

Lecture 16

Programming with Message Passing

Announcements

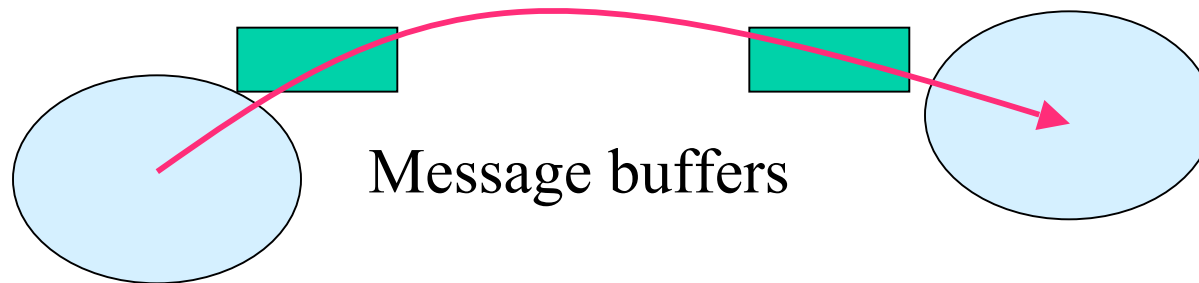
- Abe will be down on 12/1/08, plan accordingly
- Sharks and Fish

Programming with Message Passing

- **The** primary model for implementing high performance applications
- Explains fundamental behavior in multi- and single processors
- Programs execute as a set of P processes
 - We specify P when we run the program
 - Assume each process is assigned a different physical processor
- Each physical process
 - is initialized with the same code, but has private state
SPMD = “Same Program Multiple Data”
 - executes instructions at its own rate
 - has an associated *rank*, a unique integer in the range $0:P-1$
- The sequence of instructions each process executes depends on its rank and the messages it sends and receives
- Program execution is often called “bulk synchronous” or “loosely synchronous”

Message Passing

- Requires a sender and an explicit recipient
 - Must be aware of one another
- Messages are like email; to send one we specify
 - A destination
 - A message body (can be empty)
- To receive one we need similar information, including a receptacle to hold the incoming data
- Message passing performs two events
 - Memory to memory block copy
 - Synchronization signal on receiving end: “Data arrived”



A minimal interface

- Query functions
 - `nproc( )` = # processors
 - `myRank( )` = this process's rank
- *Point-to-point* communication
 - Simplest form of communication
 - Send a message to another process
`Send(Object, Destination process ID)`
 - Receive a message from another process
`Receive(Object)`
`Receive(Source process, Object)`

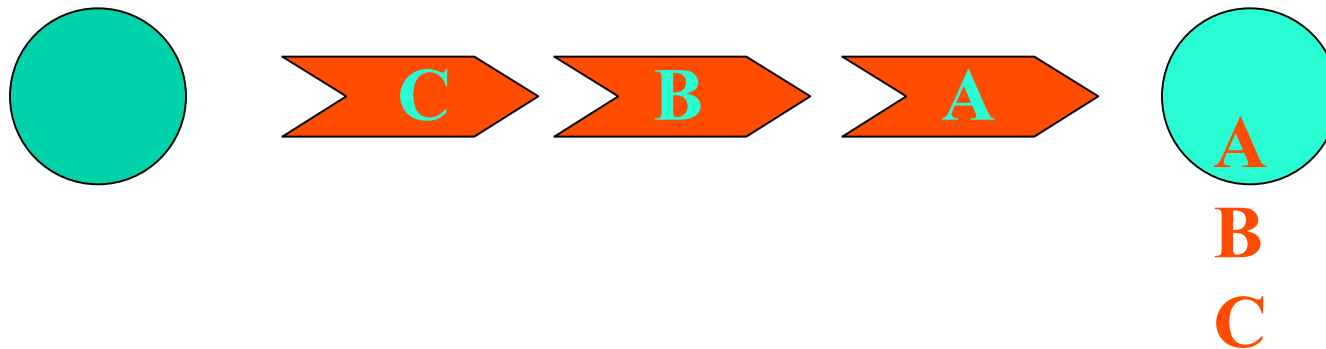
Send and Recv

- When **Send()** returns, the message is “in transit”
 - A return doesn’t tell us if the message has been received
 - Somewhere in the system
 - Safe to overwrite the buffer
- **Receive()** blocks until the message has been received
 - Safe to use the data in the buffer
- Error if the source and destination object don’t have *identical* types

