University of Notre Dame

MPI Tutorial

Part 2

High-Performance MPI

Laboratory for Scientific Computing

Fall 1998

http://www.lam-mpi.org/tutorials/nd/

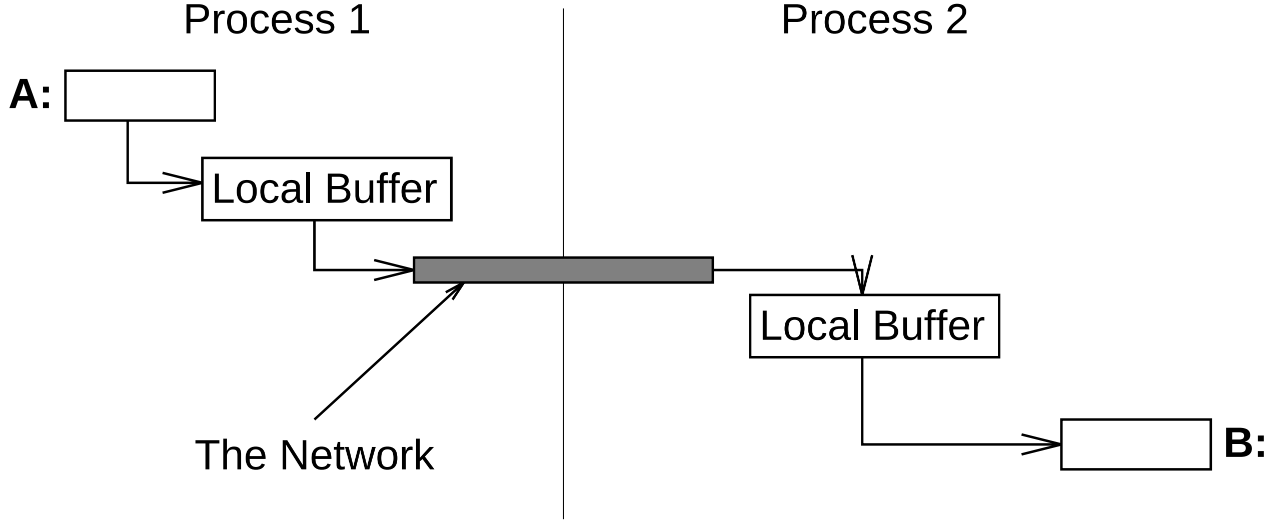
lam@lam-mpi.org

Section V

Non-Blocking Communication

Buffering Issues

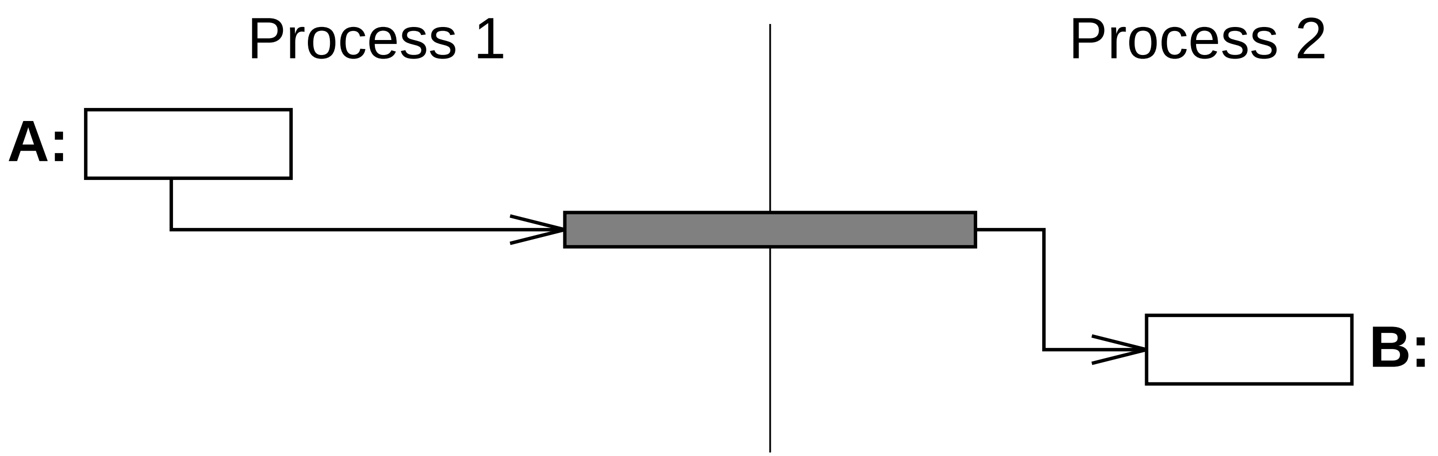
- Where does data go when you send it?
- One possibility is:



- This is not very efficient:
  - Three copies in addition to the exchange of data between processes.
  - Copies are “bad.”

