CISC 372: Parallel Programming

MPI Point-to-Point Operations

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

Point to Point Operations

- for sending a message from one process to another process

Point to Point Operations

- for sending a message from one process to another process
- sending process issues a send instruction
- receiving process issues a receive instruction

Point to Point Operations

- for sending a message from one process to another process
- sending process issues a send instruction
- receiving process issues a receive instruction
- can be considered "lower-level" than collective operations

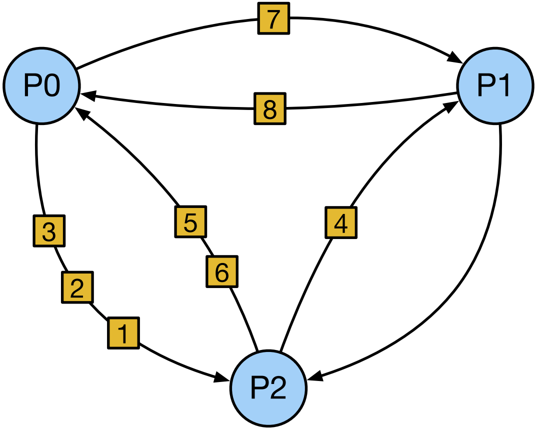
Point to Point Operations

- for sending a message from one process to another process
- sending process issues a send instruction
- receiving process issues a receive instruction
- can be considered "lower-level" than collective operations
- all collective operations can be implemented using point-to-points
  - but quality MPI implementations will provide better performance for collectives

Point to Point Operations

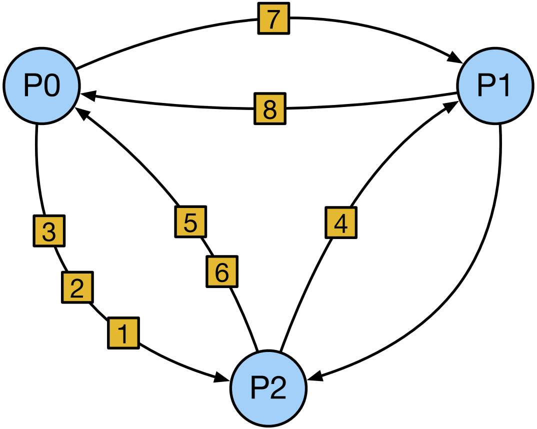
- for sending a message from one process to another process
- sending process issues a send instruction
- receiving process issues a receive instruction
- can be considered "lower-level" than collective operations
- all collective operations can be implemented using point-to-points
  - but quality MPI implementations will provide better performance for collectives
- "push" model (like the mail)
  - sending process specifies destination
  - receiving process may or may not specify source

Message channels: conceptual framework



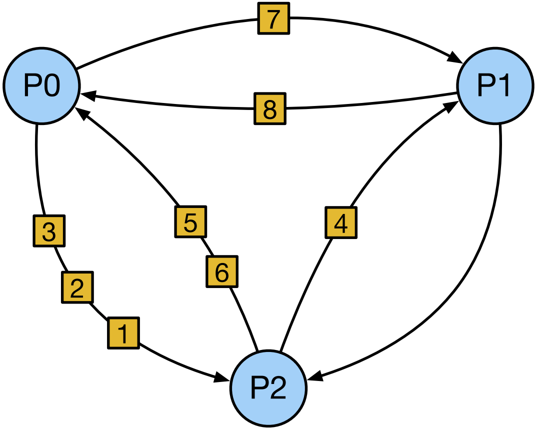
- the state of a communicator with 3 procs

Message channels: conceptual framework



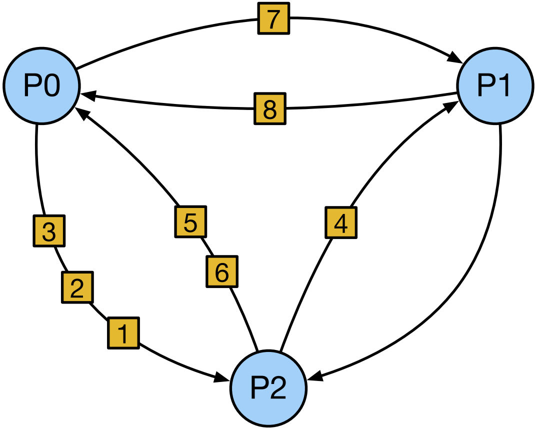
- the state of a communicator with 3 procs
- every communicator is isolated — has its own state
  - messages from one communicator are never picked up by an operation from a different communicator

