Numerical Simulation and Scientific Computing II

Lecture 1: Introduction and Distributed Parallel Computing I

Josef Weinbub, Paul Manstetten, Heinz Pettermann, Asur Vijaya Kumar, Pavan Kumar, Jesús Carrete Montana, Francesco Zonta, Kevin Sturm

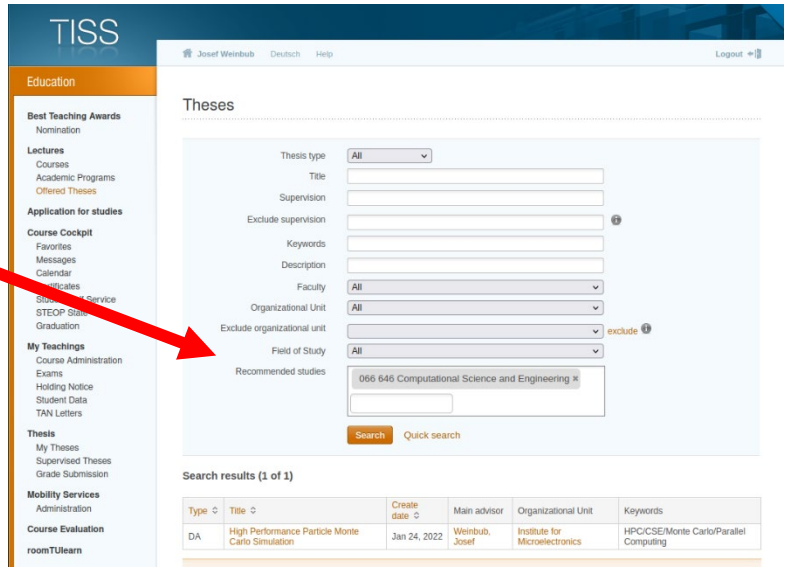
Institute for Microelectronics
TU Wien

nssc@iue.tuwien.ac.at

SS 2023

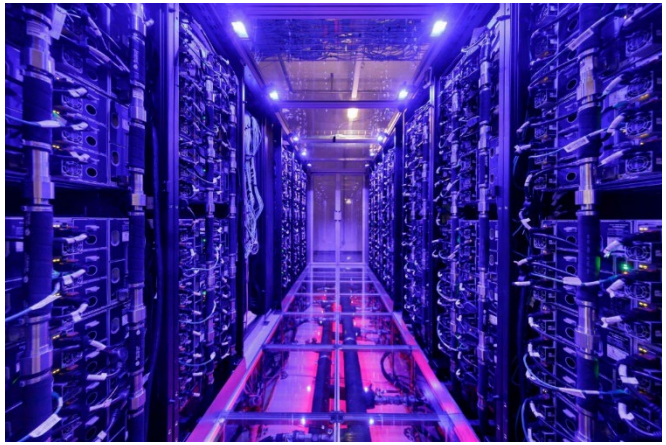
Master's Thesis Topics: Quantum Monte Carlo Simulator on Supercomputers

- **Development of new highly parallel C++ Monte Carlo Simulator**
- **Several parallelization layers: MPI/OpenMP/CUDA**
- **Tailored to Supercomputers**



Possible Tasks

- **Switchable OpenMP/CUDA kernels**
- **MPI domain decomposition**
- **Structure generator**
- **Performance studies: Algorithms, data structures ..**
- **Automated testing pipelines**
- **and many more ..**



Required knowledge: (at least one of the following)

- **C++ / MPI / OpenMP / CUDA**

Contact: josef.weinbub@tuwien.ac.at

Outline

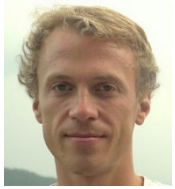
- **Introduction to the Lecture**
- Distributed Parallel Computing I
  - Primer
  - Overview
  - Process Model and Language Bindings
  - Messages and Point-to-Point Communication
  - Examples
- Quiz

Team Presentation

Core Team (Organization, MPI, ODE, FVM)

- **Josef Weinbub**

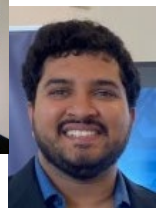
- **Paul Manstetten**
  Institute for Microelectronics



Extended Interdisciplinary Team

- **Francesco Zonta** (Fluid Dynamics)
  Institute of Fluid Mechanics and Heat Transfer

- **Heinz Pettermann,**
  Asur Vijaya Kumar, Pavan Kumar (Finite Elements)
  Institute of Lightweight Design and Structural Biomechanics



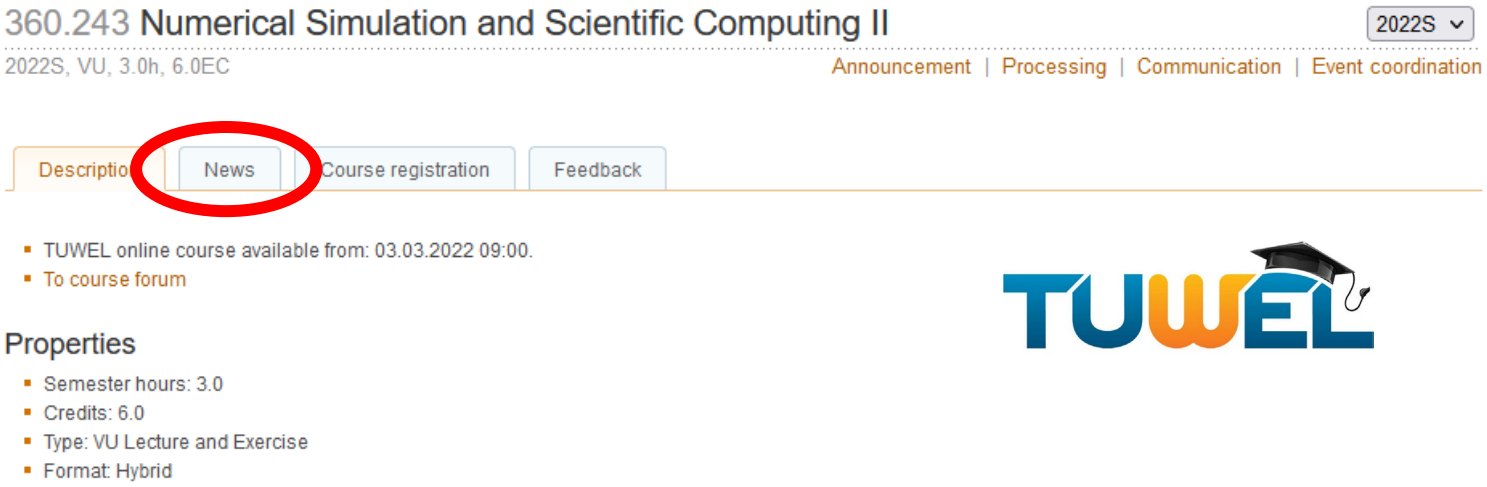
- **Jesús Carrete Montana** (Molecular Dynamics)
  Institute of Materials Chemistry

- **Kevin Sturm** (Partial Differential Equations)
  Institute of Analysis and Scientific Computing

nssc@iue.tuwien.ac.at

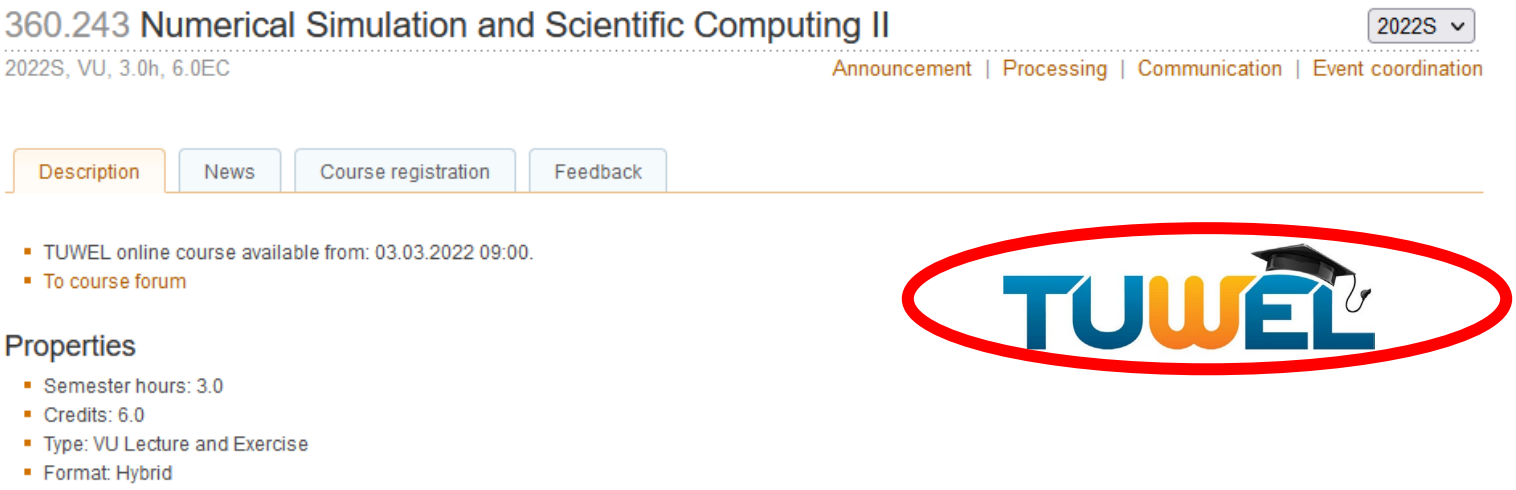
TISS

- **Acceptance to course via TISS: Check today**
- **News will be sent via TISS' notification feature: Regularly monitor your TISS News (at least weekly)!**



