www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Introduction to the MPI Programming Model

Marc Jordà, Antonio J. Peña

Montevideo, 21-25 October 2019

What will be covered in this tutorial

- **What is MPI?**
- How to write a simple program in MPI
- Running your application
- More advanced topics:
  - Non-blocking communication, collective communication, datatypes
  - One-sided communication
  - Hybrid programming with shared memory and accelerators
  - Non-blocking collectives, topologies, and neighborhood collectives

The switch from sequential to parallel computing

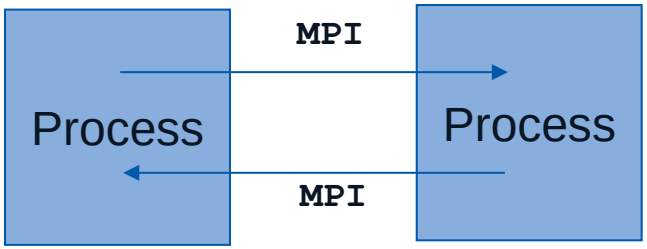
- Moore's law continues to be true, but…
  - Processor speeds no longer double every 18-24 months
  - Number of processing units double, instead
    - Multi-core chips (dual-core, quad-core, hex-core)
  - No more automatic increase in speed for software
- Parallelism is the norm
  - Lots of processors connected over a network and coordinating to solve large problems
  - Used everywhere!
    - By messaging companies for tracking and minimizing fuel routes
    - By automobile companies for car crash simulations
    - By airline industry to build newer models of flights

Sample Parallel Programming Models

- Shared Memory Programming
  - Processes share memory address space (threads model)
  - Application programmer ensures no data races/corruption (Lock/Unlock)
- Transparent Parallelization
  - Compiler works magic on sequential programs
- Directive-based Parallelization
  - Compiler needs help (e.g., OpenMP, OmpSs)
- Message Passing
  - Explicit communication between processes
    - Like sending and receiving emails
  - MPI falls in this category

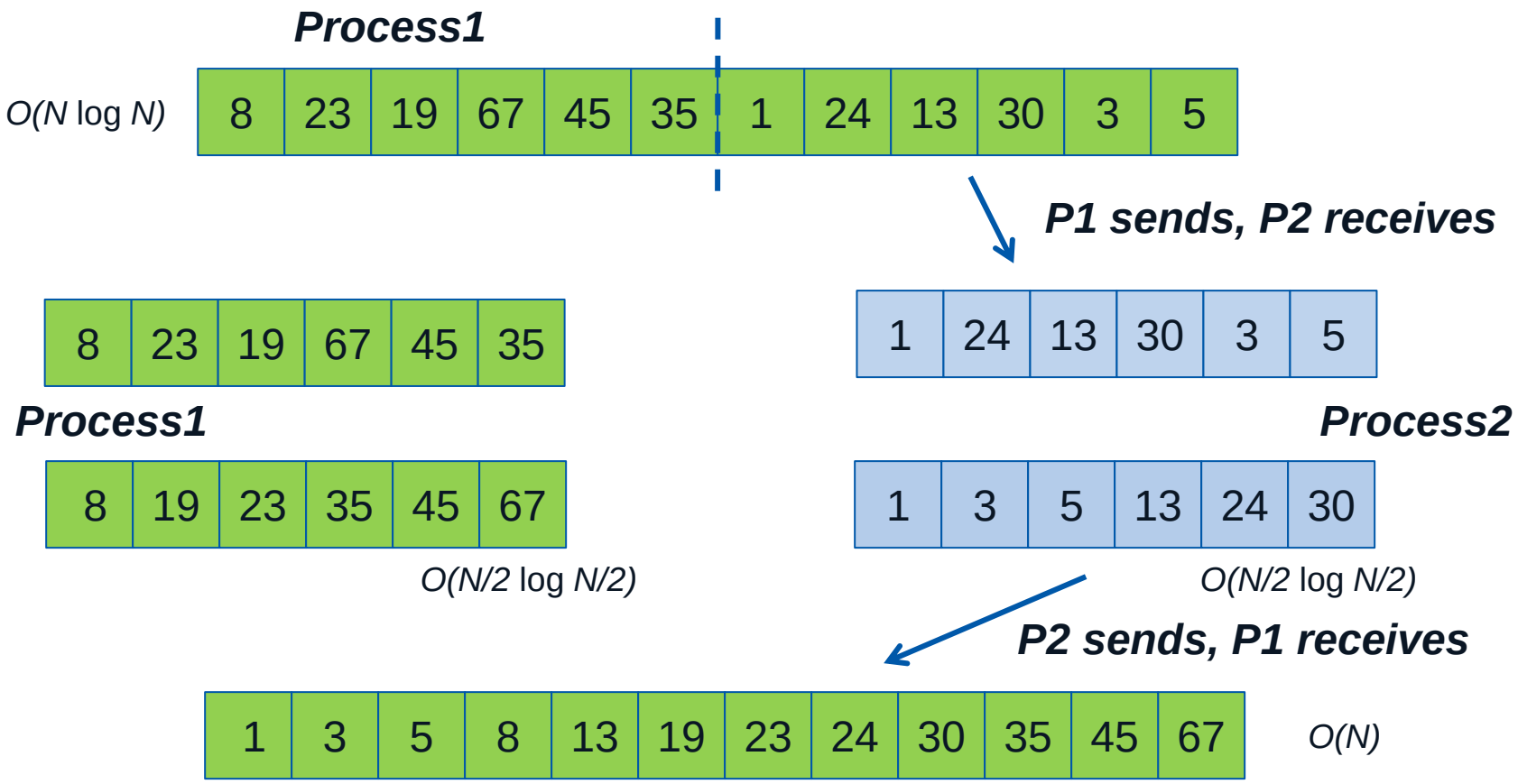
The Message-Passing Model

- *Process* (traditionally): program counter + address space
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space
- MPI is for communication among processes, which have separate address spaces
  - No inter-process load/store possible (in principle)
- Inter-process communication consists of
  - synchronization
  - movement of data from one process's address space to another's



The Message-Passing Model (an example)

- Each process has to send/receive data to/from other processes
- Example: Sorting Integers



Standardizing Message-Passing Models with MPI

- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable)

- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of message-passing issues
  - Lacked vendor support
  - Were not implemented at the most efficient level

- The MPI Forum was a collection of vendors, portability writers and users that wanted to standardize all these efforts

What is MPI?

