COL380

Introduction to

Parallel & Distributed Programming

- Shared Memory model
- Message passing model
- Task-based model
- Work-queue model
- Stream processing model
- Map-reduce model
- Client-server model

Shared Memory

Multiple Threads of Execution

OP operands	W	OP operands
OP operands	R	OP operands
OP operands		OP operands
OP operands	R	OP operands
OP operands		OP operands
OP operands		OP operands
OP operands		OP operands
OP operands		OP operands

Shared Variable Store

(May not share actual memory)

- How are ‘thread’ instantiated?
- Execution and memory model
- How do the interact?

compiler,
runtime..

Separate from HW
(Who does the execution?)

Message Passing

Multiple Threads of Execution

OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
Address space	Address space

(Could share actual memory)

- How are ‘thread’ instantiated?
- Execution and memory model
- How do the interact?

Separate from HW
(Who does the execution?)

- Inter-process Communication
  - ➡ System-call, Network infrastructure (Ethernet, Infiniband, Custom-made)
- Processor-local memory
  - ➡ Coordination with ‘network’ memory
- Access to other threads’ data through explicit Send/Recv instructions
  - ➡ Implicit synchronization semantics
  - ➡ Can double as inter-process synchronization

- Multiple "threads" of execution
  - Do not share address space (Processes)
  - Each process may further have multiple threads of control that share memory with each other

Shared Memory Model

Star a master thread
Create Sharing threads:
Process(sharedInput, myID)

Message Passing Model

Start a master thread
Create 'remote' threads
Send command/input
Collect and present results

Recv command
Process command
Send results

Shared Memory vs Message Passing

- Shared memory programs ‘appear’ similar to sequential programs (but have hidden synchronization complexity)
  - ➡ Suitable for a single unified memory system (preferable with large memory)
  - ➡ Hard to scale core/memory on a single system
- Message passing programs require coordination: a receive in one thread must match a send in another
  - ➡ Good scalability through multiple systems (preferably with a fast network)
    - Cost effectiveness, with off-the-shelf processor/ network
  - ➡ The only option for distributed applications

Example with MPI

```
#include "mpi.h"     /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);                      // start MPI
  int nProcs, myRank, dat[2] = {5,6};
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD, &nProcs); // Group size
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank); // get my rank
  If(myRank == 0)
      MPI_Send(dat, 2, MPI_INT, nProcs-1, 11, MPI_COMM_WORLD);
  If(myRank == nProcs-1)
      MPI_Recv(dat, 9, MPI_INT, 0, 11, MPI_COMM_WORLD, &status);
  MPI_Finalize();                              // stop MPI
}
```

