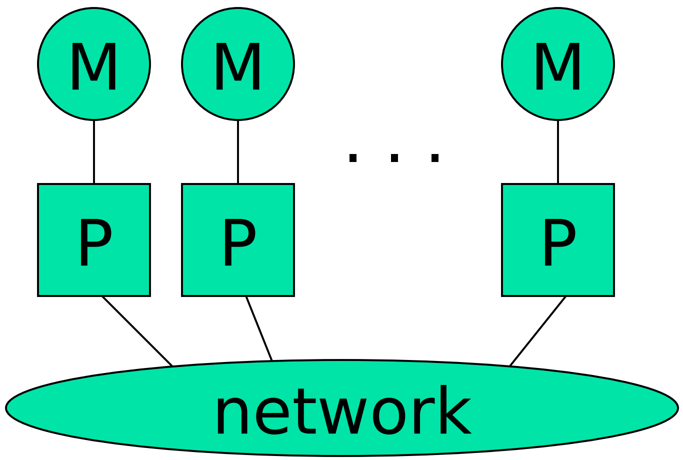
Principles of High Performance Computing (ICS 632)

Message Passing with MPI

Outline

- Message Passing
- MPI
  - Point-to-Point Communication
  - Collective Communication

Message Passing



- Each processor runs a process
- Processes communicate by exchanging messages
- They cannot share memory in the sense that they cannot address the same memory cells

- The above is a programming model and things may look different in the actual implementation (e.g., MPI over Shared Memory)
- Message Passing is popular because it is general:
  - Pretty much any distributed system works by exchanging messages, at some level
  - Distributed- or shared-memory multiprocessors, networks of workstations, uniprocessors
- It is not popular because it is easy (it’s not)

Code Parallelization

- Shared-memory programming
  - Parallelizing existing code can be very easy
    - OpenMP: just add a few pragmas
    - Pthreads: wrap work in do_work functions
  - Understanding parallel code is easy
  - Incremental parallelization is natural
- Distributed-memory programming
  - parallelizing existing code can be very difficult
    - No shared memory makes it impossible to “just” reference variables
    - Explicit message exchanges can get really tricky
  - Understanding parallel code is difficult
    - Data structured are split all over different memories
  - Incremental parallelization can be challenging

Programming Message Passing

- Shared-memory programming is simple conceptually (sort of)
- Shared-memory machines are expensive when one wants a lot of processors
- It’s cheaper (and more scalable) to build distributed memory machines
  - Distributed memory supercomputers (IBM SP series)
  - Commodity clusters
- But then how do we program them?
- At a basic level, let the user deal with explicit messages
  - difficult
  - but provides the most flexibility

Message Passing

- Isn't exchanging messages completely known and understood?
  - That's the basis of the IP idea
  - Networked computers running programs that communicate are very old and common
    - DNS, e-mail, Web, ...
- The answer is that, yes it is, we have "Sockets"
  - Software abstraction of a communication between two Internet hosts
  - Provides and API for programmers so that they do not need to know anything (or almost anything) about TCP/IP and write code with programs that communicate over the internet

Socket Library in UNIX

- Introduced by BSD in 1983
  - The “Berkeley Socket API”
  - For TCP and UDP on top of IP
- The API is known to not be very intuitive for first-time programmers
- What one typically does is write a set of “wrappers” that hide the complexity of the API behind simple function
- Fundamental concepts
  - Server side
    - Create a socket
    - Bind it to a port numbers
    - Listen on it
    - Accept a connection
    - Read/Write data
  - Client side
    - Create a socket
    - Connect it to a (remote) host/port
    - Write/Read data

Socket: server.c

```
int main(int argc, char *argv[])
{
    int sockfd, newsockfd, portno, clilen;
    char buffer[256];
    struct sockaddr_in serv_addr, cli_addr;
    int n;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = 666;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
    listen(sockfd,5);
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
    bzero(buffer,256);
    n = read(newsockfd,buffer,255);
    printf("Here is the message: %s\n",buffer);
    n = write(newsockfd,"I got your message",18);
    return 0;
}
```

Socket: client.c

```
int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    portno = 666;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    server = gethostbyname("server_host.univ.edu);
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,(char *)&serv_addr.sin_addr.s_addr,server->h_length);
    serv_addr.sin_port = htons(portno);
    connect(sockfd,&serv_addr,sizeof(serv_addr));
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    write(sockfd,buffer,strlen(buffer));
    bzero(buffer,256);
    read(sockfd,buffer,255);
    printf("%s\n",buffer);
    return 0;
}
```

Socket in C/UNIX

- The API is really not very simple
  - And note that the previous code does not have any error checking
  - Network programming is an area in which you should check ALL possible error code
  - In the end, writing a server that receives a message and sends back another one, with the corresponding client, can require 100+ lines of C if one wants to have robust code
  - This is OK for UNIX programmers, but not for everyone
  - However, nowadays, most applications written require some sort of Internet communication

Sockets in Java

- Socket class in java.net
  - Makes things a bit simpler
  - Still the same general idea
  - With some Java stuff

- Server

```
try { serverSocket = new ServerSocket(666);
} catch (IOException e) { <something> }
Socket clientSocket = null;
try { clientSocket = serverSocket.accept();
} catch (IOException e) { <something> }
PrintWriter out = new
   PrintWriter(                    clientSocket.getOutputStream()
   , true);
BufferedReader in = new BufferedReader(                    new
   InputStreamReader(clientSocket.getInputStream()));
// read from "in",  write to "out"
```

Sockets in Java

- Java client

```
try {socket = new Socket("server.univ.edu", 666);}
  catch { <something> }
out = new PrintWriter(socket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(
                        socket.getInputStream()));
 // write to out, read from in
```

- Much simpler than the C
- Note that if one writes a client-server program one typically creates a Thread after an accept, so that requests can be handled concurrently

Using Sockets for parallel programming?

