Università della Svizzera italiana

Institute of Computational Science ICS

Introduction to the Message Passing Interface (MPI)

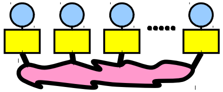
Prof. Olaf Schenk

(TA: Juraj Kardos, Radim Janalik, Aryan Eftekhari)

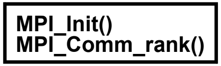
November 05, 2019

Outline

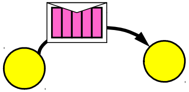
1. MPI Overview
   - one program on several processors work and data distribution



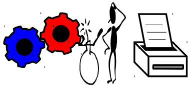
2. Process model and language bindings
   - starting several MPI processes


MPI_Init()
MPI_Comm_rank()

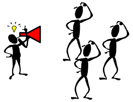
3. Messages and point-to-point communication
   - the MPI processes can communicate



4. Non-blocking communication
   - to avoid idle time and deadlocks



5. Collective communication
   - e.g., broadcast



Deploying in-class exercises on **ICSMaster**

1. C program compilation (pi0_example.zip):

```
$ module load openmpi
$ mpicc -O3 pi0.c -o pi0
```

2. Create a job configuration file to execute the program using **4** processes on **2** nodes for **30** min max:

```
$ cat prun
#!/bin/bash -l
#SBATCH --job-name=hpc-class              # job name
#SBATCH --time 00:30:00                   # wall-clock time (hrs:mins::secs)
#SBATCH --nodes=2                         # number of compute nodes
#SBATCH --ntasks=4                        # number of total MPI tasks
#SBATCH --error=job.%J.err                # error file name
#SBATCH --output=job.%J.out               # output file name
# load modules
module load gcc
module load openmpi
# your commands
mpirun $@
```

Deploying in-class exercises on **ICSMaster**

1. Submit the job for execution, passing the program name and arguments:

```
sbatch prun ./pi0 10000000000
```

2. Program output shall appear in file job.XXXXX.out (live updating)
3. Error output shall appear in file job.XXXXX.err (live updating)

Deploying in-class exercises on **your Mac**

1. Install MPI from ports:

```
$ sudo port install openmpi
```

2. Install MPI from Homebrew:

```
$ brew install open-mpi
```

3. C program compilation - same as on ICSMaster:

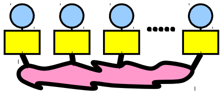
```
$ mpicc -std=c99 -O3 pi0.c -o pi0
```

4. There is no MPI job queue on your machine $\Rightarrow$ MPI programs are executed right away:

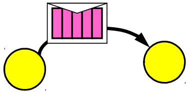
```
$ mpirun -np 8 ./pi0 10000000000
```

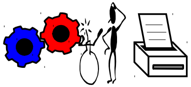
5. Program output shall appear in the owning terminal

Outline



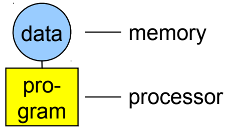
MPI_Init()
MPI_Comm_rank()



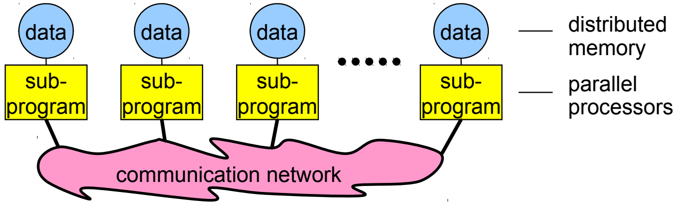


The Message-Passing Programming Paradigm

- Sequential Programming Paradigm

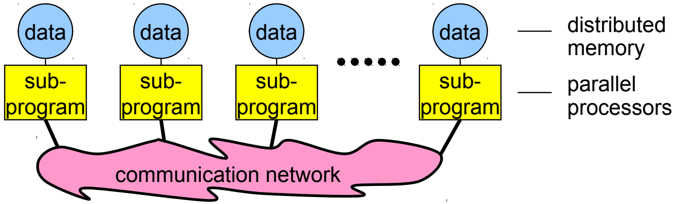


- Message-Passing Programming Paradigm



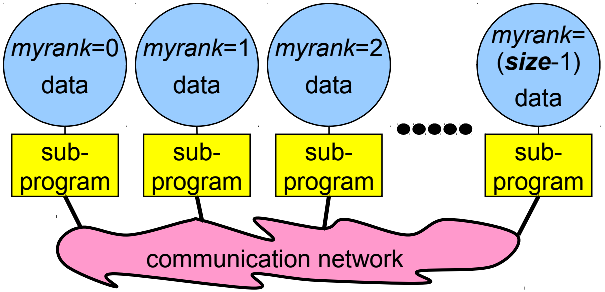
The Message-Passing Programming Paradigm

- Each processor in a message passing program runs a sub-program:
  - written in a conventional sequential language, e.g., C or C++,
  - typically the same on each processor (SPMD),
  - the variables of each sub-program have
    - the same name
    - but different locations (distributed memory) and different data!
    - i.e., all variables are private
  - communicate via special send & receive routines (message passing)



Data and Work Distribution

- the value of **myrank** is returned by special library routine
- the system of size processes is started by special MPI initialization program (mpirun or mpiexec)
- all distribution decisions are based on **myrank**
- i.e., which process works on which data



What is SPMD?