- MPI: Message Passing Interface
  - The MPI Forum organized in 1992 with broad participation by:
    - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
    - Portability library writers: PVM, p4
    - Users: application scientists and library writers
    - MPI-1 finished in 18 months
  - Incorporates the best ideas in a “standard” way
    - Each function takes fixed arguments
    - Each function has fixed semantics
      - Standardizes what the MPI implementation provides and what the application can and cannot expect
      - Each system can implement it differently as long as the semantics match
- MPI is a library API (defines functions and their semantics)
  - Is not a language or compiler specification
  - Is not a specific implementation or product

MPI-1

- MPI-1 supports the classical message-passing programming model: basic point-to-point communication, collectives, datatypes, etc
- MPI-1 was defined (1994) by a broadly based group of parallel computer vendors, computer scientists, and applications developers.
  - 2-year intensive process
- Implementations appeared quickly. Now MPI is taken for granted as vendor-supported software on parallel machines
- Free, portable implementations exist for clusters and other environments (MPICH, Open MPI)

Following MPI Standards

- MPI-2 was released in 1997
  - Several additional features including MPI + threads, MPI-I/O, remote memory access functionality and many others
- MPI-2.1 (2008) and MPI-2.2 (2009) were released with some corrections to the standard and small features
- MPI-3 (2012) added several new features to MPI
- MPI-3.1 (2015) is the latest version of the standard with minor corrections and features
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of material including tutorials, a FAQ, other MPI pages

MPI-2

- Same process of definition by MPI Forum
- MPI-2 is an extension of MPI
  - Extends the message-passing model.
    - Parallel I/O
    - Remote memory operations (one-sided)
    - Dynamic process management
  - Adds other functionality
    - C++ and Fortran 90 bindings
      - similar to original C and Fortran-77 bindings
    - External interfaces
    - Language interoperability
    - MPI interaction with threads

Overview of New Features in MPI-3

- Major new features
  - Nonblocking collectives
  - Neighborhood collectives
  - Improved one-sided communication interface
  - Tools interface
  - Fortran 2008 bindings
- Other new features
  - Matching Probe and Recv for thread-safe probe and receive
  - Noncollective communicator creation function
  - “const” correct C bindings
  - Comm_split_type function
  - Nonblocking Comm_dup
  - Type_create_hindexed_block function
- C++ bindings removed
- Previously deprecated functions removed
- MPI 3.1 added nonblocking collective I/O functions

Status of MPI-3.1 Implementations

	MPICH	MVAPICH	Open MPI	Cray	Tianhe	Intel MPI	IBM			HPE	Fujitsu	MS	MPC	NEC	Sunway	RIKEN	AMPI
							BG/Q (legacy) [1]	PE (legacy) [2]	Spectrum								
NBC	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Nbr. Coll.	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	✔	✔
RMA	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	(*)
Shr. mem	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	*
MPI_T	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	*	✔	✔	✔	✔	Q1 '19
Comm-create group	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	*	✔	✔	✔	✔	✔
F08 Bindings	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	*	✘	✔	✔	✔	✔	Q2 '19
New Dtypes	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Large Counts	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
MProbe	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
NBC I/O	✔	✔	✔	✔	✘	✔	✘	✘	✔	✔	*	✘	*	✔	✘	✔	Q2'19

[1] Open Source but unsupported

[2] No MPI_T variables exposed

* Under development (*) Partly done

Slide Updated 11/6/2018

Release dates are estimated and are subject to change at any time

✘ indicates no publicly announced plan to implement/support that feature

Pavan Balaji

Web Pointers

- MPI Standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/

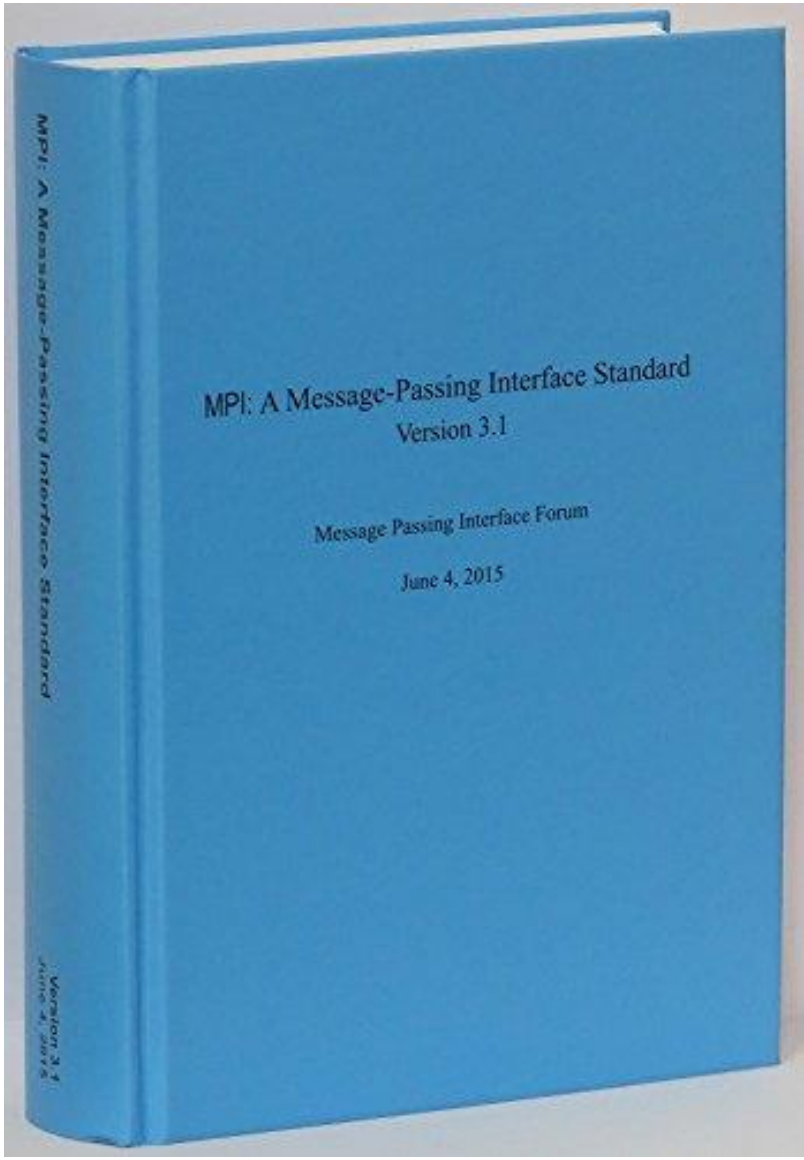
- MPI implementations:
  - MPICH : http://www.mpich.org
  - MVAPICH : http://mvapich.cse.ohio-state.edu/
  - Intel MPI: http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI: www.microsoft.com/en-us/download/details.aspx?id=39961
  - Open MPI : http://www.open-mpi.org/
  - IBM MPI, Cray MPI, HP MPI, TH MPI, …
- Several MPI tutorials can be found on the web