- One could thing of writing all parallel code on a cluster using sockets
  - n nodes in the cluster
  - Each node creates n-1 sockets on n-1 ports
  - All nodes can communicate
- Problems with this approach
  - Complex code
  - Only point-to-point communication
  - No notion of types messages
  - But
    - All this complexity could be “wrapped” under a higher-level API
    - And in fact, we’ll see that’s the basic idea
  - Does not take advantage of fast networking within a cluster/ MPP
    - Sockets have “Internet stuff” in them that’s not necessary
    - TPC/IP may not even be the right protocol!

Message Passing for Parallel Programs

- Although "systems" people are happy with sockets, people writing parallel applications need something better
  - easier to program to
  - able to exploit the hardware better within a single machine
- This "something better" right now is MPI
  - We will learn how to write MPI programs
- Let's look at the history of message passing for parallel computing

A Brief History of Message Passing

- Vendors started building dist-memory machines in the late 80's
- Each provided a message passing library
  - Caltech's Hypercube and Crystalline Operating System (CROS) - 1984
    - communication channels based on the hypercube topology
    - only collective communication at first, moved to an address-based system
    - only 8 byte messages supported by CROS routines!
    - good for very regular problems only
  - Meiko CS-1 and Occam - circa 1990
    - transputer based (32-bit processor with 4 communication links, with fast multitasking/multithreading)
    - Occam: formal language for parallel processing:

      chan1 ! data sending data (synchronous)

      chan1 ? data receiving data

      par, seq parallel or sequential block
    - Easy to write code that deadlocks due to synchronicity
    - Still used today to reason about parallel programs (compilers available)
    - Lesson: promoting a parallel language is difficult, people have to embrace it
      - better to do extensions to an existing (popular) language
      - better to just design a library

A Brief History of Message Passing

...

- The Intel iPSC1, Paragon and NX
  - Originally close to the Caltech Hypercube and CROS
  - iPSC1 had commensurate message passing and computation performance
  - hiding of underlying communication topology (process rank), multiple processes per node, any-to-any message passing, non-syn chronous messages, message tags, variable message lengths
  - On the Paragon, NX2 added interrupt-driven communications, some notion of filtering of messages with wildcards, global synchronization, arithmetic reduction operations
  - **ALL** of the above are part of modern message passing
- IBM SPs and EUI
- Meiko CS-2 and CSTools,
- Thinking Machine CM5 and the CMMD Active Message Layer (AML)

A Brief History of Message Passing

- We went from a highly restrictive system like the Caltech hypercube to great flexibility that is in fact very close to today's state-of-the-art of message passing
- The main problem was: impossible to write portable code!
  - programmers became expert of one system
  - the systems would die eventually and one had to relearn a new system
  - for instance, I learned NX!
- People started writing "portable" message passing libraries
  - Tricks with macros, PICL, P4, PVM, PARMACS, CHIMPS, Express, etc.
- The main problems was performance
  - if I invest millions in an IBM-SP, do I really want to use some library that uses (slow) sockets??
- There was no clear winner for a long time
  - although PVM had won in the end
- After a few years of intense activity and competition, it was agreed that a message passing standard should be developed
  - Designed by committee

The MPI Standard

- MPI Forum setup as early as 1992 to come up with a de facto standard with the following goals:
  - source-code portability
  - allow for efficient implementation (e.g., by vendors)
  - support for heterogeneous platforms
- MPI is not
  - a language
  - an implementation (although it provides hints for implementers)
- June 1995: MPI v1.1 (we're now at MPI v1.2)
  - http://www-unix.mcs.anl.gov/mpi/
  - C and FORTRAN bindings
  - We will use MPI v1.1 from C in the class
- Implementations:
  - well-adopted by vendors
  - free implementations for clusters: MPICH, LAM, CHIMP/MPI
  - research in fault-tolerance: MPICH-V, FT-MPI, MPIFT, etc.