Message channels: conceptual framework



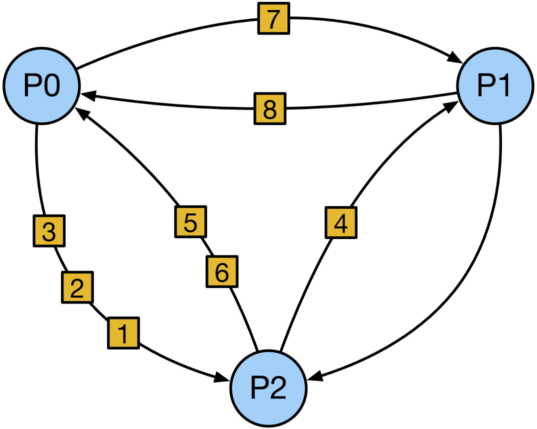
- the state of a communicator with 3 procs
- every communicator is isolated — has its own state
  - messages from one communicator are never picked up by an operation from a different communicator
- between any 2 procs, there is a p2p message channel
  - including from proc to itself (rarely used)

Message channels: conceptual framework



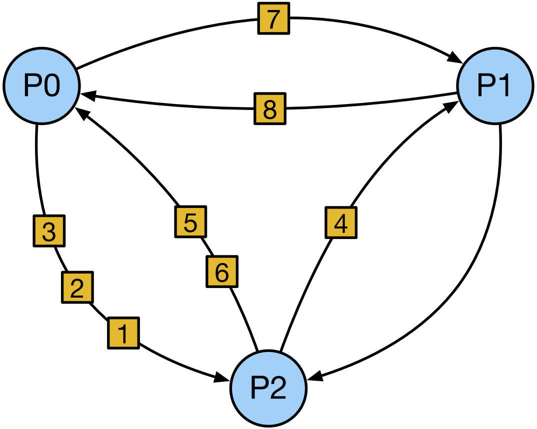
- the state of a communicator with 3 procs
- every communicator is isolated — has its own state
  - messages from one communicator are never picked up by an operation from a different communicator
- between any 2 procs, there is a p2p message channel
  - including from proc to itself (rarely used)
- `send` enqueues message

Message channels: conceptual framework



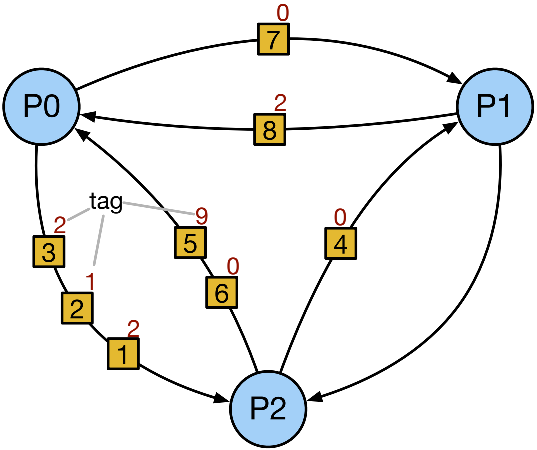
- the state of a communicator with 3 procs
- every communicator is isolated — has its own state
  - messages from one communicator are never picked up by an operation from a different communicator
- between any 2 procs, there is a p2p message channel
  - including from proc to itself (rarely used)
- `send` enqueues message
- `recv` dequeues message

Message channels: conceptual framework



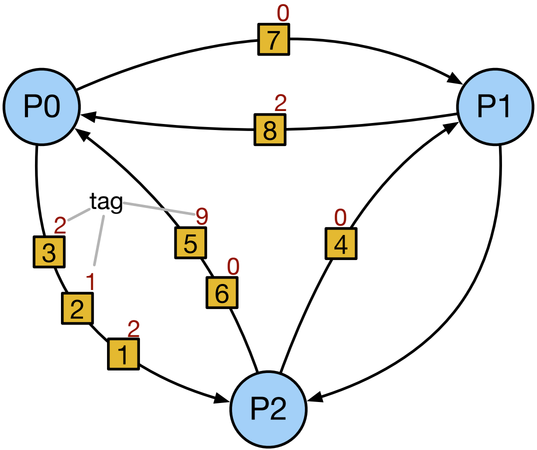
- the state of a communicator with 3 procs
- every communicator is isolated — has its own state
  - messages from one communicator are never picked up by an operation from a different communicator
- between any 2 procs, there is a p2p message channel
  - including from proc to itself (rarely used)
- `send` enqueues message
- `recv` dequeues message
- mostly a FIFO queue