TUWEL

- **Course material (slides, handouts, zoom link) provided via TUWEL course**
- **TUWEL course “reachable” via TISS course, watch out for the TUWEL icon:**



General Goals

- **Introduction to advanced methods of CSE**
  - **Distributed Parallel Computing**
  - **Methods for Ordinary Differential Equations**
  - **Classification and Analyses of Partial Differential Equations**
  - **Finite Volume Method**
  - **Finite Element Method**
  - **Fluid Dynamics**
  - **Molecular Dynamics**

Necessary Background

- **C++ and Python! (see TISS)**
  - **If not experienced: Checkout online tutorials asap!**
- **Not a formal requirement**
  yet the expected level of expertise:
  Numerical Simulation and Scientific Computing I
  360.242

Exercise Rules and Course Grade

- 4 mandatory exercises over the whole lecture
- 7 Groups of 3 students and 1 group of 2 students must be formed; student-picked.
  - Go find your group today and inform: nssc@iue.tuwien.ac.at
  - Mind the deadline! (see Course Calendar, slide 10)
  - No changes during the term!
- Code copied from other group: 0 points for both groups!
  - If copied from previous year: also 0 points.
- Each exercise will be graded separately from 0-10 points
- Access for the final exam: sum of the points ≥ 28.0 (same procedure as with NSSC I)
- Course grade: 1/4th Exercises, 3/4th Exam
- Submission details will be communicated with handout

Rules - Quizzes

- In the end of each lecture, you will receive 5 questions
- 3 questions reviewing the current lecture
- 2 questions preparing for the next
- Discussion in the beginning of next class
- Participation is voluntary but encouraged
- These questions might help you learn for the exam!

Course Calendar

March 2	Lecture
March 9	Lecture and Exercise Handout **Send group info to** nssc@iue.tuwien.ac.at
March 16	Lecture and Exercise Support
March 22	(Wednesday!) Lecture (zoom)
March 23	Exercise Support
March 30	Lecture and Exercise Submission
April 21	(Friday! Start at 10:00) Lecture and Exercise Handout (zoom)
April 27	Lecture and Exercise Support
May 4	Lecture and Exercise Submission/Handout
May 11	Lecture and Exercise Support
May 17	(Wednesday!) Lecture and Exercise Submission/Handout
May 25	Exercise Support
June 1	Exercise Support
June 7	(Wednesday!) Exercise Submission
June 15	Backup
June 22	Backup
June 29	Main Exam

- **If not otherwise states, all units start at 14:00 sharp**
- **Exercise submission is done via TUWEL/Email (see exercise handout): No need to show up on submission-only units (e.g., June 7)**
- **Will put lecture slides/exercise material on TUWEL right before slot**
- **Zoom infos in TUWEL**
- **Detailed Schedule also in TISS**

Computer Resources

- It is expected that you have access to a modern, x86-based, personal computer (laptop, home desktop, etc.)
- Linux (virtual machine, dual boot or full OS) is required
  - More information on the development environment for a particular exercise will be made available with the exercise handout

Outline

- Introduction to the Lecture
- **Distributed Parallel Computing I**
  - **Primer**
  - Overview
  - Process Model and Language Bindings
  - Messages and Point-to-Point Communication
  - Examples
- Quiz

Acknowledgments

Thanks to Rolf Rabenseifner's (HLRS)

2014 Parallel Programming Workshop Lecture Material on:

Introduction to the Message Passing Interface (MPI)

Sources

- High Performance Computing Center Stuttgart (HLRS) Online Courses https://www.hlrs.de/about-us/media-publications/teaching-training-material/
- YouTube -- search for “Introduction to MPI”, e.g., https://youtu.be/RoQJNx5npF4 -- Part I of III

Additional Courses at TU Wien

ECTS Courses (count as free electives)

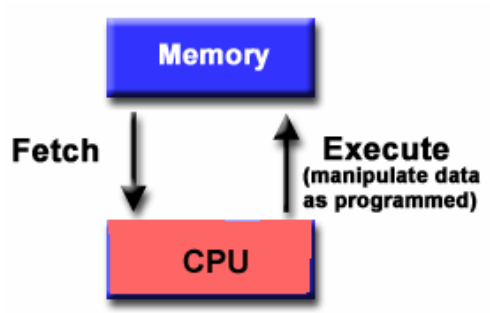
- 057.020 (Winter term)
  VSC-School I Courses in High Performance Computing
- 057.021 (Summer term)
  VSC-School II Courses in High Performance Computing

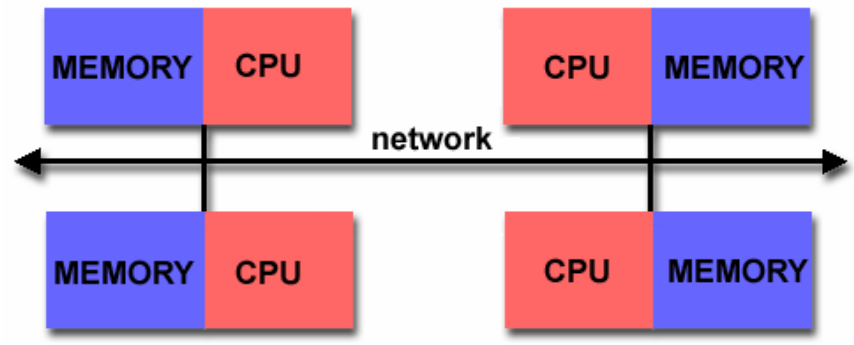
Non-ECTS Trainings (don't count as free electives)

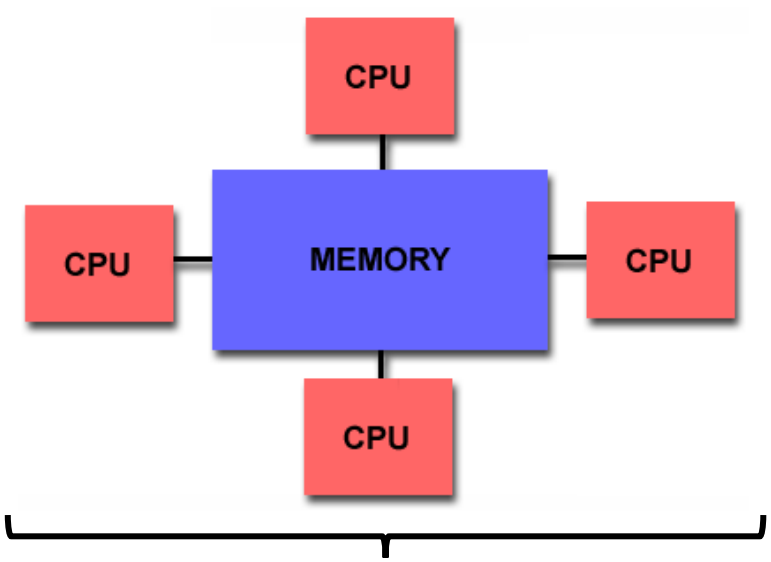
- Node-Level Performance Engineering
- OpenMP
- MPI
- Deep-Learning und GPU programming (OpenACC)
- Hybrid-programming MPI+X