SPMD Programs

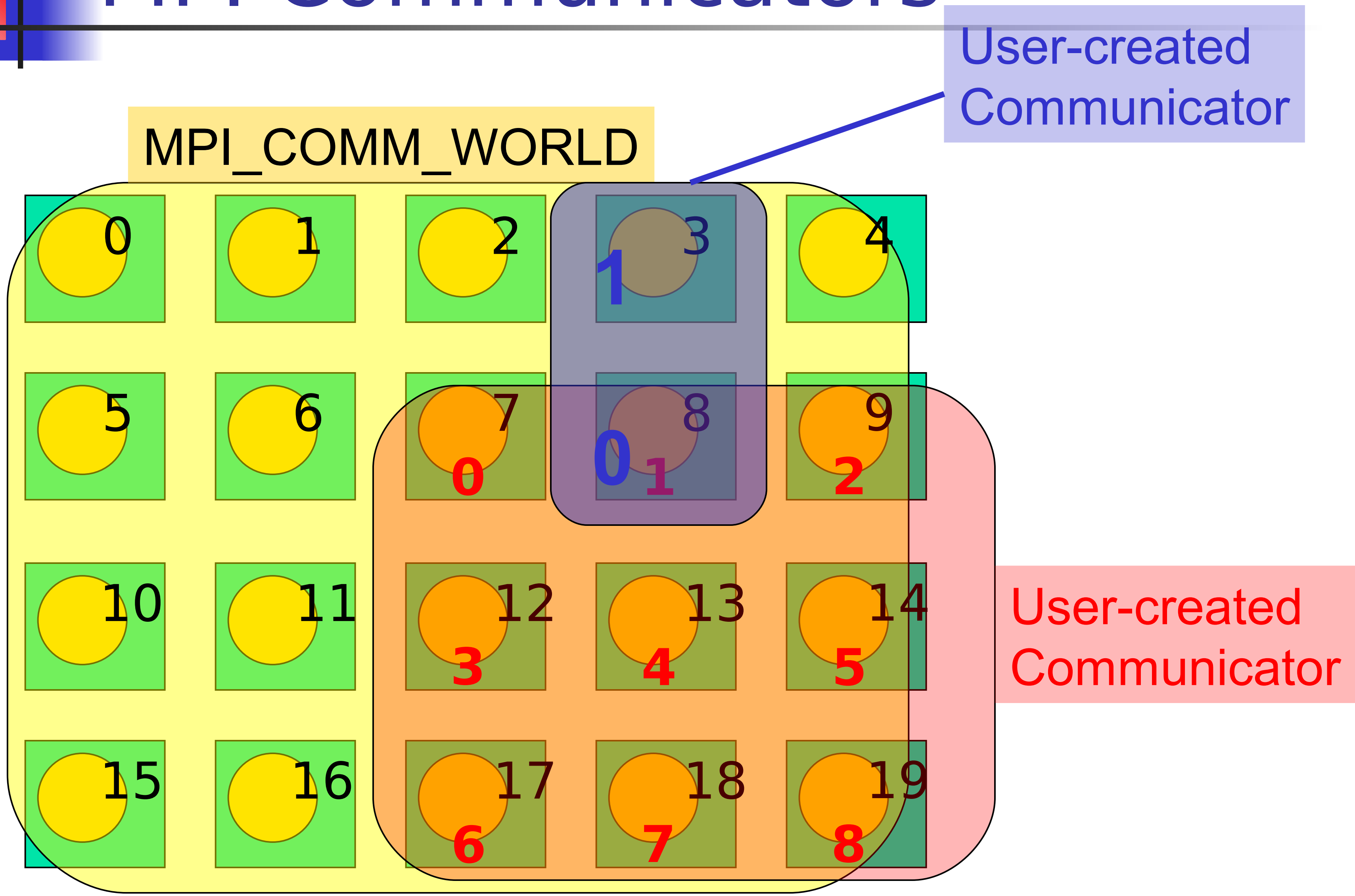
- It is rare for a programmer to write a different program for each process of a parallel application
- In most cases, people write Single Program Multiple Data (SPMD) programs
  - the same program runs on all participating processors
  - processes can be identified by some *rank*
  - This allows each process to know which piece of the problem to work on
  - This allows the programmer to specify that some process does something, while all the others do something else (common in master-worker computations)

```
main(int argc, char **argv) {
   if (my_rank == 0) { /* master */
     ... load input and dispatch ...
   } else { /* workers */
     ... wait for data and compute ...
   }
```

MPI Concepts

- Fixed number of processors
  - When launching the application one must specify the number of processors to use, which remains unchanged throughout execution
- Communicator
  - Abstraction for a group of processes that can communicate
  - A process can belong to multiple communicators
  - Makes is easy to partition/organize the application in multiple layers of communicating processes
  - Default and global communicator: ***MPI_COMM_WORLD***
- Process Rank
  - The index of a process within a communicator
  - Typically user maps his/her own virtual topology on top of just linear ranks
    - ring, grid, etc.

MPI Communicators



A First MPI Program

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int my_rank, n;
  char hostname[128];
  MPI_init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  MPI_Comm_size(MPI_COMM_WORLD,&n);
  gethostname(hostname,128);
  if (my_rank == 0) { /* master */
    printf("I am the master: %s\n",hostname);
  } else { /* worker */
    printf("I am a worker: %s (rank=%d/%d)\n",
           hostname,my_rank,n-1);
  }
  MPI_Finalize();
  exit(0);
}
```

Has to be called first, and once

Has to be called last, and once

Compiling/Running it

- Compile with mpicc
- Run with ***mpirun***

  % mpirun -np 4 my_program <args>

  - requests 4 processors for running ***my_program*** with command-line arguments
  - see the ***mpirun*** man page for more information
  - in particular the ***-machinefile*** option that is used to run on a network of workstations
- Some systems just run all programs as MPI programs and no explicit call to ***mpirun*** is actually needed
- Previous example program:

```
% mpirun -np 3 -machinefile hosts my_program
  I am the master: somehost1
  I am a worker: somehost2 (rank=2/2)
  I am a worker: somehost3 (rank=1/2)
```

(stdout/stderr redirected to the process calling mpirun)

MPI on our Cluster

- OpenMPI
  - /usr/bin/mpirun
  - /usr/bin/mpicc
- MPICH
  - /opt/mpich/gnu/bin/mpirun
  - /opt/mpich/gnu/bin/mpicc
- Your batch script should ask for >=1 nodes and call mpirun appropriately
- Remember the example we ran in class:

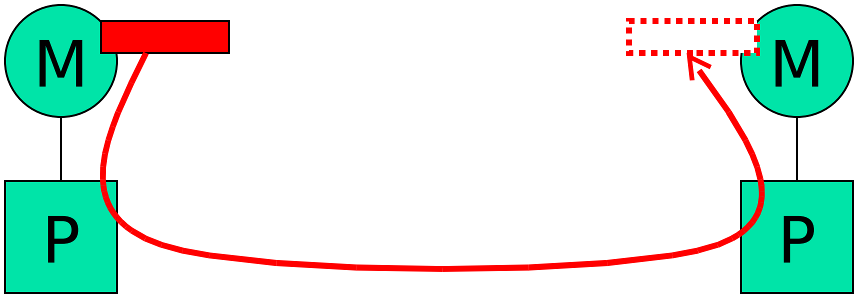
```
#
#PBS -l nodes=6
#PBS -l walltime=5:00:00
#PBS -o myprogram.out
#PBS -e myprogram.err