Process 0	Process 1
Send(x,1)	Send(x,0)
Recv(y)	Recv(y)
Print x, y	Print x, y

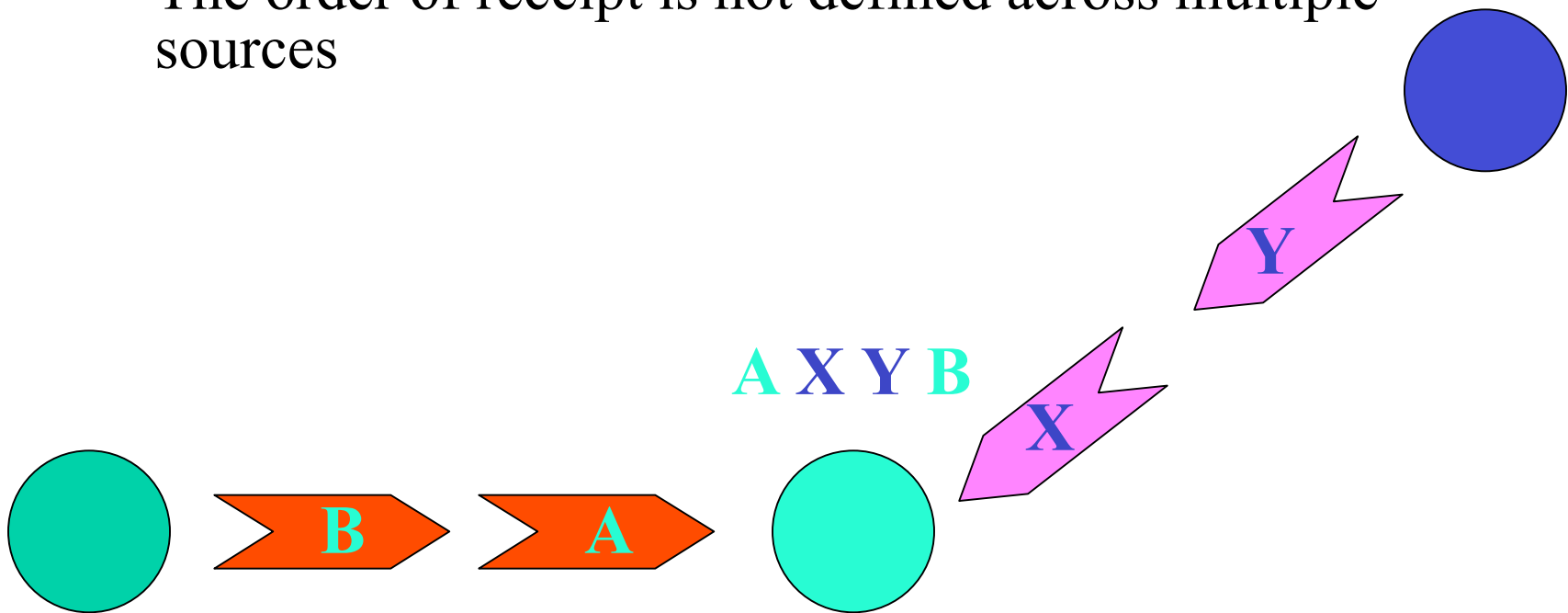
Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent
- If different processes send messages to the same destination, the order of receipt isn't defined across processes



Causality

- If different processes send messages to the same destination
 - The order of receipt is defined from a single source
 - The order of receipt is not defined across multiple sources



Non-blocking communication

- We've seen *blocking* calls that cause the program to wait for completion
- There is asynchronous, *non-blocking* communication
- These are needed to express certain algorithms
- Also used to improve performance

Non-blocking communication

- Non-blocking communication is *split-phased*
 - Phase 1: initiate communication with the immediate 'I' variant of the point- to-point call

IRecv(), ISend()

- Phase 2: synchronize

Wait()

- We can carry out unrelated computations between the two phases

- Building a blocking call

Recv() = IRecv() + Wait()

Fixing a deadlocked program

Process 0	Process 1
Recv (x)	Recv(x)
Send(y,1)	Send(y,0)

Process 0	Process 1
IRecv(x)	IRecv(x)
Send(y,1)	Send(y,0)

Restrictions on non-blocking communication

- The message buffer may not be accessed between an IRecv() (or ISend()) and its accompanying wait()
- Each pending IRecv() must have a distinct buffer

MPI

- The API is delivered as a library called **MPI**
“Message Passing Interface”
 - MPI-1 has 125 routines
 - Callable from C, C++, Fortran, etc.
 - All major vendors support MPI
 - Reference material: see
<http://www-cse.ucsd.edu/users/baden/Doc/mpi.html>
- 7 minimal routines needed by nearly every MPI program
 - start, end, and query MPI execution state (4)
 - non-blocking point-to-point message passing (3)

Functionality we'll will cover today

- Point-to-point communication
- Communicators
- Data types
- Tags
- Non-blocking communication
- Message Filtering

A first MPI program

```
main(int argc, char **argv ) {  
    MPI_Init(&argc, &argv);  
    int rank, size;  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    printf("I am process %d of %d.\n",  
           rank, size);  
    MPI_Finalize();  
}
```

Sending and receiving messages

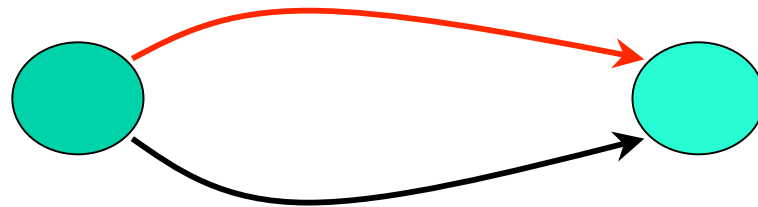
- MPI provides a rich collection of routines to move data between address spaces
- A single pair of communicating processes use *point-to-point* communication
- With *collective communication*, all the processors communicate together
- In point-to-point message passing we can filter messages in various ways
- This allows us to organize message passing activity conveniently

What's in an MPI message?

