12.950

Parallel Programming for Multicore Machines using OpenMP and MPI

Dr. C. Evangelinos, MIT

Course Syllabus

- Day 2 (OpenMP wrapup and MPI Pt2Pt):
  - EC2 cluster and Vmware image demo
  - Homework discussion
  - OpenMP 3.0 enhancements
  - Fundamentals of Distributed Memory Programming
  - MPI concepts
  - Blocking Point to Point Communications

OpenMP 3.0

- In draft form since October 2007, finalized in 2008.
- Support for tasking
- Addition of loop collapse directive
- Enhanced loop schedules SCHEDULE(AUTO)
- Improved nested parallelism support
- Autoscoping: DEFAULT(AUTO)

Tasking

- Most major new enhancement to OpenMP, allows for expressing irregular parallelism.
  - Similar to Cilc (MIT) and other vendor (C#, TBB) efforts
  - Spawn a new task and run it or queue it for running

  #pragma omp task [clauses]

  C$OMP TASK [CLAUSES]

  C$OMP END TASK

  - The clauses are if, untied, default, private, firstprivate and shared.

A linked list with tasks

```
#pragma omp parallel
{
    #pragma omp single private(p)
    {
        p = listhead ;
        while (p) {
            #pragma omp task
            process (p)
            p=next (p) ;
        }
    }
}
```

taskwait

- #pragma omp taskwait
- C$OMP TASK WAIT
  - Wait for all tasks spawned by the current task (children)
  - Partial synchronization compared to a barrier.

Better nested parallelism

- Loop collapsing

```
#pragma omp collapse(2)
    for (i = 0; i < N; ++i)
        for (j = 0; j < M; ++j)
            do_work();
```

- More routines to set and discover the nested parallel structure and control the nesting environment
- Per task internal control variables controlling nested parallelism.

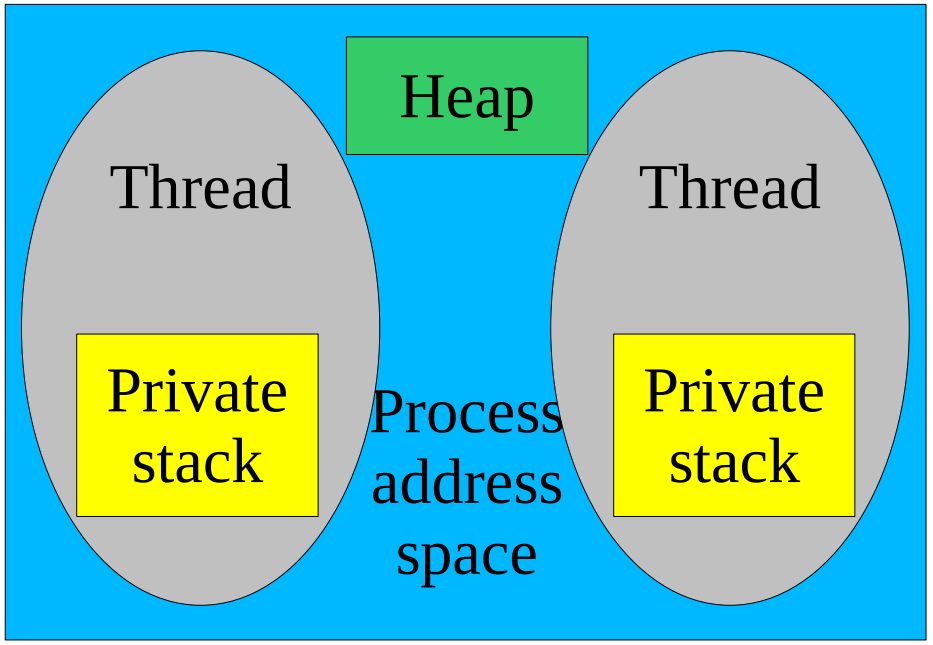
Distributed Memory Programming using MPI