Example with MPI

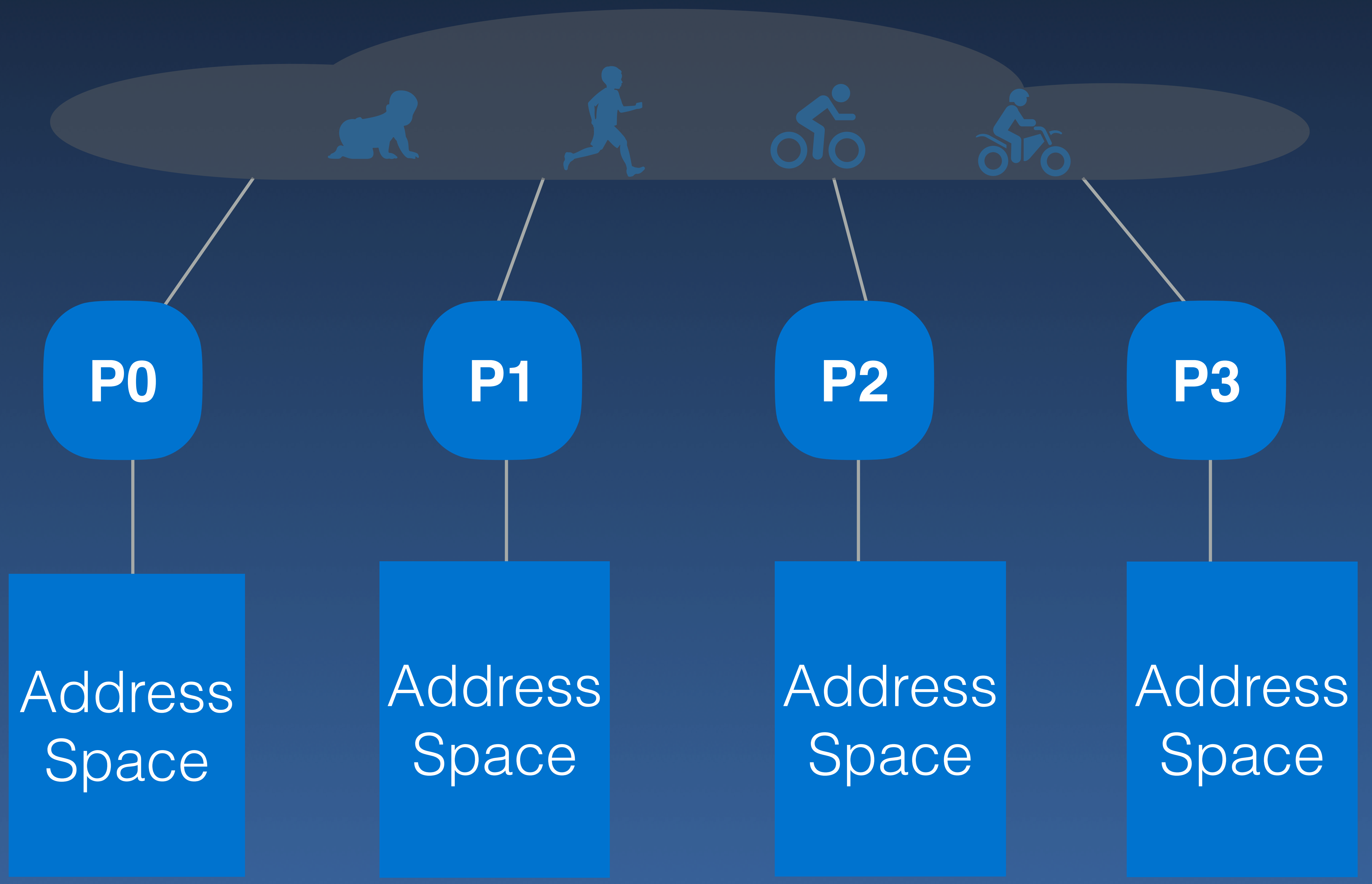
```
#include "mpi.h"      /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);                     // start MPI
  int nProcs, myRank, dat[2] = {5,6};
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_W
  MPI_Comm_rank(MPI_COMM_W
  If(myRank == 0)
      MPI_Send(dat, 2, MPI                                     LD);
  If(myRank == nProcs-1)
      MPI_Recv(dat, 9, MPI_INT, 0, 11, MPI_COMM_WORLD, &status);
  MPI_Finalize();                             // stop MPI
}
```

```
status.MPI_TAG
status.MPI_SOURCE
MPI_Get_count(&status, MPI_INT,  &count);
```