Better Buffering

- We prefer



- But this requires that either that `MPI_SEND` not return until the data has been delivered *or* that we allow a send operation to return before completing the transfer.

- In the latter case, we need to test for completion later.

Blocking and Non-Blocking Communication

- So far we have used *blocking* communication:
  - `MPI_SEND` does not complete until buffer is empty (available for reuse).
  - `MPI_RECV` does not complete until buffer is full (available for use).
- Simple, but can be prone to deadlocks:

Process 0	Process 1
Send(1)	Send(0)
Recv(1)	Recv(0)

Completion depends in general on size of message and amount of system buffering.

The semantics of blocking/non-blocking has nothing to do with when messages are sent or recieved. The difference is when the buffer is free to re-use.

Some Solutions to the Deadlock Problem

- Order the operations more carefully:

Process 0	Process 1
Send(1)	Recv(0)
Recv(1)	Send(0)

- Supply receive buffer at same time as send, with `MPI_SENDRECV`:

Process 0	Process 1
Sendrecv(1)	Sendrecv(0)

More Solutions to the Deadlock Problem

- Use non-blocking operations:

Process 0	Process 1
Irecv(1)	Irecv(0)
Isend(1)	Isend(0)
Waitall	Waitall

- Use `MPI_BSEND`
  - Copies message into a user buffer (previously supplied) and returns control to user program
  - Sends message sometime later

MPI's Non-Blocking Operations

- Non-blocking operations return (immediately) "request handles" that can be waited on and queried:

```
MPI_ISEND(start, count, datatype, dest, tag, comm,
          request)
MPI_IRECV(start, count, datatype, dest, tag, comm,
          request)
MPI_WAIT(request, status)
```

- One can also test without waiting:

```
MPI_TEST(request, flag, status)
```

Multiple Completions

- It is often desirable to wait on multiple requests.
- An example is a worker/manager program, where the manager waits for one or more workers to send it a message.

```
MPI_WAITALL(count, array_of_requests,
            array_of_statuses)
MPI_WAITANY(count, array_of_requests, index, status)
MPI_WAITSOME(incount, array_of_requests, outcount,
  array_of_indices, array_of_statuses)
```

- There are corresponding versions of test for each of these.

Probing the Network for Messages

- `MPI_PROBE` and `MPI_IPROBE` allow the user to check for incoming messages without actually receiving them
- `MPI_IPROBE` returns “`flag == TRUE`” if there is a matching message available. `MPI_PROBE` will not return until there is a matching receive available:

```
MPI_IPROBE(source, tag, communicator, flag, status)
MPI_PROBE(source, tag, communicator, status)
```

It is typically not good practice to use these functions.

MPI Send-Receive

- The send-receive operation combines the send and receive operations in one call.
- The send-receive operation performs a blocking send and receive operation using distinct tags but the same communicator.
- A send-receive operation can be used with regular send and receive operations.

```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest,
             sendtag, recvbuf, recvcount, recvtype,
             source, recvtag, comm, status)
```

- Avoids user having to order send/receive to avoid deadlock

Non-Blocking Example: Manager –¿ 1 Worker

```
/* ... Only a portion of the code */
int flag = 0;
MPI_Status status;
double buffer[BIG_SIZE];
MPI_Request request;

/* Send some data */
MPI_Isend(buffer, BIG_SIZE, MPI_DOUBLE, dest, tag,
          MPI_COMM_WORLD, &request);

/* While the send is progressing, do some useful work */
while (!flag && have_more_work_to_do) {
  /* ...do some work... */
  MPI_Test(&request, &flag, &status);
}
/* If we finished work but the send is still pending, wait */
if (!flag)
  MPI_Wait(&request, &status);
/* ... */
```

Non-Blocking Example: Manager –¿ 4 Workers

```
/* ... Only a portion of the code */
MPI_Status status[4];
double buffer[BIG_SIZE];
MPI_Request requests[4];
int i, flag, index, each_size = BIG_SIZE / 4;

/* Send out the data to the 4 workers */
for (i = 0; i < 4; i++)
  MPI_Isend(buffer + (i * each_size), each_size, MPI_DOUBLE, i + 1,
            tag, MPI_COMM_WORLD, &requests[i]);

/* While the sends are progressing, do some useful work */
for (i = 0; i < 4 && have_more_work_to_do; i++) {
  /* ...do some work... */
  MPI_Testany(4, requests, &flag, &index, &status[0]);
  if (!flag)
    i--;
}

/* If we finished work but still have sends pending, wait for the rest*/
if (i < 4)
  MPI_Waitall(4, requests, status);
/* ... */
```

The 5 Sends

MPI_SEND Normal send. Returns after the message has been copied to a buffer OR after the message "on its way".