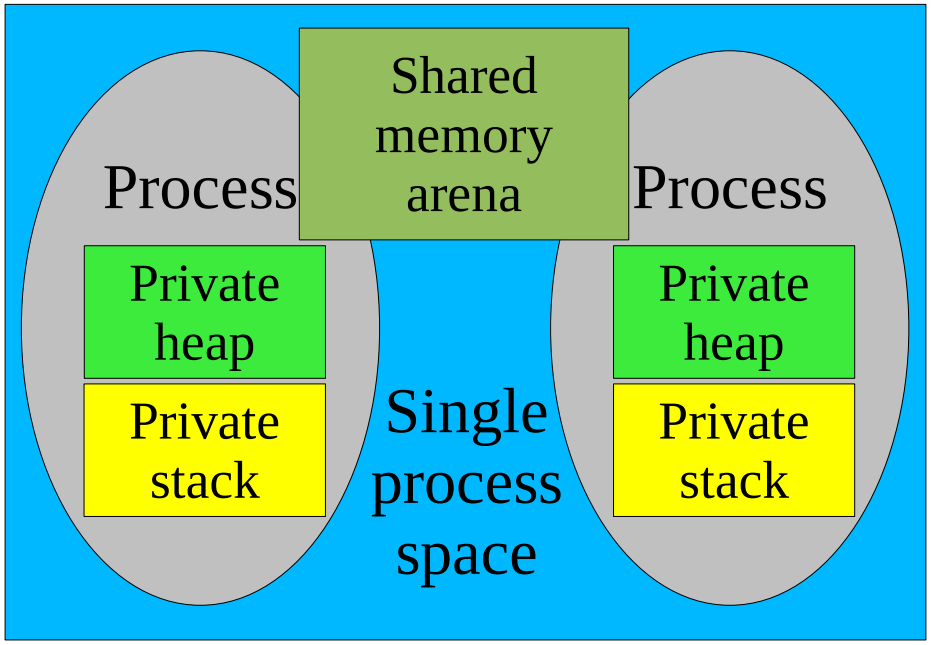
Acknowledgments

- MPI Forum
- Joel Malard, Alan Simpson (EPCC)
- Rolf Rabenseifner, Traugott Streicher (HLRS)
- The MPICH team at ANL
- The LAM/MPI team at Indiana University

Shared Memory Programming

- Under the assumption of a single address space one uses multiple control streams (threads, processes) to operate on both private and shared data.
- Shared data: synchronization, communication, work
  - In shared arena/mmaped file (multiple processes)
  - In the heap of the process address space (multiple threads)





Distributed Memory Programming

- Multiple independent control streams (processes in general) operating on separate data and coordinating by communicating data and information.
- Most common communication method: Message Passing
- Multiple Program Multiple Data (MPMD) (eg. master-slave)
  - Single Program Multiple Data (SPMD) (eg. data parallel)

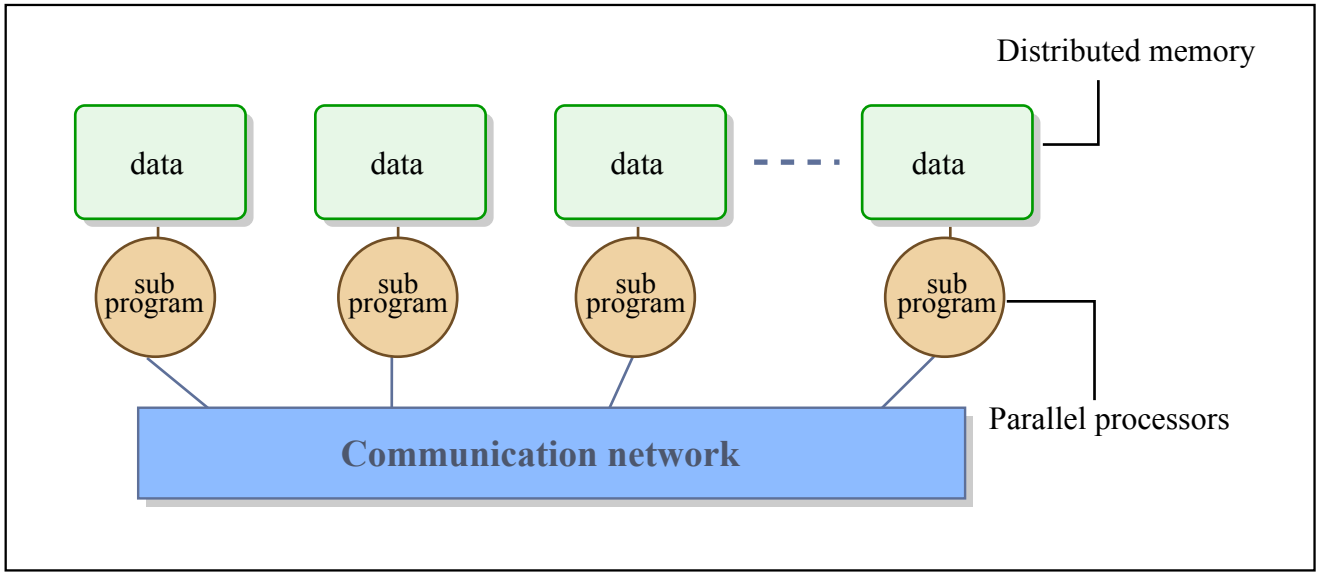


Figure by MIT OpenCourseWare.