Tags



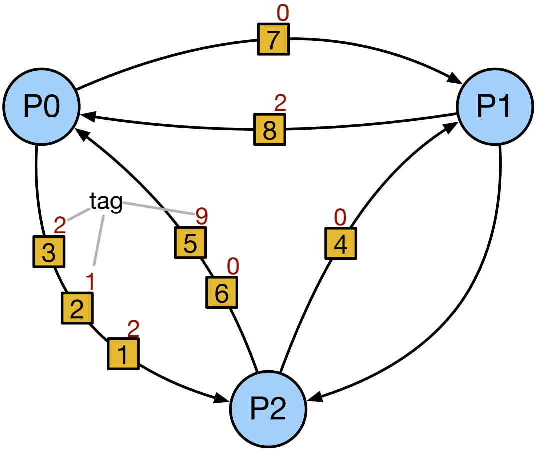
- each message has a tag

Tags



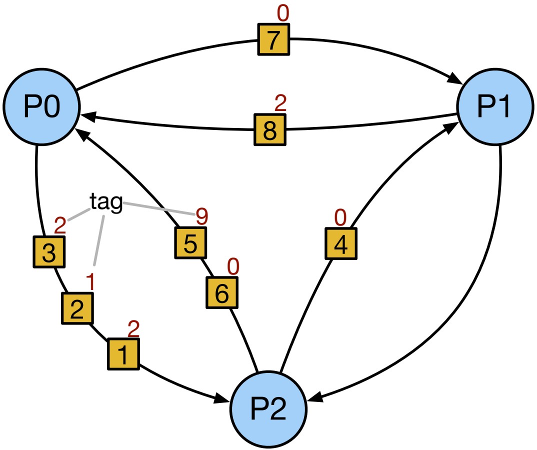
- each message has a tag
- an `int` specified by the sender

Tags



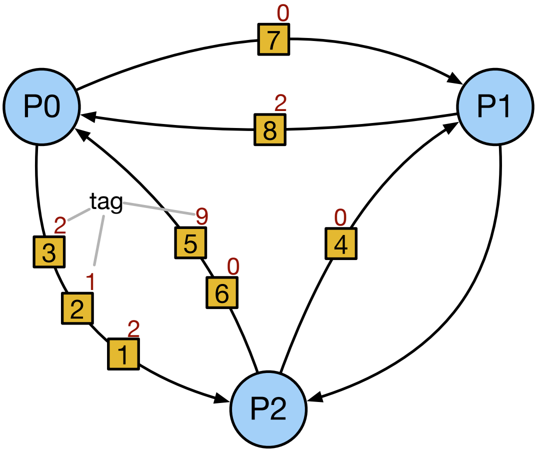
- each message has a tag
- an `int` specified by the sender
- the receiver may specify a tag
  - or can specify "any tag"

Tags



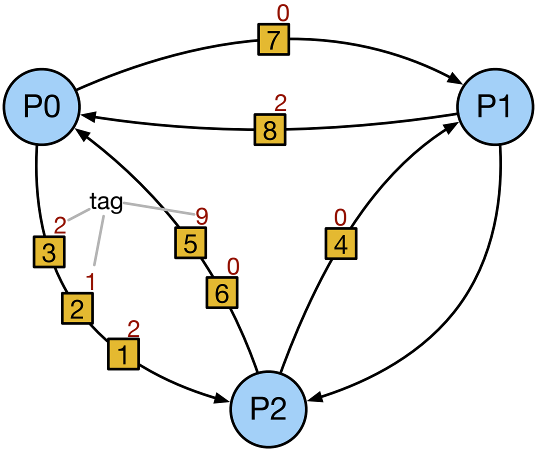
- each message has a tag
- an `int` specified by the sender
- the receiver may specify a tag
  - or can specify "any tag"
- if P2 issues recv from P0 with tag 2
  - P2 will receive message 1

