The MPI Message-passing Standard
Lab Time Hands-on

SPD Course
2018-2019
Massimo Coppola

UNIVERSITÀ DI PISA
ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

Remember!

- Simplest programs do not need much beyond Send and Recv, still…
- Each process lives in a separate memory space
  - Need to initialize all your data structures
  - Need to initialize your instance of the MPI library
  - Use MPI_COMM_WORLD
  - Need to define all your DataTypes
  - Should you make assumptions on process number?
  - How portable will your program be?
- Check your MPI man page about launching
  - E.g. **`mpirun –np 4 myprogram parameters`**

Initializing the runtime

- MPI_Init()
  - Shall be called before using any MPI calls (very few exceptions)
  - Initializes the MPI runtime for all processes in the running program, some kind of handshaking implied
    - e.g. creates **MPI_COMM_WORLD**
  - check its arguments!
- MPI_Finalize()
  - Frees all MPI resources and cleans up the MPI runtime, taking care of any operation pending
  - Any further call to MPI is forbidden
  - some runtime errors can be detected at finalize
    - e.g. calling finalize with communications still pending and unmatched

Note on mpich

- Mpich installation on Linux (centos 7) requires this in your .bash_profile

```
##### MPICH
export PATH=/usr/local/bin:/usr/lib64/mpich/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/lib:/usr/lib64/mpich/lib:$LD_LIBRARY_PATH
export MANPATH=/usr/share/man/mpich/:`manpath`
export PATH
```

- Mpirun becomes mpiexec, e.g.

```
mpiexec -np 2 ./pingpong "Hello world(s)"
```

  – explicit relative path to the executable

Do nothing (in parallel!) C example

```c
#include <stdio.h>
#include <mpi.h>
int main (int argc , char ** argv)   {
 int worldsize, myrank;
 MPI_Comm myComm;

 MPI_Init(&argc, &argv);

 MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

 MPI_Comm_dup(MPI_COMM_WORLD, &myComm);

 printf("Process %d of %d starting\n",
   myrank, worldsize);

 MPI_Finalize();
 return 0;
}
```

Post installation

- We rely on CLI-based compilation ad execution
- Assuming you install MPI with a package manager: binaries, headers and link-libraries will be ready for use
- You may need to add some dirs to your search path
- MPI program compilation and linking relies on your sequential compiler (GCC, LLVM …)
- MPI defines wrappers: **mpicc**, **mpicxx**, **mpif77**, **mpif90**
  - Each one call your sequential compiler passing options that enable MPI libraries and headers
- MPI provides a launcher: **mpirun** or **mpiexec** usually, plus variants (e.g. **mpirun_rsh**)
- Some implementations need a connection daemon to be active on each physical machine for MPI processes to be able to communicate (check your README files)
- The default configs of firewall and binary signing on your machine may need adjusting

Exercise 1

- Define the classical ping-pong program with 2 processes
  - they send back and fort a data buffer, the second process executes an operation on the data (e.g. sum 1).
  - Verify after a given number N of iterations, that the expected result is achieved.
  - Add printouts close to communications
  - Does it work? Why?
- Generalize the ping-pong example to N processes
  - Each process sends to the next one, with some processes being special, e.g.
  - Token ring (a process has to start and stop the token)
  - One-way pipeline (one process starts, one only receives)
  - Can you devise the proper communicator structure?

Getting your identity

- MPI_Comm_rank
  - After the MPI_Init
  - Returns the rank of the current process within a specified communicator
  - For now let's just use ranks related to MPI_COMM_WORLD
  - Example:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

Writing "structured" MPI

- We'll never stress this enough
  - Aim at separation of concern : avoid chaotically mixing up MPI primitives and sequential code
  - When possible, write a separate function/class for each type of process in your program
    - Parametric wrt to sequential program parameters and arguments, AND wrt parallel environment
    - E.g. Operates in a give communicator with known assumptions
    - Global initialization done by all processes, local initialization may be done locally (e.g. build a worker-specific communicator inside the farm implementation)
  - Sometimes it may be possible to write MPI code which is generic and may be reused → try to decouple these parts into separate functions

Exercise 2

- Build datatypes for
  - a square matrix of arbitrary element types and constant size 120*120
  - a column of the matrix
  - a row of the matrix
  - a group of 3 columns of the matrix
  - the upward and downward diagonals of the matrix

