**Lecture 10-11 - MPI programming**

**Basic concepts**

**Launching MPI applications**

A MPI application can be launched via the mpirun command. The MPI application is launched on several specified nodes (hosts) and in a specified number of instances on each node.

Each instance starts the execution from the main() function and has the same command-line arguments (specified in the mpirun command). Each instance can retrieve (via MPI\_Comm\_size() and MPI\_Comm\_rank()) the number of instances launched (in the same mpirun command) and the own ID.

**The MPI setup**

The mpirun command normally uses ssh to connect to the remote nodes in order to launch there instances of the MPI application. For that purpose, it is best to prepare a [public key authentication setup for SSH](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/side-note-ssh-setup.html).

In addition, mpirun (actually, the remote daemon) must be able to find an executable with the same name and in the same location on each of the nodes. This can be achieved either by setting up a NFS or other networked sharing of files. It is enough, however, simply to copy the executable on all of the nodes before invoking mpirun.

The nodes can be of distinct architecture. In that case, the executable must, of course, be distinct: each node must have the executable of the appropriate architecture.

**Basic API**

An MPI program must call MPI\_Init() in the beginning and MPI\_Finalize() in the end.

Finding out the number of launched instances is done via MPI\_Comm\_size(), with MPI\_COMM\_WORLD as the identifier. Finding out one's own ID is done via MPI\_Comm\_rank().

Basic communication operations consist in sending and receiving an array of elements of the same type (array of integers, array of doubles, etc).

There are two kinds of communications:

* buffered communication, where the sender puts the data in a buffer and continues execution. If the receiver comes after the sender, then it takes the data from the buffer; if it comes before the sender, it has to wait for the sender.
* Synchronous communication, where the sender waits for the receiver to be ready. Whichever of the processes (the sender of the receiver) comes first, it is blocked until the other gets to the communication operation. Then, the communication is done and each process continues execution.

The receive operation is MPI\_Recv(). The send operations are: MPI\_Ssend() (synchronous), MPI\_Bsend() (buffered) and MPI\_Send() (unspecified, implementation-defined).

See the example [mpi1.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/mpi1.cpp), where a first process sends a number to a second one, the second adds 1 and sends the result to the third one, and so on, until the last process adds 1 and prints the result.

Other operations include:

* Broadcast — MPI\_Bcast() - see example [bcast-mpi.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/bcast-mpi.cpp)
* Launching a send or receive without waiting — MPI\_Isend() and MPI\_Irecv(). These calls return immediately; to actually wait for the completion of the operation, call MPI\_Wait() or MPI\_Waitany.
* Scatter-gather: MPI\_Scatter(), MPI\_Gather(), MPI\_Allgather()

**Other simple example**

Add all numbers in a vector. See solution [sum-mpi.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/sum-mpi.cpp):

* how to distribute input data;
* how each slave process gets its part as well as the meta-data (chunk size, etc)
* how the result gets gathered back at the root process

Another solution is given in [sum-scatter-mpi.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/sum-scatter-mpi.cpp). This one uses MPI\_Scatter() and MPI\_Gather() to distribute input data to workers and to gather the results.

**Algorithm design isses**

**Divide and conquer algorithms**

* Divide operations should stop when all the processes received work items;
* The parent process should not idle; so, when dividing the work, one part remains on the parent and only the other are sent to child processes (in the hierarchy).

Example: distributed merge-sorting:

* [mergesort-simplified-mpi.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/mergesort-simplified-mpi.cpp) (simplified version) or
* [mergesort-mpi.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/mergesort-mpi.cpp) (optimizing copying from one buffer to the other, and having children nodes deduce the parent, process pool size and vector size).
* [quicksort-mpi.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/quicksort-mpi.cpp) a quicksort implementation

**Interesting algorithms — Cannon's matrix multiplication**

Idea (consider, for simplicity, the product of two square matrices *N*x*N*, to be computed on *K*x*K* processors, where *K* divides *N*):

* divide all 3 matrices (the two inputs and the output) into *K*x*K* equal blocks;
* each process is responsible of computing one block;
* in the beginning, each process gets one block of each input matrix and computes their product;
* then, each process passes the block from the first matrix to the rigth and the block from the second matrix down, computes their product and adds it to the result block (actually, both are circular permutations);
* After *K* such circular permutations, each process has its result and the algorithm ends.

Note that the initial distribution of blocks must be done in such a way that each process has two blocks that it has to multiply together. This involves some circular permutations of rows or columns of blocks.

See the implementation in [matrix-mpi.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/matrix-mpi.cpp)