Latest MPI 3.1 Standard in Book Form

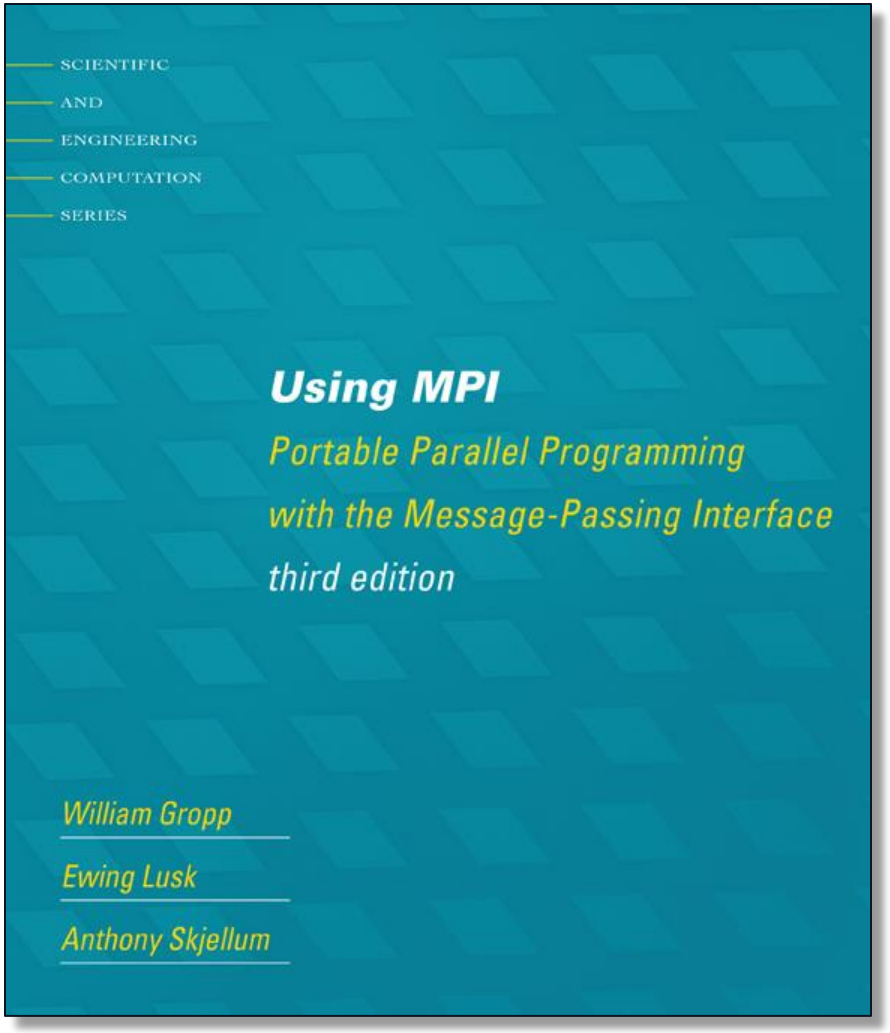
Available from amazon.com

http://www.amazon.com/dp/B015CJ42CU/

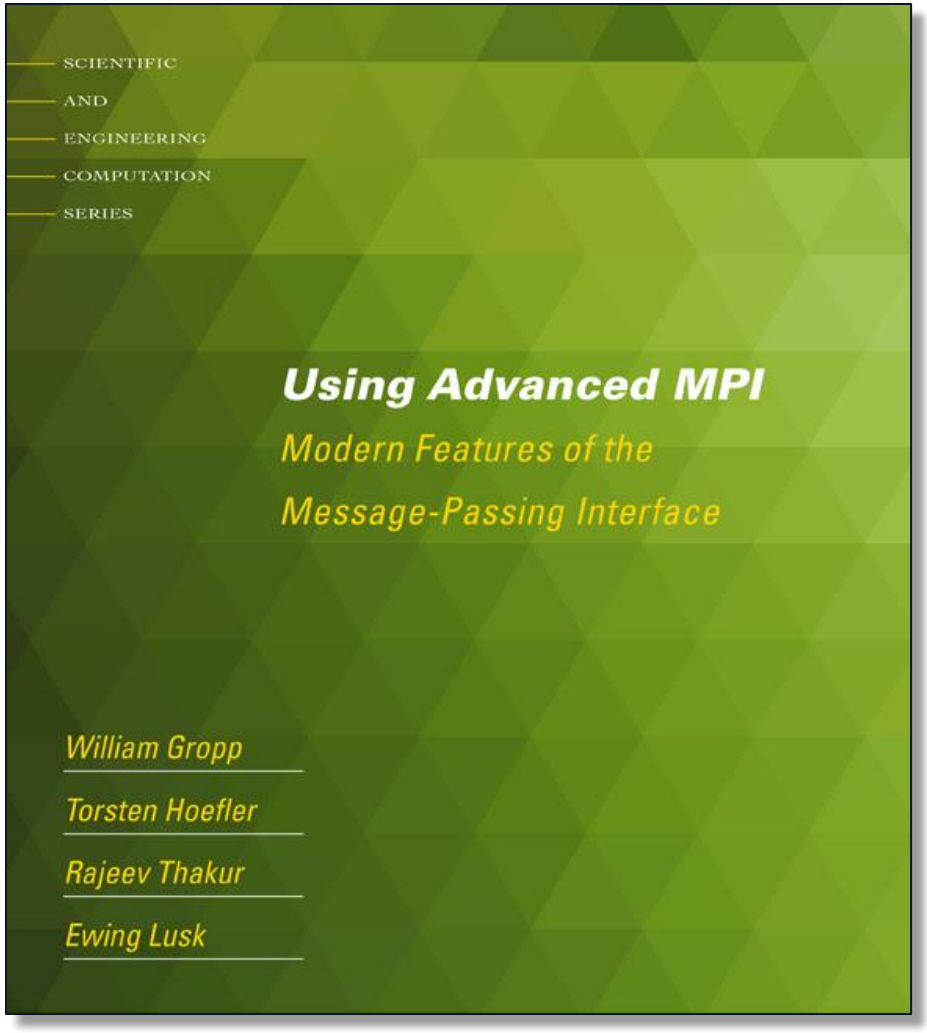
Targeted to implementors of MPI libraries, not so much for MPI users.