Tags



- each message has a tag
- an `int` specified by the sender
- the receiver may specify a tag
  - or can specify "any tag"
- if P2 issues recv from P0 with tag 2
  - P2 will receive message 1
- if P2 issues recv from P0 with tag 1
  - P2 will receive message 2
  - the first (oldest) message in queue with matching tag

Tags



- each message has a tag
- an `int` specified by the sender
- the receiver may specify a tag
  - or can specify "any tag"
- if P2 issues recv from P0 with tag 2
  - P2 will receive message 1
- if P2 issues recv from P0 with tag 1
  - P2 will receive message 2
  - the first (oldest) message in queue with matching tag
- if P2 issues recv from P0 with "any tag"
  - P2 will receive message 1

MPI_Send

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

`buf`	address of send buffer (`void*`)
`count`	number of elements in buffer (`int`)
`datatype`	data type of elements in buffer (`MPI_Datatype`)
`dest`	rank of destination process (`int`)
`tag`	integer to attach to message envelope (`int`)
`comm`	communicator (`MPI_Comm`)

MPI_Send

`MPI_Send(buf, count, datatype, dest, tag, comm)`

`buf`	address of send buffer (`void*`)
`count`	number of elements in buffer (`int`)
`datatype`	data type of elements in buffer (`MPI_Datatype`)
`dest`	rank of destination process (`int`)
`tag`	integer to attach to message envelope (`int`)
`comm`	communicator (`MPI_Comm`)

- message envelope
  - source rank
  - destination rank
  - tag
  - communicator
- tag can be used by receiver to select which message to receive

MPI_Recv

```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

`buf`	address of receive buffer (`void*`)
`count`	number of elements in buffer (`int`)
`datatype`	data type of elements in buffer (`MPI_Datatype`)
`source`	rank of source process (`int`)
`tag`	tag of message to receive (`int`)
`comm`	communicator (`MPI_Comm`)
`status`	pointer to status object (`MPI_Status*`)

MPI_Recv

```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

buf	address of receive buffer (void*)
count	number of elements in buffer (int)
datatype	data type of elements in buffer (MPI_Datatype)
source	rank of source process (int)
tag	tag of message to receive (int)
comm	communicator (MPI_Comm)
status	pointer to status object (MPI_Status*)

- count must be at least as large as count of incoming message
  - otherwise, undefined behavior

MPI_Recv

`MPI_Recv(buf, count, datatype, source, tag, comm, status)`

buf	address of receive buffer (void*)
count	number of elements in buffer (int)
datatype	data type of elements in buffer (MPI_Datatype)
source	rank of source process (int)
tag	tag of message to receive (int)
comm	communicator (MPI_Comm)
status	pointer to status object (MPI_Status*)

- count must be at least as large as count of incoming message
  - otherwise, undefined behavior
- status: object to store envelope information on received message
  - source, tag, count
  - if you don't need it, use MPI_STATUS_IGNORE

MPI_Recv

`MPI_Recv(buf, count, datatype, source, tag, comm, status)`

buf	address of receive buffer (void*)
count	number of elements in buffer (int)
datatype	data type of elements in buffer (MPI_Datatype)
source	rank of source process (int)
tag	tag of message to receive (int)
comm	communicator (MPI_Comm)
status	pointer to status object (MPI_Status*)

- count must be at least as large as count of incoming message
  - otherwise, undefined behavior
- status: object to store envelope information on received message
  - source, tag, count
  - if you don't need it, use MPI_STATUS_IGNORE
- why would you need to know source and tag when you already specified them?