cd $PBS_O_WORKDIR
mpirun -np 6 -machinefile $PBS_NODEFILE ./hello_world
```

Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Data Types
- One slide on MPI-2

Point-to-Point Communication



- Data to be communicated is described by three things:
  - address
  - data type of the message
  - length of the message
- Involved processes are described by two things
  - communicator
  - rank
- Message is identified by a “tag” (integer) that can be chosen by the user

Point-to-Point Communication

- Two modes of communication:
  - Synchronous: Communication does not complete until the message has been received
  - Asynchronous: Completes as soon as the message is “on its way”, and hopefully it gets to destination
- MPI provides four versions
  - synchronous, buffered, standard, ready

Synchronous/Buffered sending in MPI

- Synchronous with MPI_Ssend
  - The send completes only once the receive has succeeded
    - copy data to the network, wait for an ack
    - The sender has to wait for a receive to be posted
    - No buffering of data
- Buffered with MPI_Bsend
  - The send completes once the message has been buffered internally by MPI
    - Buffering incurs an extra memory copy
    - Doe not require a matching receive to be posted
    - May cause buffer overflow if many bsends and no matching receives have been posted yet

Standard/Ready Send

- Standard with MPI_Send
  - Up to MPI to decide whether to do synchronous or buffered, for performance reasons
  - The rationale is that a correct MPI program should not rely on buffering to ensure correct semantics
- Ready with MPI_Rsend
  - May be started *only* if the matching receive has been posted
  - Can be done efficiently on some systems as no hand-shaking is required

MPI_RECV

- There is only one MPI_Recv, which returns when the data has been received.
  - only specifies the **MAX** number of elements to receive
- Why all this junk?
  - Performance, performance, performance
  - MPI was designed with constructors in mind, who would endlessly tune code to extract the best out of the platform (LINPACK benchmark).
  - Playing with the different versions of MPI_?send can improve performance without modifying program semantics
  - Playing with the different versions of MPI_?send can modify program semantics
  - Typically parallel codes do not face very complex distributed system problems and it's often more about performance than correctness.
  - You'll want to play with these to tune the performance of your code in your assignments

Example: Sending and Receiving

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int i, my_rank, nprocs, x[4];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  if (my_rank == 0) { /* master */
    x[0]=42; x[1]=43; x[2]=44; x[3]=45;
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    for (i=1;i<nprocs;i++)
      MPI_Send(x,4,MPI_INT,i,0,MPI_COMM_WORLD);
  } else { /* worker */
    MPI_Status status;
    MPI_Recv(x,4,MPI_INT,0,0,MPI_COMM_WORLD,&status);
  }
  MPI_Finalize();
  exit(0);
}
```

destination and source

user-defined tag

Max number of elements to receive

Can be examined via calls like MPI_Get_count(), etc.

Example: Deadlock

... *MPI_Ssend()* *MPI_Recv()* ...	**Deadlock**	... *MPI_Ssend()* *MPI_Recv()* ...
... *MPI_Buffer_attach()* *MPI_Bsend()* *MPI_Recv()* ...	**No Deadlock**	... *MPI_Buffer_attach()* *MPI_Bsend()* *MPI_Recv()* ...
... *MPI_Buffer_attach()* *MPI_Bsend()* *MPI_Recv()* ...	**No Deadlock**	... *MPI_Ssend()* *MPI_Recv()* ...

What about MPI_Send?

- MPI_Send is either synchronous or buffered....
- With , running “some” version of MPICH

Deadlock

```
...
MPI_Send()
MPI_Recv()
...
```

Data size > 127999 bytes

Data size < 128000 bytes

```
...
MPI_Send()
MPI_Recv()
...
```

No Deadlock

- Rationale: a correct MPI program should not rely on buffering for semantics, just for performance.
- So how do we do this then? ...

Non-blocking communications

- So far we've seen blocking communication:
  - The call returns whenever its operation is complete (MPI_SSEND returns once the message has been received, MPI_BSEND returns once the message has been buffered, etc..)
- MPI provides non-blocking communication: the call returns immediately and there is another call that can be used to check on completion.
- Rationale: Non-blocking calls let the sender/receiver do something useful while waiting for completion of the operation (without playing with threads, etc.).

Non-blocking Communication

- MPI_Issend, MPI_Ibsend, MPI_Isend, MPI_Irsend, MPI_Irecv

```
MPI_Request request;
MPI_Isend(&x,1,MPI_INT,dest,tag,communicator,&request);
MPI_Irecv(&x,1,MPI_INT,src,tag,communicator,&request);
```

- Functions to check on completion: MPI_Wait, MPI_Test, MPI_Waitany, MPI_Testany, MPI_Waitall, MPI_Testall, MPI_Waitsome, MPI_Testsome.

```
MPI_Status status;
MPI_Wait(&request, &status) /* block */
MPI_Test(&request, &status) /* doesn't block */
```

Example: Non-blocking comm


```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int i, my_rank, x, y;
  MPI_Status status;
  MPI_Request request;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  if (my_rank == 0) { /* P0 */
    x=42;
    MPI_Isend(&x,1,MPI_INT,1,0,MPI_COMM_WORLD,&request);
    MPI_Recv(&y,1,MPI_INT,1,0,MPI_COMM_WORLD,&status);
    MPI_Wait(&request,&status);
 } else if (my_rank == 1) { /* P1 */
    y=41;
    MPI_Isend(&y,1,MPI_INT,0,0,MPI_COMM_WORLD,&request);
    MPI_Recv(&x,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
    MPI_Wait(&request,&status);
  }
  MPI_Finalize(); exit(0);
}
```

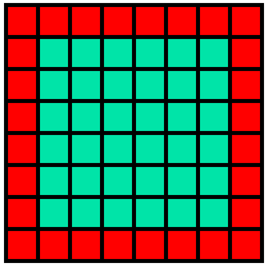
Use of non-blocking comms

- In the previous example, why not just swap one pair of send and receive?
- Example:
  - A logical linear array of N processors, needing to exchange data with their neighbor at each iteration of an application
  - One would need to orchestrate the communications:
    - all odd-numbered processors send first
    - all even-numbered processors receive first
  - Sort of cumbersome and can lead to complicated patterns for more complex examples
  - In this case: just use MPI_Isend and write much simpler code
- Furthermore, using MPI_Isend makes it possible to overlap useful work with communication delays:

```
MPI_Isend()
<useful work>
MPI_Wait()
```

Iterative Application Example

```
for (iterations)
  update all cells
  send boundary values
  receive boundary values
```