SPMD & MPMD

- SPMD can be considered a special case of MPMD
- MPMD can be implemented as a SPMD program which depending on some criterion executes the relevant constituent code

```
switch (myid) {
    case 0:
           run_prog1();
           breaksw;
    case 1:
           run_prog2();
           breaksw;}
```

- All data is private to each process. Some of it however may map to the same physical variables as the corresponding private data in other processes (eg. shadow cells, boundary data):
  - Communication is required to enforce consistency
  - Communication can be direct or collective

MPI: Message Passing Interface

- A standard API for message passing communication and process information lookup, registration, grouping and creation of new message datatypes.
  - Point to point comms: ([non]blocking, [a]synchronous)
  - Collective comms: one-many, many-one, many-many
- Code parallelization cannot be incremental
- Supports coarse level parallelism *and parallel I/O*
- Fortran 77 and C support, C++/Fortran90 in MPI-2
- *Very large* API (128), MPI-2 document adds quite a lot (152). Most users do not use but a fraction of it.
- Performance oriented standard.

MPI history

- Many message passing libraries in the past:
  - TCGMSG, P4, PARMACS, EXPRESS, NX, MPL, **PVM**
  - Vendor specific, research code, application driven
- 1992-94 the Message Passing Forum defines a standard for message passing (targeting MPPs)
- Evolving standards process:
  - 1994: MPI 1.0: Basic comms, Fortran 77 & C bindings
  - 1995: MPI 1.1: errata and clarifications
  - 1997: MPI 2.0: single-sided comms, I/O, process creation, Fortran 90 and C++ bindings, further clarifications, many other things. Includes MPI-1.2.
  - 2008: MPI 1.3, 2.1: combine 1.3 and 2.0, corrections & clarrifications
  - 2009: MPI 2.2: corrections & clarrifications
  - MPI 3.0 standardization in progress.

The Genealogy of MPI

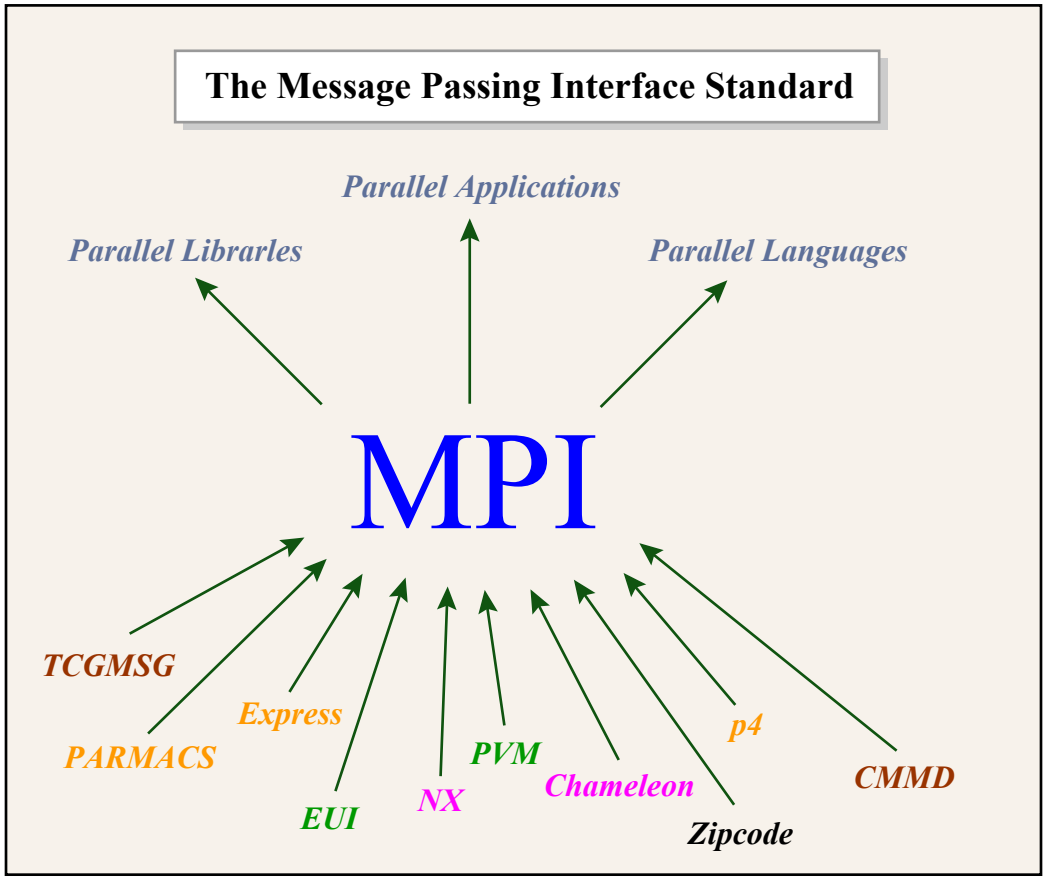


Figure by MIT OpenCourseWare.