Example: p2p.c

```
#include<stdio.h>
#include<mpi.h>
int main() {
  int message, rank;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    message = 173;
    MPI_Send(&message, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
  } else if (rank == 1)  {
    MPI_Recv(&message, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Proc 1 received: %d\n", message);
  }
  MPI_Finalize();
}
```

Example: p2p.c

```
#include<stdio.h>
#include<mpi.h>
int main() {
  int message, rank;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    message = 173;
    MPI_Send(&message, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
  } else if (rank == 1)  {
    MPI_Recv(&message, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Proc 1 received: %d\n", message);
  }
  MPI_Finalize();
}
```

```
> mpiexec -n 4 ./p2p.exec
Proc 1 received: 173
```

Example: using different tags: tags.c

```
/* tags.c: demonstration of receiving messages out of order using tags.  Note that
   this program is not safe --- technically, it could deadlock.  But if it does not
   deadlock, the messages will be received in the reverse order.  */
#include<stdio.h>
#include<mpi.h>
int main() {
  int message, rank;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    message = 1; MPI_Send(&message, 1, MPI_INT, 1, 1, MPI_COMM_WORLD); // tag=1
    message = 2; MPI_Send(&message, 1, MPI_INT, 1, 2, MPI_COMM_WORLD); // tag=2
  } else if (rank == 1)  {
    MPI_Recv(&message, 1, MPI_INT, 0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // tag=2
    printf("Proc 1 received: %d\n", message);
    MPI_Recv(&message, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // tag=1
    printf("Proc 1 received: %d\n", message);
  }
  MPI_Finalize();
}
```

MPI_ANY_TAG

- a recv can use MPI_ANY_TAG for the tag argument
- receive a message from sender with "any tag"
- it will always match the oldest message from the sender
- execution is deterministic — one and only one thing can happen

Example: using MPI_ANY_TAG: anytag.c

```
/* anytag: the messages will be received in the order sent.  The MPI_ANY_TAG recv
   must match the oldest message sent from proc 0 */
#include<stdio.h>
#include<mpi.h>
int main() {
  int message, rank;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    message = 1;
    MPI_Send(&message, 1, MPI_INT, 1, 1, MPI_COMM_WORLD); // tag=1
    message = 2;
    MPI_Send(&message, 1, MPI_INT, 1, 2, MPI_COMM_WORLD); // tag=2
  } else if (rank == 1)  {
    MPI_Recv(&message, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Proc 1 received: %d\n", message);
    MPI_Recv(&message, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Proc 1 received: %d\n", message);
  }
  MPI_Finalize();
}
```

Getting the status

`status` is a C `struct`

Getting the status

`status` is a C `struct`

- getting the rank of the source
  - `status.MPI_SOURCE`

Getting the status

`status` is a C `struct`

- getting the rank of the source
  - `status.MPI_SOURCE`
- getting the tag of the message
  - `status.MPI_TAG`

Getting the status

`status` is a C `struct`

- getting the rank of the source
  - `status.MPI_SOURCE`
- getting the tag of the message
  - `status.MPI_TAG`
- getting the error code
  - `status.MPI_ERROR`

Getting the status

`status` is a C `struct`

- getting the rank of the source
  - `status.MPI_SOURCE`
- getting the tag of the message
  - `status.MPI_TAG`
- getting the error code
  - `status.MPI_ERROR`
- getting the size ("count") of the message
  - not simply a field in the struct
  - need to use function `MPI_Get_count`

Example: status.c

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

int main() {
  char message[100];
  int rank;
  MPI_Status status;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    strcpy(message,"Hello, from proc 0!");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  } else if (rank == 1)  {
    MPI_Recv(message, 100, MPI_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    printf("Proc 1 received: \"%s\"\n", message);
    printf("source=%d tag=%d \n", status.MPI_SOURCE, status.MPI_TAG);
  }
  MPI_Finalize();
}
```

status.c output

Note that in C, a string is a sequence of `char` ending with the "null terminating char" `'\0'`. The number of characters in the string is therefore `strlen(message) + 1` $= 19 + 1 = 20$.

```
> mpiexec status.exec
Proc 1 received: "Hello, from proc 0!"
source=0 tag=99
```

MPI_Get_count

