PPCES 2011: MPI - Lab

24 March 2011
https://sharepoint.campus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/work.tar.gz
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 principles of point-to-point communication accomplishing tasks 1-3. In the **afternoon**, you will put hands on tasks practicing the usage of collectives and MPI in general (tasks 5-7). Furthermore, you get to know Vampir which is initially a performance analysis tool but also very good suited to visualize message passing, which we will be using for this workshop.

Before you start

Before you can start, login to one of our Linux cluster frontends (cluster-linux, cluster-linux-nehalem) using the X-Win32. Then, download the MPI Lab from the PPCES webpage. Extract it to a suitable location (e.g. create directory MPILab, go there and extract using "tar –zxf ../work.tar.gz").
It contains the stubs for the exercises described below. Where applicable there will be intermediate solutions. 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 clean                              # clean directory
make vampir                             # 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 the folder ***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 print the received value and send the 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 with an additional message.
c) What is the behavior of the program for NPROCS=1 and NPROCS=3. Modify the code to provide an error message for too few processes and to complete without an error for larger process counts
d) Implement part b) of the assignment without explicitly sending the number of elements.

e) Bonus task: Implement a loop to send/receive different message sizes. How does the message size influence the time being spend in MPI functions? You may use MPI_Wtime() to measure process local time and set the maximal 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 situation if not done in the correct way. A send-receive skeleton can be found in the folder ***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-x.
b) Modify the given program using MPI_send and MPI_Recv such that it is a correct MPI-program and completes execution.
c) Can the send and receive operation be replace 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 send and received to and from the next higher rank.
Hint: Will a special treatment be necessary for the last rank?

3. Count-Down-Ring

Using send and receive, implement a round robin communication passing an integer value along using the stub given in directory ***3_countdownRing***. Each time this value is received it should be decremented by a random number (given function **random_dec**). Once this value reaches zero, the process that is currently updating it should notify all other processes of its rank. Every process then should display the process which decremented the counter to zero. A stub for this exercise can be found in the folder countdown. You can modify the maximal value for the countdown by:

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

Compare the following example output for this exercise:

```
> $MPIEXEC -np 4 countdown.exe 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
```

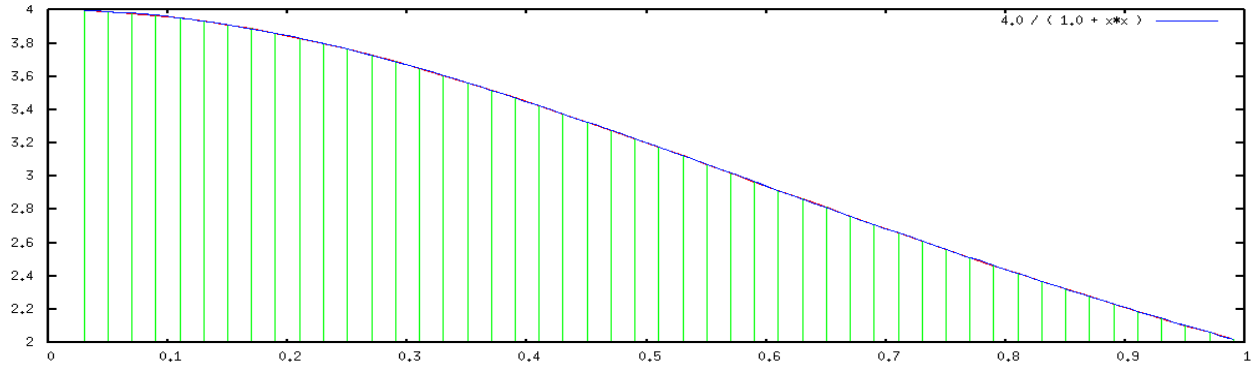
4. Master Worker

The master-worker concept is an often used technique in MPI applications. Thereby, one master process distributes work onto the other processes (=worker). In this exercise, you should implement this concept using (again) point-to-point communication routines.
The trapezoid- integration of a function serves as a basis for this exercises. Thereby, the following formula is parallelized:

$$\int_{x_0}^{x_1} f(x)\,dx \approx \frac{x_1 - x_0}{N} \sum_{i=1}^{N} f(x_i); \quad x_i = x_0 + \frac{(i - 0.5)}{N} * (x_1 - x_0)$$

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

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



For this assignment please open the directory ***4_integration***.

a) Compile and execute the given example with different process counts, e.g. NPROCS=2 NPROCS=4. Why is there no speedup? You can use Vampir for easier insight.
b) Modify the given example such that the work is correctly done in parallel.
c) Refactor (rewrite) your program such that rank 0 calls a function called master and every other process calls a function slave. The program still should compute the integral in parallel, with the work-distribution occurring in the master function and the work-processing in the slave 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.
Which effect does this have on the work-balancing?