MPI_BSEND Buffered send. Returns after the message has been copied to an internal MPI buffer (previously supplied by the user).

MPI_SSEND Synchronous send. Returns after the message reaches the receiver.

MPI_RSEND Ready Send. The matching receive must be posted before the send executes. Returns once the message has left the send buffer.

MPI_ISEND Immediate send. Returns immediately. You may not modify contents of the message buffer until the send has completed (MPI_WAIT, MPI_TEST).

Homework – Manager / Worker

- Objective: Calculate an average in parallel workers
- Write a program to do the following:
  - Process 0 (the manager) should only use non-blocking communications
  - The manager should send 100 integers to every other processor (e.g., $0 \ldots 99$ to processor 1, $100 \ldots 199$ to processor 2, etc.)
  - All other processors (the workers) should receive the integers, calculate their sum, and return it to the manager
  - The manager should receive the results from the workers and output the average of all the numbers (i.e., $0 \ldots (\texttt{size} * 100) - 1$)

Section VI

Persistent Communication

Persistent Communication Requests

- Save arguments of a communication call
- Take overhead out of subsequent calls (e.g., in a loop)
- `MPI_SEND_INIT` creates a communication request that completely specifies a standard send operation
- `MPI_RECV_INIT` creates a communication request that completely specifies a standard recv operation
- Similar routines for ready, synchronous, and buffered send modes

MPI_SEND_INIT

MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer
IN	count	number of elements sent
IN	datatype	type of each element
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator
OUT	request	communication request

MPI_SEND_INIT bindings

```
int MPI_Send_init(void* buf, int count,
                  MPI_Datatype datatype, int dest,
                  int tag, MPI_Comm comm,
                  MPI_Request *request)

Prequest Comm::Send_init(const void* buf, int count,
                  const Datatype& datatype, int dest,
                  int tag) const

MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG,
             COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG,
COMM, REQUEST, IERROR
```

MPI_RECV_INIT

MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)

OUT	buf	initial address of receive buffer
IN	count	number of elements received
IN	datatype	type of each element
IN	source	rank of source or MPI_ANY_SOURCE
IN	tag	message tag or MPI_ANY_TAG
IN	comm	communicator
OUT	request	communication request

MPI_RECV_INIT bindings

```
int MPI_Recv_init(void* buf, int count,
                  MPI_Datatype datatype,
                  int source,
                  int tag, MPI_Comm comm,
                  MPI_Request *request)

Prequest  Comm::Recv_init(void* buf, int count,
                  const Datatype& datatype,
                  int source,
                  int tag) const
```

MPI_RECV_INIT bindings (cont.)

```
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG,
             COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
REQUEST, IERROR
```

Persistent Communication Requests

- To start a send or receive:

```
MPI_START (REQUEST, IERR)
MPI_START_ALL (COUNT, REQUESTARRAY, IERR)
```

- The wait and test routines can be used to block until completion, or to check on status

MPI_START

MPI_START(request)

INOUT	request	communication request

```
int MPI_Start(MPI_Request *request)

void Prequest::Start()

MPI_START(REQUEST, IERROR)
INTEGER REQUEST, IERROR
```

MPI_START_ALL

MPI_STARTALL(count, array_of_requests)

IN	count	list length
INOUT	array_of_requests	array of requests

MPI_START_ALL bindings

```
int MPI_Startall(int count,
                 MPI_Request *array_of_requests)

static void Prequest::Startall(int count,
            Prequest array_of_requests[])

MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
```

Homework - Persistent Communication

- Rewrite the ring program with persistent communication requests.
- Write a program to do the following:
  - Process 0 should read in a single integer (> 0) from standard input
  - Use MPI send and receive to pass the integer around a ring
  - Use the user-supplied integer to determine how many times to pass the message around the ring
  - Process 0 should decrement the integer each time it is received.
  - Processes should exit when they receive a “0”.