New Tutorial Books on MPI



Basic MPI

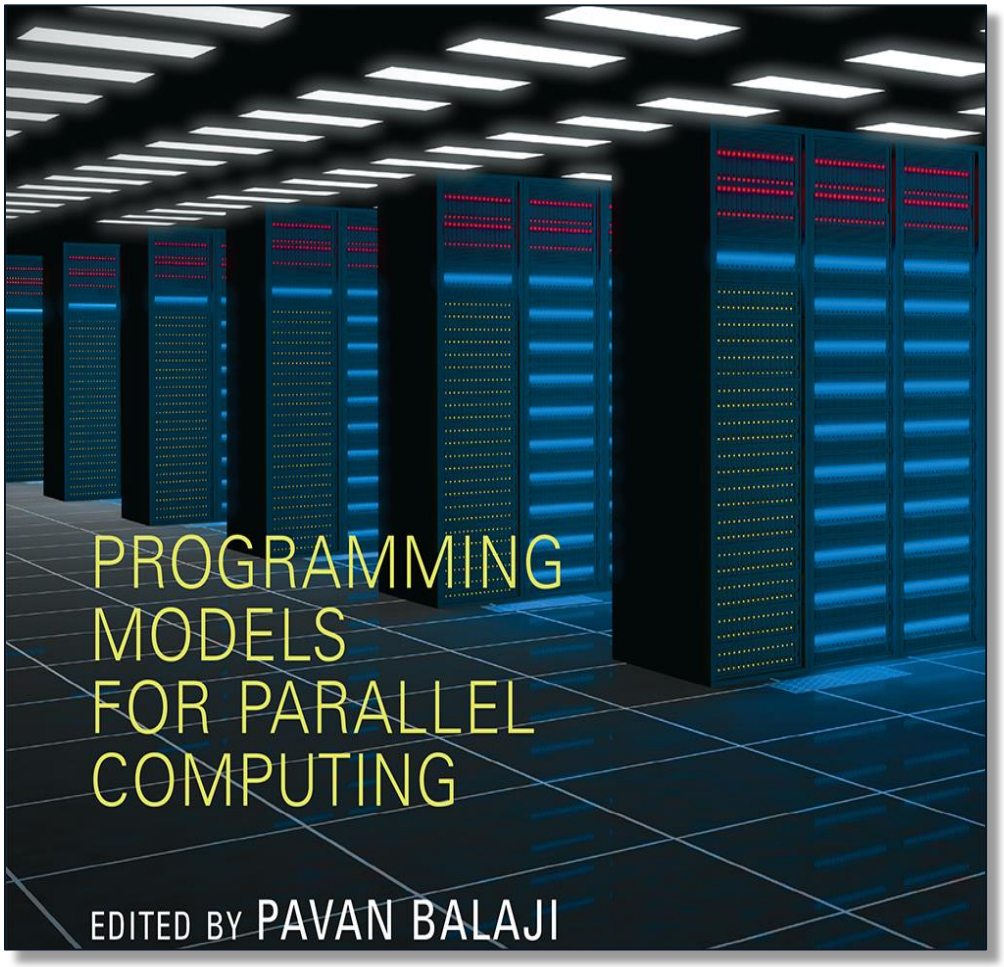


Advanced MPI, including MPI-3

New Book on Parallel Programming Models

Edited by Pavan Balaji

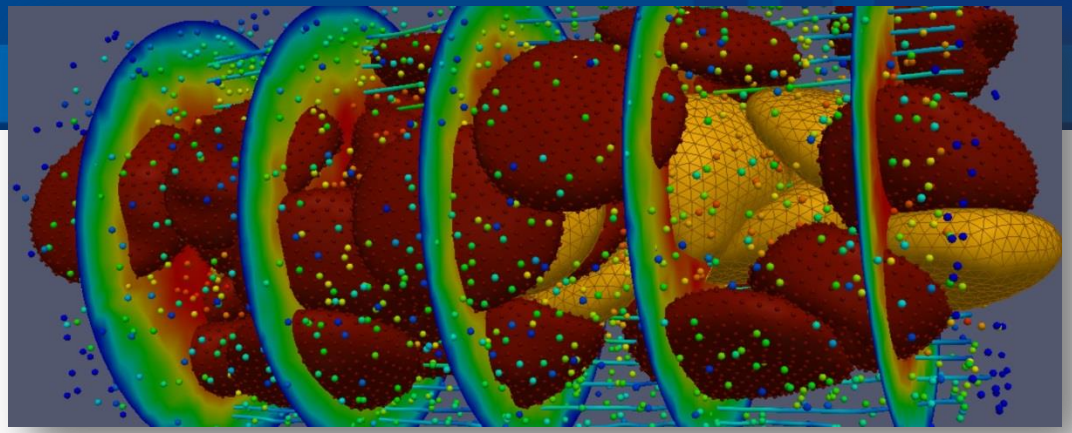
- ***MPI:*** W. Gropp and R. Thakur
- ***GASNet:*** P. Hargrove
- ***OpenSHMEM:*** J. Kuehn and S. Poole
- ***UPC:*** K. Yelick and Y. Zheng
- ***Global Arrays:*** S. Krishnamoorthy, J. Daily, A. Vishnu, and B. Palmer
- ***Chapel:*** B. Chamberlain
- ***Charm++:*** L. Kale, N. Jain, and J. Lifflander
- ***ADLB:*** E. Lusk, R. Butler, and S. Pieper
- ***Scioto:*** J. Dinan
- ***SWIFT:*** T. Armstrong, J. M. Wozniak, M. Wilde, and I. Foster
- ***CnC:*** K. Knobe, M. Burke, and F. Schlimbach
- ***OpenMP:*** B. Chapman, D. Eachempati, and S. Chandrasekaran
- ***Cilk Plus:*** A. Robison and C. Leiserson
- ***Intel TBB:*** A. Kukanov
- ***CUDA:*** W. Hwu and D. Kirk
- ***OpenCL:*** T. Mattson



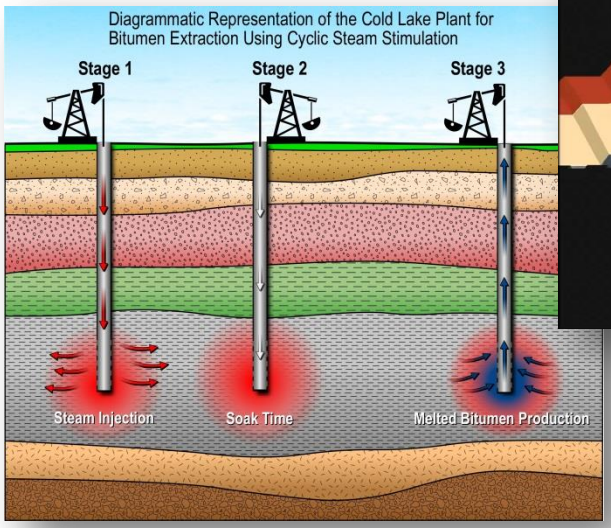
Applications (Science and Engineering)

- MPI is widely used in large scale parallel applications in science and engineering
  - Atmosphere, Earth, Environment
  - Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
  - Bioscience, Biotechnology, Genetics
  - Chemistry, Molecular Sciences
  - Geology, Seismology
  - Mechanical Engineering - from prosthetics to spacecraft
  - Electrical Engineering, Circuit Design, Microelectronics
  - Computer Science, Mathematics