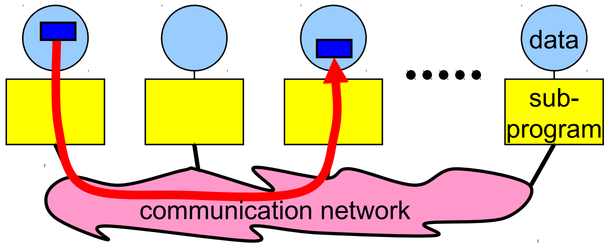
- Single Program, Multiple Data
- Same (sub-)program runs on each processor

- MPI allows also MPMD, i.e., Multiple Program, ...
- but some vendors may be restricted to SPMD
- MPMD can be emulated with SPMD

- Emulation of Multiple Program (MPMD), Example:

```
int main(int argc, char **argv)
{
  ...
  if (myrank < .... /* process should run the ocean model */)
  {
    ocean( /* arguments */ );
  }else{
    weather( /* arguments */ );
  }
  ...
}
```

Messages



- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:

- sending process rank	- receiving process rank
- source location	- destination location
- source data type	- destination data type
- source data size	- destination buffer size

Access

- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
  - mail box
  - phone line
  - fax machine
  - etc.
- MPI:
  - sub-program must be linked with an MPI library
  - sub-program must use include file of this MPI library
  - the total program (i.e., all sub-programs of the program) must be started with the MPI startup tool

Addressing

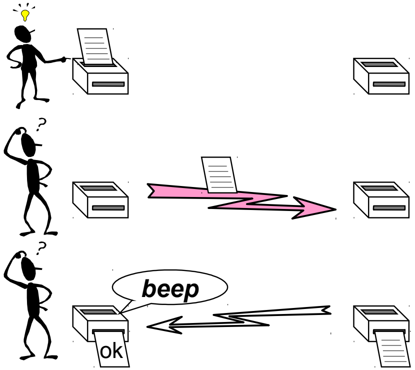
- Messages need to have addresses to be sent to
- Addresses are similar to:
  - mail addresses
  - phone number
  - fax number
  - etc.
- MPI: addresses are ranks of the MPI processes (sub-programs)

Point-to-Point Communication

- Simplest form of message passing
- One process sends a message to another
- Different types of point-to-point communication:
  - synchronous send
  - buffered = asynchronous send

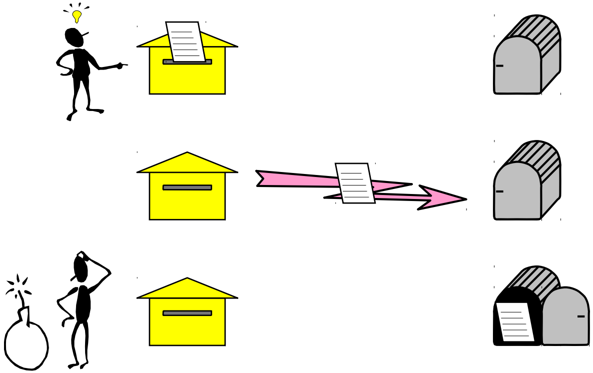
Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the beep or okay-sheet of a fax.



Buffered = Asynchronous Sends

- Only know when the message has left

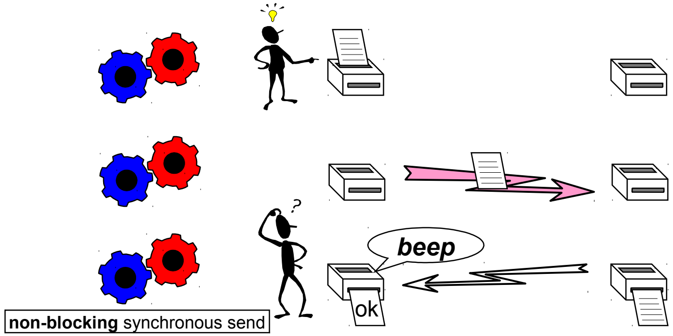


Blocking Operations

- Operations are local activities, e.g.,
  - sending (a message)
  - receiving (a message)
- Some operations may block until another process acts:
  - synchronous send operation blocks until receive is posted;
  - receive operation blocks until message was sent.
- Relates to the completion of an operation.
- Blocking subroutine returns only when the operation has completed.