- Perform a test of the datatypes within the code of exercise 1
  - Initialize the matrix in a known way, perform computation on the part that you pass along (e.g. multiply or increment its elements) and check the result you receive back

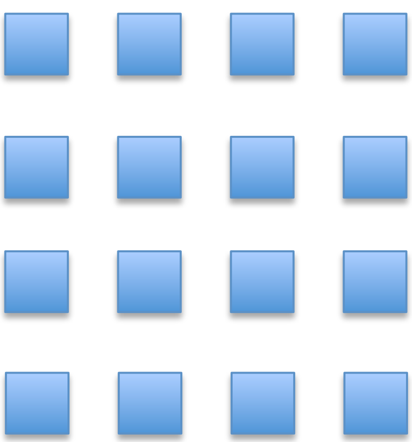
Derived Datatypes – don't forget

- MPI_TYPE_COMMIT(datatype)
  - Mandatory to enables a newly defined datatype for use in all other MPI primitives
  - Consolidates datatype definition, making it permanent
  - May compile internal information needed to the MPI library runtime
    - e.g. : optimized routines for data packing & unpacking
- MPI_TYPE_FREE(datatype)
  - Free library memory used by a datatype that is no longer needed

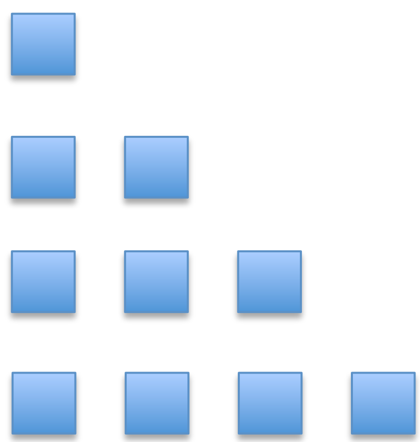
Exercise 3

- Define a datatype for a square matrix **with parametric size**
  - Define a datatype for its lower triagular matrix
  - Define one for its upper triangular.
- Test the them within the code of exercise 1

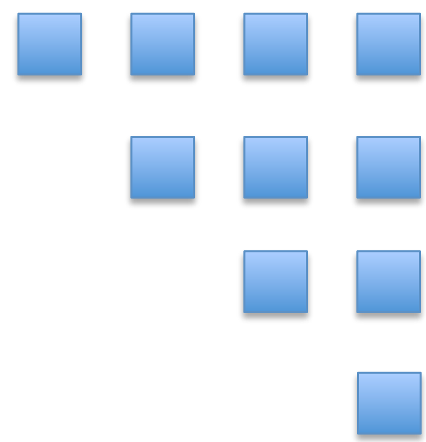
$A_{i,j}$ *i,j* in *1.. n*



$A_{i,j}$ $i \geq j$

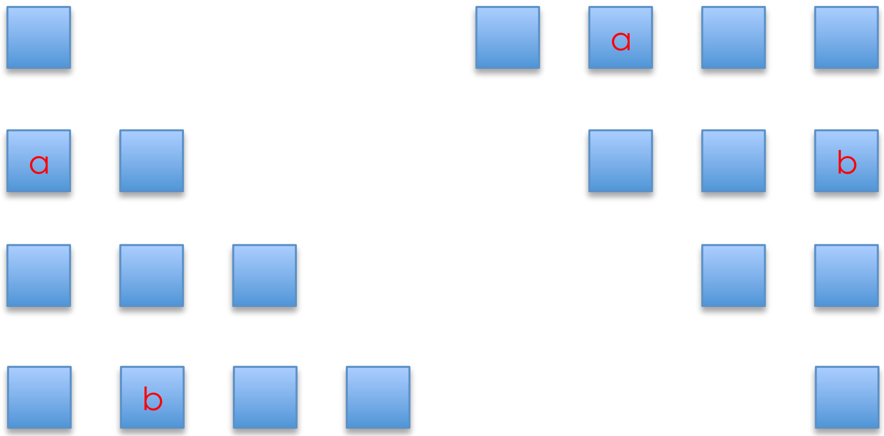


$A_{i,j}$ $i \leq j$



Exercise 3 (cont.)