Turbo machinery (Gas turbine/compressor)

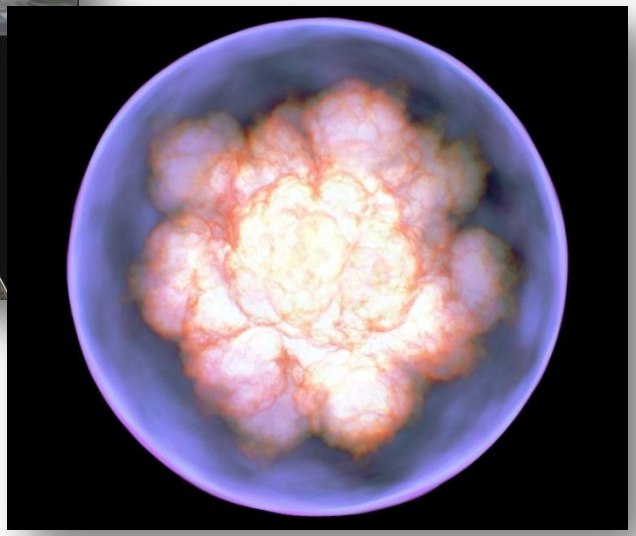


Biology (heart murmur simulation)



Drilling application

Transportation & traffic application



Astrophysics application

Reasons for Using MPI

- **Standardization** - The only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries
- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance
- **Functionality** – Rich set of features
- **Availability** - A variety of implementations are available, both vendor and public domain

Important considerations while using MPI

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

What will be covered in this tutorial

- What is MPI?
- **How to write a simple program in MPI**
- Running your application
- More advanced topics (tomorrow):
  - Non-blocking communication, collective communication, datatypes
  - One-sided communication
  - Hybrid programming with shared memory and accelerators
  - Non-blocking collectives, topologies, and neighborhood collectives

Compiling and Running MPI applications (more details later)

- MPI is a library
  - Applications can be written in C, C++ or Fortran and appropriate calls to MPI can be added where required
- Compilation:
  - Regular applications:
    - `gcc test.c -o test`
  - MPI applications
    - `mpicc test.c -o test`
- Execution:
  - Regular applications
    - `./test`
  - MPI applications (running with 16 processes)
    - `mpiexec –n 16 ./test`