```
MPI_Get_count(status, datatype, count)
```

`status`	pointer to status object (`MPI_Status*`)
`datatype`	data type of elements received (`MPI_Datatype`)
`count`	pointer to variable in which to return result (`int*`)

MPI_Get_count

`MPI_Get_count(status, datatype, count)`

`status`	pointer to status object (`MPI_Status*`)
`datatype`	data type of elements received (`MPI_Datatype`)
`count`	pointer to variable in which to return result (`int*`)

- should only be called after `status` has been filled in by receive

MPI_Get_count

`MPI_Get_count(status, datatype, count)`

`status`	pointer to status object (`MPI_Status*`)
`datatype`	data type of elements received (`MPI_Datatype`)
`count`	pointer to variable in which to return result (`int*`)

- should only be called after `status` has been filled in by receive
- `datatype` should be same as used in receive

MPI_Get_count

`MPI_Get_count(status, datatype, count)`

status	pointer to status object (MPI_Status*)
datatype	data type of elements received (MPI_Datatype)
count	pointer to variable in which to return result (int*)

- should only be called after status has been filled in by receive
- datatype should be same as used in receive
- sets count to the number of elements received

MPI_Get_count

`MPI_Get_count(status, datatype, count)`

`status`	pointer to status object (`MPI_Status*`)
`datatype`	data type of elements received (`MPI_Datatype`)
`count`	pointer to variable in which to return result (`int*`)

- should only be called after `status` has been filled in by receive
- `datatype` should be same as used in receive
- sets `count` to the number of elements received
- note
  - `count` specified in receive statement and message `count` can differ
  - receive buffer must be big enough to hold incoming message
  - memory in receive buffer after message count will not be altered

Example: getting the count: count.c

The following lines are added to proc 1:

```
int count;
MPI_Get_count(&status, MPI_CHAR, &count);
printf("source=%d tag=%d count=%d\n",
       status.MPI_SOURCE, status.MPI_TAG, count);
```

Example: getting the count: count.c

The following lines are added to proc 1:

```
int count;
MPI_Get_count(&status, MPI_CHAR, &count);
printf("source=%d tag=%d count=%d\n",
       status.MPI_SOURCE, status.MPI_TAG, count);
```

This sets count to the actual number of characters (MPI_CHAR) received.

Example: getting the count: count.c

The following lines are added to proc 1:

```
int count;
MPI_Get_count(&status, MPI_CHAR, &count);
printf("source=%d tag=%d count=%d\n",
       status.MPI_SOURCE, status.MPI_TAG, count);
```

This sets count to the actual number of characters (MPI_CHAR) received.

```
> mpiexec -n 4 ./count.exec
Proc 1 received: "Hello, from proc 0!"
source=0 tag=99 count=20
```

Note the null terminating character is counted.

Synchronization and deadlock

- a receive operation must block until a matching message arrives

Synchronization and deadlock

- a receive operation must block until a matching message arrives
- this can lead to deadlocks if you are not careful; see deadlock.c

```
#include<stdio.h>
#include<mpi.h>
int main() {
  int message, rank;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    message = 173;
    printf("Proc 0: was I supposed to do something?\n");
  } else if (rank == 1)  {
    MPI_Recv(&message, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Proc 1 received: %d\n", message);
  }
  MPI_Finalize();
}
```

Synchronization and deadlock

- a receive operation must block until a matching message arrives
- this can lead to deadlocks if you are not careful; see deadlock.c

```
#include<stdio.h>
#include<mpi.h>
int main() {
  int message, rank;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    message = 173;
    printf("Proc 0: was I supposed to do something?\n");
  } else if (rank == 1)  {
    MPI_Recv(&message, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Proc 1 received: %d\n", message);
  }
  MPI_Finalize();
}
```

```
mpiexec -n 4 ./deadlock.exec
Proc 0: was I supposed to do something?
^C[mpiexec@basie.local] Sending Ctrl-C to processes as requested
```

Synchronization and potential deadlock

- a send operation . . .

Synchronization and potential deadlock

- a send operation . . .
  - may complete even if a matching receive operation has not been executed
    - the message will be stored in a system buffer (channel)

Synchronization and potential deadlock

- a send operation . . .
  - may complete even if a matching receive operation has not been executed
    - the message will be stored in a system buffer (channel)
  - or it may block until a matching receive is available
    - the message can then be copied directly from send buffer to recv buffer

Synchronization and potential deadlock

- a send operation ...
  - may complete even if a matching receive operation has not been executed
    - the message will be stored in a system buffer (channel)
  - or it may block until a matching receive is available
    - the message can then be copied directly from send buffer to recv buffer
- the choice is up to the MPI implementation

Synchronization and potential deadlock

- a send operation ...
  - may complete even if a matching receive operation has not been executed
    - the message will be stored in a system buffer (channel)
  - or it may block until a matching receive is available
    - the message can then be copied directly from send buffer to recv buffer
- the choice is up to the MPI implementation
- the decision can be made differently at each send operation

Synchronization and potential deadlock

- a send operation . . .
  - may complete even if a matching receive operation has not been executed
    - the message will be stored in a system buffer (channel)
  - or it may block until a matching receive is available
    - the message can then be copied directly from send buffer to recv buffer
- the choice is up to the MPI implementation
- the decision can be made differently at each send operation
- you cannot assume anything

Synchronization and potential deadlock

- a send operation . . .
  - may complete even if a matching receive operation has not been executed
    - the message will be stored in a system buffer (channel)
  - or it may block until a matching receive is available
    - the message can then be copied directly from send buffer to recv buffer
- the choice is up to the MPI implementation
- the decision can be made differently at each send operation
- you cannot assume anything
- a correct program will behave correctly regardless of how this decision is made

Example may_deadlock.c: a potential deadlock

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

int main() {
  int message, rank;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    message = 173;
    MPI_Send(&message, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
  } else if (rank == 1)  {
    printf("Proc 1: was I supposed to do something?\n");
  }
  MPI_Finalize();
}
```