http://typo3.vsc.ac.at/research/vsc-research-center/vsc-school-seminar/

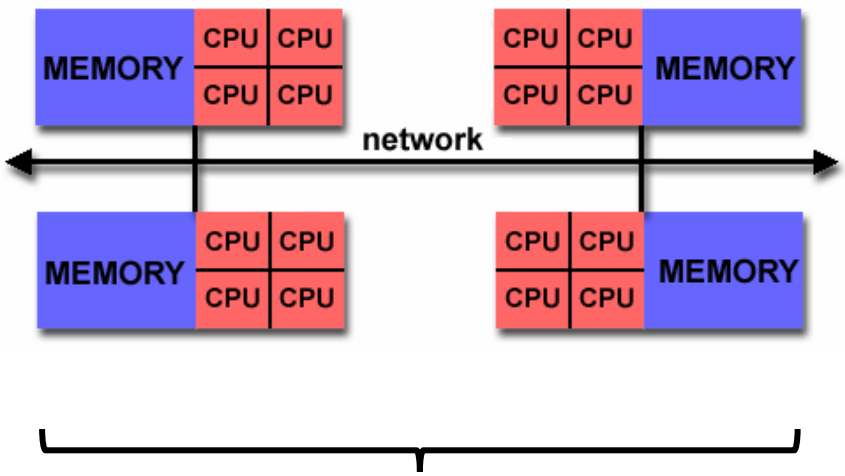
Parallel Computer Architectures







Easy to use, but not scalable



Difficult to use, but scalable

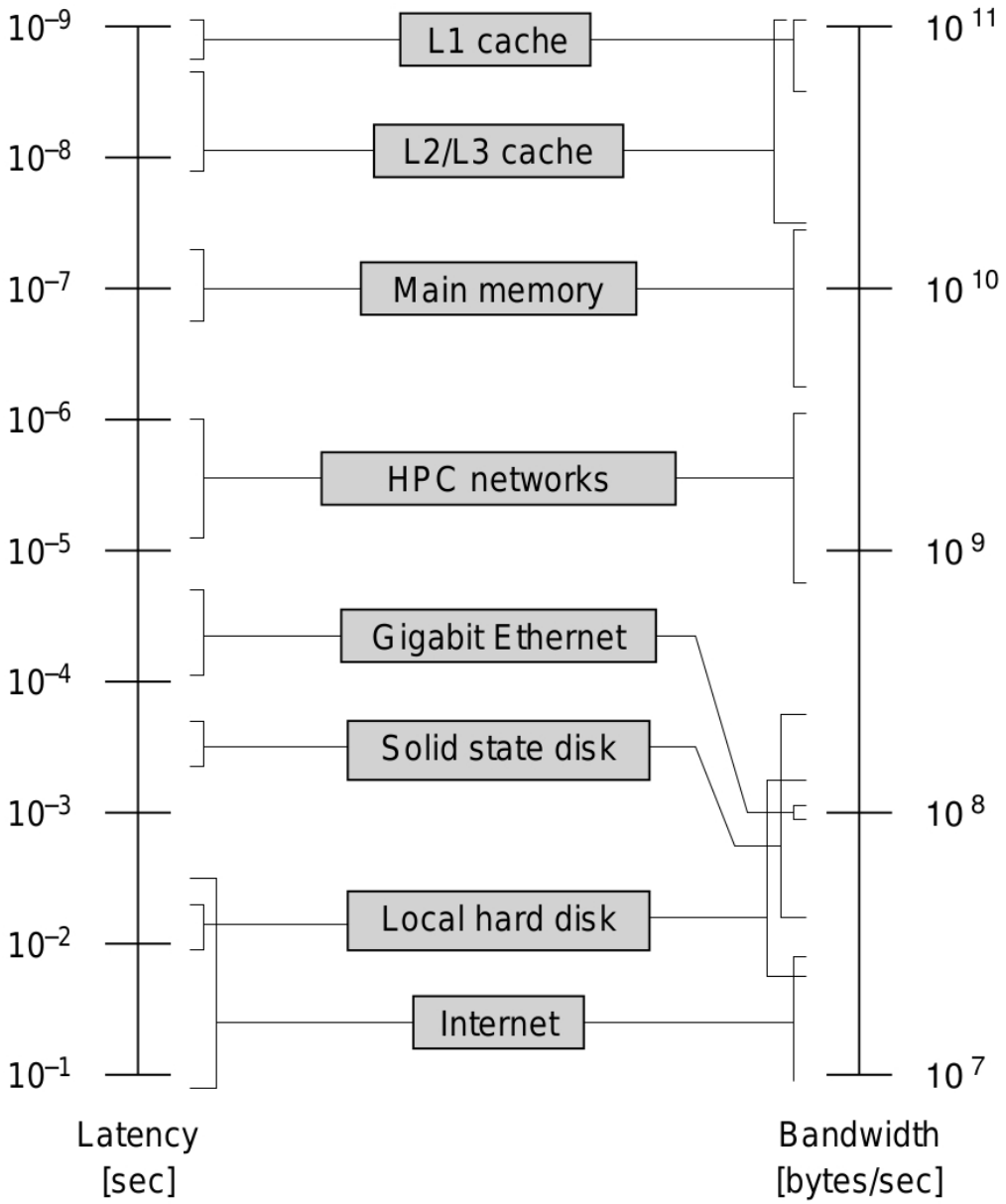
Source: Lyle N. Long, PSU

Distributed Parallel Computing Clusters

IBM Blue Gene/P: Intrepid
Argonne National Laboratory

Source: Wikipedia

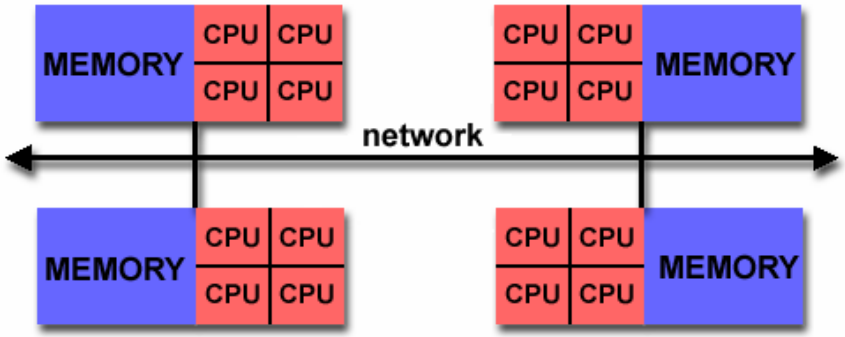
Latency / Bandwidth Hierarchy



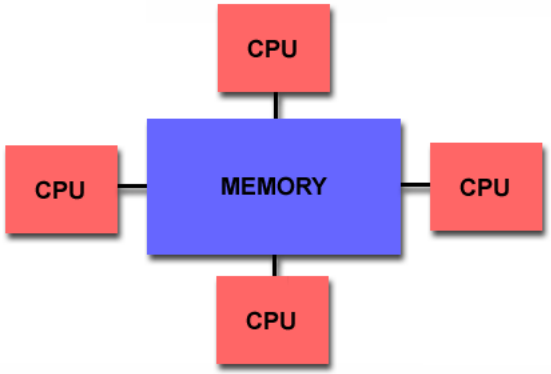
Processes and Threads

- **Process**
  - **A process is a program in execution**

Data exchange happens explicitly via sending messages between processes!



- **Thread**
  - **Unit of CPU utilization**
  - **A single process can contain multiple threads**
  - **Threads belonging to the same process can share resources**



Outline

- Introduction to the Lecture
- Distributed Parallel Computing I
  - Primer
  - **Overview**
  - Process Model and Language Bindings
  - Messages and Point-to-Point Communication
  - Examples
- Quiz

We will gradually increase the level of detail! Topics will re-emerge in later sections in a more detailed form.

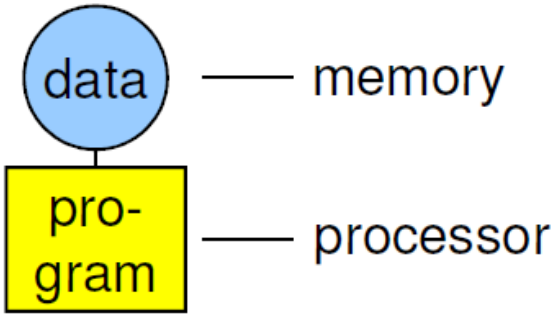
Message Passing Interface (MPI)

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
  int myid, numprocs;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  printf("I am %d out of %d\n", myid, numprocs);
  MPI_Finalize();
  return 0;
}
```

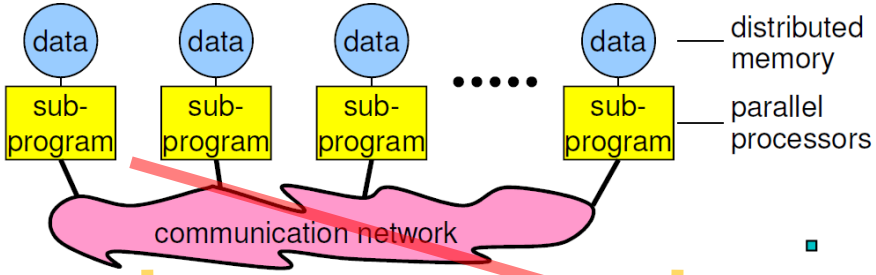
```
> mpiexec –n 4 ./hello
I am 3 out of 4
I am 1 out of 4
I am 0 out of 4
I am 2 out of 4
```

Message Passing Programming Paradigm

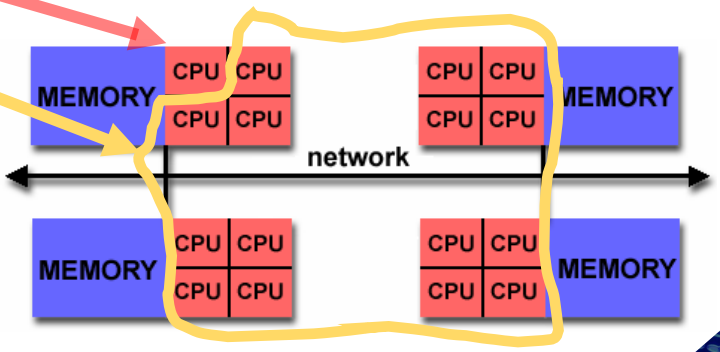
- **Sequential Programming Paradigm**



- **Message Passing Programming Paradigm**

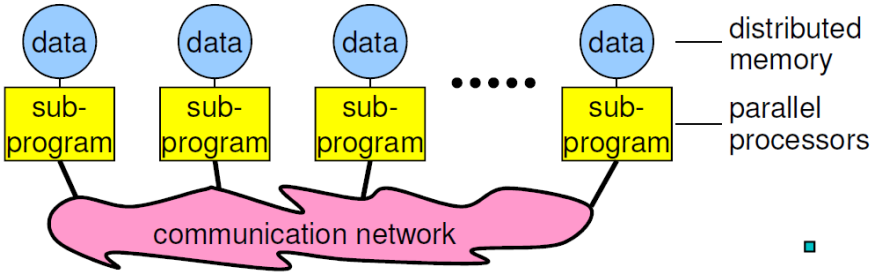