MPI Basics

- **mpi.h** and **mpif.h** include files for C/C++ and Fortran

```
error = MPI_Xxxxx(argument, ...);

call mpi_xxxxx(argument, ..., error)
```

- Notice that all routines have to return an error value
- mpi module for Fortran90
- MPI_ namespace reserved for MPI
- MPI:: namespace for C++
- In C/C++ (in)out arguments passed as pointers
- Always start with MPI_Init() and end with MPI_Finalize() *(or MPI_Abort())*.Both calls need to be made by all processes

Minimal MPI subset

- There is a minimal subset of MPI that allows users to write functional parallel programs without learning all hundreds of MPI functions and going through hoops:
  - MPI_Init()/MPI_Finalize()/MPI_Abort()
  - MPI_COMM_WORLD
  - MPI_Comm_size()/MPI_Comm_rank()
  - MPI_Send()/MPI_Recv()
  - MPI_Isend()/MPI_Irecv()/MPI_Wait()
- You *might be able* actually to stick to the 6 functions in red if your message sizes are small enough
- For performance as well as code compactness reasons you will need to at least use collective comms.

Initialization

- int MPI_Init(int *argc, char ***argv)
- MPI_INIT(ier), integer ier
- Called before any other MPI call. Only
  - MPI_Initialized(int *flag)
  - MPI_INITIALIZED(flag, ierror), logical flag

  is allowed before it.
- It initializes the MPI environment for the process.

Communicators and handles

- A communicator is an ordered set of processes, that remains constant from its creation until its destruction
- All MPI processes form the MPI_COMM_WORLD communicator
- Users can create their own (subset) communicators
- MPI_COMM_WORLD gets created at the very beginning and is a handle defined in the include files
- handles are predefined constants in the include files, that are of integer type for Fortran and special typedefs for C/C++
- Handles describe MPI objects and datatypes

Communicator size

- MPI_Comm_size(MPI_Comm comm, int *size)
- MPI_COMM_SIZE(comm, size, ier)
  - all arguments are integers, henceforth not repeated
- Returns the size of the communicator set.