5. Stepping-stone to global communication

Often data needs to be distributed from a single process to all processes or collected from all the processes to a single rank of the MPI program. MPI provides dedicated function for this. In this exercise you will be implementing your own global operations like broadcast, scatter, gather and all-to-all using only MPI-Send and receive functions. Please change to directory ***5_myGlobals***. There you will find the stub for this exercise. Note, that you can compile and execute this exercise at any time to observe data distribution. The given code is a simple toy example, that will be used to practice message passing by transmitting process specific random numbers to the neighboring processes. You may specify a rank number after the executable at startup time, which will be plotting its data structures to investigate the communication behavior.

a) Implement the bcast_Int function. It should distribute the integer data from the process with the rank equal to root to all other processes in the communicator comm. After the operation all processes should have a copy of the data of the process with rank “root”
b) Implement the scatter_Int function. This function should distribute the contest of the send-buffer to the participating processes. The first “sendcnt” integers of sendbuffer should be available in the receive-buffer on rank0, the next “sendcnt” integers and rank 1 and so forth.
c) Implement the gather_Int function, which is the reverse operation of the scatter_Int. It should collect recvcnt integers from each process in the communicator and store them in the receive buffer of the root-ranked process.
d) Implement an all to all. This is a combination of scatter on all processes. Rank 0 scatters its data to 1..nProcs-1, rank 1 to 0,2,… nProcs-1 and so forth.
e) Now use the MPI-Collectives. What is the difference?
f) You may have observed that the output may have been shuffled. Consider potential reasons for this.

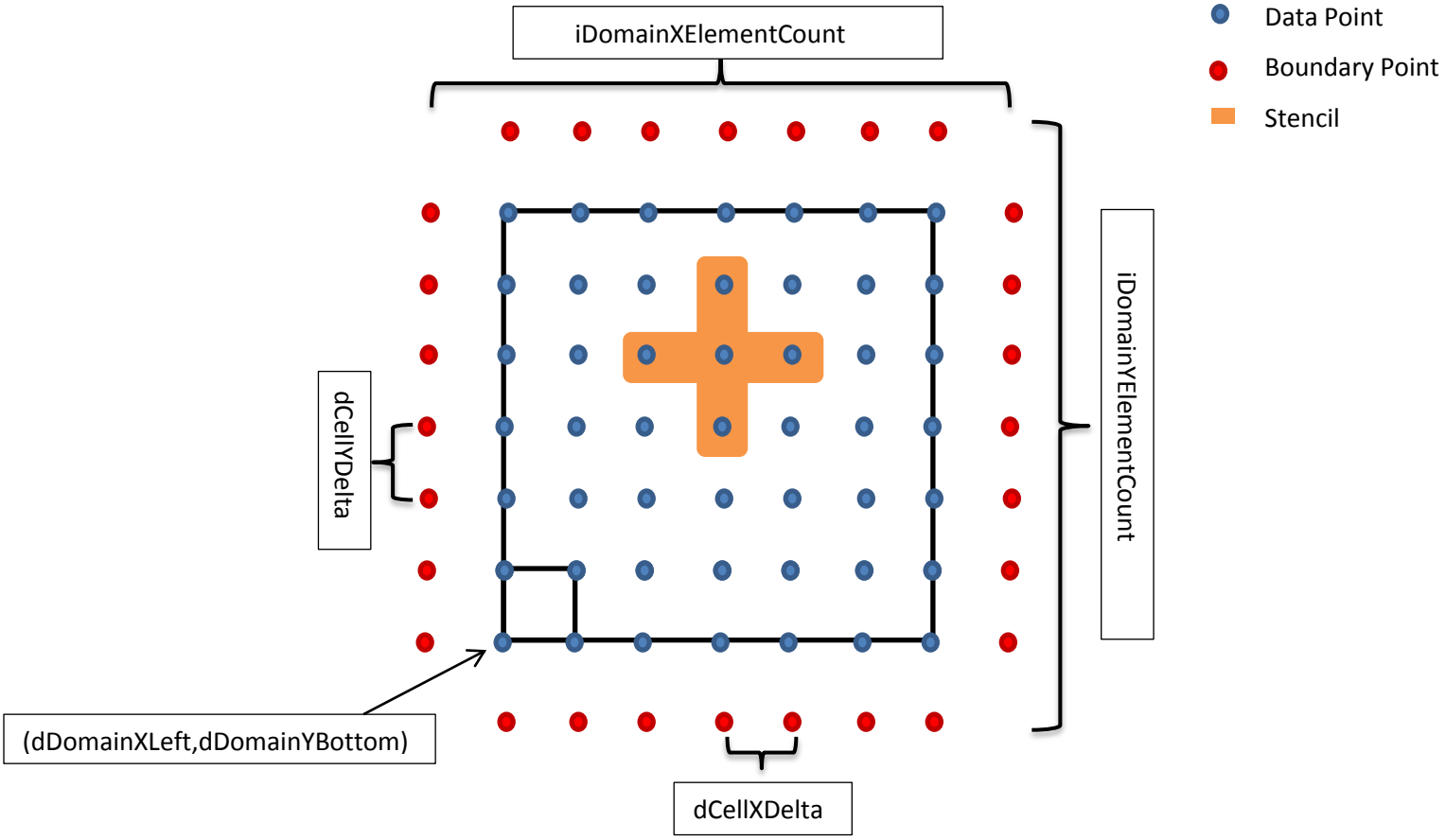
Note: Don’t forget testing your code with different process counts.