```
// identify specific process which is being executed
// on specific CPU core
if(myrank == 0) {
    // do some central organization
}
else {
    // all the other ranks should do something useful
}
```



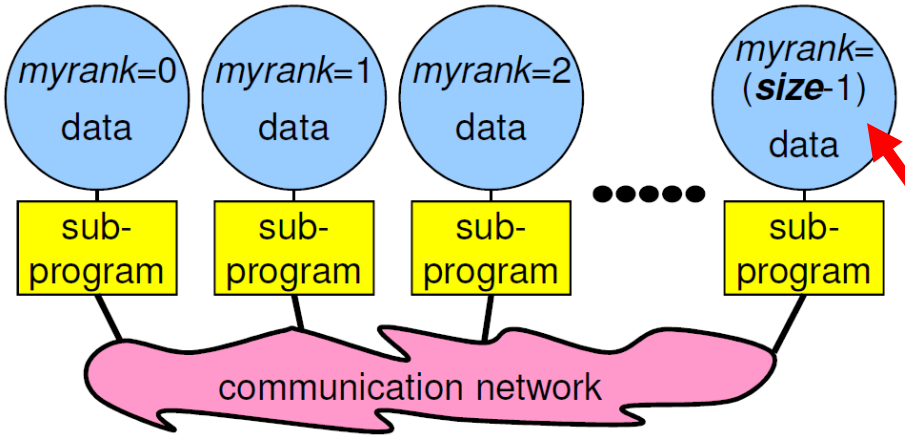
Message Passing Programming Paradigm

- **For instance: Each CPU core in a message passing program runs a *sub-program (SPMD-single program, multiple data):***
  - **written in a conventional sequential language, e.g., C/C++ or Fortran,**
  - **typically the same on each processor,**
  - **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

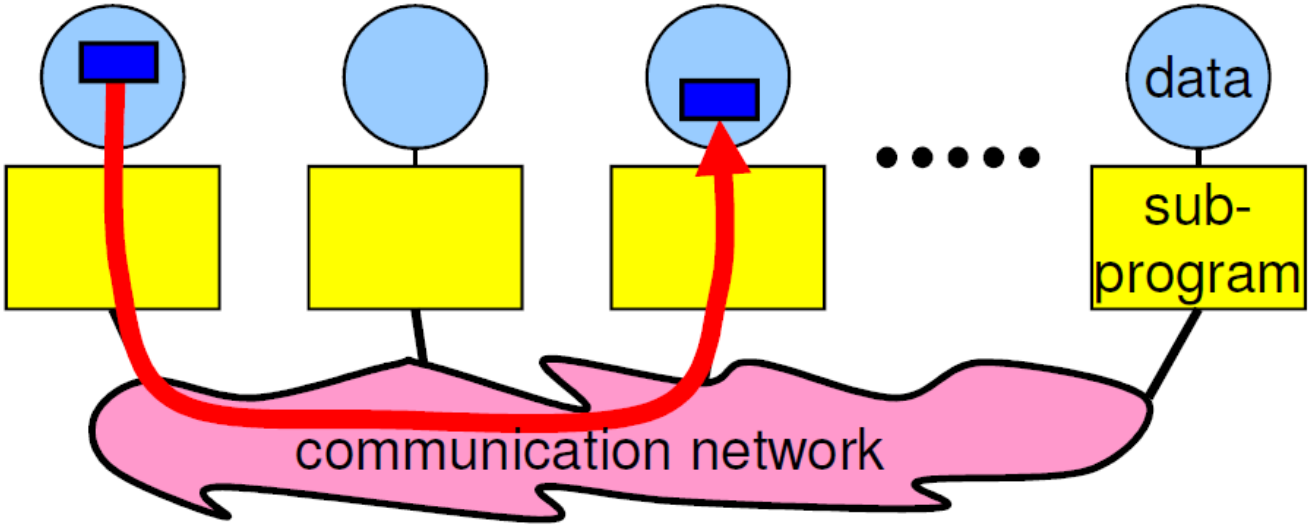
- the value of *myrank* is returned by special library routine
- the system of *size* processes is started by special MPI initialization
- program (*mpirun* or *mpiexec*)
- all distribution decisions are based on *myrank*
- i.e., which process works on which data