- In the two-process program
  - initialize randomly a square matrix
  - send the lower triangular and
  - receive it back as upper triangular in the same buffer.
- Is the result a symmetric matrix?
  - How do you need to modify one of the two triangular datatypes in order to achieve that?
- In the end we want $A_{i,j} = B_{j,i}$



Exercise 4

- How do you implement an asynchronous communication with given asynchrony?
  - Implement a communication with asynchrony 1
  - Implement a communication with asynchrony K

- Assigned asynchrony of degree K: asynchronous communication (sender does not block) which becomes synchronous if more than K messages are still pending.
- Receiver can skip at most K receives before sender blocks

- Can you rely on MPI buffering?
- How would you implement a fixed size buffer?

Exercise 5

- Build a task farm skeleton program aiming at general reusability of MPI code
    - Should allow to change the data structures, computing functions and possibly load distribution policies without changing the MPI implementation code
  - Simplifying assumptions
    - single emitter and collector
    - stream generation and consumption are functions called within the emitter and collector processes
    - explicitly manage End-of-stream conditions via messages/tags
  - Separation of concerns
    - Each kind of process is a C function
    - Each computing task is a function called by the generic process
  - Different communication and load balancing strategies
    - Simple round-robin, explicit task request, degree of worker buffering
    - explicit task request, implicit request via Ssend,
  - What pros and cons in using separate communicators for the farm skeleton and its substructures?
    - Think of how you could implement some common extensions of the basic farm semantics: initial/periodic worker initialization, workers with status and status collection, work stealing strategies

Exercise 5 – example computation

- Mandelbrot set
- Compute the escape time (number of iterations before diverging) of the Z=Z^2+c complex sequence for any starting point c
  - c within the square (-2,-2) (2,2)
- Computation cannot be optimized, has rather high variance
- You can aggregate several points in a single task
  - Passing a square or a row of points to compute can be quite effective in the emitter, only needing two coordinates and the number of samples to take

http://en.wikibooks.org/wiki/Fractals/Iterations_in_the_complex_plane/Mandelbrot_set

```
int GiveEscapeTime(double C_x, double C_y, int iMax, 
double _ER2)
{   int i;
    double Zx, Zy;
    double Zx2, Zy2; /* Zx2=Zx*Zx;  Zy2=Zy*Zy  */
    Zx=0.0; /* initial value of orbit = critical point 
Z= 0 */
    Zy=0.0;
    Zx2=Zx*Zx;
    Zy2=Zy*Zy;

    for (i=0;i<iMax && ((Zx2+Zy2)<_ER2);i++)
    { Zy=2*Zx*Zy + C_y;
      Zx=Zx2-Zy2 +C_x;
      Zx2=Zx*Zx;
      Zy2=Zy*Zy;
    };
 return i;
}

/* Example of the worker function computing the escape 
time for a single point on the complex plane.
Here a sequence escapes if its squared modulo becomes 
greater than _ER2
_ER2 == 4 usually (modulo >= 2 implies divergence)

*/
```

Exercise 5 (cont.)

- Pitfalls and suggestions
  - Can you just change a communicator and plug the farm source code in a different program?
  - Stream management should never depend on knowing the stream length in advance
  - How do you add task grain management? Can you dynamically vary the grain?
    - Aggregation of a square or row of points in a single task is problem-specific → nice feat but it is not a general form of farm grain control
  - Can you model the execution time of the farm from a small execution and try to predict for a longer one? How do the grid resolution and iteration parameters, as well as choices about communication and load balancing affect the prototype?

Exercise 6

- Add to the farm skeleton a mechanism to reinitialize the workers
  - The stream computation depends on the status; each part of the stream (substream) is associated with a specific status
  - Example: the status is the max number of iteration in Mandelbrot
    - You cannot just assume to send the status within the job (status updates may be sporadic and quite larger than ordinary tasks)
  - How do you send/receive status updates (ISSend, IBSend, Ssend versus non-determinism control in the worker receives)
  - Should you serialize the communications and how? (adding a progressive identifier to the task, the status messages or both, and how to link them)
  - Manage substream ordering in the emitter (chose semantics: no ordering, reordering the results by the tasks, reordering the result by substreams but not by the tasks