- Would deadlock with MPI_Ssend, and maybe deadlock with MPI_Send, so must be implemented with MPI_Isend
- Better version that uses non-blocking communication to achieve communication/computation overlap (aka latency hiding):

```
for (iterations)
   initiate sending of boundary values to neighbours;
   initiate receipt of boundary values from neighbours;
   update non-boundary cells;
   wait for completion of sending of boundary values;

   wait for completion of receipt of boundary values;
   update boundary cells;
```

- Saves cost of boundary value communication if hardware/software can overlap comm and comp

Non-blocking communications

- Almost always better to use non-blocking
  - communication can be carried out during blocking system calls
  - communication and communication can overlap
  - less likely to have annoying deadlocks
  - synchronous mode is better than implementing acks by hand though
- However, everything else being equal, non-blocking is slower due to extra data structure bookkeeping
  - The solution is just to benchmark
- When you do your programming assignments, you will play around with different communication types

More information

- There are many more functions that allow fine control of point-to-point communication
- Message ordering is guaranteed
- Detailed API descriptions at the MPI site at ANL:
  - Google “MPI”. First link.
  - Note that you should check error codes, etc.
- Everything you want to know about deadlocks in MPI communication

http://andrew.ait.iastate.edu/HPC/Papers/mpicheck2/mpicheck2.htm

Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Data Types
- One slide on MPI-2

Collective Communication

- Operations that allow more than 2 processes to communicate simultaneously
  - barrier
  - broadcast
  - reduce
- All these can be built using point-to-point communications, but typical MPI implementations have optimized them, and it's a good idea to use them
- In all of these, all processes place the same call (in good SPMD fashion), although depending on the process, some arguments may not be used

Barrier

- Synchronization of the calling processes
  - the call blocks until all of the processes have placed the call
- No data is exchanged
- Similar to an OpenMP barrier

```
...
MPI_Barrier(MPI_COMM_WORLD)
...
```

Broadcast

- One-to-many communication
- Note that multicast can be implemented via the use of communicators (i.e., to create processor groups)

```
...
MPI_Bcast(x, 4, MPI_INT, 0,
   MPI_COMM_WORLD)
...
```

Rank of the root

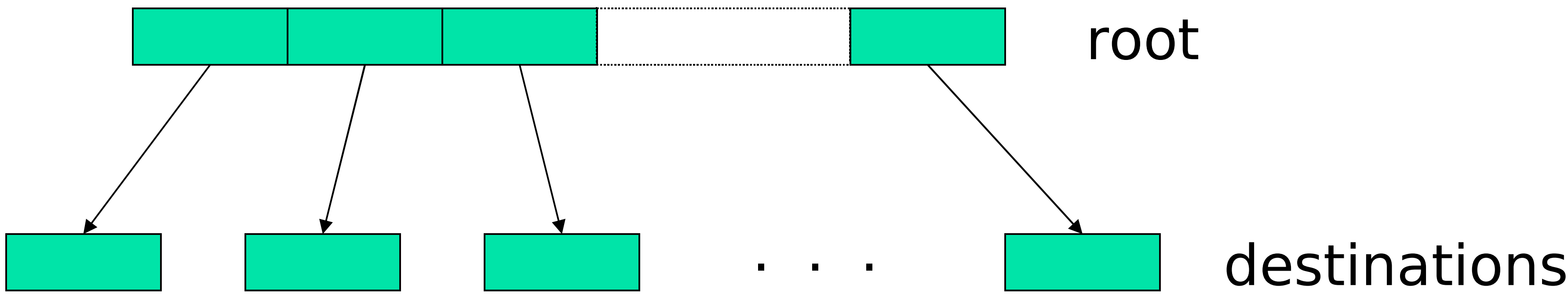
Broadcast example

- Let's say the master must send the user input to all workers

```
int main(int argc,char **argv) {
   int my_rank;
   int input;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
   if (argc != 2) exit(1);
    if (sscanf(argv[1],"%d",&input) != 1) exit(1);
   MPI_Bcast(&input,1,MPI_INT,0,MPI_COMM_WORLD);
   ...
}
```

Scatter

- One-to-many communication
- Not sending the same message to all




```
...
MPI_Scatter(x, 100, MPI_INT, y, 100, MPI_INT, 0,
   MPI_COMM_WORLD)
...
```

Send buffer

Data to send to each

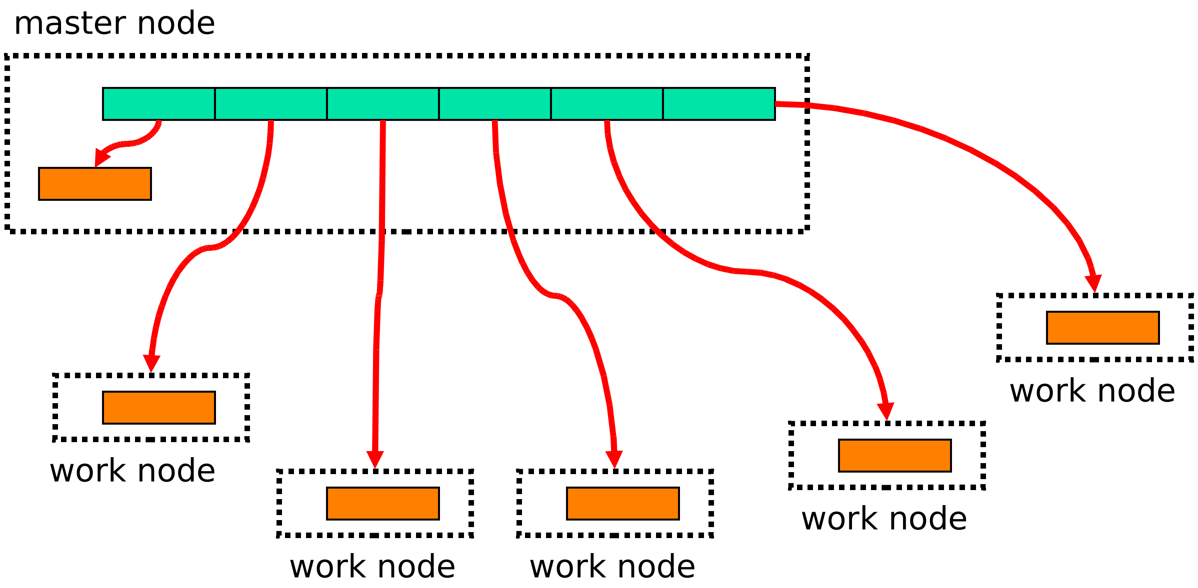
Receive buffer

Data to receive

Rank of the root

This is actually a bit tricky

- The root sends data to itself!