Non-Blocking Operations

- Non-blocking operation: returns immediately and allow the sub-program to perform other work.
- At some later time the sub-program must **test** or **wait** for the completion of the non-blocking operation.



Non-Blocking Operations (cont‘d)

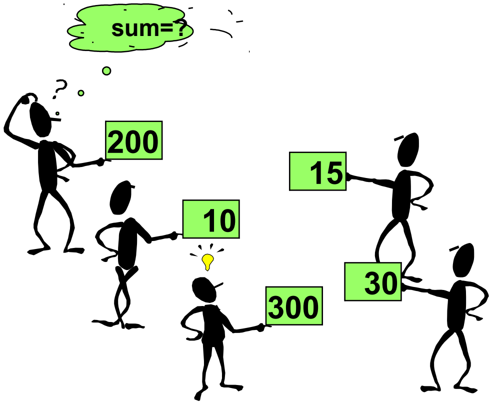
- All non-blocking operations must have matching wait (or test) operations. (Some system or application resources can be freed only when the non-blocking operation is completed.)
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation.
- Non-blocking operations are not the same as sequential subroutine calls: the operation may continue while the application executes the next statements!

Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow optimized internal implementations, e.g., tree based algorithms
- Can be built out of point-to-point communications.

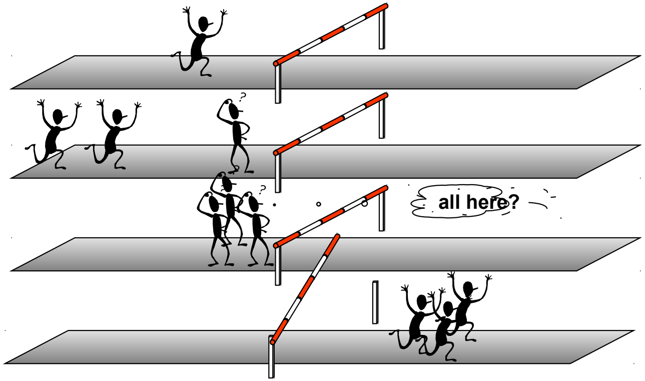
Reduction Operations

- Combine data from several processes to produce a single result.



Barriers

- Synchronize processes.



MPI Forum

- **MPI-1 Forum**
  - First message-passing interface standard.
  - Sixty people from forty different organizations.
  - Users and vendors represented, from US and Europe.
  - Two-year process of proposals, meetings and review.
  - Message-Passing Interface document produced.
  - MPI-1.0 — June,1994.
  - MPI-1.1 — June 12, 1995.

MPI-2 and MPI-3 Forum

- **MPI-2 Forum**
  - Same procedure (e-mails, and meetings in Chicago, every 6 weeks).
  - MPI-2: Extensions to the Message-Passing Interface (July 18, 1997).

  containing:
  - MPI-1.2 — mainly clarifications.
  - MPI-2.0 — extensions to MPI-1.2.

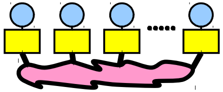
- **MPI-3 Forum**
  - Started Jan. 14-16, 2008 (1st meeting in Chicago)
  - Using e-mails, wiki, meetings every 8 weeks (Chicago and San Francisco), and telephone conferences
  - MPI-2.1 — June 23, 2008, mainly combining MPI-1 and MPI-2 books to one book
  - MPI-2.2 — September 4, 2009: Clarifications and a few new func.
  - MPI-3.0 — September 21, 2012: Important new functionality

Goals and Scope of MPI

- **MPI‘s primary goals**
  - To provide a message-passing interface.
  - To provide source-code portability.
  - To allow efficient implementations.
- **It also offers:**
  - A great deal of functionality.
  - Support for heterogeneous parallel architectures.
- **With MPI-2:**
  - Important additional functionality.
  - No changes to MPI-1.
- **With MPI-2.1, 2.2, 3.0:**
  - Important additional functionality to fit on new hardware principles.
  - Deprecated MPI routines moved to chapter “Deprecated Functions”
  - With MPI-3.0, some deprecated features were removed

Outline

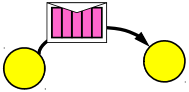
1 MPI Overview
- one program on several processors work and data distribution



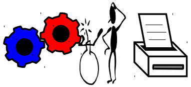
2 Process model and language bindings
- **starting several MPI processes**

MPI_Init()
MPI_Comm_rank()

3 Messages and point-to-point communication
- the MPI processes can communicate



4 Non-blocking communication
- to avoid idle time and deadlocks



5 Collective communication
- e.g., broadcast

MPI Header files and MPI Function Format

```
#include <mpi.h>
error = MPI_Xxxxxx( parameter, ... );
MPI_Xxxxxx( parameter, ... );
```

definition in the standard:

```
MPI_Comm_rank(..., int *rank);
MPI_Recv(..., MPI_Status *status);
```

usage in your code:

```
main...
{
    int myrank;
    MPI_Status rcv_status;
    MPI_Comm_rank(..., &myrank);
    MPI_Recv(..., &rcv_status);
}
```

- MPI_...... namespace is reserved for MPI constants and routines, i.e. application routines and variable names must not begin with MPI_.