- To send a message we need
 - A destination
 - A tag
 - A message body (can be empty)
 - A context (called a “communicator” in MPI)
- To receive a message we need similar information, including a receptacle to hold the incoming data

MPI Tags

- One way of screening messages is with tags
- Each sent message is accompanied by a user-defined integer *tag*:
 - Receiving process can use this information to organize or filter messages
 - **MPI_ANY_TAG** inhibits screening.



Communicators

- Another way of screening messages is through a communicator
- A communicator is a name-space (or a context) describing a set of processes that may communicate
- MPI defines a default communicator **MPI_COMM_WORLD** containing all processes
- MPI provides the means of generating uniquely named subsets (advanced MPI)

Send and Recv

```
const int Tag=99;
int msg[2] = { rank, rank * rank};
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2,
             MPI_INT, 1,
             Tag, MPI_COMM_WORLD, &status);
}
else MPI_Send(msg, 2,
              MPI_INT, 0,
              Tag, MPI_COMM_WORLD);
```

Message Buffer

Message length

Send and Recv

```
const int Tag=99;
int msg[2] = { rank, rank * rank};
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2,
             MPI_INT, 1,
             Tag, MPI_COMM_WORLD, &status);
}
else MPI_Send(msg, 2,
              MPI_INT, 0,
              Tag, MPI_COMM_WORLD);
```

The diagram illustrates the mapping of MPI function arguments to specific parameters:

- Data type:** A teal box labeled "Data type" has two teal arrows pointing to the `MPI_INT` argument in both `MPI_Recv` and `MPI_Send` calls.
- SOURCE Process ID:** A red box labeled "SOURCE Process ID" has a red arrow pointing to the `1` argument in the `MPI_Recv` call.
- Destination Process ID:** A red box labeled "Destination Process ID" has a red arrow pointing to the `0` argument in the `MPI_Send` call.

Send and Recv

```
const int Tag=99;
int msg[2] = { rank, rank * rank};
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2,
             MPI_INT, 1,
             Tag, MPI_COMM_WORLD, &status);
}
else MPI_Send(msg, 2,
               MPI_INT, 0,
               Tag, MPI_COMM_WORLD);
```

Message Tag

Communicator

MPI Datatypes

- MPI messages have a specified length
- The unit depends on the type of the data
- The length in bytes is $\text{sizeof}(\text{type}) \times \#$ elements
- We don't we use the # bytes as the length
 - Heterogeneous machines with different storage representations
 - Performance

MPI Datatypes

- Because MPI is a library, we specify the type (and hence length) of an element
- To this end MPI specifies a set of built-in types, corresponding to the primitive types of the language from which MPI is called
- In C: **MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR, MPI_LONG, MPI_UNSIGNED, MPI_BYTE,...**
- Also defined types, e.g. structs,

Message status

- An MPI_Status variable is a struct that contains the sending processor and the message tag
- This information is useful when we haven't filtered the messages
- We may also access the length of the received message (may be shorter than the message buffer)

```
MPI_Recv( message, count,  
          TYPE, MPI_ANY_SOURCE,  
          MPI_ANY_TAG, COMMUNICATOR,  
          &status);
```

```
MPI_Get_count(&status, TYPE, &recv_count );  
status.MPI_SOURCE    status.MPI_TAG
```

Non-blocking communication in MPI

- An extra request argument is required

`MPI_Request request;`

`MPI_Irecv(buf, count, type, source, tag, comm,
&request)`

- We use the request variable to specify which message we are synchronizing in `MPI_Wait()`