- Arguments #1, #2, and #3 are only meaningful at the root

Scatter Example

- Partitioning an array of input among workers

```
int main(int argc,char **argv) {
   int *a;
   double *revbuffer;
   ...
   MPI_Comm_size(MPI_COMM_WORLD,&n);
   <allocate array recvbuffer of size N/n>

   if (my_rank == 0) { /* master */
       <allocate array a of size N>
   }
   MPI_Scatter(a, N/n, MPI_INT,
                  recvbuffer, N/n, MPI_INT,
                  0, MPI_COMM_WORLD);
   ...
}
```

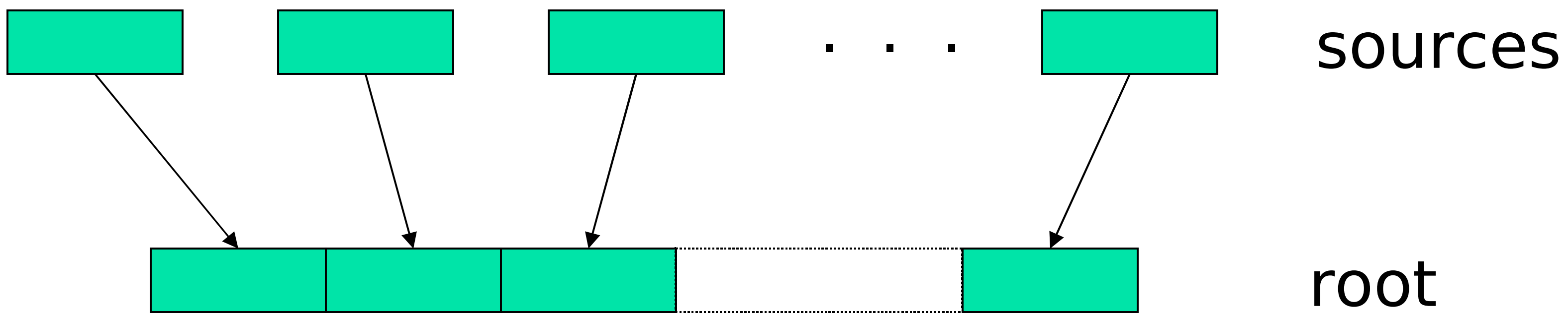
Scatter Example

- Without redundant sending at the root

```
int main(int argc,char **argv) {
   int *a;
   double *revbuffer;
   ...
   MPI_Comm_size(MPI_COMM_WORLD,&n);
   if (my_rank == 0) { /* master */
        <allocate array a of size N>
        <allocate array recvbuffer of size N/n>
        MPI_Scatter(a, N/n, MPI_INT,
                    MPI_IN_PLACE, N/n, MPI_INT,
                    0, MPI_COMM_WORLD);
   } else { /* worker */
        <allocate array recvbuffer of size N/n>
        MPI_Scatter(NULL, 0, MPI_INT,
                    recvbuffer, N/n, MPI_INT,
                    0, MPI_COMM_WORLD);
   }
   ...
}
```

Gather

- Many-to-one communication
- Not sending the same message to the root




```
...
MPI_Gather(x, 100, MPI_INT, y, 100, MPI_INT, 0, MPI_COMM_WORLD)
...
```

Send buffer

Receive buffer

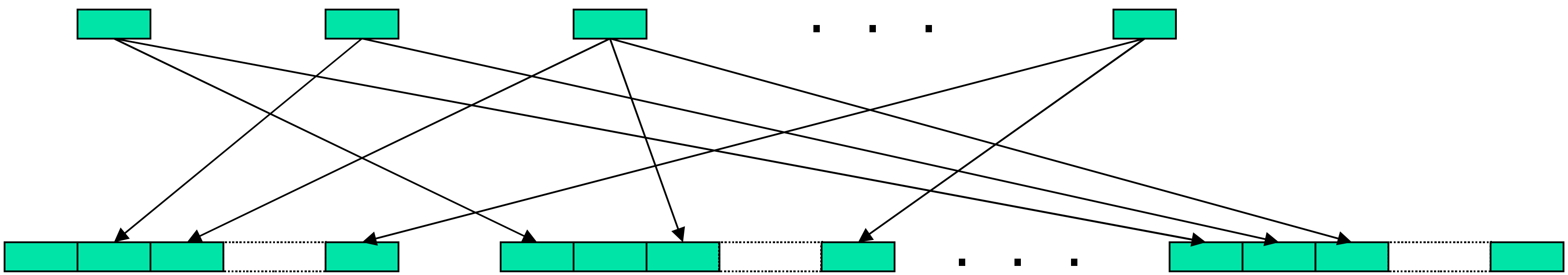
Rank of the root

Data to send from each

Data to receive

Gather-to-all

- Many-to-many communication
- Each process sends the same message to all
- Different Processes send different messages



```
...
MPI_Allgather(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)
...
```