Section VII

User-Defined Datatypes

Datatypes and Heterogeneity

- MPI datatypes have two main purposes:
  - Heterogeneity — parallel programs between different processors
  - Noncontiguous data — structures, vectors with non-unit stride, etc.
- Basic datatypes, corresponding to the underlying language, are predefined.
- The user can construct new datatypes at run time; these are called *derived datatypes*.
- Datatypes can be constructed recursively
- Avoids packing/unpacking

Datatypes in MPI

Elementary: Language-defined types (e.g., `MPI_INT` or `MPI_DOUBLE_PRECISION`)

Vector: Separated by constant "stride"

Contiguous: Vector with stride of one

Hvector: Vector, with stride in bytes

Indexed: Array of indices

Hindexed: Indexed, with indices in bytes

Struct: General mixed types (for C structs etc.)

MPI C Datatypes Revisited

MPI datatype	C datatype
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`

MPI C Datatypes Revisited (cont.)

MPI datatype	C datatype
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI C++ Datatypes

MPI datatype	C++ datatype
`MPI::CHAR`	`signed char`
`MPI::SHORT`	`signed short int`
`MPI::INT`	`signed int`
`MPI::LONG`	`signed long int`
`MPI::UNSIGNED_CHAR`	`unsigned char`
`MPI::UNSIGNED_SHORT`	`unsigned short int`
`MPI::UNSIGNED`	`unsigned int`

MPI C++ Datatypes (cont.)

MPI datatype	C++ datatype
`MPI::UNSIGNED_LONG`	`unsigned long int`
`MPI::FLOAT`	`float`
`MPI::DOUBLE`	`double`
`MPI::LONG_DOUBLE`	`long double`
`MPI::BYTE`	
`MPI::PACKED`	

MPI Fortran Datatypes Revisited

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

Type Contiguous

- Simplest derived data type
- Constructs a type map consisting of replications of a datatype in contiguous locations.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count (nonnegative integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_contiguous(int count,
   MPI_Datatype oldtype, MPI_Datatype *newtype)

Datatype Datatype::Create_contiguous(int count) const

MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

Vectors

29	30	31	32	33	34	35
22	23	24	25	26	27	28
15	16	17	18	19	20	21
8	9	10	11	12	13	14
1	2	3	4	5	6	7

To specify this column (in row order), we can use

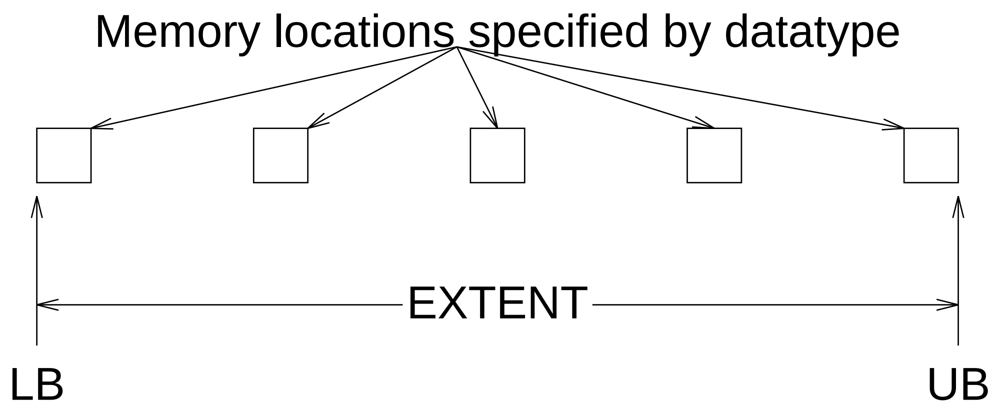
```
MPI_TYPE_VECTOR(count, blocklen, stride, oldtype,
                newtype);
MPI_TYPE_COMMIT(newtype);
```

The exact code for this is

```
MPI_TYPE_VECTOR(5, 1, 7, MPI_DOUBLE, newtype);
MPI_TYPE_COMMIT(newtype);
```

Extents

- The extent of a datatype is (normally) the distance between the first and last member (in bytes).



- You can set an artificial extent by using `MPI_UB` and `MPI_LB` in `MPI_TYPE_STRUCT`.

Extent and Size

- The **size** returns the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype.
- What is the size of the vector in the previous example?
- What is the extent?