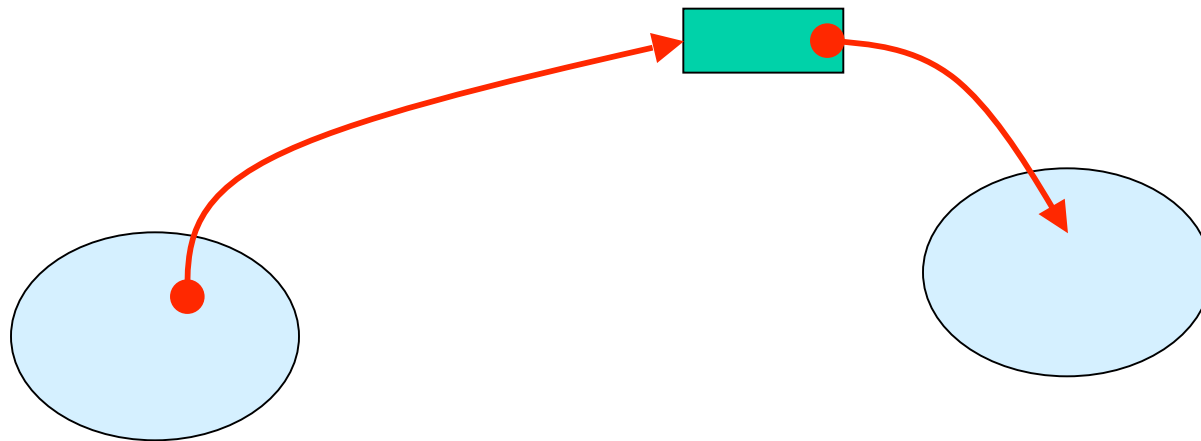
`MPI_Wait(&request, &status)`

- Making above 3 calls in succession is equivalent to

`MPI_Recv(buf, count, type, source,
tag, comm, &status)`

Buffering

- If there is not a pending receive, then an incoming message is placed in an anonymous system buffer
- When the receive gets posted, the message is moved into the user specified buffer
- Double copying reduces communication performance
- Non-blocking communication can help avoid this problem



Rendezvous

- When a long message is to be sent, MPI first checks if the recipient has sufficient storage to receive the message
- If so, then it sends the message. This is called a *rendezvous* implementation. What are the advantages and disadvantages?

Eager limits

- In an *eager* implementation, we just send the message
- In practice, MPI implementations switch between the two modes
- The *eager limit* is the longest message that can be sent in eager mode
- Maximum value on IBM SP systems is 256K
- See M. Banikazemi et al., IEEE TPDS, 2001, “MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems”

Sends that block

- Consider the following example of an “unsafe” program
- It may deadlock if there isn’t enough storage to receive the incoming message (s)

Process 0	Process 1
Send(x,1)	Send(x,0)
Recv(y)	Recv(y)

Avoiding an unsafe program

- The system has pre-allocated storage for the incoming messages so there's no possibility of running out of storage

Process 0	Process 1
IRecv(x) Send(y,1)	IRecv(x) Send(y,0)

Communication performance

- Communication performance is a major factor in determining the overall performance of an application
- Let the message have a length n
- The simplest communication cost model is
transfer time = $\alpha + \beta^{-1} n$
 α = message startup time
 β = peak bandwidth (bytes per second)
 n = message length

Startup and bandwidth

- The startup term dominates when the message is sufficiently short

$$\alpha \gg \beta^{-1} n$$

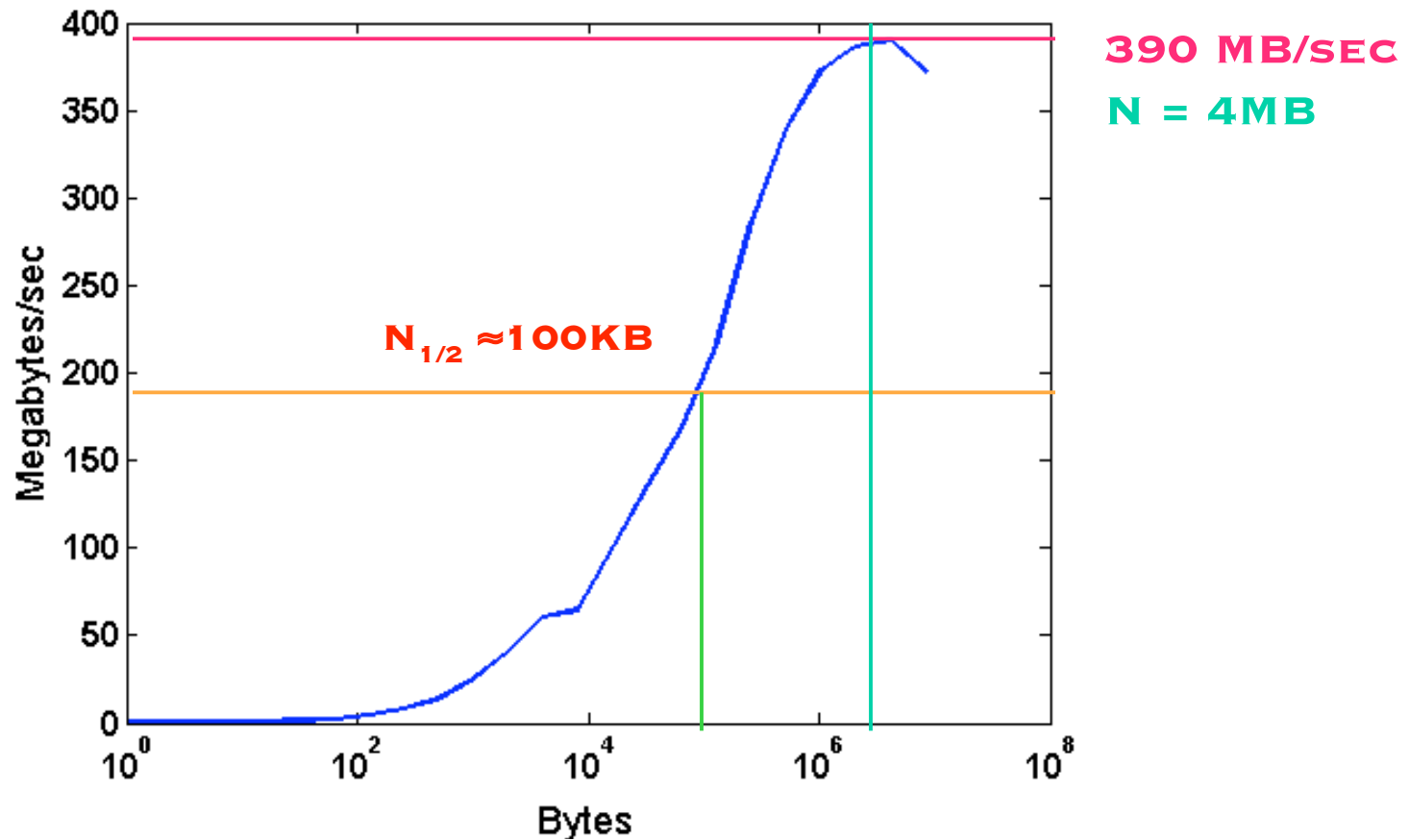
- The bandwidth term dominates when the message is sufficiently long