Initializing MPI

```
int MPI_Init( int *argc, char ***argv); // C
```

```
#include <mpi.h>
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    ...
}
```

- In MPI-2.0 and higher could also be: `MPI_Init(NULL, NULL);`
- Must be first MPI routine that is called
  (only a few exceptions, e.g., `MPI_Initialized`)

Starting the MPI Program

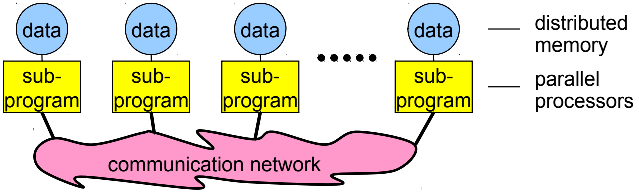
- Start mechanism is implementation-dependent:
  - Most implementations:

```
$ mpirun -np number_of_processes ./executable
```

  - With MPI-2 and later:

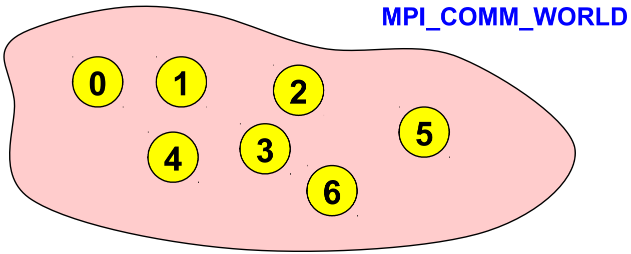
```
$ mpiexec -n number_of_processes ./executable
```

- The parallel MPI processes exist at least after `MPI_Init` was called.



Communicator MPI_COMM_WORLD

- All processes (= sub-programs) of one MPI program are combined in the **communicator** MPI_COMM_WORLD.
- MPI_COMM_WORLD is a predefined **handle** in **mpi.h**.
- Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)



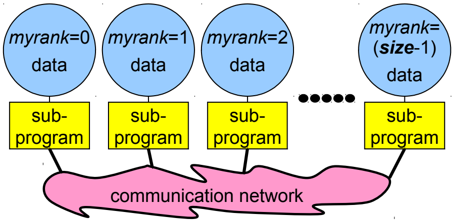
Handles

- Handles identify **MPI** objects.
- For the programmer, handles are
  - predefined constants in mpi.h
    - Example: `MPI_COMM_WORLD`
    - Can be used in initialization expressions or assignments.
    - The object accessed by the predefined constant handle exists and does not change only between `MPI_Init` and `MPI_Finalize`.
  - values returned by some MPI routines, to be stored in variables, that are defined as
    - in C: special MPI typedefs, e.g., `MPI_Comm`
- Handles refer to internal MPI data structures

Rank

- The rank identifies different processes.
- The rank is the basis for any work and data distribution.
- C:

```
int rank;
int MPI_Comm_rank( MPI_Comm comm, int *rank);
```



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

Size / Exiting MPI

- **How many processes are contained within a communicator?**
  - C:

```
int size;
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- **Exiting MPI**
  - C:

```
int MPI_Finalize();
```

  - Must be called last by all processes.
  - After `MPI_Finalize`:
    - Further MPI-calls are forbidden
    - Especially re-initialization with `MPI_Init` is forbidden

In Class Exercise: Hello World

- Write a minimal MPI program which prints “hello world” by each MPI process.
- Compile and run it on a single processor.
- Run it on several processors in parallel.
- Modify your program so that:
  - every process writes its rank and the size of `MPI_COMM_WORLD`,
  - only process ranked 0 in `MPI_COMM_WORLD` prints “hello world”.
- Why is the sequence of the output non-deterministic?