Example: C Structures

```
struct {
    char     display[50];   /* Name of display */
    int      maxiter;       /* max # of iterations */
    double   xmin, ymin;    /* lower left corner of rectangle */
    double   xmax, ymax;    /* upper right corner */
    int      width;         /* of display in pixels */
    int      height;        /* of display in pixels */
} cmdline;

/* set up 4 blocks */
int          blockcounts[4] = {50,1,4,2};
MPI_Datatype types[4];
MPI_Aint     displs[4];
MPI_Datatype cmdtype;

/* initialize types and displs with addresses of items */
MPI_Address(&cmdline.display, &displs[0]);
MPI_Address(&cmdline.maxiter, &displs[1]);
MPI_Address(&cmdline.xmin,    &displs[2]);
MPI_Address(&cmdline.width,   &displs[3]);
types[0] = MPI_CHAR;
types[1] = MPI_INT;
types[2] = MPI_DOUBLE;
types[3] = MPI_INT;
for (i = 3; i >= 0; i--)
    displs[i] -= displs[0];
MPI_Type_struct(4, blockcounts, displs, types, &cmdtype);
MPI_Type_commit(&cmdtype);
```

Structures

- Structures are described by *arrays* of
  - number of elements (`array_of_len`)
  - displacement or location (`array_of_displs`)
  - datatype (`array_of_types`)

```
MPI_Type_struct(count, array_of_len,
                array_of_displs,
                array_of_types, &newtype);
```

C++ Objects

- Objects are combinations of data and functions
  - Literally, a C `struct` with function pointers
  - Can associate actions with functions on the object (e.g., construction, destruction)
- MPI is only built upon moving data, not functions
  - MPI can only "fill" an object's data, just like a `struct`
  - Does not automatically perform any actions or functions on the object

C++ Objects

- Ramifications:
  - Objects have to be instantiated on receiving side before they can be received
  - A member (or `friend`) function must receive the data buffer and "fill" the object (and vice versa for sending; a member/`friend` function must marshall the data and send the buffer)
  - MPI does not combine the receive and instantiation (nor the send with destruction)
  - Other products can literally move objects from one process to another (SOM, CORBA, DCOM), but are more "distributed" rather than "parallel"
- Alternatives:
  - Object Oriented MPI (OOMPI): `http://www.osl.iu.edu/research/oompi/`

Vectors Revisited

- This code creates a datatype for an arbitrary number of elements in a row of an array stored in Fortran order (column first).

```
int blens[2], displs[2];
MPI_Datatype types[2], rowtype;
blens[0]  = 1;
blens[1]  = 1;
displs[0] = 0;
displs[1] = number_in_column * sizeof(double);
types[0]  = MPI_DOUBLE;
types[1]  = MPI_UB;
MPI_Type_struct(2, blens, displs, types, &rowtype);
MPI_Type_commit(&rowtype);
```

- To send n elements, you can use

```
MPI_Send(buf, n, rowtype, ...);
```

Structures Revisited

- When sending an array of structures, it is important to ensure that MPI and the C compiler have the same value for the size of each structure.
- Most portable way to do this is to use `MPI_UB` in the structure definition for the end of the structure. In the previous example, this would be:

```
/* initialize types and displs with addresses of items */

MPI_Address(&cmdline.display, &displs[0]);
MPI_Address(&cmdline.maxiter, &displs[1]);
MPI_Address(&cmdline.xmin,    &displs[2]);
MPI_Address(&cmdline.width,   &displs[3]);
MPI_Address(&cmdline+1,       &displs[4]);

types[0] = MPI_CHAR;
types[1] = MPI_INT;
types[2] = MPI_DOUBLE;
types[3] = MPI_INT;
types[4] = MPI_UB;

for (i = 4; i >= 0; i--)
  displs[i] -= displs[0];

MPI_Type_struct(5, blockcounts, displs, types, &cmdtype);
MPI_Type_commit(&cmdtype);
```

Interleaving Data

- By moving the UB inside the data, you can interleave data.
- Consider the matrix

To rank 0 →	0	8	16	24	32	40	48	56	← To rank 2
	1	9	17	25	33	41	49	57	
	2	10	18	26	34	42	50	58	
	3	11	19	27	35	43	51	59	
To rank 1 →	4	12	20	28	36	44	52	60	← To rank 3
	5	13	21	29	37	45	53	61	
	6	14	22	30	38	46	54	62	
	7	15	23	31	39	47	55	63	

- We wish to send 0-3,8-11,16-19, and 24-27 to rank 0; 4-7,12-15,20-23, and 28-31 to rank 1; etc.
- How can we do this with `MPI_SCATTERV`?