$$\beta^{-1} n \gg \alpha$$

Half power point

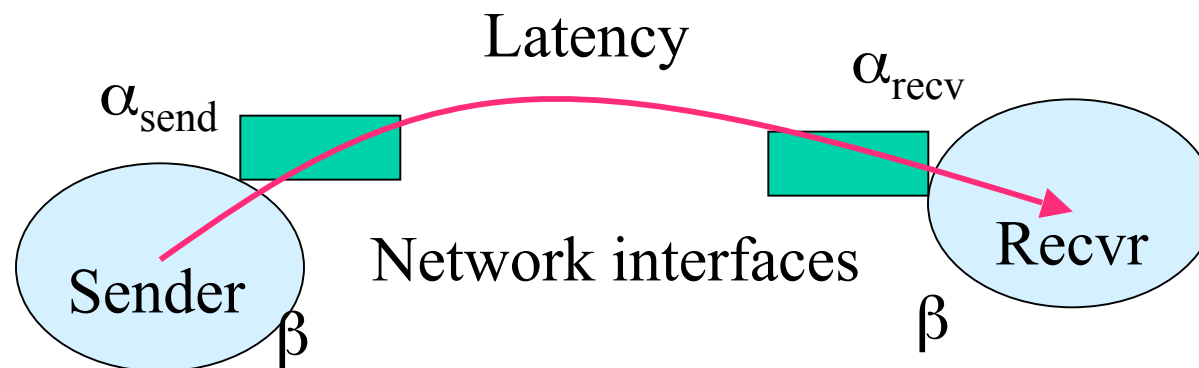
- Let $T(n)$ = time to send a message of length n
- Let $\beta(n)$ = the effective bandwidth
$$\beta^{-1}(n) = n / T(n)$$
- We define the ***half power point*** $n_{1/2}$ as the message size required to achieve $\frac{1}{2} \beta_{\infty}$
$$\frac{1}{2} \beta^{-1}_{\infty} = n_{1/2} / T(n_{1/2}) \Rightarrow \beta^{-1}(n_{1/2}) = \frac{1}{2} \beta^{-1}_{\infty}$$
- In theory, this occurs when $\alpha = \beta^{-1}_{\infty} n_{1/2} \Rightarrow n_{1/2} = \alpha / \beta^{-1}_{\infty}$
- Doesn't generally predict actual value of $n_{1/2}$
- For SDSC's DataStar machine
 - $\alpha \approx 7.6 \mu\text{s}$, $\beta_{\infty} \approx 1580 \text{ Mbytes/sec} \Rightarrow n_{1/2} \approx 12\text{KB}$
 - The actual value of $n_{1/2} \approx 38\text{KB}$
 - In Assignment #2, you'll explore this phenomenon

Typical bandwidth curve (SDSC Blue Horizon)



Where does the time go?