Process Identification

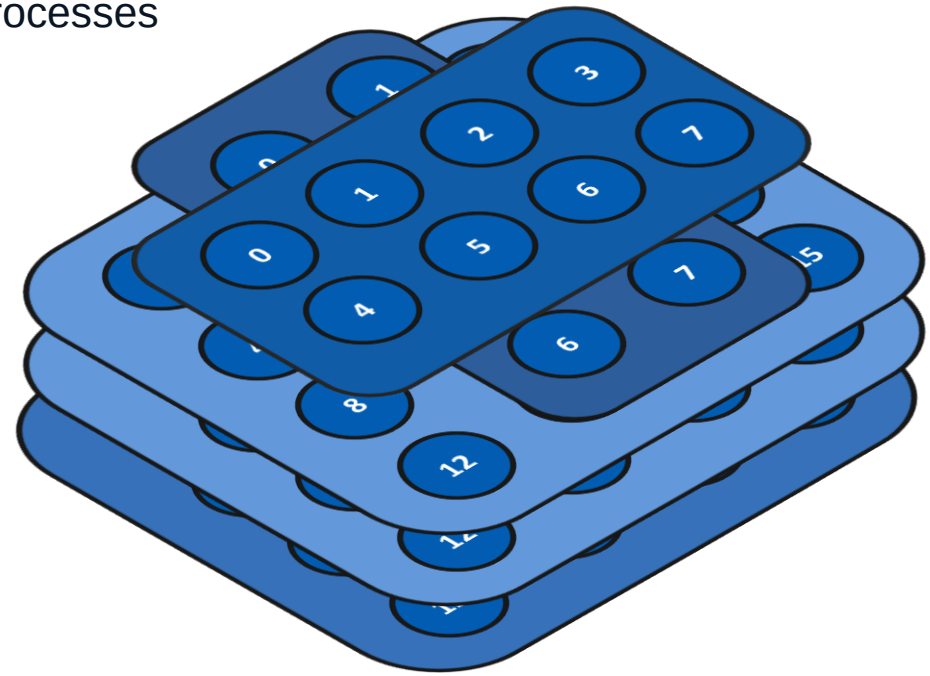
- MPI processes are grouped
  - When an MPI application starts, the group of all processes is initially given a predefined name called `MPI_COMM_WORLD`
  - The same group can have many names
    - But simple programs do not have to worry about multiple names

- A process is identified by a unique number within each communicator, called *rank*
  - For **different communicators**, the **same process can have different ranks**
    - So the meaning of a “rank” is only defined when you specify the comm.

Communicators

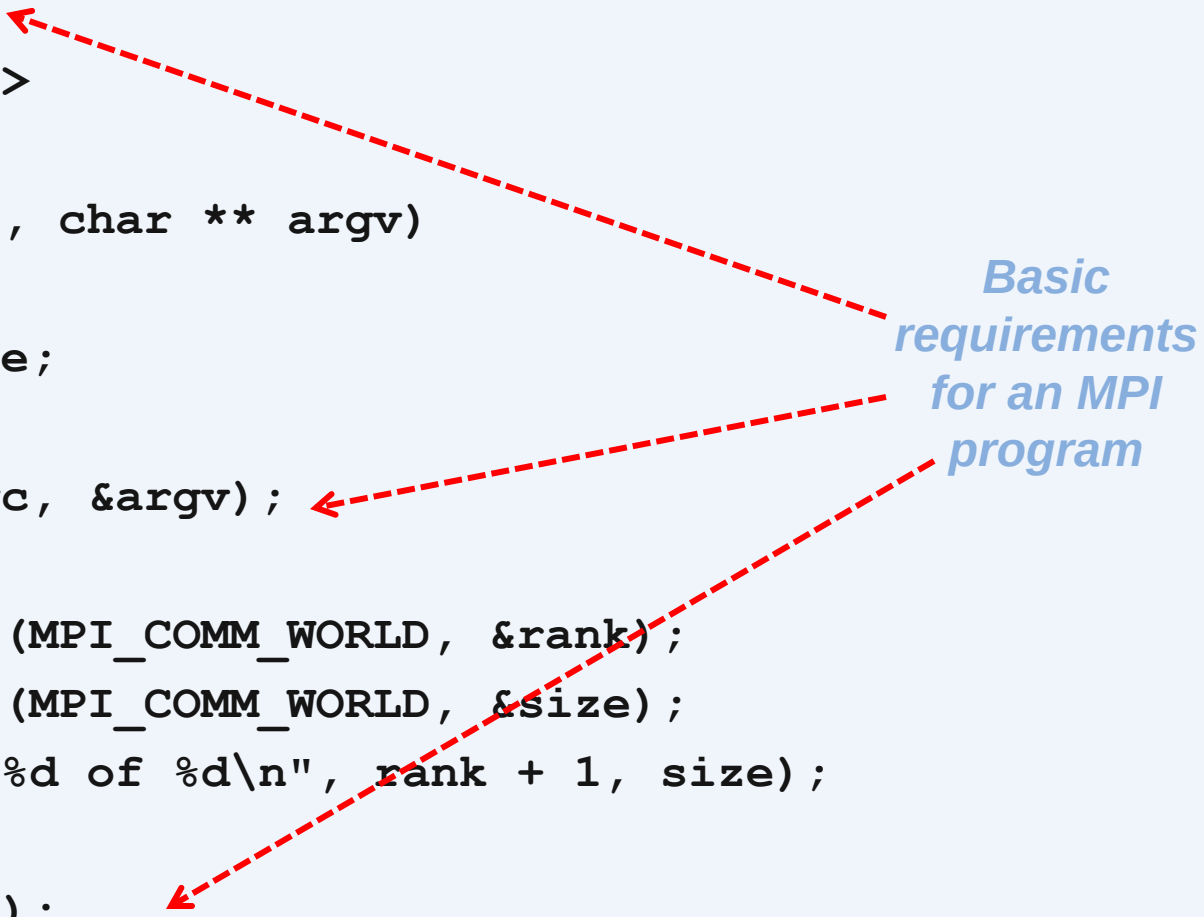
Can be thought of as independent communication layers over a group of processes

Messages in one layer will not affect messages in another



Simple MPI Program Identifying Processes

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

int main(int argc, char ** argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank + 1, size);