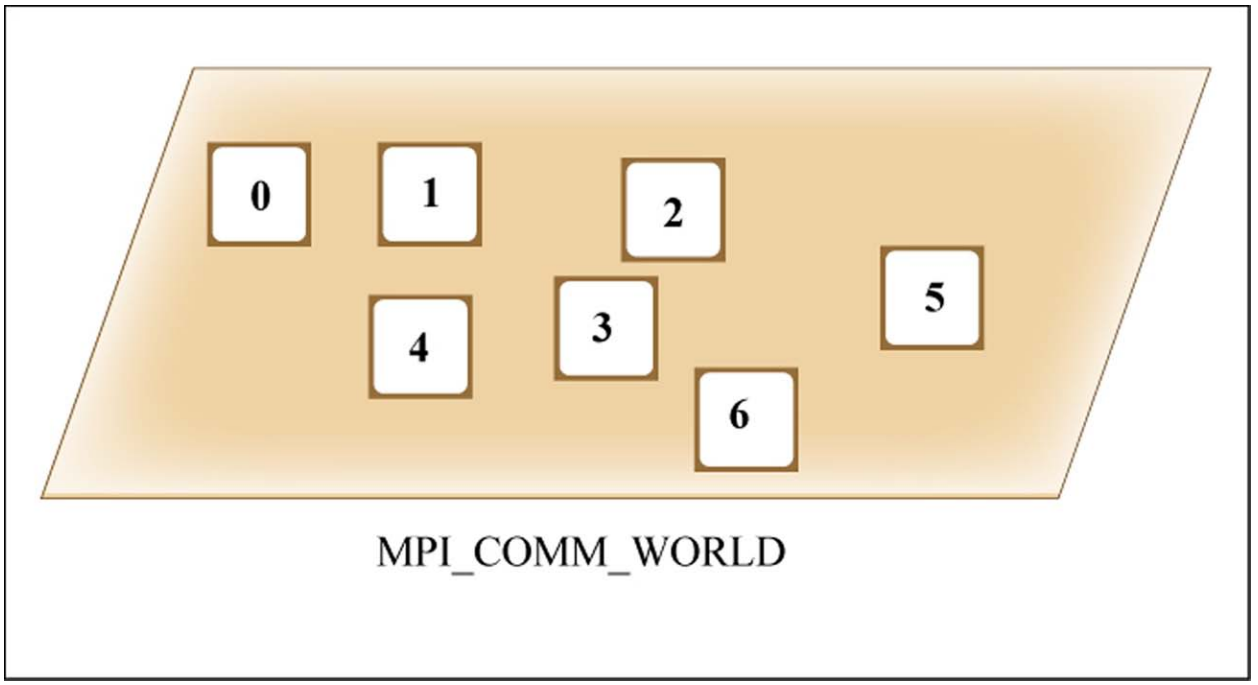
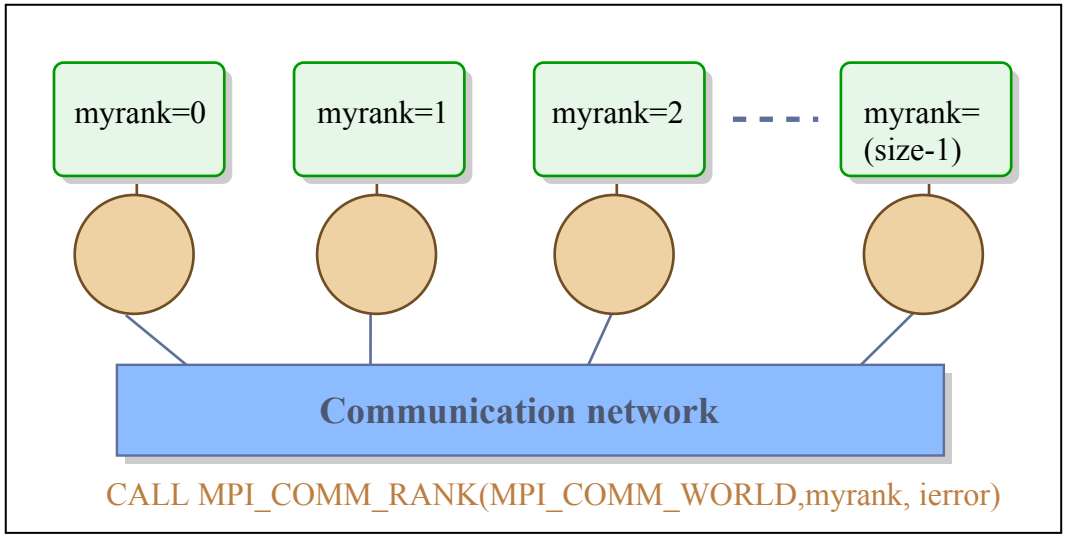


Figure by MIT OpenCourseWare.

Process rank

- MPI_Comm_rank(MPI_Comm comm, int *rank)
- MPI_COMM_RANK(comm, rank, ier)
- Returns the rank of the process in the communicator
- If communicators A and B are different and the process belongs to both of them, its rank in one of them is unrelated to its rank in the other
- Its value is between 0 and (size-1)



Exiting

- For a graceful exit all processes need to call **last**:
  - MPI_Finalize()
  - MPI_FINALIZE(ierror)
- If a process catches an error that cannot be corrected, a user can call:
  - MPI_Abort(MPI_Comm comm, int errorcode)
  - MPI_ABORT(comm, errorcode, ier)
  - This will make a best attempt to abort all other tasks in the communicator set. Currently works only for MPI_COMM_WORLD. The errorcode is usually the return value of the parallel executable.

The usual "hello world"

```
      IMPLICIT NONE
      INCLUDE "mpif.h"
      INTEGER ierror, rank, size
      CALL MPI_INIT(ierror)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank,
ierror)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
      IF (rank .EQ. 0) THEN
        WRITE(*,*) 'I am process', rank, ' out of', size,
     &             ': Hello world!'
      ELSE
        WRITE(*,*) 'I am process', rank, ' out of', size
      END IF
      CALL MPI_FINALIZE(ierror)
      END
```

Communications

- So far we have not explicitly exchanged any information between processes.
- Communications can be between two processes *(point to point)* or between a group of processes *(collective)*
- Communications involve **arrays** of data organized as MPI datatypes:
  - Datatypes can be predefined with a mapping to host language basic datatypes
  - They can also be user-defined, as structures of basic or other user-defined datatypes
  - User defined datatypes hide the complexity of the data exchange and leave it to the MPI library to optimize it