```
I am 2 of 4
Hello world
I am 0 of 4
I am 3 of 4
I am 1 of 4
```

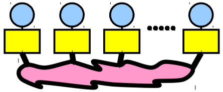
In Class Exercise: Hello World (Solution)

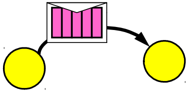
Advanced exercise: Hello World with deterministic output

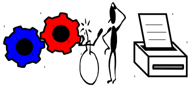
1. Discuss with your neighbor, what must be done, that the output of all MPI processes on the terminal window is in the sequence of the ranks.
2. Or is there no chance to guarantee this?

In Class Exercise: Hello World (Solution)

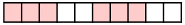
Outline







Messages

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
  - Basic datatype (see next slide).
  - Derived datatypes. 
- Derived datatypes can be built up from basic or derived datatypes.
- Datatype handles are used to describe the type of the data in the memory.

Example: message with 5 integers

2345	**654**	**96574**	**-12**	**7676**

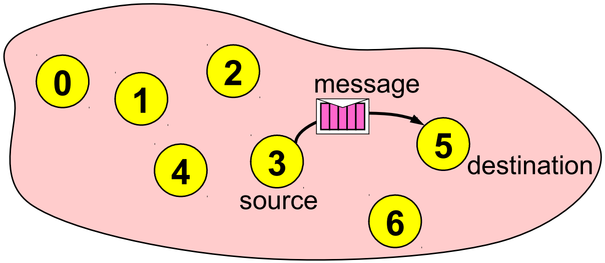
MPI Basic Datatypes – C

MPI Datatype	C datatype	Remarks
MPI_CHAR	char	Treated as printable character
MPI_SHORT	signed short int	
MPI_INT	signed int	
MPI_LONG	signed long int	
MPI_LONG_LONG	signed long long	
MPI_SIGNED_CHAR	signed char	Treated as integral value
MPI_UNSIGNED_CHAR	unsigned char	Treated as integral value
MPI_UNSIGNED_SHORT	unsigned short int	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_UNSIGNED_LONG_LONG	unsigned long long	
MPI_FLOAT	float	
MPI_DOUBLE	double	
MPI_LONG_DOUBLE	long double	
MPI_BYTE		
MPI_PACKED		

Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator, e.g., `MPI_COMM_WORLD`.
- Processes are identified by their ranks in the communicator.

communicator:



Sending a Message: MPI_Send

- C:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm);
```

- buf is the starting point of the message with count elements, each described with datatype.
- dest is the rank of the destination process within the communicator comm.
- tag is an additional non-negative integer piggyback information, additionally transferred with the message.
- The tag can be used by the program to distinguish different types of messages.

Sending a Message: MPI_Ssend

- C:

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm);
```

- buf is the starting point of the message with count elements, each described with datatype.
- dest is the rank of the destination process within the communicator comm.
- tag is an additional non-negative integer piggyback information, additionally transferred with the message.
- The tag can be used by the program to distinguish different types of messages.

Receiving a Message: MPI_Recv

- C:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status);
```

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.
- Envelope information is returned in status.
- status is output argument (highlighted).
- Only messages with matching tag are received.

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message datatypes must match.
- Receiver's buffer must be large enough.

Wildcarding

- Receiver can wildcard.
- To receive from any source – source = `MPI_ANY_SOURCE`
- To receive from any tag – tag = `MPI_ANY_TAG`
- Actual source and tag are returned in the receiver's status parameter.

Communication Modes

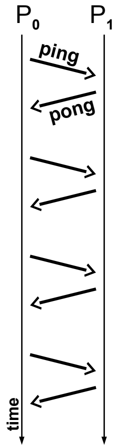
- Send communication modes:
  - synchronous: `MPI_SSEND`
    – Only completes when the receive has started
  - standard: `MPI_SEND`
    – Tries to complete when the send buffer is available for reuse (copies into an internal buffer). May fall back to `MPI_SSEND`, if internal buffer is not large enough for the message.
- Receiving all modes: `MPI_RECV`
  – Completes when a message has arrived (no `SRECV` mode)

In Class Exercise: Ping-pong

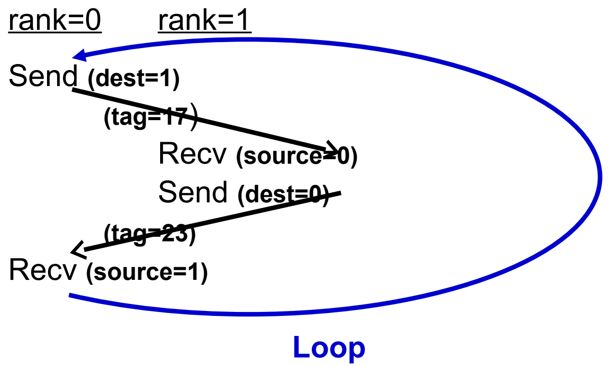
- Write a program according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message, process 1 sends a message back to process 0 (pong)
- Repeat this ping-pong with a loop of length 50
- Add timing calls before and after the loop:

```
double MPI_Wtime(void);
```

- Only at process 0: print out the transfer time of **one** message in μs, i.e. $\Delta_{time}/(2 \times 50) \times 10^6$