    MPI_Finalize();
    return 0;
}
```



Code Example

- *intro-hello.c*

Data Communication

- Data communication in MPI is like email exchange
  - One process sends a copy of the data to another process (or a group of processes), and the other process receives it
- Communication requires the following information:
  - Sender has to know:
    - Whom to send the data to (receiver's process rank)
    - What kind of data to send (100 integers or 200 characters, etc.)
    - A user-defined "tag" for the message (think of it as an email subject; allows the receiver to understand what type of data is being received)
  - Receiver "might" have to know:
    - Who is sending the data (OK if the receiver does not know; in this case sender rank will be `MPI_ANY_SOURCE`, meaning anyone can send)
    - What kind of data is being received (partial information is OK: I might receive *up to* 1000 integers)
    - What the user-defined "tag" of the message is (OK if the receiver does not know; in this case tag will be `MPI_ANY_TAG`)

More Details on Describing Data for Communication

- MPI Datatype is very similar to a C or Fortran datatype
  - `int` → `MPI_INT`
  - `double` → `MPI_DOUBLE`
  - `char` → `MPI_CHAR`
- More complex datatypes are also possible:
  - E.g., you can create a structure datatype that comprises other datatypes → a char, an int and a double.
  - Or, a vector datatype for the columns of a matrix
- The "count" in `MPI_SEND` and `MPI_RECV` refers to how many **datatype elements** should be communicated

MPI Basic (Blocking) Send

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

- The message buffer is described by (**buf, count, datatype**)
- The target process is specified by **dest** and **comm**
  - **dest:** rank of the target process in the **comm** communicator
- **tag** is a user-defined "type" for the message
- When this function returns, the data has been delivered to the system and the buffer can be reused
  - The message may not have been received by the target process

MPI Basic (Blocking) Receive

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

- Waits until a matching (on **source**, **tag**, **comm**) message is received from the system, and the buffer can be used.
- **source** is rank in communicator **comm**, or **MPI_ANY_SOURCE**.
- Receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error.
- **status** contains further information:
  - Who sent the message (can be used if you used **MPI_ANY_SOURCE**)
  - How much data was actually received
  - What tag was used with the message (can be used if you used **MPI_ANY_TAG**)
  - **MPI_STATUS_IGNORE** can be used if we don't need any additional information

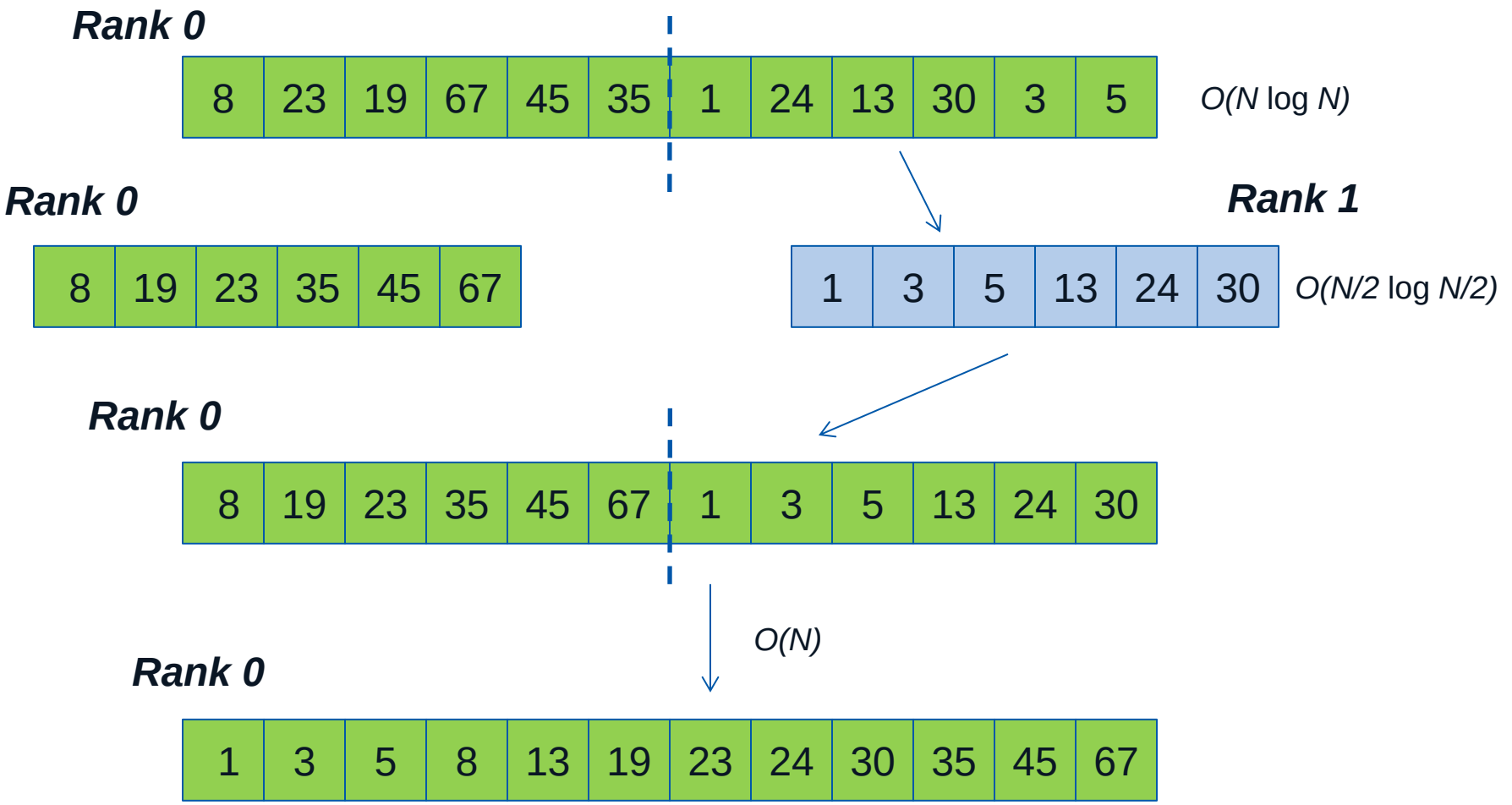
Simple Communication in MPI

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

int main(int argc, char ** argv)
{
    int rank, data[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

    MPI_Finalize();
    return 0;
}
```

Code Example

- *intro-sendrecv.c*

Parallel Sort using MPI Send/Recv



Parallel Sort using MPI Send/Recv (contd.)