An Interleaved Datatype

- To define a block of this matrix (in C):

```
MPI_Type_vector(4, 4, 8, MPI_DOUBLE, &vec);
```

- To define a block whose extent is just one entry:

```
blens[0]  = 1;   blens[1] = 1;
types[0]  = vec; types[1] = MPI_UB;
displs[0] = 0;   displs[1] = sizeof(double);
MPI_Type_struct(2, blens, displs, types,
                &block);
```

Scattering a Matrix

- We set the displacements for each block as the location of the first element in the block.
- This works because `MPI_SCATTERV` uses the extents to determine the start of each piece to send.

```
scdispls[0] = 0;   sendcounts[0] = 1;
scdispls[1] = 4;   sendcounts[1] = 1;
scdispls[2] = 32;  sendcounts[2] = 1;
scdispls[3] = 36;  sendcounts[3] = 1;

MPI_Scatterv(sendbuf, sendcounts, scdispls, block,
             recvbuf, 16, MPI_DOUBLE, 0,
             MPI_COMM_WORLD);
```

Lab - Datatypes

- Create a datatype called submatrix that consists of elements in alternate rows and alternate columns of the given original matrix.
- Use MPI_SENDRECV to send the submatrix from a process to itself and print the results. To test this program you can run the program on just one processor.
- For example, if the given matrix is:

```
1  2  3   4  5  6
7  8  9  10 11 12
13 14 15 16 17 18
```