MPI Basic Datatypes

MPI Datatypes for C	C datatypes	MPI Datatypes for Fortr	Fortran datatypes
MPI_CHAR	signed char		
MPI_SHORT	signed short		
MPI_INT	signed int	MPI_INTEGER	INTEGER
MPI_LONG	signed long		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short		
MPI_UNSIGNED_INT	unsigned int		
MPI_UNSIGNED_LONG	unsigned long		
MPI_FLOAT	float	MPI_REAL	REAL
MPI_DOUBLE	double	MPI_DOUBLE_PRECIS	DOUBLE PRECISION
MPI_LONG_DOUBLE	long double		
		MPI_CHAR	CHARACTER(1)
		MPI_LOGICAL	LOGICAL
		MPI_COMPLEX	COMPLEX
MPI_BYTE		MPI_BYTE	
MPI_PACKED		MPI_PACKED	

Messages

- The purpose is the "exchange" of information, much the mail system. Thus every message *(document)* has:
- A sender address (rank, *like a business address*)
- A message location (starting address, *like the document's location*)
- A message datatype (what is being sent)
- A message size (how big is it in datatype units)
- A message tag
- A destination address (rank, *like another business address*)
- A destination location (*that cabinet in the office of so-and-so*)
- Compatible datatype and size combo in order to fit
- Matching tag

Point to Point Comms

- Blocking comms: Block until completed (*send stuff on your own*)
- Non-blocking comms: Return without waiting for completion (*give them to someone else*)
- Forms of Sends:
  - Synchronous: message gets sent only when it is known that someone is already waiting at the other end (*think fax*)
  - Buffered: message gets sent and if someone is waiting for it so be it; otherwise it gets saved in a temporary buffer until someone retrieves it. (*think mail*)
  - Ready: Like synchronous, only there is no ack that there is a matching receive at the other end, just a programmer's assumption! **(Use it with extreme care)**

Point to Point messages

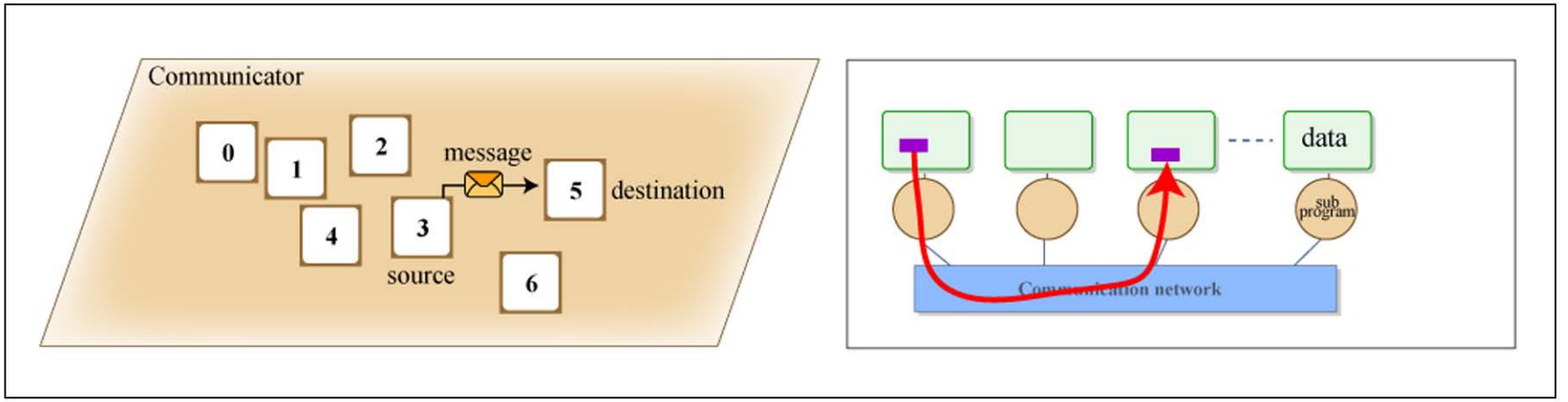


Figure by MIT OpenCourseWare.