In Class Exercise: Ping-pong



```
if (my_rank == 0)
{
    MPI_Send( ... dest=1 ...);
    MPI_Recv( ... source=1 ...);
}
else
{
    MPI_Recv( ... source=0 ...);
    MPI_Send( ... dest=0 ...);
}
```

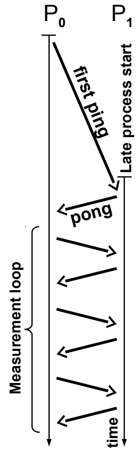
In Class Exercise: Ping-Pong (Solution)

- show template pingpong_template/pingpong.c
- show solution pingpong/pingpong.c

Advanced Exercises – Ping pong latency and bandwidth

- Exclude startup time problems from measurements:
  - Execute a first ping-pong outside of the measurement loop
- latency = transfer time for short messages
- bandwidth = message size (in bytes) / transfer time
- Print out message transfer time and bandwidth
  - for following send modes:
    - for standard send (MPI_Send)
    - for synchronous send (MPI_Ssend)
  - for following message sizes:

```
  8  B (e.g., one double or double precision value)
512  B (= 8*64 bytes)
 32 KB (= 8*64**2 bytes)
  2 MB (= 8*64**3 bytes)
```



In Class Exercise: Ping-Pong (Solution)

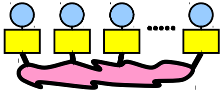
- show solution pingpong/pingpong_advanced_send.cpp (`MPI_Send`)
- show solution pingpong/pingpong_advanced_ssend.cpp (`MPI_Ssend`)

Results for ping-pong on ICSMaster (2 processes, 2 nodes)

2 bytes	0.000022 sec	1.485531 MB/s
2048 bytes	0.000018 sec	111.906391 MB/s
131072 bytes	0.000093 sec	1413.472037 MB/s
8388608 bytes	0.002661 sec	3152.936229 MB/s

Outline

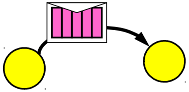
1 MPI Overview
- one program on several processors work and data distribution



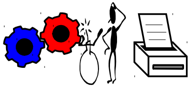
2 Process model and language bindings
- starting several MPI processes

```
MPI_Init()
MPI_Comm_rank()
```

3 Messages and point-to-point communication
- the MPI processes can communicate



4 **Non-blocking communication**
- **to avoid idle time and deadlocks**



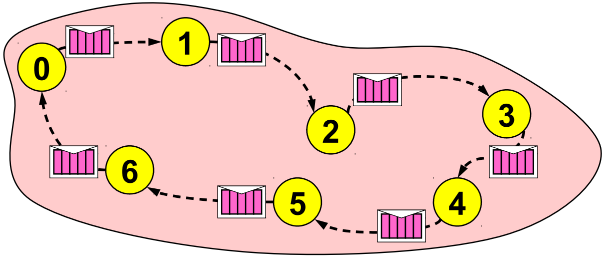
5 Collective communication
- e.g., broadcast

Deadlock in a process ring

- The following code in each MPI process:

```
MPI_Ssend(..., right_rank, ...);
MPI_Recv(..., left_rank, ...);
```

will block and never return, because `MPI_Recv` cannot be called in the right-hand MPI process



- Same problem with standard send mode (`MPI_Send`), if MPI implementation chooses synchronous protocol
- Solution:
  - `MPI_Sendrecv`, or
  - nonblocking Send and Receive

MPI_Sendrecv

- MPI_Sendrecv is just like an MPI_Send followed by an MPI_Recv, except that it's much better than that.
- With MPI_Send and MPI_Recv, these are your choices:
  - Everyone calls MPI_Recv, and then everyone calls MPI_Send.
  - Everyone calls MPI_Send, and then everyone calls MPI_Recv.

Why `MPI_Sendrecv`?

- Suppose that everyone calls `MPI_Recv`, and then everyone calls `MPI_Send`.
  - Well, these routines are synchronous (also called blocking), meaning that the communication has to complete before the process can continue with the program.
  - That means that, when everyone calls `MPI_Recv`, they're waiting for someone else to call `MPI_Send`. We call this deadlock.
- Suppose that everyone calls `MPI_Send`, and then everyone calls `MPI_Recv`.
  - Well, this will only work if there's enough buffer space available to hold everyone's messages until after everyone is done sending.
  - Sometimes, there isn't enough buffer space.

Why MPI_Sendrecv?