- The submatrix created should look like:

```
 1  3  5
13 15 17
```

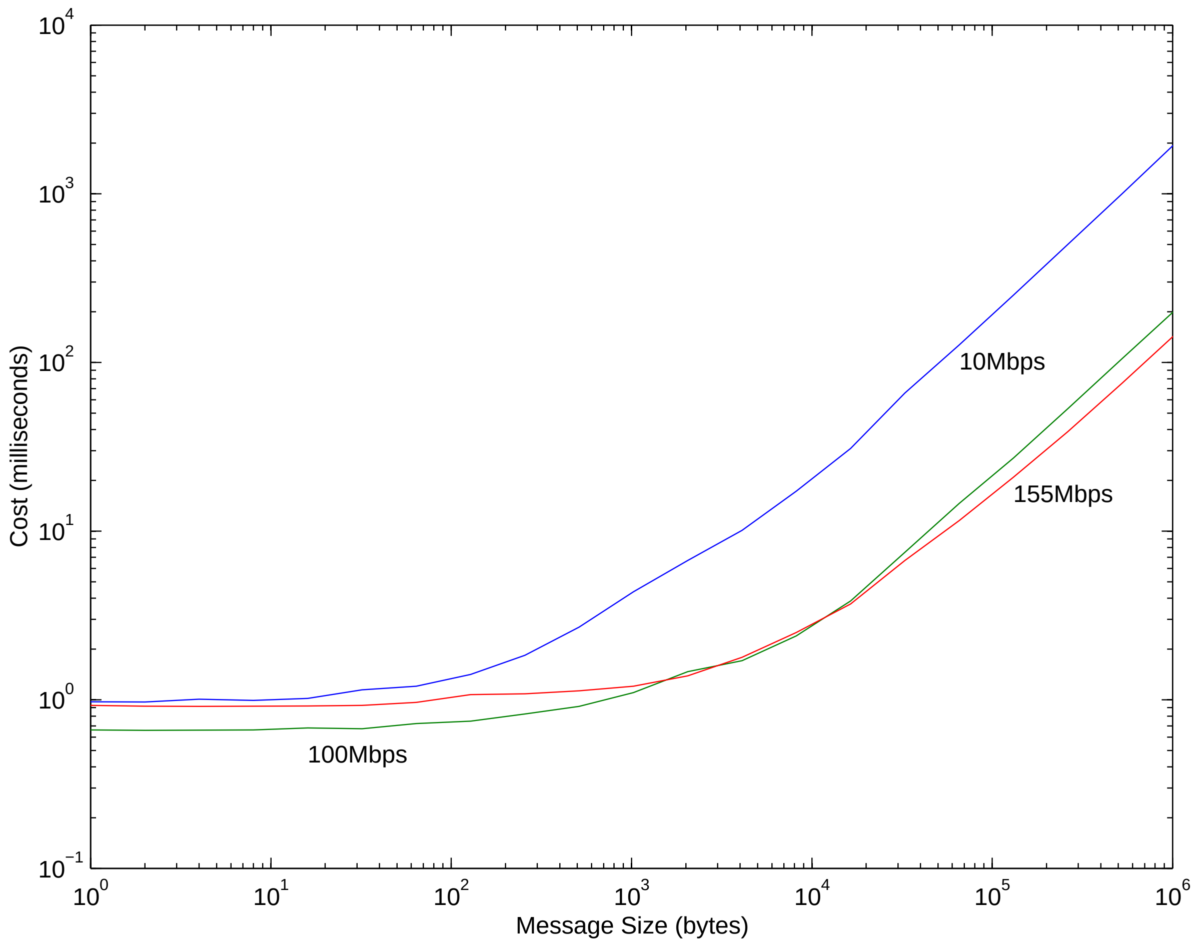
Section VIII

MPI Idioms for High-performance

Latency and Bandwidth

- Latency [=] time
  - A measure of a duration of time for an operation to transpire
  - A measure of a “fixed cost” associated with an operation
  - Includes *overhead* costs in software and hardware
  - *Zero message latency* or “startup time”. Time to send an empty message
- Bandwidth [=] bytes/time
  - A measure of a rate of transfer
  - A measure of the size dependent cost of an operation
  - *Asymptotic* bandwidth is the rate for sending an infinitely long message
  - *Contended* bandwidth is the actual bandwidth of a network considering congestion from multiple transfers

Latency and Bandwidth



Message Size and Frequency

- Size and frequency are not inter-changeable
- The total cost of sending a message is

$$T_{total} = T_{latency} + N/Bandwidth$$

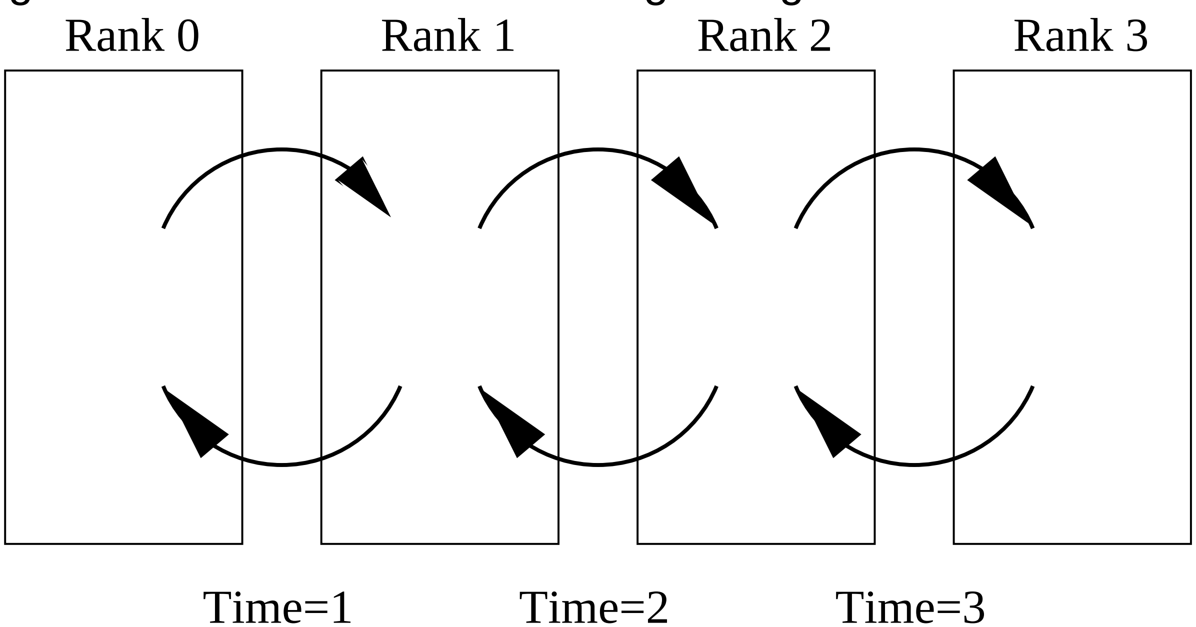
- The choice of size and frequency affects performance
- Multiple small messages should be collected into larger messages

Message Size and Frequency

- Cluster-based parallel machines:
  - Favor fewer and larger message because **latency** is high
  - For example, 100Mbps switched ethernet, all messages under approximately 1K bytes take the same amount of time to transmit
- “Real” parallel machines:
  - Latency is lower (faster interconnection between CPUs)
  - Tradeoff point may be different

Serialization

- Consider the following communication pattern — we wish each process to exchange a data item with its left and right neighbors.