6. My first own parallelization

The program in directory ***6_ghostCells*** solves a finite difference discretization of the Helmholtz equation

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

using the Jacobi iterative method.



In this exercise you should apply everything you have learned today to parallelize the given serial version of the Jacobi-solver, as this is what you have do after this course on your own. Change to the directory ***6_jacobi***. There you will find a serial implementation of that iteration method. Use this implementation as a basis for your own parallel version.
In principle the algorithm works as follows:

a. Init u_old to zero
b. Compute an approximation of every u (a discretization point of your solution vector) based on the former approximation u_old
c. Swap u and u_old
d. Continue from b) until there is no significant change anymore

The figure on the top of this page describes some of the important variables of this implementation.
Note: Even though the number of data points in each direction is 7 one needs 9 points to include the boundary.

You may use the following steps as a guide-line:

What are the dependencies for the computation of a single discretization?
What communication method is suitable for computing the error?
How would you partition the domain? Is there a difference between Fortran and C?

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 (Oracle Grid Engine, formerly known as SGE). A batch script must be submitted using the qsub command (or alternatively using the qmon GUI which is not covered here). A batch script is basically a shell script containing all commands which must be executed, and all (recommended) necessary options for the batch system. It is also possible to provide all options as qsub command line parameters. Change to directory ***7_batch*.** An example batch script is the submit.sh together with the sample MPI-program sendRecv.c. Note, that the batch script can (and should!) be tested interactively before submitting. The interactive test lets you directly find eventual errors omitting the batch system waiting time. To let the batch file run interactively, it must be marked as “executable” by the command:

```
chmod 755 submit.sh
```

and then execute it by

```
./submit.sh
```

Note: the MPI_EXEC will only execute one process, as the environment variable FLAGS_MPI_BATCH will be defined by the batch system. 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:

```
qsub submit.sh
```

The batch system tells you the job ID of the submitted job:

```
Your job xxxxxx ("submit.sh") has been submitted
```

The status of all your jobs can be controlled by the qstat command. Only waiting jobs and jobs in an error state are shown; finished jobs are not listed.
You can cancel a job with the qdel command:

```
qdel xxxxxx
```

where xxxxxx is the job ID.

Compile and run this program to check your batch script.
Submit your script to the batch system and wait for its completion.

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