Distributed Memory

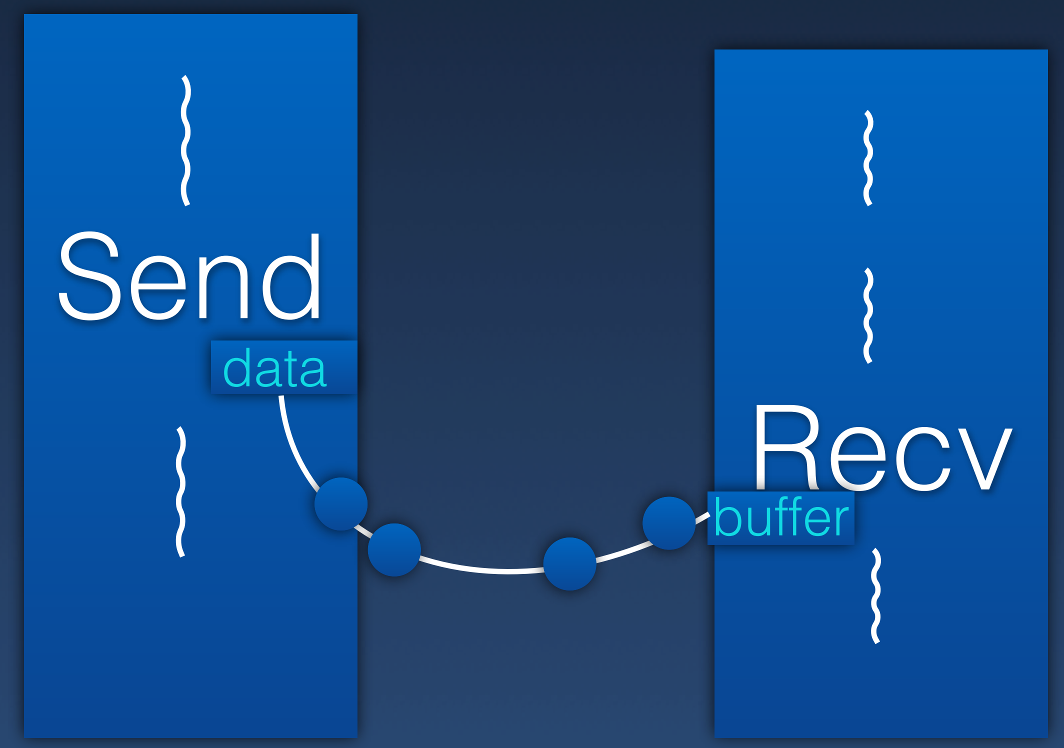


Subodh Kumar


istributed Memo
NIC Buffer
OS Buffer
P0
User Buffers: A[], *B
Many hidden buffers
NIC Buffer
OS Buffer
P1
User Buffers: A[], *B
P2

Message Semantics

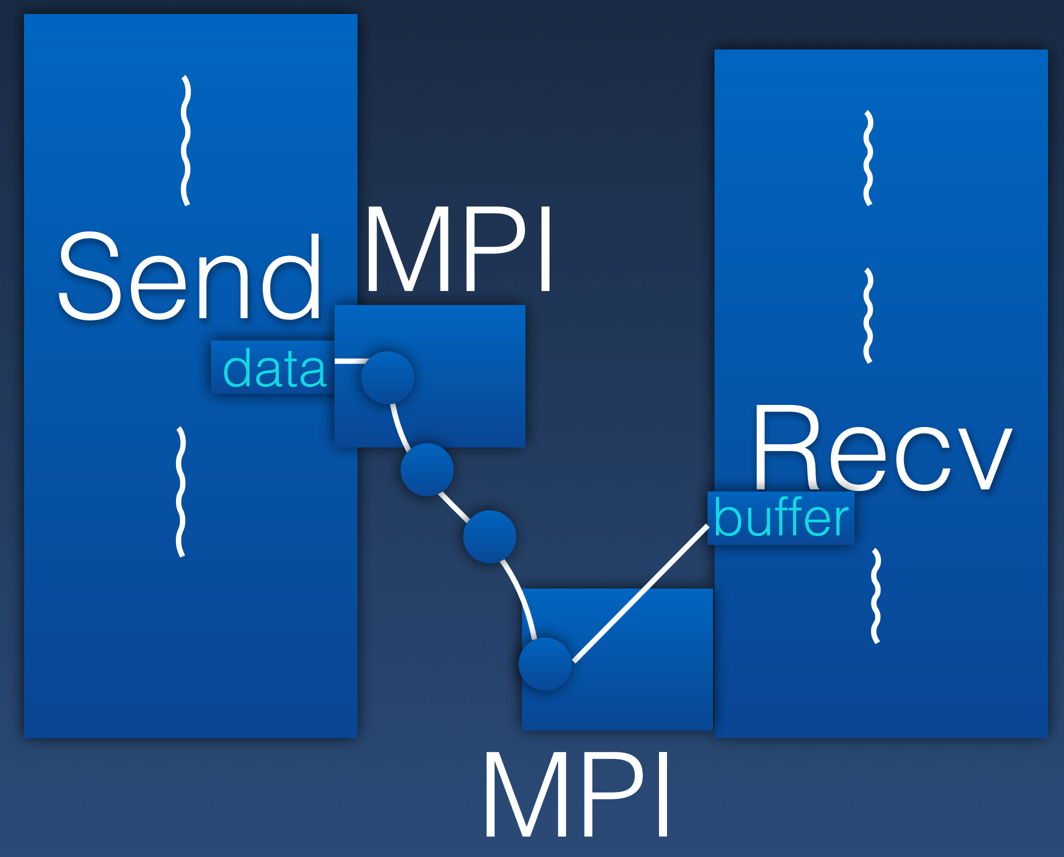
- Variables, Buffers, and Packets
  - ➡ Application to application
- Lossy?
  - ➡ Deal with loss
  - ➡ Acks
- FIFO?
- Point to Point vs Collective?
- Addressing?



Subodh Kumar

Message Semantics

- Variables, Buffers, and Packets
  - ➡ Application to application
- Lossy?
  - ➡ Deal with loss
  - ➡ Acks
- FIFO?
- Point to Point vs Collective?
- Addressing?