Serialization

- One approach
  - Everyone sends right
  - Everyone receives from the left
  - Everyone sends left
  - Everyone receives from the right
- Nice, parallel approach (?)

Serialization

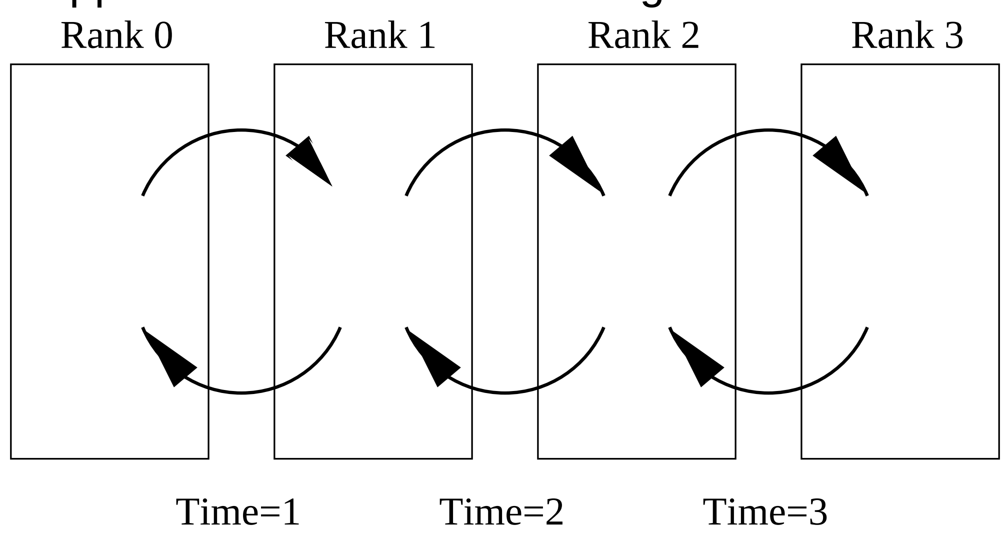
- MPI implementation (code snippet)

```
if (my_rank != size - 1)
  MPI_Send(right);
if (my_rank != 0)
  MPI_Recv(left);
if (my_rank != 0)
  MPI_Send(left);
if (my_rank != size - 1)
  MPI_Recv(right);
```

- What is wrong with this approach?

Avoiding Serialization

- The suggested approach may induce serialization of the communication
- The sends may not complete until there is a matching receive (why?)
- Initially there will only be one receive posted
- Creates a daisy-chain effect
- What would happen if we wanted to exchange data around a ring?



A Better Implementation

- Code snippet:

```
if (my_rank != 0)
  MPI_Irecv(left);
if (my_rank != size - 1)
  MPI_Irecv(right);
if (my_rank != size - 1)
  MPI_Send(right);
if (my_rank != 0)
  MPI_Send(left);
/* ... wait for recvs to complete */
```

A Better Implementation

- Why is this better?
- How can you receive data before it is sent?

Overlapping Communication and Computation

- There is lots of “wasted” time spent waiting for sends and receives to complete
- Better to do some computation while waiting
- Use non-blocking sends and receives
- **BUT:** Be aware that communication is not guaranteed to take place in the background with non-blocking operations

Overlapping Communication and Computation

1. Post non-blocking (perhaps persistent) receive
2. Post (non-blocking, persistent) send
3. While receive has not completed
   - do some computation
4. Handle received message
5. Wait for sent messages to complete

Overlapping Communication and Computation

- What MPI calls would you use for each step above?
- Why do we want to wait for sent messages to complete?
- What does it mean for the sent messages to complete?

Overlapping Communication and Computation

- Code snippet:

```
if (my_rank != 0)
  MPI_Irecv(left);
if (my_rank != size - 1)
  MPI_Irecv(right);
if (my_rank != size - 1)
  MPI_Isend(right);
if (my_rank != 0)
  MPI_Isend(left);
/* Do some computation */
/* ... wait for sends and recvs to complete */
```

Non-blocking "Gotchas"

- Be careful about abusing number of outstanding asynchronous communication requests
  - Causes more overhead in the MPI layer
  - Buffers for the pending sends and receives can be expensive memory-wise, which will also hurt performance

Make sure you understand the difference between non-blocking communication and background communication operations.

Lab - Idioms

- Implement the very first lab exercise
- Algorithm (for each processor)
  - Initialize $x =$ number of neighbors
  - Update x with average of neighbor's values of x
  - Repeat until done