Exchanging data

Exchanging data

- suppose two processes wish to `exchange some data`
  - proc 0 wants to send something to proc 1, and
  - proc 1 wants to send something to proc 0

Exchanging data

- suppose two processes wish to `exchange some data`
  - proc 0 wants to send something to proc 1, and
  - proc 1 wants to send something to proc 0
- very common scenario

Exchanging data

- suppose two processes wish to `exchange some data`
  - proc 0 wants to send something to proc 1, and
  - proc 1 wants to send something to proc 0
- very common scenario
- how to it safely?
  - must be correct
  - must not deadlock

Exchange 1: Incorrect: will deadlock!

- both procs try to receive before sending

```
int main() {
  int rank, myNumber, otherNumber;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    myNumber = 10;
    MPI_Recv(&otherNumber, 1, MPI_INT, 1, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&myNumber, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
  } else if (rank == 1)  {
    myNumber = 20;
    MPI_Recv(&otherNumber, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&myNumber, 1, MPI_INT, 0, 9, MPI_COMM_WORLD);
  }
  printf("Process %d: received %d\n", rank, otherNumber);
  MPI_Finalize();
}
```

Exchange 2: Unsafe: may deadlock!

- both procs send before receiving — what if MPI tries to execute both sends synchronously?

```
int main() {
  int rank, myNumber, otherNumber;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    myNumber = 10;
    MPI_Send(&myNumber, 1, MPI_INT, 1, 99, MPI_COMM_WORLD);
    MPI_Recv(&otherNumber, 1, MPI_INT, 1, 99, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else if (rank == 1)  {
    myNumber = 20;
    MPI_Send(&myNumber, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
    MPI_Recv(&otherNumber, 1, MPI_INT, 0, 99, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
  printf("Process %d: received %d\n", rank, otherNumber);
  MPI_Finalize();
}
```

Exchange 3: Correct: procs alternate

- one proc sends, then receives; the other proc receives, then sends

```
int main() {
  int rank, myNumber, otherNumber;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    myNumber = 10;
    MPI_Send(&myNumber, 1, MPI_INT, 1, 99, MPI_COMM_WORLD);
    MPI_Recv(&otherNumber, 1, MPI_INT, 1, 99, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else if (rank == 1)  {
    myNumber = 20;
    MPI_Recv(&otherNumber, 1, MPI_INT, 0, 99, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&myNumber, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
  }
  printf("Process %d: received %d\n", rank, otherNumber);
  MPI_Finalize();
}
```

Exchanging with MPI_Sendrecv

- this situation is so common, MPI provides a function to deal with it
- MPI_Sendrecv combines one send and one receive operation into a single command
- both operations execute concurrently

MPI_Sendrecv

```
MPI_Sendrecv(sbuf, scount, stype, dest, stag,
             rbuf, rcount, rtype, source, rtag,
             comm, status)
```

sbuf	address of send buffer (void*)
scount	number of elements in send buffer (int)
stype	data type of elements in sbuf (MPI_Datatype)
dest	rank of destination process (int)
stag	integer to attach to message envelope (int)
rbuf	address of receive buffer (void*)
rcount	length of receive buffer (int)
rtype	data type of elements to be received (MPI_Datatype)
source	rank of sending process (int)
rtag	tag of message to receive (int)
comm	communicator (MPI_Comm)
status	pointer to status object for receive (MPI_Status*)