- MPI_Sendrecv allows each processor to simultaneously send to one processor and receive from another. For example, P_1 could send to P_0 while simultaneously receiving from P_2.

```
#include <mpi.h>

int MPI_Sendrecv(void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source,
                 int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status);
```

MPI_Sendrecv_replace

- MPI_Sendrecv_replace sends and receives using a single buffer:

```
#include <mpi.h>

int MPI_Sendrecv_replace(void *buf,
                         int count,
                         MPI_Datatype datatype,
                         int dest,
                         int sendtag,
                         int source,
                         int recvtag,
                         MPI_Comm comm,
                         MPI_Status *status);
```

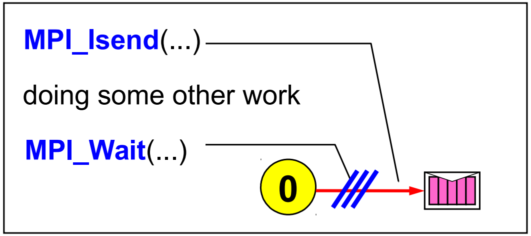
Non-Blocking Communications

Separate communication into three phases:

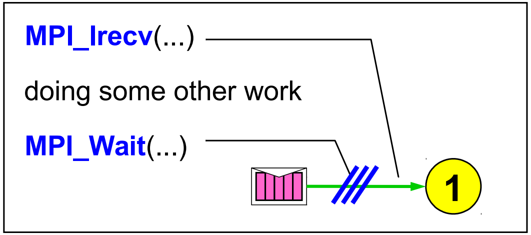
1. Initiate non-blocking communication
   - returns **I**mmediately
   - routine name starting with `MPI_I...`
2. Do some work (perhaps involving other communications?)
3. Wait for non-blocking communication to complete

Non-Blocking Examples

- Non-blocking **send**:



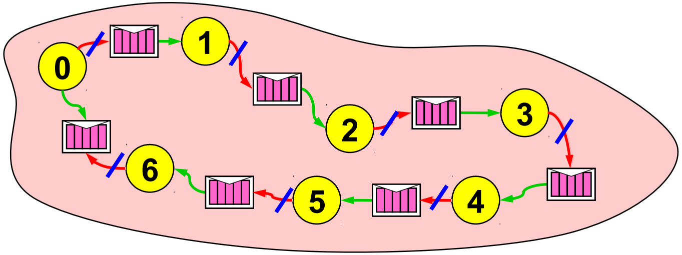
- Non-blocking **receive**:



/// – waiting until operation locally completed

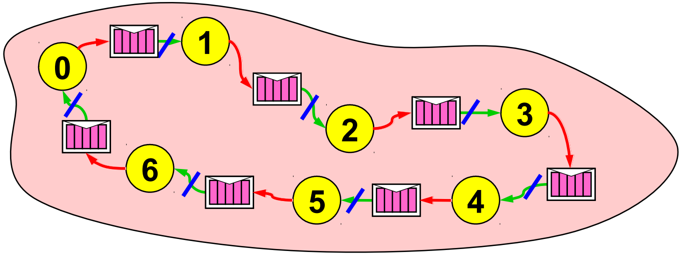
Non-Blocking Send

- Initiate non-blocking send
  - 🟥 in the ring example: Initiate non-blocking send to the right neighbor
- Do some work:
  - 🟩 in the ring example: Receiving the message from left neighbor
- Now, the message transfer can be completed
- Wait for non-blocking send to complete: 🟦



Non-Blocking Receive

- Initiate non-blocking receive
  - 🟥 in the ring example: Initiate non-blocking receive from left neighbor
- Do some work:
  - 🟩 in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for non-blocking receive to complete: 🟦



Handles and `MPI_Request`

Predefined handles

- defined in `mpi.h`
- communicator, e.g., `MPI_COMM_WORLD`
- datatype, e.g., `MPI_INT`, `MPI_INTEGER`, ...

Handles **can** also be stored in local variables:

- memory for datatype handle: `MPI_Datatype name_of_var`
- memory for communicator handles: `MPI_Comm name_of_var`

Request handles

- are used for **non-blocking communication**
- **must** be stored in local variables: `MPI_Request name_of_var`
- the value
  - **is generated** by a non-blocking communication routine
  - **is used** (and freed) in the `MPI_WAIT` routine

Non-blocking Synchronous Send / Non-blocking Receive

- C:

```
MPI_Issend(buf, count, datatype, dest, tag, comm, OUT &request_handle);
..
MPI_Wait(INOUT &request_handle, &status);
```

- buf must not be modified between `MPI_Issend` and `MPI_Wait`

- C:

```
MPI_Irecv(buf, count, datatype, source, tag, comm, OUT &request_handle);
...
MPI_Wait(INOUT &request_handle, &status);
```

- buf must not be used between `MPI_Irecv` and `MPI_Wait`

Completion

- C:

```
MPI_Wait(&request_handle, &status);
MPI_Test(&request_handle, &flag, &status);
```