```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
  int myid, numprocs;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  printf("I am %d out of %d\n", myid, numprocs);
  MPI_Finalize();
  return 0;
}
```

Messages

- **Messages are packets of data moving between sub-programs**
- **Necessary information for the message passing system:**
  - **sending process – receiving process** } **the ranks**
  - **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)

Reception

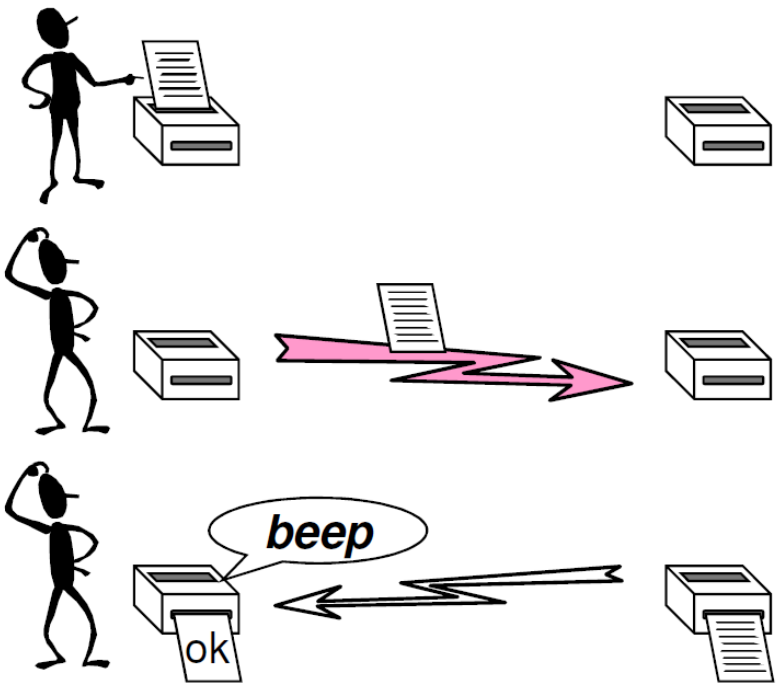
- **All messages must be received.**

Point-to-Point Communication

- Simplest form of message passing
- One process sends a message to another.
- Different types of point-to-point communication modes:
  - synchronous
  - buffered = asynchronous
  - (there is also “standard” and “ready”, more on that later)

Communication Mode: Synchronous

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



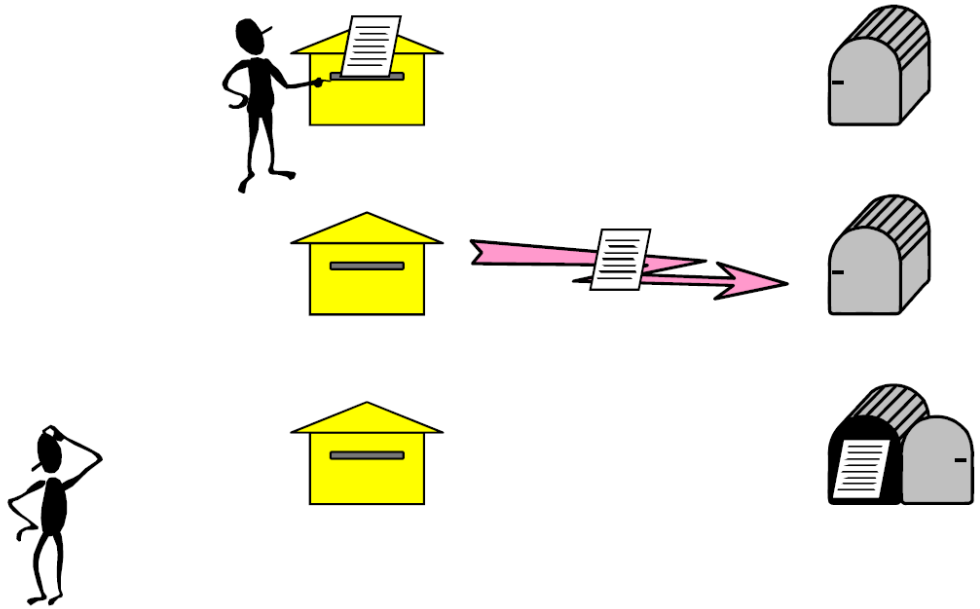
Send can be started whether or not a matching receive was posted.

Nonetheless, send will only complete successfully <u>if</u> a matching receive is posted <u>and</u> the receive process has started.

Operation is thus <u>non-local</u> as completion depends on occurrence of matching receive.

Communication Mode: Buffered (Asynchronous)

- **Only know when the message has left.**



Send can be started whether or not a matching receive was posted (same as synchronous mode).

However, the buffered send <u>may complete before</u> a matching receive is posted.

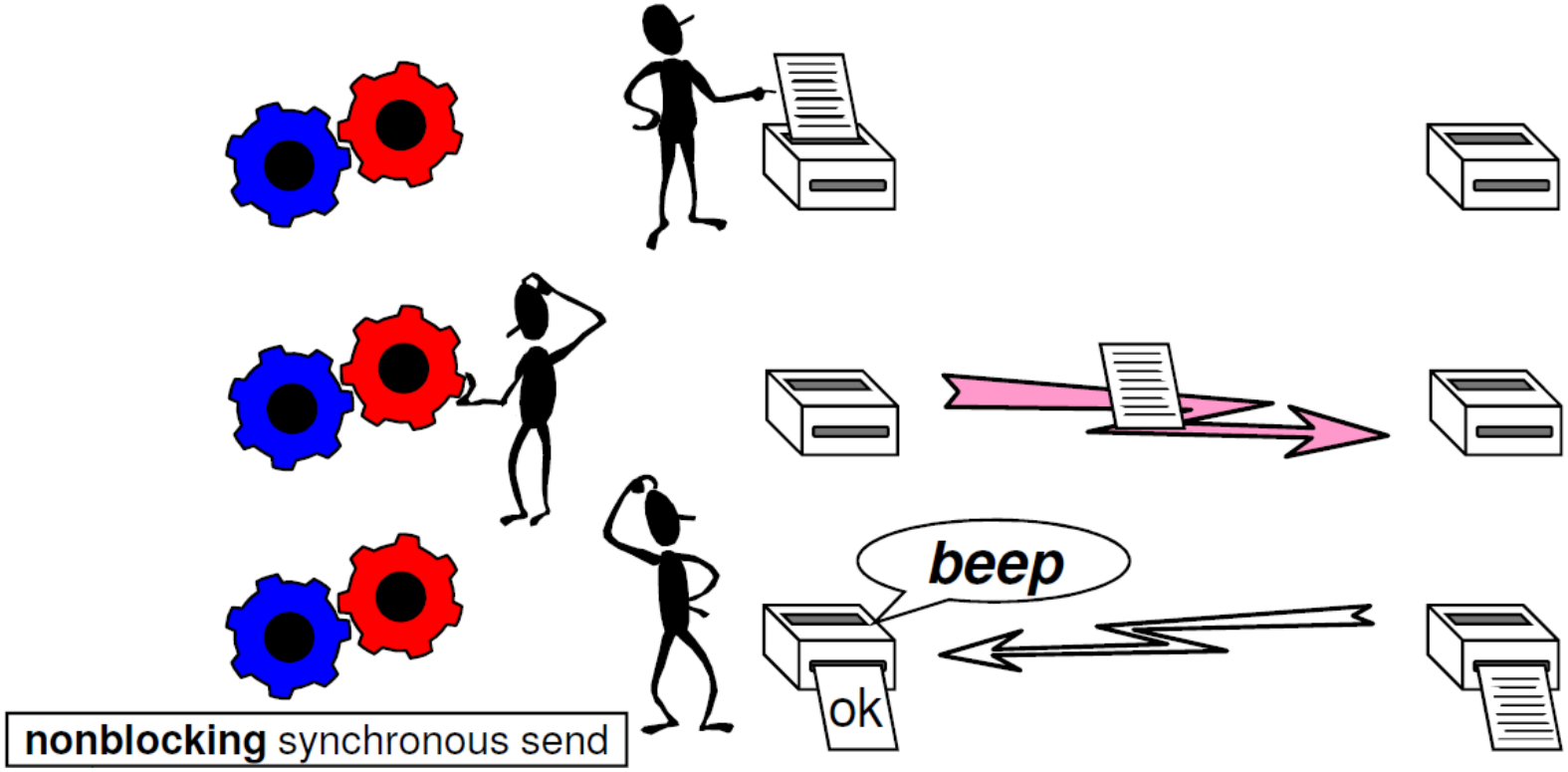
Operation is thus <u>local</u> as completion does not depend on occurrence of matching receive.

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 send buffer is safe to reuse / recv buffer contains the intended data.

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)

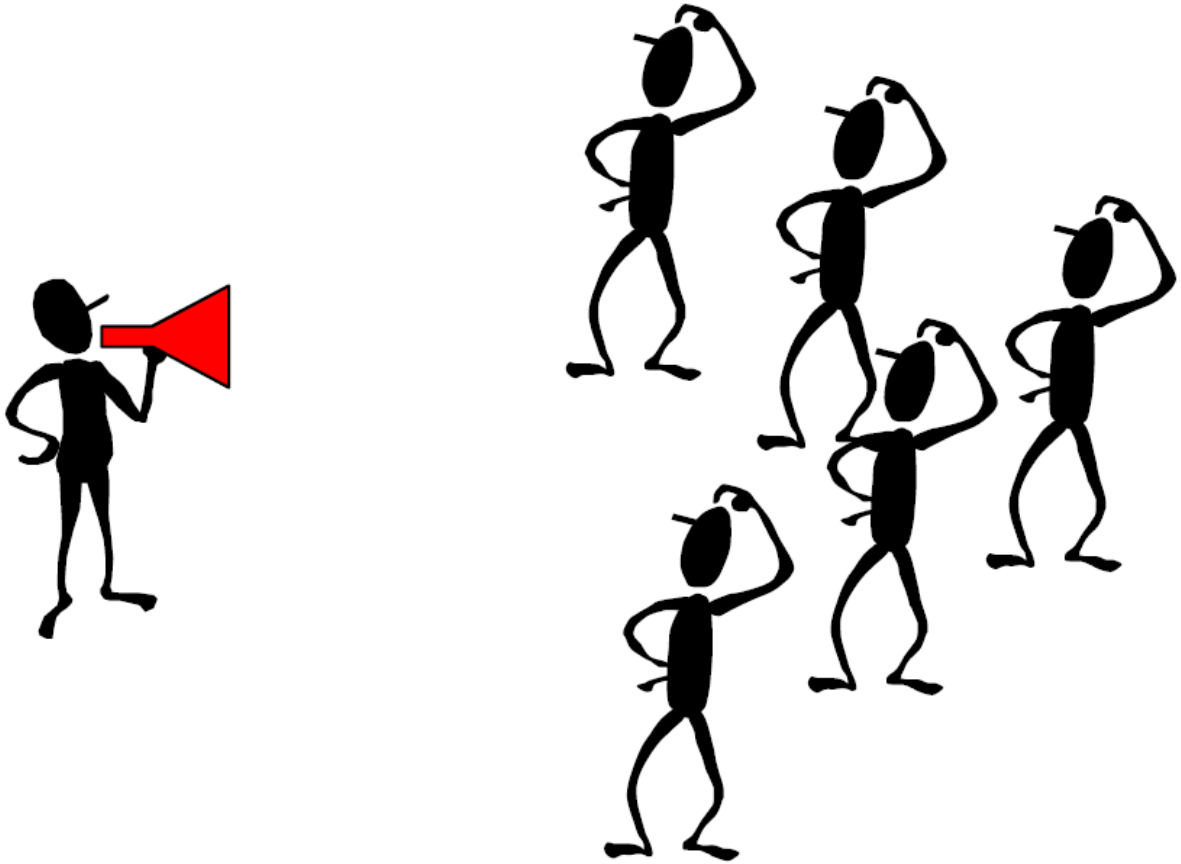
- **All non-blocking operations must have matching wait (or test) operations.**
  (Some system or application resources can be freed only when the non-blocking operation is completed.)
- **<u>A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation.</u>**
- **Non-blocking is not the same as buffered:**
  - **Blocking send returns after data has been copied out of sender memory**
  - **Non-blocking returns as soon as possible (before completion): it may not have sent/buffered the data!**
  - **Requires separate complete-call; if finished, data can be changed.**

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.

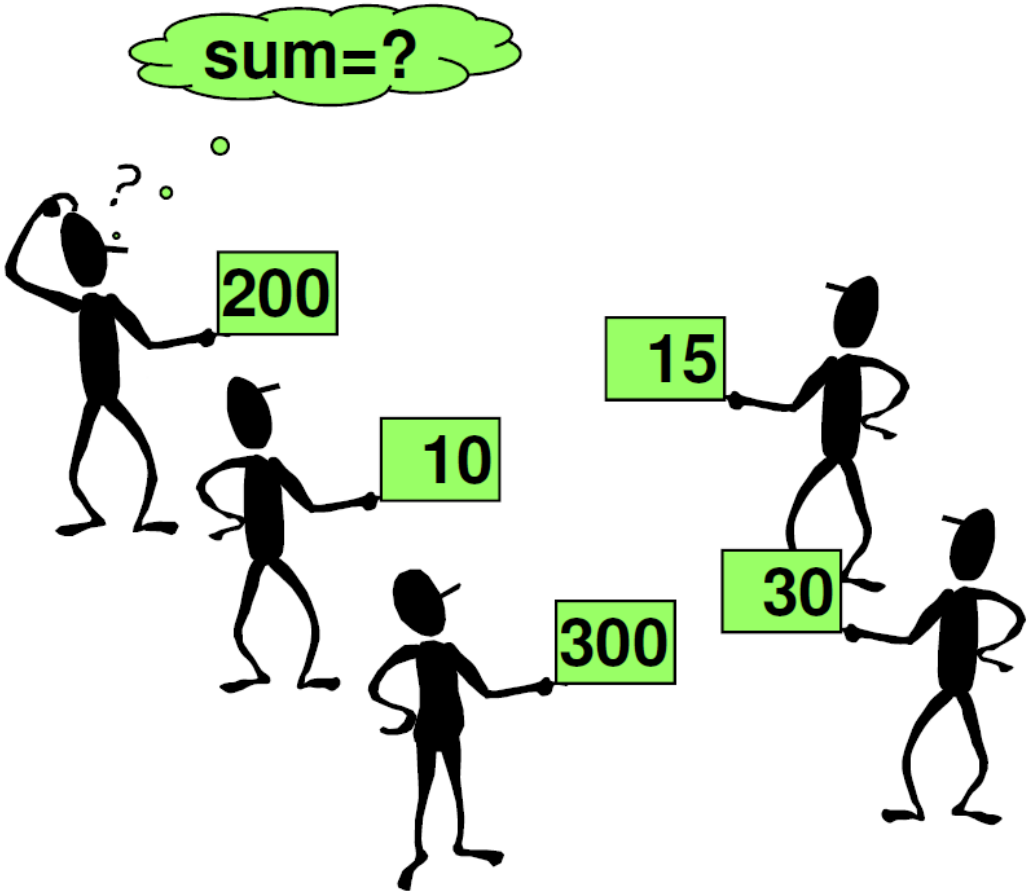
Broadcast

- **A one-to-many communication.**



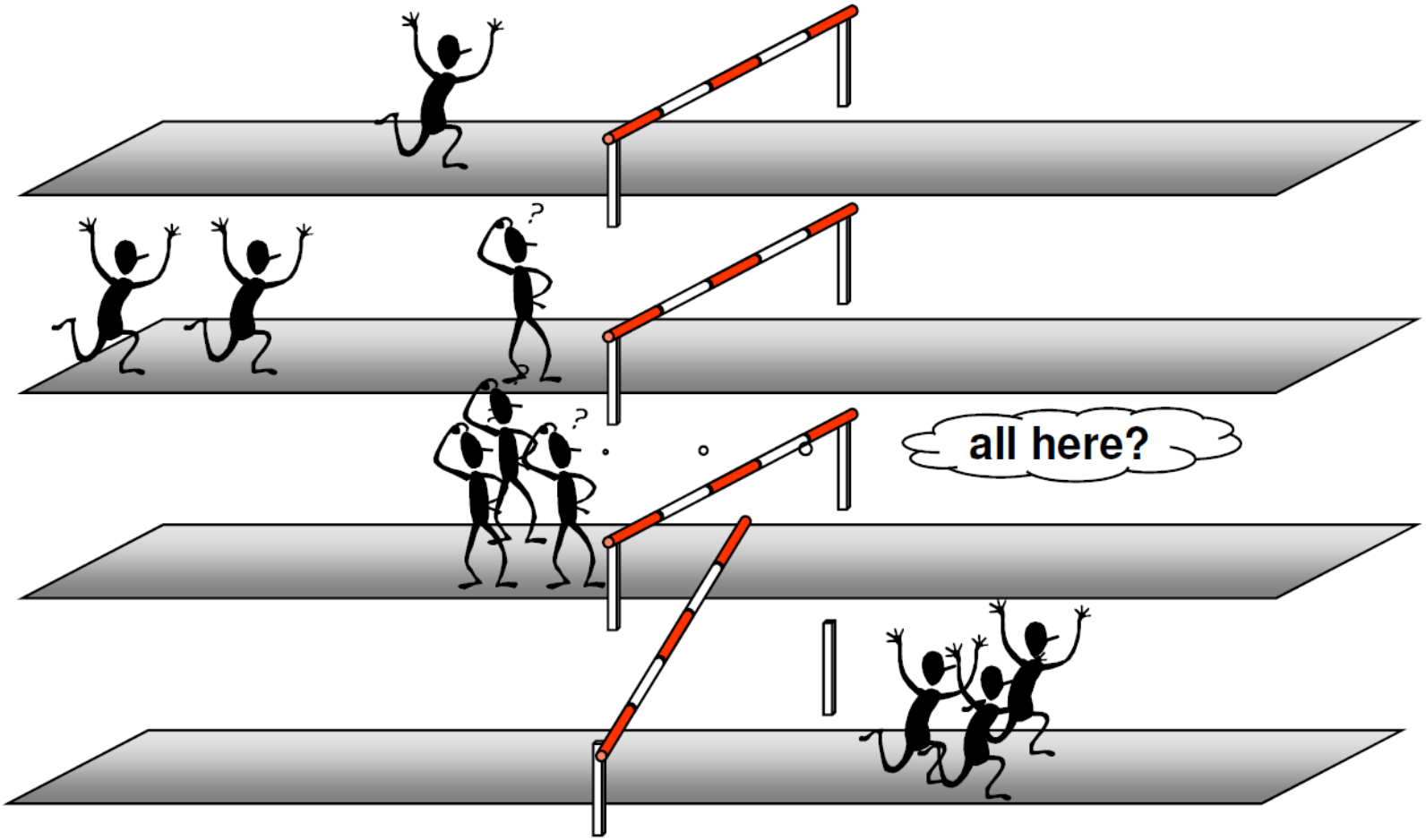
Reduction Operations

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



Barriers

- **Synchronize processes.**



MPI Forum

- MPI is a standard
  - Each MPI routine is defined
- MPI Forum: the standardization forum www.mpi-forum.org/
- MPI-1.0 June 1994
- MPI-3.1 June 2015
- MPI-4.0 June 2021
- Many MPI libraries ((mostly) adhering to the standard) are available
  - Open source: OpenMPI, MPICH
  - Native support for C/C++, Fortran (this course focuses only on C/C++)

Outline

- Introduction to the Lecture
- Distributed Parallel Computing I
  - Primer
  - Overview
  - **Process Model and Language Bindings**
  - Messages and Point-to-Point Communication
  - Examples
- Quiz

Header Files and Function Format

- **Header Files**