Local calls to an MPI stub on each node (or process),
Which can manage buffers and
handle all requests in the background

- MPI is for inter-process communication
  - ➡ Process creation
  - ➡ Data communication (Buffering, Book-keeping ..)
  - ➡ Synchronization

Functions, Types, Constants

- Allows
  - ➡ Synchronous communication
  - ➡ Asynchronous communication
    - compare to shared memory

- High-level constructs
  - broadcast, reduce, scatter/gather message
  - Collective functions
- Interoperable across architectures

Running MPI Programs

- **Compile:** mpiCC -o exec code.cpp
  - script to compile and link
  - Automatically adds include, library flags
- **Run:**
  - mpirun -host host1,host2 exec args
  - Or, use hostfile
- **Useful:**
  - mpirun -mca <key> <value>

- mpirun -mca mpi_show_handle_leaks 1
- mpirun -mca btl openib,tcp
- mpirun -mca btl_tcp_min_rdma_size
- Check out “ompi_info”

- Key based remote shell execution
- Use ssh-keygen to create public-private key pair
  - ➡ Private key stays in subdirectory ~/.ssh on your client
  - ➡ Public key on server in ~/.ssh/authorized_keys
  - ➡ Test: ‘ssh <server> ls’ works
  - ➡ On HPC, client and server share the same home directory
- PBS automatically creates appropriate host files
  - ➡ See also: -l select=2:ncpus=1:mpiprocs=1 -l place=scatter

Example

```
#include "mpi.h"      /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);                        // start MPI
  int nProcs, myRank, dat[2] = {5,6};
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD, &nProcs); // Group size
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank); // get my rank
  If(myRank == 0)
      MPI_Send(dat, 2, MPI_INT, nProcs-1, 11, MPI_COMM_WORLD);
  If(myRank == nProcs-1)
      MPI_Recv(dat, 9, MPI_INT, 0, 11, MPI_COMM_WORLD, &status);
  MPI_Finalize();                               // stop MPI
}
```

Example

```
#include "mpi.h"      /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);                     // start MPI
  int nProcs, myRank, dat[2] = {5,6};
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_W
  MPI_Comm_rank(MPI_COMM_W
  If(myRank == 0)
      MPI_Send(dat, 2, MPI                                  LD);
  If(myRank == nProcs-1)
      MPI_Recv(dat, 9, MPI_INT, 0, 11, MPI_COMM_WORLD, &status);
  MPI_Finalize();                             // stop MPI
}
```

```
status.MPI_TAG

status.MPI_SOURCE

MPI_Get_count(&status, MPI_INT, &count);
```

Process Organization

- Communicator

  May be used to support communication Topology

  - ➡ Groups of processes sharing a context
  - ➡ Intra and inter-communicator

  Predefined constant: MPI_COMM_WORLD

- Context
  - ➡ “communication universe”
  - ➡ Messages across context have no ‘interference’
- Groups
  - ➡ Collection of processes (can build hierarchy)
  - ➡ Ordered

  Use group-rank to address

Starting and Ending

- MPI_Init(&argc, &argv);

  Use MPI_Init_thread in multi-threaded processes

  - Needed before any other MPI call

```
int nump, id;
MPI_Comm_size (MPI_COMM_WORLD, &nump);
MPI_Comm_rank (MPI_COMM_WORLD, &id);
```

- MPI_Finalize();
  - Required

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

• message contents	block of memory
• count	number of items in message
• message type	MPI_Datatype of each item
• destination	rank of recipient
• tag	integer “message identifier”
• communicator	

Send/Receive

Blocking calls (Progress condition is satisfied on return)

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

MATCHING (Per context)

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

message contents	blo
count	nu
message type	MI
destination	ra
tag	int
communicator	

message contents	memory buffer to store received message
count	space in buffer, overflow error if too small
message type	type of each item
source	sender's rank (or MPI_ANY_SOURCE)
tag	message identifier (or MPI_ANY_TAG)
communicator	
status	information about message received

Send/Receive

Blocking calls (Progress condition is satisfied on return

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

MATCHING (Per context)

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

s block of memory
number of items in messag
MPI_Datatype of each iten
rank of recipient
integer "message identifier

• message contents	memory buffer to store received message
• count	space in buffer, overflow error if too small
• message type	type of each item
• source	sender's rank (or MPI_ANY_SOURCE)
• tag	message identifier (or MPI_ANY_TAG)
• communicator	
• status	information about message received