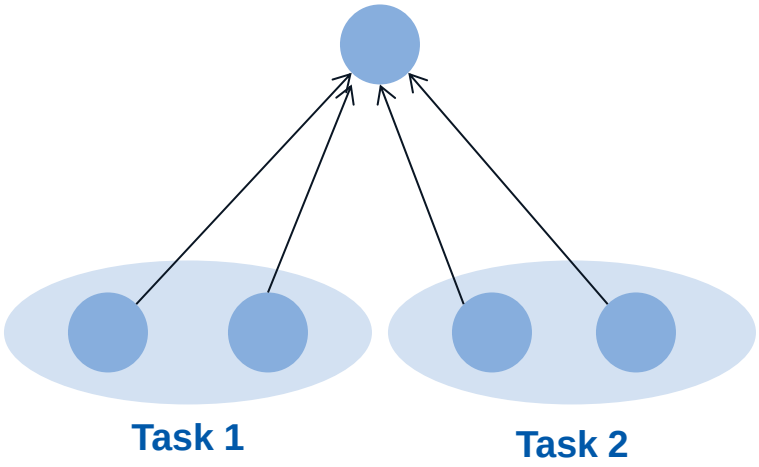
```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
    int rank, a[1000], b[500];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Send(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD);
        sort(a, 500);
        MPI_Recv(b, 500, MPI_INT, 1, 0, MPI_COMM_WORLD,
                                MPI_STATUS_IGNORE);

        /* Serial: Merge array b and sorted part of array a */
    }
    else if (rank == 1) {
        MPI_Recv(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD,
                                MPI_STATUS_IGNORE);
        sort(b, 500);
        MPI_Send(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize(); return 0;
}
```

- The status object is used after completion of a receive to find the actual length, source, and tag of a message
- Status object is MPI-defined type and provides information about:
  - The source process for the message (**`status.MPI_SOURCE`**)
  - The message tag (**`status.MPI_TAG`**)
  - Error status (**`status.MPI_ERROR`**)
- The number of elements received is given by:

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

`status`	return status of receive operation (status)
`datatype`	datatype of each receive buffer element (handle)
`count`	number of received elements (integer) (OUT)

Using the “status” field



- Each “worker process” computes some task (maximum 100 elements) and sends it to the “master” process together with its group number
- The “tag” field can be used to represent the task
- Data count is not fixed (maximum 100 elements)
- Order in which workers send output to master is not fixed (different workers = different source ranks, and different tasks = different tags)

Using the “status” field (contd.)

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

int main(int argc, char ** argv)
{
    [...snip...]

    if (rank != 0)  /* worker process */
        MPI_Send(data, /*0..100*/, MPI_INT, 0, task_id,
                 MPI_COMM_WORLD);
    else {  /* master process */
        for (i = 0; i < size - 1; i++) {
            MPI_Recv(data, 100, MPI_INT, MPI_ANY_SOURCE,
                     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_INT, &count);
            printf("worker ID: %d; task ID: %d; count: %d\n",
                   status.MPI_SOURCE, status.MPI_TAG, count);
        }
    }

    [...snip...]
}
```

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **MPI_Init – initialize the MPI library (must be the first routine called)**
  - **MPI_Comm_size - get the size of a communicator**
  - **MPI_Comm_rank – get the rank of the calling process in the communicator**
  - **MPI_Send – send a message to another process**
  - **MPI_Recv – send a message to another process**
  - **MPI_Finalize – clean up all MPI state (must be the last MPI function called by a process)**

- For better performance, however, you need to use other MPI features

What will be covered in this tutorial

- What is MPI?
- How to write a simple program in MPI
- **Running your application**
- More advanced topics:
  - Non-blocking communication, collective communication, datatypes
  - One-sided communication
  - Hybrid programming with shared memory and accelerators
  - Non-blocking collectives, topologies, and neighborhood collectives

Compiling MPI programs

- Compilation Wrappers
  - For C programs: **`mpicc test.c -o test`**
  - For C++ programs: **`mpicxx test.cpp -o test`**
  - For Fortran programs: **`mpifort test.f90 -o test`**
- You can link other libraries are required too
  - To link to a math library: **`mpicc test.c -o test -lm`**
- You can just assume that “mpicc” and friends have replaced your regular compilers (gcc, gfortran, etc.)

Running MPI programs (no resource manager)

- Launch 16 processes on the local node:
  - **`mpiexec -n 16 ./test`**
- Launch 16 processes on 4 nodes (each has 4 cores)
  - **`mpiexec -hosts h1:4,h2:4,h3:4,h4:4 -n 16 ./test`**
    - Runs the first four processes on h1, the next four on h2, etc.
  - **`mpiexec -hosts h1,h2,h3,h4 -n 16 ./test`**
    - Runs the first process on h1, the second on h2, etc., and wraps around
    - So, h1 will have the 1st, 5th, 9th and 13th processes
- If there are many nodes, it might be easier to create a host file
  - **`cat hf`**

    `h1:4`

    `h2:2`
  - **`mpiexec -hostfile hf -n 16 ./test`**

Interaction with Resource Managers

- Resource managers such as SGE, PBS, SLURM or Loadleveler are common in many managed clusters
- For example with SLURM, you can create a script such as:

```
#! /bin/bash
#SBATCH --ntasks=4
#SBATCH --tasks-per-node=2
#SBATCH --cpus-per-task=1
srun ./test
```

- Job can be submitted as: `sbatch test.sh`
  - "srun" will automatically get from SLURM the info of tasks, nodes and CPUs and will start the appropriate number of MPI ranks.
- The usage is similar for other resource managers

Debugging MPI programs

- Parallel debugging is trickier than debugging serial codes
  - Many processes computing; getting the state of one failed process is usually hard
  - Commercial parallel debuggers such as Totalview and DDT
- E.g., with totalview:
  - `totalview -args mpiexec -n 6 ./test`
- With gdb on one process:
  - `mpiexec -n 4 ./test : -n 1 gdb ./test : -n 1 ./test`
  - Launches the 5th process under “gdb” and all other processes normally

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
marc.jorda@bsc.es, antonio.pena@bsc.es