Send buffer

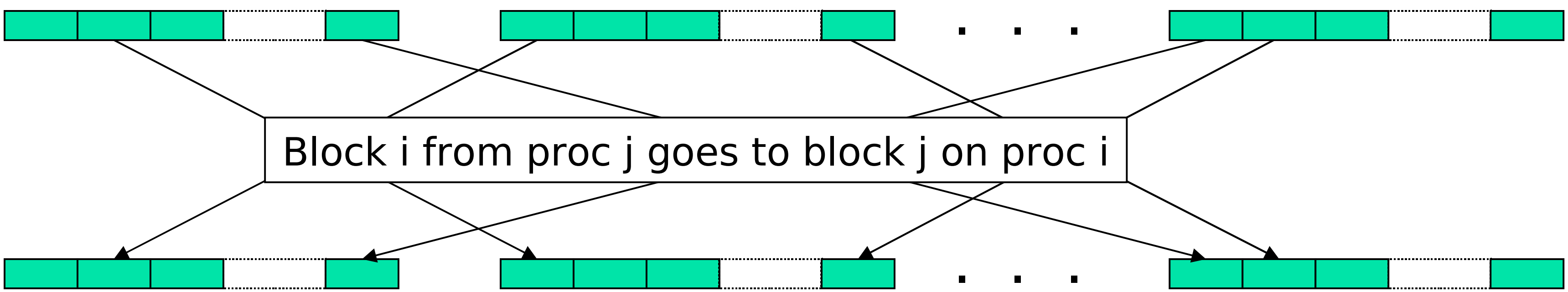
Data to send to each

Receive buffer

Data to receive

All-to-all

- Many-to-many communication
- Each process sends a different message to each other process




```
...
MPI_Alltoall(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)
...
```

Send buffer

Data to send to each

Receive buffer

Data to receive

Reduction Operations

- Used to compute a result from data that is distributed among processors
  - often what a user wants to do anyway
    - e.g., compute the sum of a distributed array
  - so why not provide the functionality as a single API call rather than having people keep re-implementing the same things
- Predefined operations:
  - MPI_MAX, MPI_MIN, MPI_SUM, etc.
- Possibility to have user-defined operations

MPI_Reduce, MPI_Allreduce

- MPI_Reduce: result is sent out to the root
  - the operation is applied element-wise for each element of the input arrays on each processor
  - An output array is returned
- MPI_Allreduce: result is sent out to everyone

```
...
MPI_Reduce(x, r, 10, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD)
...
```

input array | output array | array size | root

```
...
MPI_Allreduce(x, r, 10, MPI_INT, MPI_MAX, MPI_COMM_WORLD)
...
```

MPI_Reduce example

MPI_Reduce(sbuf,rbuf,6,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)

sbuf

P0	3	4	2	8	12	1
P1	5	2	5	1	7	11
P2	2	4	4	10	4	5
P3	1	6	9	3	1	1

rbuf

P0	11	16	20	22	24	18

MPI_Scan: Prefix reduction

- Process i receives data reduced on process 0 to i.

sbuf

P0	3	4	2	8	12	1
P1	5	2	5	1	7	11
P2	2	4	4	10	4	5
P3	1	6	9	3	1	1

rbuf

P0	3	4	2	8	12	1
P1	8	6	7	9	19	12
P2	10	10	11	19	23	17
P3	11	16	12	22	24	18

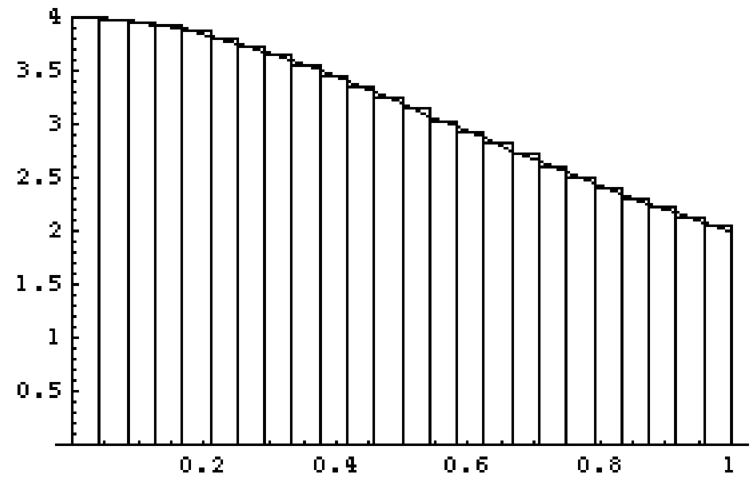
```
MPI_Scan(sbuf,rbuf,6,MPI_INT,MPI_SUM,MPI_COMM_WORLD)
```

And more...

- Most broadcast operations come with a version that allows for a stride (so that blocks do not need to be contiguous)
  - MPI_Gatherv(), MPI_Scatterv(), MPI_Allgatherv(), MPI_Alltoallv()
- MPI_Reduce_scatter(): functionality equivalent to a reduce followed by a scatter
- All the above have been created as they are common in scientific applications and save code
- All details on the MPI Webpage

Example: computing π

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$




```
int n; /* Number of rectangles */
int nproc, myrank;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_Size(MPI_COMM_WORLD,&nproc);
if (my_rank == 0)  read_from_keyboard(&n);
/* broadcast number of rectangles from root
   process to everybody else */
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
mypi = integral((n/nproc) * my_rank, (n/nproc) * (1+my_rank) - 1)
/* sum mypi across all processes, storing
   result as pi on root process */
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Using MPI to increase memory

- One of the reasons to use MPI is to increase the available memory
  - I want to sort an array
  - The array is 10GB
  - I can use 10 computers with each 1GB of memory
- Question: how do I write the code?
  - I cannot declare

```
#define SIZE (10*1024*1024*1024)
char array[SIZE]
```

Global vs. Local Indices

- Since each node gets only 1/10th of the array, each node declares only an array on 1/10th of the size
  - processor 0: char array[SIZE/10];
  - processor 1: char array[SIZE/10];
  - ...
  - processor p: char array[SIZE/10];
- When processor 0 references array[0] it means the first element of the global array
- When processor i references array[0] it means the (SIZE/10*i) element of the global array

Global vs. Local Indices

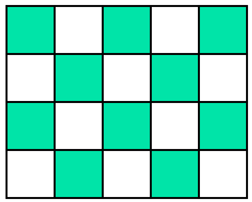
- There is a mapping from/to local indices and global indices
  - It can be a mental gymnastic
    - requires some potentially complex arithmetic expressions for indices
  - One can actually write functions to do this
    - e.g. global2local()
    - When you would write “a[i] * b[k]” for the sequential version of the code, you should write “a[global2local(i)]*b[global2local(k)]”
    - This may become necessary when index computations become too complicated
    - More on this when we see actual algorithms

Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Data Types
- One slide on MPI-2

More Advanced Messages

- Regularly strided data



Blocks/Elements of a matrix

- Data structure