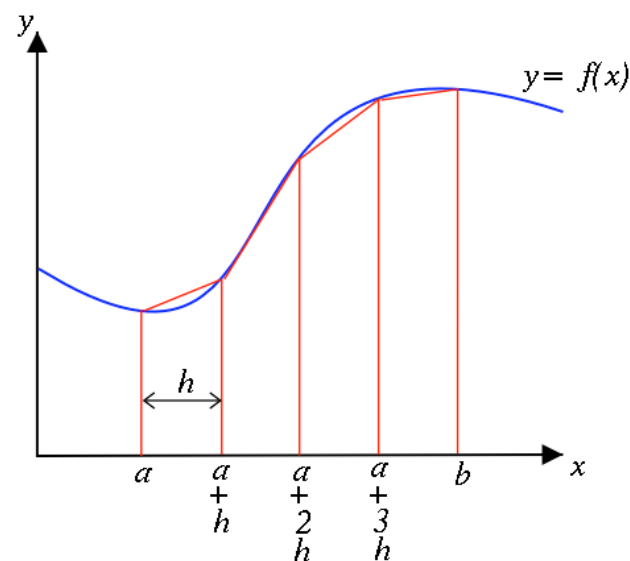
- Under ideal conditions...
 - There is a pending receive waiting for an incoming message, which is transmitted directly to and from the users message buffer
 - There is no other communication traffic
- Assume a contiguous message



The trapezoidal rule

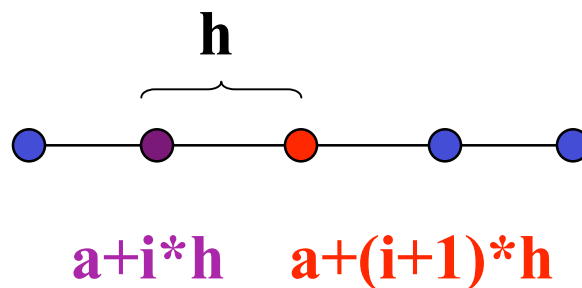
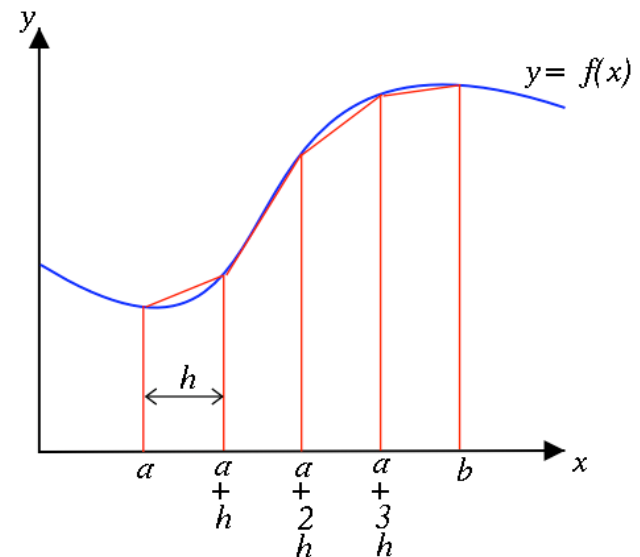
- Use the trapezoidal rule to numerically approximate the definite integral

$$\int_a^b f(x) dx$$



How the trapezoidal rule works

- Divide the interval $[a,b]$ into n segments of size $h=1/n$
- Approximate the area under an interval using a trapezoid
- Area under the i^{th} trapezoid
 $\frac{1}{2} (f(a+i \times h) + f(a+(i+1) \times h)) \times h$
- Area under the entire curve
 $\approx$ sum of all the trapezoids



Reference material

- For a discussion of the trapezoidal rule

<http://metric.ma.ic.ac.uk/integration/techniques/definite/numerical-methods/trapezoidal-rule>

- A applet to carry out integration

<http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Numerical/Integration>

- Code (from Pacheco hard copy text)

Serial Code

[\\$PUB/Pacheco/ppmpi_c/chap04/serial.c](#)

Parallel Code

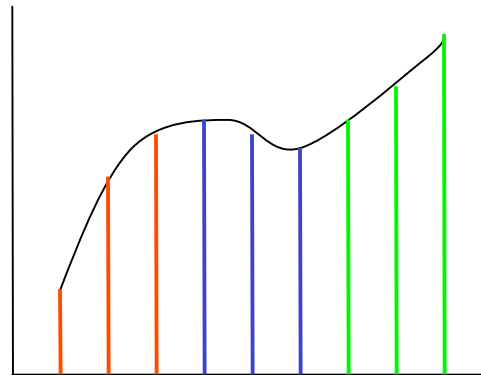
[\\$PUB/Pacheco/ppmpi_c/chap04/trap.c](#)

Serial code (Following Pacheco)

```
main() {  
    float f(float x) { return x*x; }           // Function we're integrating  
  
    float h = (b-a)/n;                          // h = trapezoid base width  
                                                // a and b: endpoints  
                                                // n = # of trapezoids  
  
    float integral = (f(a) + f(b))/2.0;  
  
    float x; int i;  
  
    for (i = 1, x=a; i <= n-1; i++) {  
        x += h;  
        integral = integral + f(x);  
    }  
    integral = integral*h;  
}
```

The parallel algorithm

- Partition the integration interval into sub-intervals, one per processor
- Each processor computes the integral on its local subdomain
- All combine their local integrals into a global one