```
#include <mpi.h>
```

- **MPI Namespace**
  - **MPI_... namespace is reserved for MPI constants and routines**
  - **Application routines and variable names must not begin with MPI_**

- **Function Format**

```
error = MPI_Xxxxxx( parameter, ... );
MPI_Xxxxxx( parameter, ... );
```

Initializing MPI

- **First routine to be called**

```
int MPI_Init( int *argc, char ***argv)
```

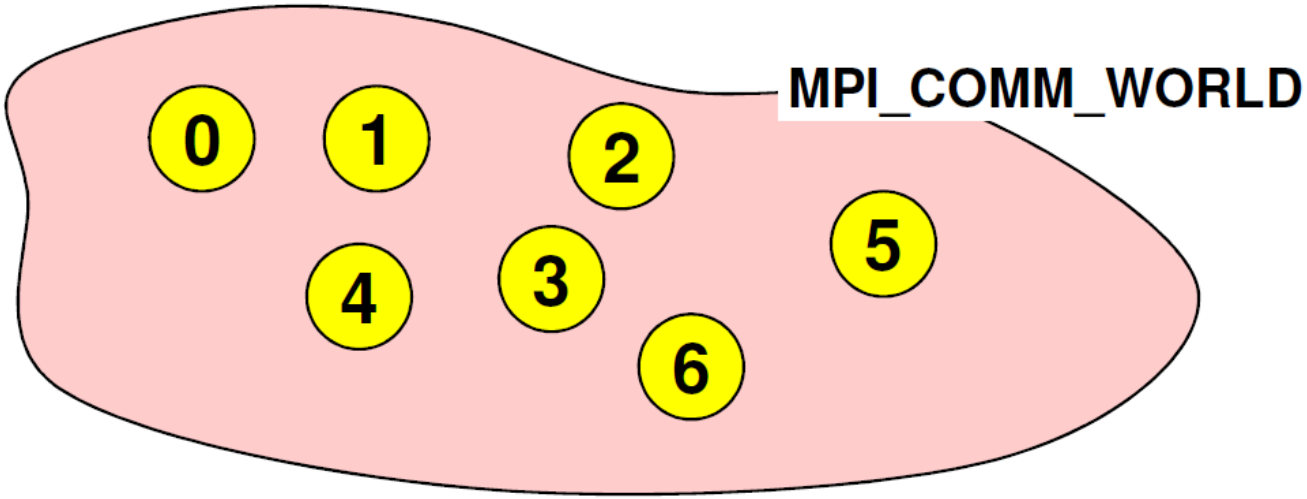
```
#include <mpi.h>
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  ...
}
```

Starting the MPI Program

- **Start mechanism is implementation dependent**
- **mpirun –np *number_of_processes ./executable***
  (most implementations)
- **mpiexec –n *number_of_processes ./executable***
  (with MPI-2 and later)

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)



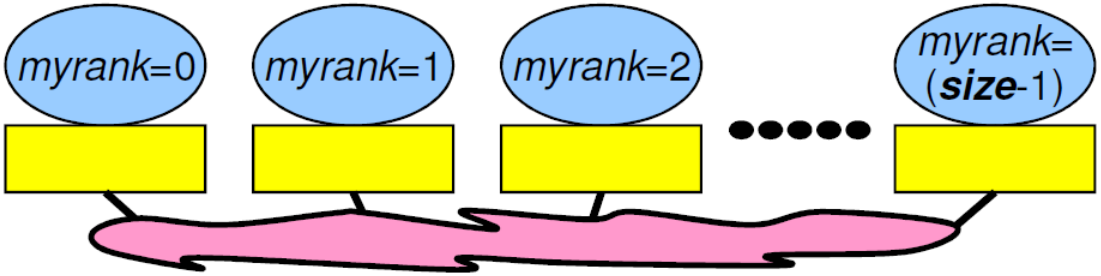
Handles

- <u>Handles</u> 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

Rank

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

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