Synchronous vs. Asynchronous

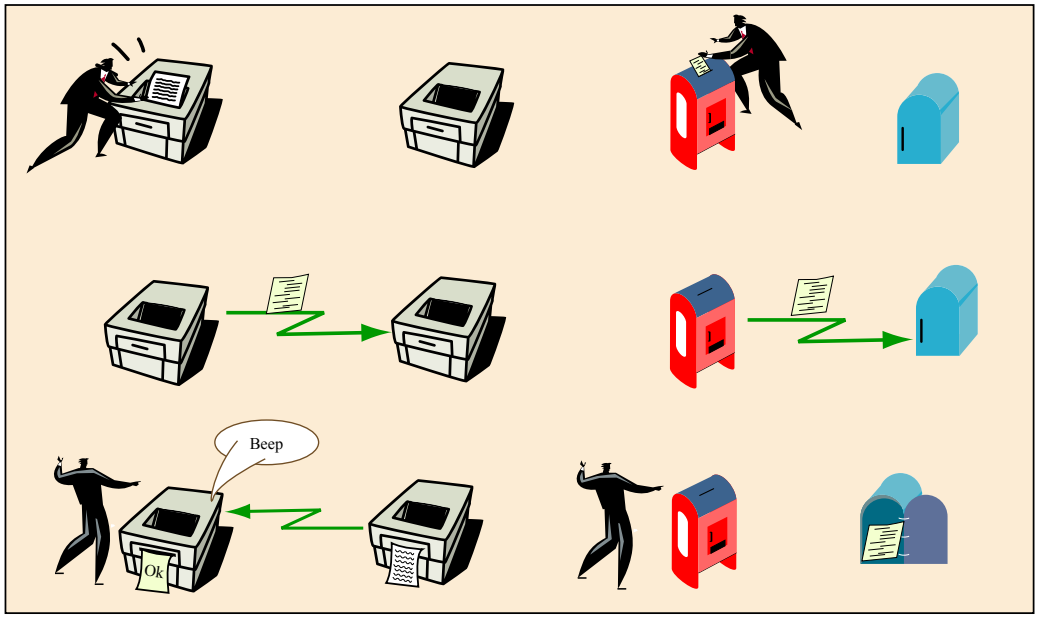


Figure by MIT OpenCourseWare.

MPI blocking standard send

- MPI_Send(void *buf, int cnt, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
- MPI_SEND(buf, cnt, type, dest, tag, comm, ier)
  - buf is an array of variable type in Fortran...
- buf is the starting address of the array
- cnt is its length
- type is its MPI datatype
- comm is the communicator context
- int is the rank of the destination process in comm
- tag is an extra distinguishing number, like a note

Other blocking sends

- MPI_Ssend: Synchronous send
  - The sender notifies the receiver; after the matching receive is posted the receiver acks back and the sender sends the message.
- MPI_Bsend: Buffered (asynchronous) send
  - The sender notifies the receiver and the message is either buffered on the sender side or the receiver side according to size until a matching receive forces a network transfer or a local copy respectively.
- MPI_Rsend: Ready send
  - The receiver is notified and the data starts getting sent immediately following that. Use at your own peril!

Things to consider

- Depending on the MPI implementation, MPI_Send may behave as either MPI_Ssend or MPI_Bsend.
  - Usually buffered for small messages, synchronous for larger ones. Very small messages may piggypack on the initial notification sent to the receiver. The switchover point can be tunable (see P4_SOCKBUFSIZE for MPICH)
  - For buffered sends it uses an internal system buffer
- MPI_Bsend may use a system buffer but cannot be guaranteed to work without a user-specified buffer setup using
  - MPI_Buffer_attach(void *buffer, int size)
- MPI_Ssend can lead to deadlocks

Blocking send performance

- Synchronous sends offer the highest asymptotic data rate (AKA bandwidth) but the startup cost (latency) is very high, and they run the risk of deadlock.
- Buffered sends offer the lowest latency but:
  - suffer from buffer management complications
  - have bandwidth problems because of the extra copies and system calls
- Ready sends *should* offer the best of both worlds but are so prone to cause trouble they are to be avoided!
- Standard sends are usually the ones that are most carefully optimized by the implementors. For large message sizes they can always deadlock.

MPI blocking receive

- Irrespective of the send employed, there is one blocking receive operation on the other end
- MPI_Recv(void *buf, int cnt, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Status *stat)
- MPI_RECV(buf, cnt, type, src, tag, comm, stat, ier)
- buf is the starting address of the target array
- cnt is its length, type is its MPI datatype
- comm is the communicator context
- src is the rank of the source process in comm
- tag needs to match along with src and comm

Issues with MPI_Recv

- Upon completion of the call the message is stored in the target array and can safely be used
  - However if buf had the wrong datatype/size combination the message was probably either truncated or padded with garbage.
  - Message envelope information (source, tag along with information that can give the size using MPI_Get_count) is stored in the MPI_Status (structure/integer array)
  - Along with error code can be used for debugging purposes
- MPI_ANY_SOURCE and MPI_ANY_TAG are wildcards for matching receives *(less work)*
- Fortran status is int array of size MPI_STATUS_SIZE

Message passing restrictions

- Order is preserved. For a given channel (sender, receiver, communicator) message order is enforced:
  - If P sends to Q, messages *sa* and *sb* in that order, that is the order they will be received at B, even if *sa* is a medium message sent with MPI_Bsend and *sb* is a small message sent with MPI_Send. Messages do not overtake each other.
  - If the corresponding receives *ra* and *rb* match both messages (same tag) again the receives are done in order of arrival.
  - This is actually a performance drawback for MPI *but* helps avoid major programming errors.