```
struct {
    int a;
    double b;
}
```

- A set of variables

```
int a; double b; int x[12];
```

Problems with current messages

- Packing strided data into temporary arrays wastes memory
- Placing individual MPI_Send calls for individual variables of possibly different types wastes time
- Both the above would make the code bloated

→ Motivation for MPI's "derived data types"

Derived Data Types

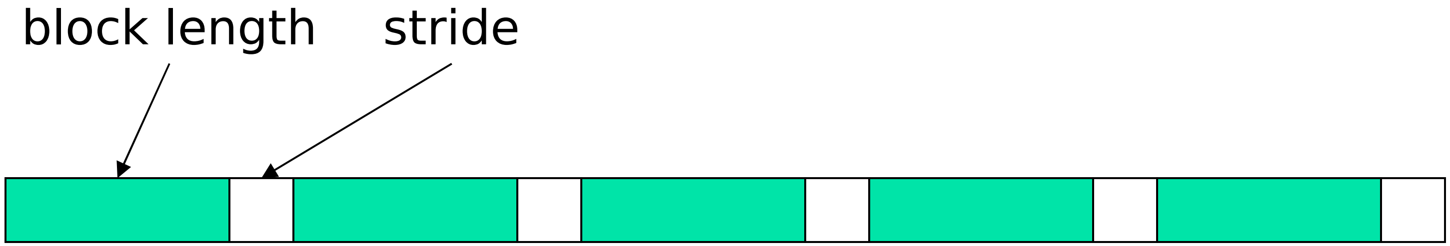
- A data type is defined by a “type map”
  - set of <type, displacement> pairs
- Created at runtime in two phases
  - Construct the data type from existing types
  - Commit the data type before it can be used
- Simplest constructor: contiguous type

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

MPI_Type_vector()

```
int MPI_Type_vector(int count,
      int blocklength, int stride
      MPI_Datatype oldtype,

      MPI_Datatype *newtype)
```

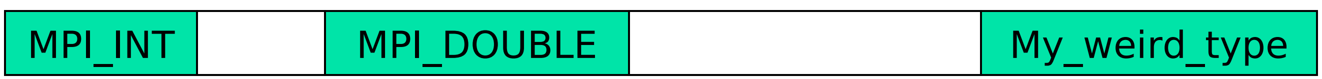


MPI_Type_indexed()

```
int MPI_Type_indexed(int count,
    int *array_of_blocklengths,
    int *array_of_displacements,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

MPI_Type_struct()

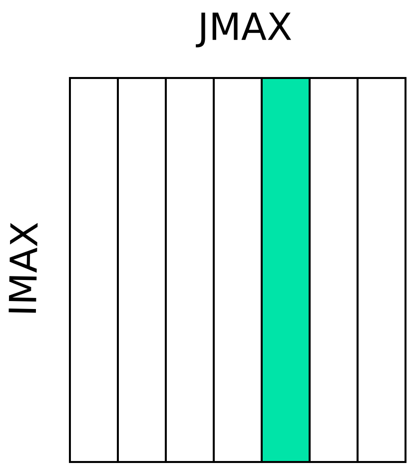
```
int MPI_Type_struct(int count,
    int *array_of_blocklengths,
    MPI_Aint *array_of_displacements,
    MPI_Datatype *array_of_types,
    MPI_Datatype *newtype)
```

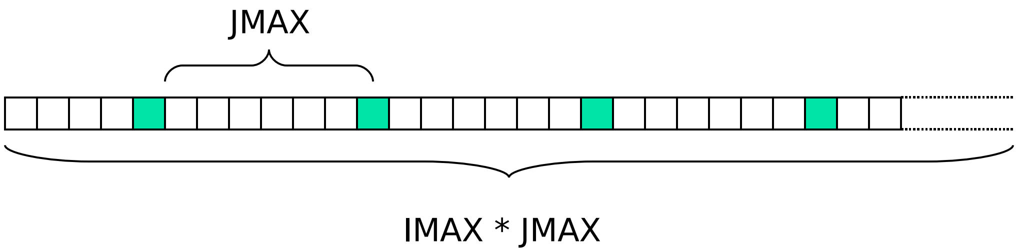


Derived Data Types: Example

- Sending the 5th column of a 2-D matrix:

```
double results[IMAX][JMAX];
MPI_Datatype newtype;
MPI_Type_vector (IMAX, 1, JMAX, MPI_DOUBLE, &newtype);
MPI_Type_Commit (&newtype);
MPI_Send(&(results[0][4]), 1, newtype, dest, tag, comm);
```





Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Data Types
- One slide on MPI-2

MPI-2

- MPI-2 provides for:
  - Remote Memory
    - put and get primitives, weak synchronization
    - makes it possible to take advantage of fast hardware (e.g., shared memory)
    - gives a shared memory twist to MPI
  - Parallel I/O
    - we'll talk about it later in the class
  - Dynamic Processes
    - create processes during application execution to grow the pool of resources
    - as opposed to "everybody is in MPI_COMM_WORLD at startup and that's the end of it"
    - as opposed to "if a process fails everything collapses"
    - a MPI_Comm_spawn() call has been added (akin to PVM)
  - Thread Support
    - multi-threaded MPI processes that play nicely with MPI
  - Extended Collective Communications
  - Inter-language operation, C++ bindings
  - Socket-style communication: open_port, accept, connect (client-server)
- MPI-2 implementations are now available