```
CALL MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierror)
```

Size

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

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

Exiting MPI

- **<u>Must be</u> called last by all processes.**
- **User must ensure the completion of all pending communications (locally) before calling finalize**
- **After MPI_Finalize:**
  - **Further MPI-calls are forbidden**
  - **Especially re-initialization with MPI_Init is forbidden**
  - **May abort all processes except “rank==0” in MPI_COMM_WORLD**

```
int MPI_Finalize()
```

Outline

- Introduction to the Lecture
- Distributed Parallel Computing I
  - Primer
  - Overview
  - Process Model and Language Bindings
  - **Messages and Point-to-Point Communication**
  - Examples
- Quiz

Messages

- **A message contains a number of elements of some particular datatype.**
- **MPI datatypes:**
  - **Basic datatype.**
  - **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

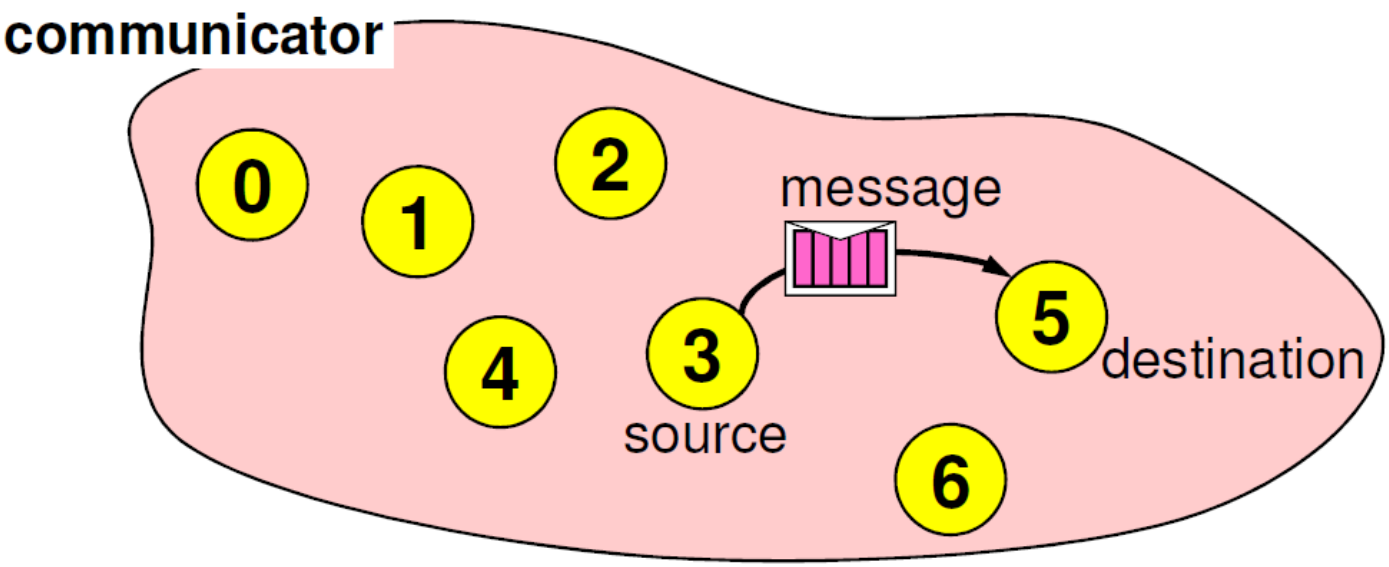
MPI Datatype	C datatype	Remarks
MPI_CHAR	char	Treated as printable character
MPI_SHORT	signed short int	
MPI_INT	signed int	
MPI_LONG	signed long int	
MPI_LONG_LONG	signed long long	
MPI_SIGNED_CHAR	signed char	Treated as integral value
MPI_UNSIGNED_CHAR	unsigned char	Treated as integral value
MPI_UNSIGNED_SHORT	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		

Further datatypes, see, e.g., MPI-3.0, Annex A.1

Includes also special C++ types, e.g., bool, see page 666

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.



Sending a Message

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

Example

```
int number = 4;
int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if(myrank == 0)
  // send number to rank 1 using tag 0
  MPI_Send(&number, 1, MPI_INT, 1, 0,
  MPI_COMM_WORLD);
```

- **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 nonnegative 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

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

Example

```
int number;
int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if(myrank == 1)
  // recv number from rank 0 using tag 0
  MPI_Recv(&number, 1, MPI_INT, 0, 0,
  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- ***buf*/count/datatype describe the receive buffer.**
- **Receiving the message sent by process with rank source in comm.**
- **Envelope information is returned in *status*.**
- **On can pass MPI_STATUS_IGNORE instead of a status argument.**
- **Output arguments are printed *blue-cursive*.**
- **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.**
- **Buffer's type must match with the datatype handle (in the send and receive call)**
- **Message datatypes must match.**
- **Receiver's buffer must be large enough.**

Wildcarding

- **Receiver can wildcard.**
- **To receive from any <u>source:</u>**
  source = MPI_ANY_SOURCE
- **To receive from any <u>tag:</u>**
  tag = MPI_ANY_TAG
- **Actual source and tag are returned in the receiver's *status* parameter.**

Example

```
int number;
int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if(myrank == 1)
  // recv number from any rank using tag 0
  MPI_Recv(&number, 1, MPI_INT, MPI_ANY_SOURCE, 0,
  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Communication Envelope

- **Envelope information is returned from MPI_RECV in *status*.**

```
MPI_Status status;
status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR
```

- **Receive Message Count**

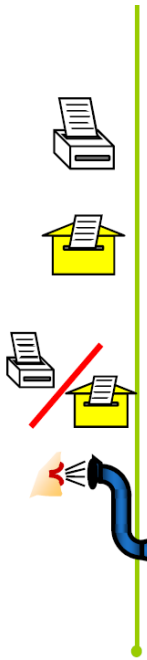
```
int MPI_Get_count(      MPI_Status*     status,
                        MPI_Datatype    datatype,
                        int*            count)
```

(Adapted) Example

```
int number, number_amount;
int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if(myrank == 1) {
  MPI_Status status;
  // recv number from rank 0 using tag 0
  MPI_Recv(&number, 1, MPI_INT, 0, 0,  MPI_COMM_WORLD, &status);
  MPI_Get_count(&status, MPI_INT, &number_amount);
  printf(“Rank 1 received %d numbers. Message source = %d, tag = %d\n",
         number_amount, status.MPI_SOURCE, status.MPI_TAG);
}
```

Communication Modes

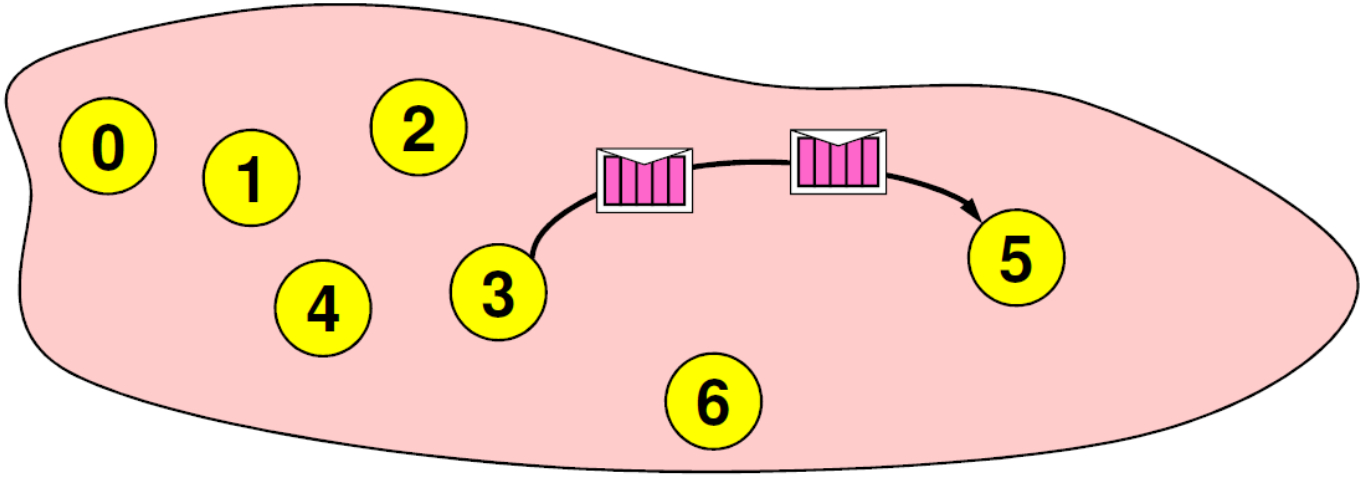
- **Send communication modes:**
  - **synchronous send → MPI_SSEND**
  - **buffered [asynchronous] send → MPI_BSEND**
  - **standard send → MPI_SEND**
  - **Ready send → MPI_RSEND**
- **Receiving all modes → MPI_RECV**



Sender mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Standard send **MPI_SEND**	Either synchronous or buffered	uses an internal buffer
Ready send **MPI_RSEND**	May be started **only** if the matching receive is already posted!	highly dangerous!
Receive **MPI_RECV**	Completes when a message has arrived	same routine for all communication modes

Message Order Preservation

- Rule for messages on the same connection,
- i.e., same communicator, source, and destination rank:
- Messages do not overtake each other.
- This is true even for non-synchronous sends.
- If the here shown two receives match both messages, then the order is preserved.



Outlook Collective Communications

- Two kinds
  - Data movement (broadcast, scatter, gather, etc.)
  - Collective computation (min, max, sum, logical OR etc.)
- Advantages
  - More convenient
- Can be used as short cut for an ensemble of P2P operations
  - More efficient
- Encapsulate sophisticated algorithms
- Implementation can take advantage of the structure of a machine to optimize and increase parallelism in these operations

Broadcast

- **Broadcasts a message from the process with rank root to all other processes of the group.**
  - **buf = starting address of buffer**
  - **count = number of entries in buffer**
  - **datatype = data type of buffer**
  - **root = rank of broadcast root**
  - **comm = communicator**