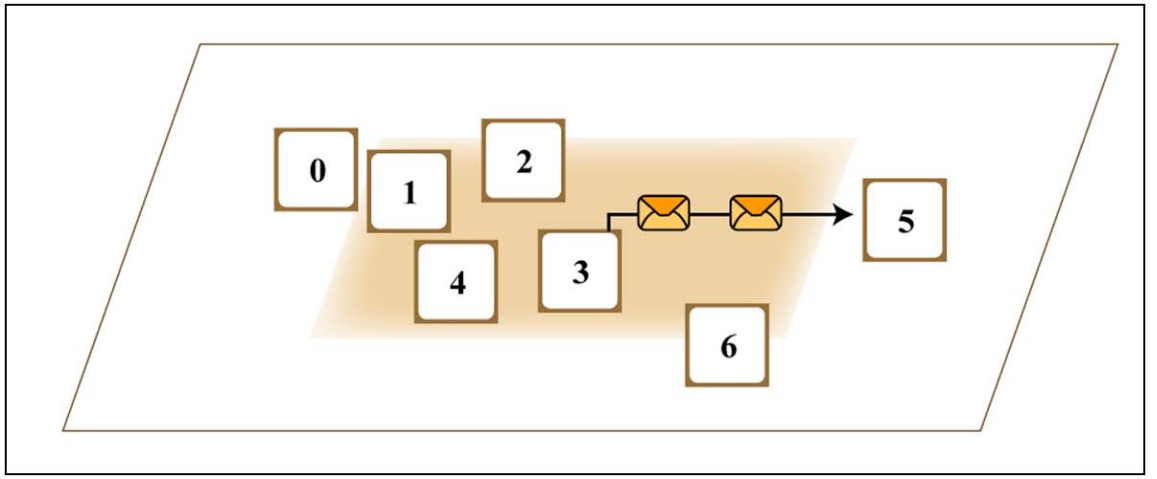


Figure by MIT OpenCourseWare.

Ping-Pong

- Test send-receive pairs in the simplest of scenarios:
  - A pair of processes exchanging messages back and forth.
  - Use double precision utility function MPI_Wtime to get wallclock time in seconds as a benchmark
  - The variations of the benchmark allow us to measure latency, the time for a very small message to be exchanged, as well as bandwidth, the rate at which a very large message gets sent out.

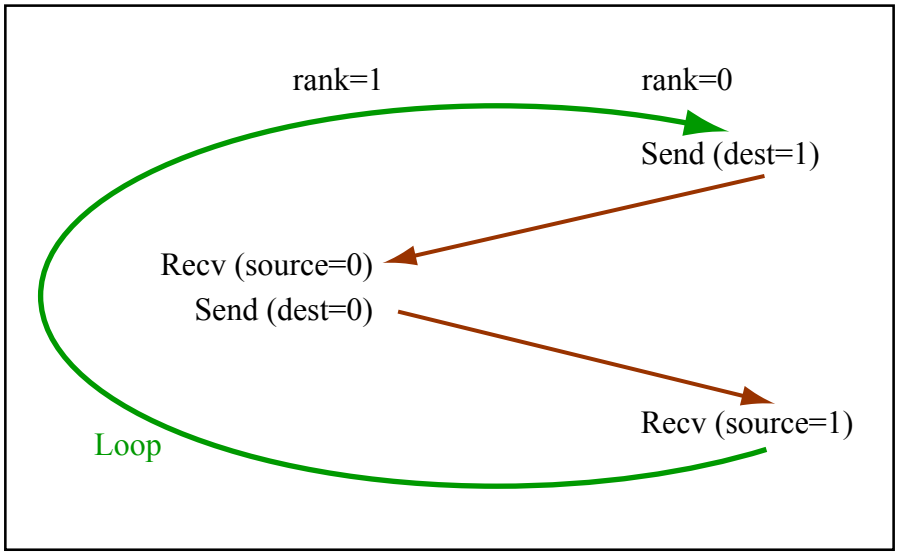


Figure by MIT OpenCourseWare.

```
if (myrank == 0) {
    MPI_Send(..,1,..);
    MPI_Recv(..,1,..);
} else {
    MPI_Recv(..,0,..);
    MPI_Send(..,0,..);
}
```

12.950 Parallel Programming for Multicore Machines Using OpenMP and MPI

IAP 2010