS=<#processes>]           # run the program
make clean                               # clean the directory
make totalview                           # run program with TotalView debugger
```

For exercise 6 the following additional targets are available in addition to the common set:

```
make runParallel [NPROCS=<#processes>]   # run the program in parallel (make run = serial)
make runReports [NPROCS=<#processes>]    # profile with ARM Performance Reports
```

0. Hello, MPI!

Lecture Sessions: MPI Overview, MPI Concepts

The purpose of this exercise is to get you familiar with the very basics of MPI programming on the RWTH Compute cluster. Start with the minimal program in directory ***0_helloMPI*** and add a line that makes each process print its rank and the total number of processes in the MPI program. Follow the instructions in the source code. Use the following commands to compile and run your program:

C/C++:

```
$MPICC -o hello hello.c
$MPIEXEC -n 4 ./hello
```

Fortran:

```
$MPIFC -o hello hello.f90
$MPIEXEC -n 4 ./hello
```

1. Ping Pong

Lecture Sessions: Blocking Point-to-Point Communication

One basic MPI program using point-to-point communication is the “ping pong” between two MPI processes. A ping-pong program skeleton can be found in directory ***1_pingPong***. Complete the source code parts marked with “TODO”.

a) Make the first process of the MPI program transmit its input to the second process. The second process should then print the received value and send it back with an opposite sign to the first process, which should again print the received value.

b) Make each rank send an individually and randomly selected number of elements. Let the other process know in advance the size of the array by explicitly sending it as an additional message.

c) What is the behaviour of the program for NPROCS=1 and NPROCS>2? Modify it to display an error message when started with too few processes and to execute properly with more than two 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 wall-clock time and go with array size as high as 2^{26} elements to make the impact of the data size clearly visible.

2. More Sending and Receiving

Lecture Sessions: Blocking Point-to-Point Communication, Non-Blocking Point-to-Point Communication

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

a) Take a look at the given program source code and execute it with two processes. What is happening? Hint: You can abort the program execution by hitting Ctrl-c.

b) Modify the program to use **MPI_Send()** and **MPI_Recv()** such that it becomes a correct MPI program and completes execution.

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

d) Modify your code to utilise non-blocking communication operations.

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 needed for the last rank?

3. Count-Down Ring

Lecture Sessions: Blocking Point-to-Point Communication, Non-Blocking Point-to-Point Communication

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

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

An example output for this exercise follows (note how message lines from different ranks might become intermixed):