```
int MPI_Bcast(    void*          buf,
                  int            count,
                  MPI_Datatype   datatype,
                  int            root,
                  MPI_Comm       comm)
```

Reduce

- **Combines the elements in the input buffer of each process using the operation op and returns the combined value in the output buffer of the process with rank root**
  - **sendbuf = address of send buffer**
  - **recvbuf = address of receive buffer**
  - **count = number of elements in send buffer**
  - **datatype = data type of elements of send buffer**
  - **op = reduce operation**
  - **root = rank of root process**
  - **comm = communicator**

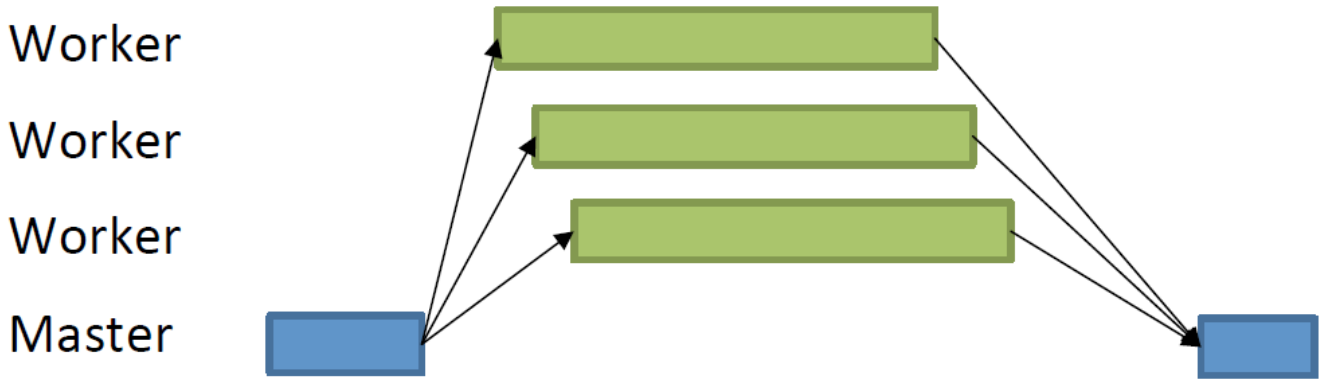
```
int MPI_Reduce(  void*          sendbuf,
                 void*          recvbuf,
                 int            count,
                 MPI_Datatype   datatype,
                 MPI_Op         op,
                 int            root,
                 MPI_Comm       comm)
```

Outline

- Introduction to the Lecture
- Distributed Parallel Computing I
  - Primer
  - Overview
  - Process Model and Language Bindings
  - Messages and Point-to-Point Communication
  - **Example**
- Quiz

Master Worker

- **Idea: self-scheduling algorithm**
  - **Master coordinates processing of tasks by providing input data to workers and collecting results**
- **Suitable if**
  - **Workers need not communicate with one another**
  - **Amount of work each worker must perform is difficult to predict**
- **Example: matrix-vector multiplication**



Matrix-Vector Multiplication

$$A \cdot \vec{b} = \vec{c}$$

Unit of work:

dot product of one row of matrix A with vector b

Master
- Broadcasts b to each worker
- Sends one row to each worker
- Loop
  - Receives dot product from whichever worker sends one
  - Sends next row to that worker
  - Termination of loop if all rows are handed out

Worker
- Receives broadcast value of b
- while-Loop
  - Receives row from A
  - Forms dot product
  - Returns result back to master

Matrix-Vector Multiplication: Common Part 1

```
#include <stdio.h>
#include <stdlib.h>
#include “mpi.h”
#define MAX_ROWS 1000 // upper limits
#define MAX_COLS 1000 // upper limits
#define MIN(a, b) ((a) > (b) ? (b) : (a))
#define DONE MAX_ROWS+1
int main(int argc, char **argv) {
  double** A;
  double* b, * c, * buffer;
  double ans;
  int myid, master, numprocs, i, j, numsent, sender, done, anstype, row, rows, cols;

  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  A = (double **)calloc(MAX_ROWS, sizeof(double *));
  for (i=0; i<MAX_ROWS; i++)
    A[i] = (double *)calloc(MAX_COLS, sizeof(double));
  b = (double *)calloc(MAX_COLS, sizeof(double));
  c = (double *)calloc(MAX_ROWS, sizeof(double));
  buffer = (double *)calloc(MAX_COLS, sizeof(double));
  ...
```

This is a minimal working examples, try it yourself!

Matrix-Vector Multiplication: Common Part 2

```
  …
  master = 0;
  rows = 2; // for debugging …
  cols = 2; // for debugging …
  if (myid == master)  {  /* master code: next slides */
  } else               {  /* worker code: next slides */
  }
  // result c vector finished: considering the test values, all element values should be 3.0
  for (i=0; i<MAX_ROWS; i++)
    free(A[i]);
  free(A);
  free(b);
  free(c);
  free(buffer);
  MPI_Finalize();
  return 0;
}
```

Matrix-Vector Multiplication: Master Part 1

Master

- **Broadcasts b to each worker**
- **Sends one row to each worker**
- **Loop**
  - **Receives dot product from whichever worker sends one**
  - **Sends next task to that worker**
  - **Termination if all tasks are handed out**

```
…
if (myid == master)   {
/* Initialize A and b (arbitrary) */
  A[0][0] = 1.0;
  A[0][1] = 2.0;
  A[1][0] = 1.0;
  A[1][1] = 2.0;
  b[0] = 1.0;
  b[1] = 1.0;

numsent = 0;
/* Send b to each worker process */
MPI_Bcast(b, cols, MPI_DOUBLE, master, MPI_COMM_WORLD);
/* Send a row to each worker process; tag with row number */
for (i = 0; i < MIN(numprocs - 1, rows); i++) {
  MPI_Send(&A[i][0], cols, MPI_DOUBLE, i+1, i, MPI_COMM_WORLD);
  numsent++;
}
…
```

Matrix-Vector Multiplication: Master Part 2

```
...
for (i = 0; i < rows; i++) {
  MPI_Recv(&ans, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
  sender = status.MPI_SOURCE;
  /* row is tag value */
  anstype = status.MPI_TAG;
  c[anstype] = ans;
  /* send another row */
  if (numsent < rows) {
    MPI_Send(&A[numsent][0], cols,
                MPI_DOUBLE, sender,
                 numsent, MPI_COMM_WORLD);
    numsent++;
  } else {
    /* Tell sender that there is no more work */
    MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE, sender, DONE, MPI_COMM_WORLD);
  }
}
} /* end of master specific part*/
```

Master

- **Broadcasts b to each worker**
- **Sends one row to each worker**
- **Loop**
  - **Receives dot product from whichever worker sends one**
  - **Sends next task to that worker**
  - **Termination if all tasks are handed out**

MPI_BOTTOM (indicates end of address space) as placeholder!

Matrix-Vector Multiplication: Worker Part

```
else    {     /* start of worker code*/
MPI_Bcast(b, cols, MPI_DOUBLE, master, MPI_COMM_WORLD);
/* Skip if more processes than work */
done = myid > rows;
while (!done) {
  MPI_Recv(buffer, cols, MPI_DOUBLE, master, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);
  done = status.MPI_TAG == DONE;
  if (!done) {
    row = status.MPI_TAG;
    ans = 0.0;
    for (i = 0; i < cols; i++) {
      ans += buffer[i] * b[i];
    }
    MPI_Send(&ans, 1, MPI_DOUBLE, master, row, MPI_COMM_WORLD);
  }
}
} /* end of worker code */
```

Worker

- **Receives broadcast value of b**
- **while-Loop**
  - **Receives row from A**
  - **Forms dot product**
  - **Returns answer back to master**

Outline

- Introduction to the Lecture
- Distributed Parallel Computing I
  - Primer
  - Overview
  - Process Model and Language Bindings
  - Messages and Point-to-Point Communication
  - Examples
- **Quiz**

Quiz

- **Q1: How is it ensured that a specific message is received by a specific process?**
- **Q2: What is the first and last routine to be called in a MPI program?**
- **Q3: Is “MPI_Init” executed by one, several or all MPI processes?**
- **Q4: Name typical reduction operations?**
- **Q5: How can a point-to-point communication be made non-blocking? What potential advantage is there?**