First version of the parallel code

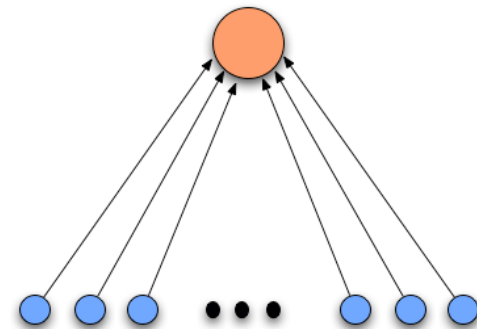
```
local_n = n/p;           // Number of trapezoids; assume p divides n evenly
float local_a = a + my_rank*local_n*h,
      local_b = local_a + local_n*h,
      integral = Trap(local_a, local_b, local_n, h);

if (my_rank == ROOT) {  // Sum the integrals calculated by all the processes
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag, WORLD, &status);
        total += integral;
    }
} else
    MPI_Send(&integral, 1, MPI_FLOAT, ROOT, tag, WORLD);
```

Improvements

- The result does not depend on the order in which the sums are taken, except to within roundoff
- We use a linear time algorithm to accumulate contributions, but there are other orderings

```
for (source = 1; source < p; source++) {  
    MPI_Recv(&integral, 1, MPI_FLOAT,  
             MPI_ANY_SOURCE, tag,  
             WORLD, &status);  
  
    total += integral;  
}
```



Improved parallel code

- We can often improve performance by taking advantage of global knowledge about communication
- Instead of using point to point communication operations to accumulate the sum, use *collective* communication **nt**

```
local_n = n/p;  
float local_a = a + my_rank*local_n*h,  
      local_b = local_a + local_n*h,  
      integral = Trap(local_a, local_b, local_n, h);  
MPI_Reduce( &integral, &total, 1,  
            MPI_FLOAT, MPI_SUM,  
            ROOT, WORLD)
```

Collective communication in MPI

- Collective operations are called by **all** processes within a communicator
- Broadcast: distribute data from a designated “root” process to all the others
`MPI_Bcast(in, count, type, root, comm)`
- Reduce: combine data from all processes and return to a designated root process
`MPI_Reduce(in, out, count, type, op, root, comm)`

Broadcast

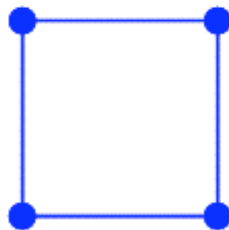
- The root process transmits of m pieces of data to all the $p-1$ other processors
- With the linear ring algorithm this processor performs $p-1$ sends of length m
 - Cost is $(p-1)(\alpha + \beta m)$
- Another approach is to use the *hypercube algorithm*, which has a logarithmic running time

What is a hypercube?

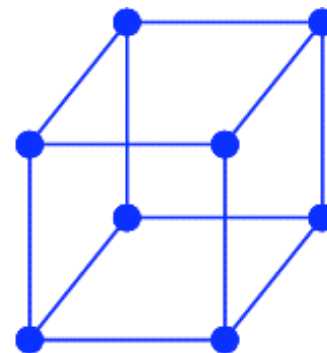
- A hypercube is a d -dimensional graph with 2^d nodes
- A 0-cube is a single node, 1-cube is a line connecting two points, 2-cube is a square, etc
- Each node has d neighbors



1D



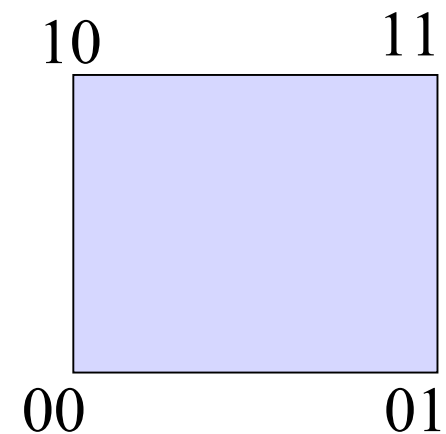
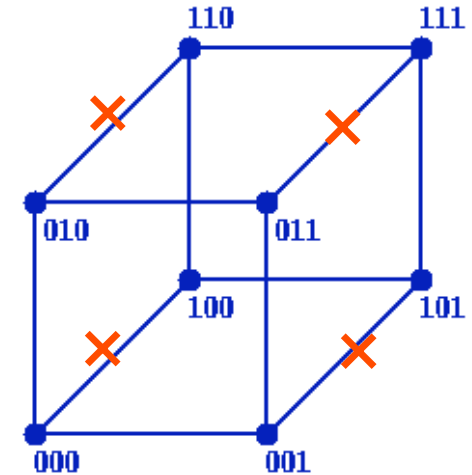
2D