```
> make run NPROCS=3 N=45
Counting down from 45
Process 1 has received the bomb (38 on the clock) and is still alive!
Process 2 has received the bomb (35 on the clock) and is still alive!
Process 0 has received the bomb (31 on the clock) and is still alive!
Process 0 has received the bomb (13 on the clock) and is still alive!
Process 1 has received the bomb (24 on the clock) and is still alive!
Process 1 has received the bomb (6 on the clock) and is still alive!
Process 2 has received the bomb (17 on the clock) and is still alive!
Process 2 has received the bomb (4 on the clock) and is still alive!
Process 0 lost
I am process 0 and 0 is the loser
I am process 2 and 0 is the loser
I am process 1 and 0 is the loser
```

4. Controller-Worker Pattern (optional)

Lecture Sessions: Blocking Point-to-Point Communication, Non-Blocking Point-to-Point Communication

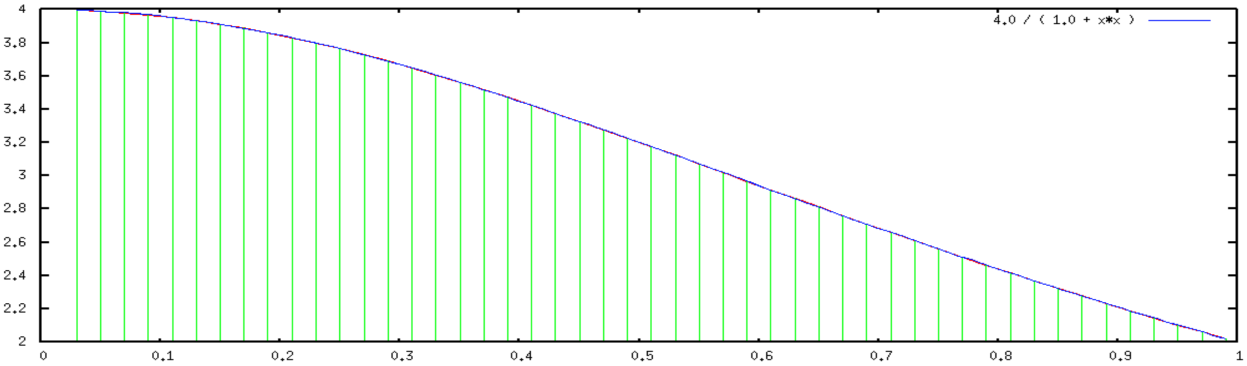
The controller-worker pattern is an often-used approach to solving the work imbalance problem in MPI applications. A small number of processes (the controllers; usually just one) distribute work items to one or more processes (the workers). In this exercise, you should implement this pattern using point-to-point communication routines.

The trapezoid numeric function integration rule serves as basis for this exercise. The following formula is parallelised:

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

To test this integration algorithm, we implement one of the integral representations of π (to simulate an irregular problem, we also introduce an artificial delay proportional to x^2; see **func**):

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



The skeleton code for this assignment is located in directory ***4_integration***.

a) Compile and then execute the given program with different number of MPI processes, e.g. with NPROCS=2, NPROCS=4, etc. Why is no speedup being observed?

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

c) Refactor (rewrite) the program and place all controller code into a function called **controller** and all worker code into a function called **worker**. Have rank 0 call **controller** and every other process call **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) Distributing the function evaluation at individual points is not very efficient. Modify the code to work on intervals (blocks of points) instead of on individual points. What effect does the block size have on the work balance?

5. Your Own Collective Communication

Lecture Sessions: Collective Communication

Often data in parallel algorithms needs to be moved around in a very structured fashion, e.g. broadcasted from a single rank to all ranks or distributed in chunks among the ranks. MPI provides for that purpose specialised operations called collective communication operations (*collectives* for short). In this exercise you will implement your own collectives, namely broadcast, scatter, gather, all-to-all, and sum reduction, using only the point-to-point MPI operations. The code skeletons are located in directory ***5_myGlobals***. Those come with a built-in debugging mechanism that allows you to observe the data distribution. If you specify a process rank as an argument to the executable, that process will dump its data structures so you can investigate the communication behavior (if not specified, rank 0 will be debugged).

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 ranks should hold the same integer value as the one that rank *root* has.

b) Implement the **scatter_int** function. This function should distribute the content of the send buffer in process *root* to *all* ranks in the communicator. The first *sendcnt* integers of *sendbuffer* should become 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* rank in the communicator and store them in the receive buffer of rank *root*. The received data chunks should be ordered according to the rank of the process where they originate.

d) Implement the **alltoall_int** function. Its operation is a combination of a scatter and a gather. Rank 0 scatters its data to ranks 0, 1, …, nProcs-1, then rank 1 scatters to 0, 1, …, nProcs-1 and so forth. Receivers should order the pieces following the rank of the sender.

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 the ranks of all processes.
Note: 0 + 1 + … + numProcs-1 = *numProcs*(numProcs-1)/2*; compare it with the value your code computes.

f) Now add the corresponding collective MPI calls where noted. Compare the results?

g) You may have observed that the output has been shuffled. Consider the potential reasons for it. Try to fix it.

Note: Don’t forget to test your code with different process counts. Be careful when a process sends data to itself.

6. Performance Analysis of a Jacobi Solver

Lecture Sessions: ARM Performance Reports

The program in directory ***6_jacobi*** solves numerically using the Jacobi method with successive over-relaxation a finite difference version of the screened Poisson equation:

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

In this exercise you analyze the performance of an already parallelized Jacobi solver.

In general, the algorithm works as follows:

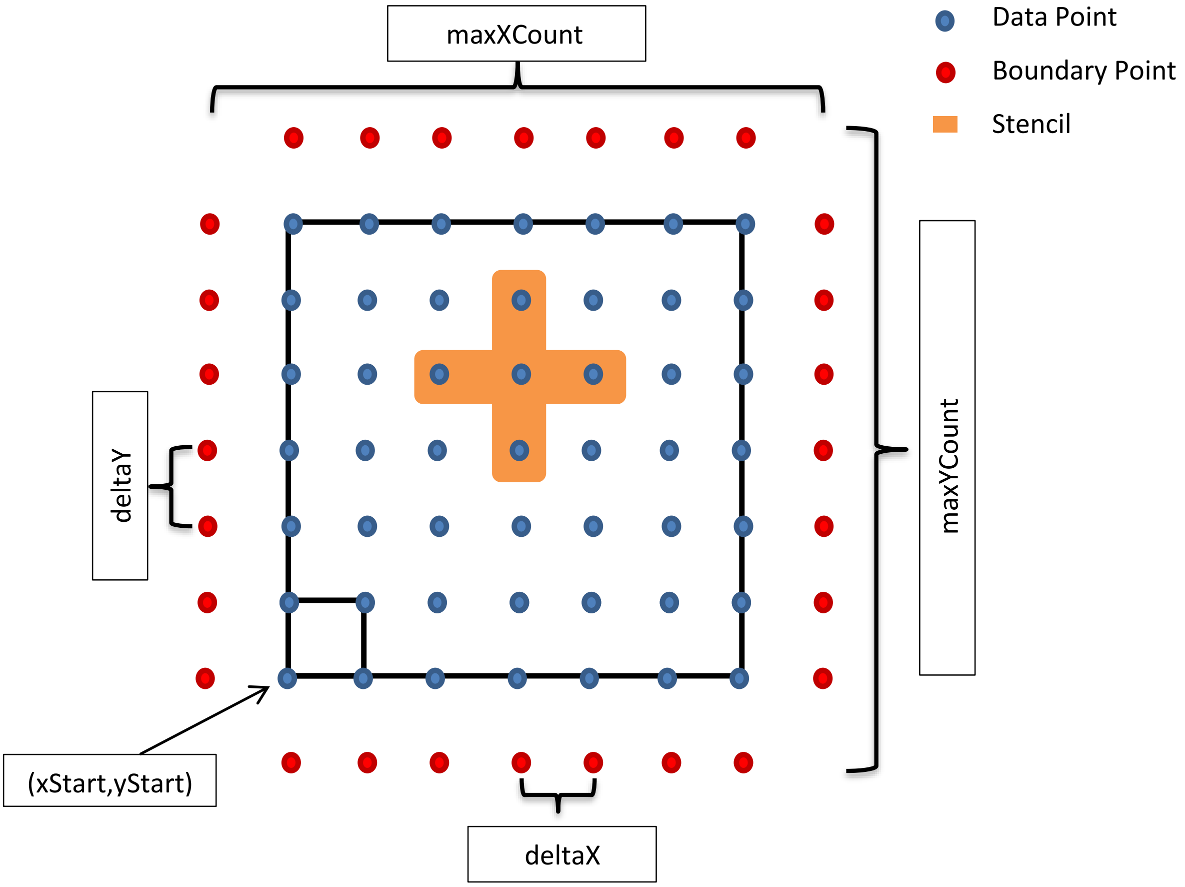
a. *u_old* is initialised to zero

b. A new approximation of *u* is computed at every point (a discretisation point in your solution vector) based on the previous approximation *u_old* (the whole procedure is implemented in function **one_jacobi_iteration**)

c. *u* and *u_old* are swapped.

d. Steps b) and c) are repeated over and over until the method converges (i.e. values no longer change significantly after the Jacobi iteration) or the pre-set maximum number of iterations is reached.

.



The parallelized version of the Jacobi solver is available in the file **jacobi.c / jacobi.f90**. We have provided the original sequential version of the solver as **jacobi_serial.c** / **jacobi_serial.f90**.

You can examine the performance of the Jacobi solver using a multitude of tools available on the RWTH Compute Cluster. Among those tools are: ARM Performance Reports, Vampir, and Cube. The easiest to use is ARM Performance Reports (perf-reports), which presents in a very concise form an overview of the program performance characteristics. To generate a report, use the following command:

```
make runReports
```

This will run the program with four MPI processes (unless NPROCS is set). Once the program execution is complete, the report will be available in both text and HTML formats. Use a web browser (e.g., firefox) to see the output. Identify how much time was spent in MPI operations and how much of the compute time was spent in vector operations. Is the application memory-bound (which means that most of the CPU time is spent in memory accesses)? Especially have a look at the breakdown for that.

The hotspot of the application is the function **one_jacobi_iteration**, i.e., it is the function in which most of the time of the program execution is spent. It calculates the new approximations of every point according to the equation described above. There is a performance problem introduced in this code due to the way how the nested for-loop iterates over the matrix. The matrix itself is stored (for both the C and the FORTRAN code variant) in **column-major order**. Such an order defines how the matrix elements are mapped to memory addresses in linear memory. Column-major order means that consecutive elements of a column are stored next to each other in memory, whereas row-major order means that consecutive elements of a row are next to each other in memory. Assume you store an $n \ x \ n$ matrix A in linear memory. Then, you could declare an array M of size n^2 and use the following rule to map the elements a_{ij} to the array elements of M:

map(i, j) = M[j * n + i] for column-major order
map(i, j) = M[i * n + j] for row-major order

The nested for-loop in the function **one_jacobi_iteration** iterates over the matrix (stored in column-major order) row by row and not column by column. This leads to a performance penalty because the caches of the CPU cannot be used efficiently. Fix the nested for loop in the function **one_jacobi_iteration** such that the iteration is done column by column (i.e., first calculate all new elements of the first column, then the second column and so on). After you have fixed the problem, run ARM Performance Reports again on the program and compare the result to the previous execution. How does the performance behavior of the application change?

7. Using the Batch System (optional)

Long-running computations and benchmarks should be submitted as jobs to our batch system SLURM[1]. Jobs are represented by batch scripts – shell scripts containing all commands which need to be executed together with a list of resource requirements and options for the batch system. The **sbatch** command is used to submit jobs. It is also possible to provide options as command-line arguments to **sbatch**. In directory ***7_batch*** you will find an example script ***submit_claix2018.sh*** together with the sample MPI program ***jacobi.c***. The batch script could (and should!) be tested interactively before submission. 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, use the following commands (don't forget to compile the program first):

```
chmod 755 submit_claix2018.sh
./submit_claix2018.sh
```

Note: The program 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_claix2018.sh
```

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

```
sbatch  submit_claix2018.sh
```

The batch system then responds to the submission with the job ID of the queued job:

```
Submitted batch job xxxxxx.
```

The status of all your jobs can be monitored by the **squeue** command. Only pending (PD) and running (R) jobs as well as jobs in error state are shown; finished jobs are not listed. It might take (quite) some time before the job starts!

```
squeue -u $USER -a
```

You can cancel a job at any time with the **scancel** command:

```
scancel xxxxxx
```

where *xxxxxx* is the job numeric ID from the output of the **sbatch** command

[1] More information about batch jobs can be found in the IT Center documentation Wiki