- one must
  - WAIT or
  - loop with TEST until request is completed, i.e., `flag == 1` (C) or `.TRUE.` (Fortran)

You have several request handles:

- Wait or test for completion of **one** message: `MPI_Waitany` / `MPI_Testany`
- Wait or test for completion of **all** messages: `MPI_Waitall` / `MPI_Testall`

Exercise – MPI Ring

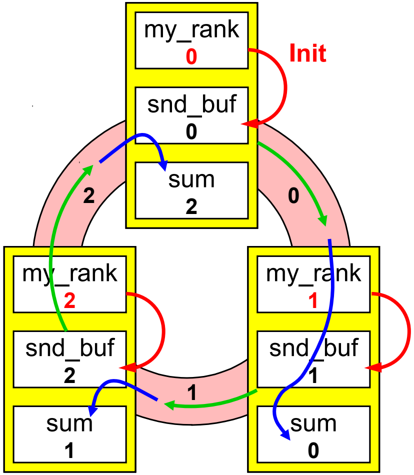
- A set of MPI processes are arranged in a ring.
- The program shall rotate information around a ring, each process calculates sum of all ranks

Algorithm:

1. Each process stores its rank in `MPI_COMM_WORLD` into an integer variable `snd_buf`.
2. Each process passes this on to its neighbor on the right.
3. Each processor calculates the sum of all values.
4. Repeat **2-4** with "size" iterations (size = number of processes), i.e. **each process calculates sum of all ranks.**

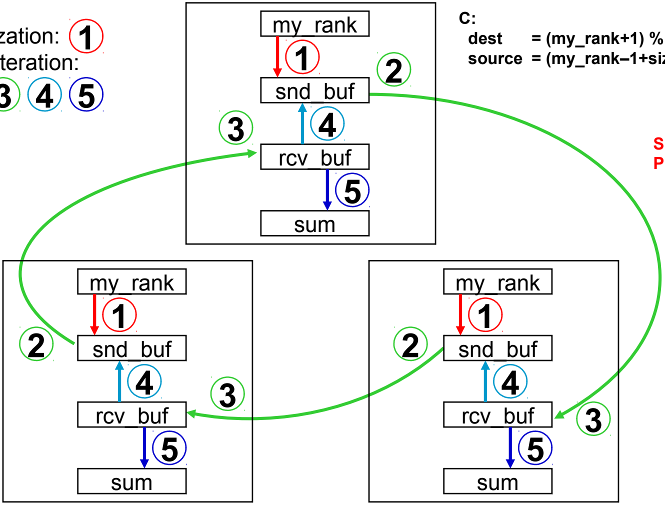
Use non-blocking `MPI_Issend`:

- to avoid deadlocks
- to verify the correctness, because blocking synchronous send will cause a deadlock



Exercise – Rotating information around a ring

Initialization: 1
Each iteration:
2 3 4 5



C:

```
dest    = (my_rank+1) % size;
source  = (my_rank–1+size) % size;
```

Single Program !!!

In Class Exercise: Ring (Solution)

- show solution pingpong/ring.c

Advanced Exercises – Irecv instead of Issend

- Substitute the `Issend-Recv-Wait` method by the `Irecv-Ssend-Wait` method in your ring program.

Or

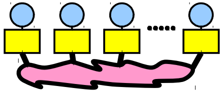
- Substitute the `Issend-Recv-Wait` method by the `Irecv-Issend-Waitall` method in your ring program.

In Class Exercise: Ring (Solution)

- show solution pingpong/ring_advanced_irecv_issend.c
- show solution pingpong/ring_advanced_irecv_ssend.c

Outline

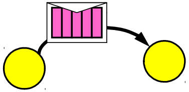
1 MPI Overview
- one program on several processors work and data distribution



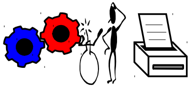
2 Process model and language bindings
- starting several MPI processes

MPI_Init()
MPI_Comm_rank()

3 Messages and point-to-point communication
- the MPI processes can communicate

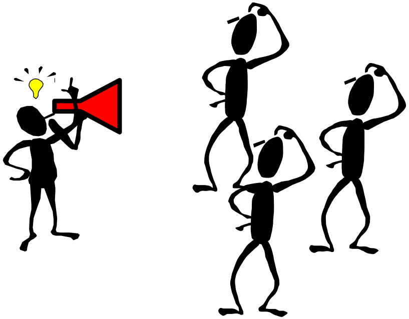


4 Non-blocking communication
- to avoid idle time and deadlocks



5 **Collective communication**
- **e.g., broadcast**

Broadcast and Collective Communication



- A one-to-many communication
- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

Broadcast and Collective Communication

- Use slides on collective communications (second PDF on icorsi)