3D

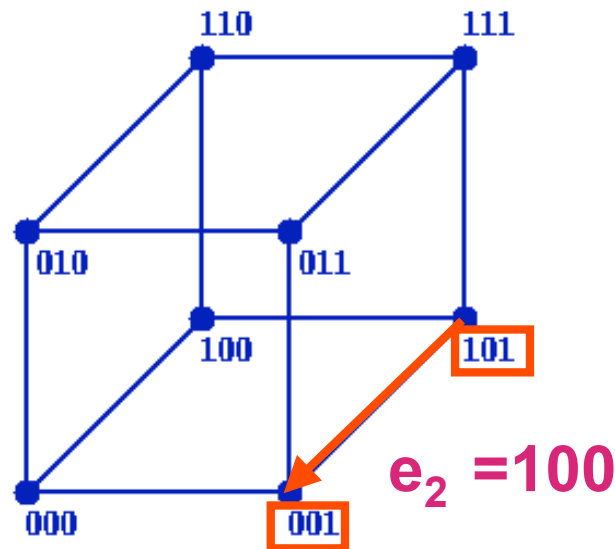
Properties of hypercubes

- A hypercube with p nodes has $\lg(p)$ dimensions
- *Inductive construction*: we may construct a d -cube from two $(d-1)$ dimensional cubes
- **Diameter**: What is the maximum distance between any 2 nodes?
- **Bisection bandwidth**: How many cut edges (mincut)



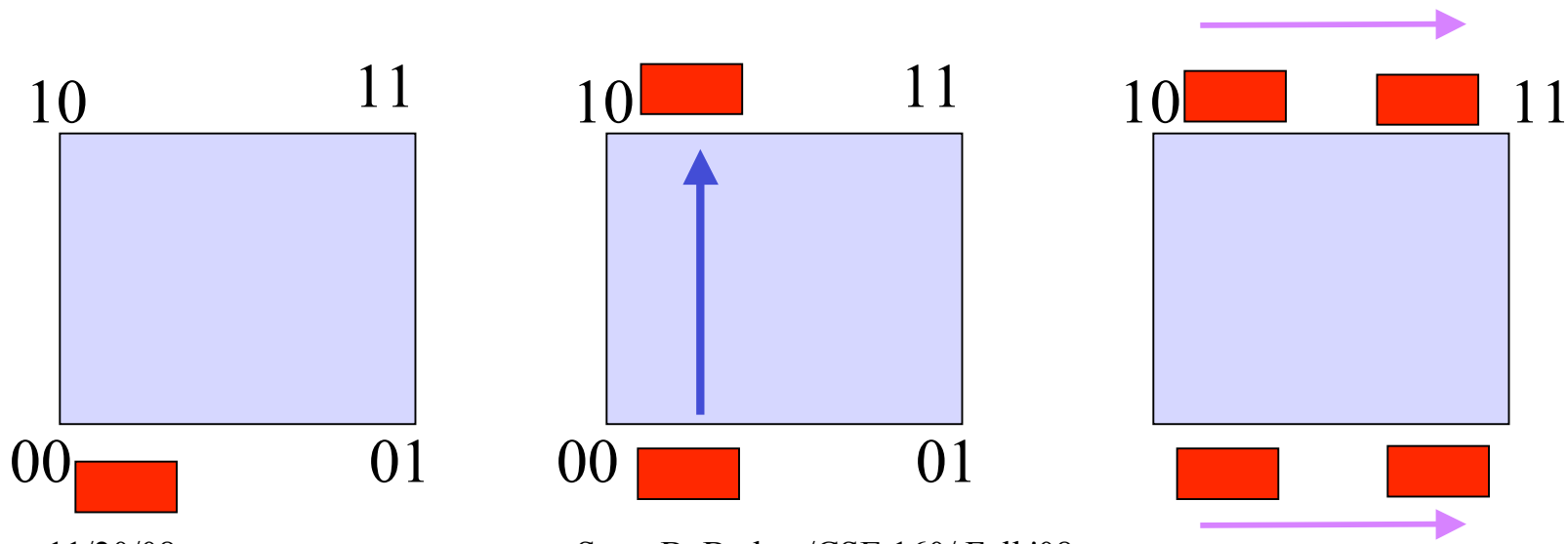
Bookkeeping

- Label nodes with a binary reflected grey code
<http://www.nist.gov/dads/HTML/graycode.html>
- Neighboring labels differ in exactly one bit position $001 = 101 \otimes e_2$, $e_2 = 100$



Hypercube broadcast algorithm with $p=4$

- Processor 0 is the root, sends its data to its hypercube “buddy” on processor 2 (10)
- Proc 0 & 2 send data to respective buddies



Reduction

- We may use the hypercube algorithm to perform reductions as well as broadcasts
- Another variant of reduction provides all processes with a copy of the reduced result

Allreduce()

- Equivalent to a *Reduce* + *Bcast*
- A clever algorithm performs an *Allreduce* in one phase rather than having perform separate reduce and broadcast phases

Final version

```
int local_n = n/p;

float local_a = a + my_rank*local_n*h,
      local_b = local_a + local_n*h,
      integral = Trap(local_a, local_b, local_n, h);

MPI_Allreduce( &integral, &total, 1,
               MPI_FLOAT, MPI_SUM, WORLD)
```