Semantics and uses of MPI_Sendrecv

- combines a send statement and a receive statement into one statement

Semantics and uses of MPI_Sendrecv

- combines a send statement and a receive statement into one statement
- both operations post simultaneously

Semantics and uses of `MPI_Sendrecv`

- combines a send statement and a receive statement into one statement
- both operations post simultaneously
- as if two threads are spawned, one to manage the send, the other the receive

Semantics and uses of `MPI_Sendrecv`

- combines a send statement and a receive statement into one statement
- both operations post simultaneously
- as if two threads are spawned, one to manage the send, the other the receive
- the operation completes only after both the send and receive complete

Semantics and uses of `MPI_Sendrecv`

- combines a send statement and a receive statement into one statement
- both operations post simultaneously
- as if two threads are spawned, one to manage the send, the other the receive
- the operation completes only after both the send and receive complete
- solves the deadlocking problem for data exchange

Semantics and uses of `MPI_Sendrecv`

- combines a send statement and a receive statement into one statement
- both operations post simultaneously
- as if two threads are spawned, one to manage the send, the other the receive
- the operation completes only after both the send and receive complete
- solves the deadlocking problem for data exchange
- cyclic exchange
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$
  - process of rank i
    - sends to $i+1$ (modulo numProcs)
    - receives from $i-1$ (modulo numProcs)

Semantics and uses of `MPI_Sendrecv`

- combines a send statement and a receive statement into one statement
- both operations post simultaneously
- as if two threads are spawned, one to manage the send, the other the receive
- the operation completes only after both the send and receive complete
- solves the deadlocking problem for data exchange
- cyclic exchange
  - $0 \to 1 \to 2 \to 3 \to 0$
  - process of rank i
    - sends to $i+1$ (modulo numProcs)
    - receives from $i-1$ (modulo numProcs)
- shift
  - $0 \to 1 \to 2 \to 3$
  - proc 0 only sends
  - proc `nprocs` -1 only receives
  - or use `MPI_PROC_NULL`

Exchange 4: Correct: MPI_Sendrecv

```
int main() {
  int rank, myNumber, otherNumber;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    myNumber = 10;
    MPI_Sendrecv(&myNumber, 1, MPI_INT, 1, 99, &otherNumber, 1, MPI_INT, 1, 99,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else if (rank == 1) {
    myNumber = 20;
    MPI_Sendrecv(&myNumber, 1, MPI_INT, 0, 99, &otherNumber, 1, MPI_INT, 0, 99,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
  printf("Process %d: received %d\n", rank, otherNumber);
  MPI_Finalize();
}
```

Cyclic exchange: cycle.c

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

int main() {
  int nprocs, rank;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  const int right = (rank + 1)%nprocs, left = (rank + nprocs - 1)%nprocs;
  int rbuf, sbuf = 100 + rank;
  MPI_Sendrecv(&sbuf, 1, MPI_INT, right, 0, &rbuf, 1, MPI_INT, left, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("Proc %d: received %d\n", rank, rbuf);
  MPI_Finalize();
}
```

- note use of `rank + nprocs - 1` to avoid a negative argument to modulo operator

Shift exchange: shift.c

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

int main() {
  int nprocs, rank;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  const int right = rank < nprocs - 1 ? rank + 1 : MPI_PROC_NULL,
    left = rank > 0 ? rank - 1 : MPI_PROC_NULL;
  int rbuf, sbuf = 100 + rank;
  MPI_Sendrecv(&sbuf, 1, MPI_INT, right, 0, &rbuf, 1, MPI_INT, left, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  if (rank > 0) printf("Proc %d: received %d\n", rank, rbuf);
  MPI_Finalize();
}
```

- a send or receive to `MPI_PROC_NULL` is a no-op

Semantics: Non-interaction with collectives

- an MPI program can use both point-to-point and collective operations
- point-to-point and collective operations exist in two separate universes
  - there is no "matching" between p2p and collective operations
  - a message sent by a p2p can never be received by a collective
  - a message sent by a collective can never be received by a p2p