Lecture 8

Writing parallel programs with MPI
Measuring performance

Announcements

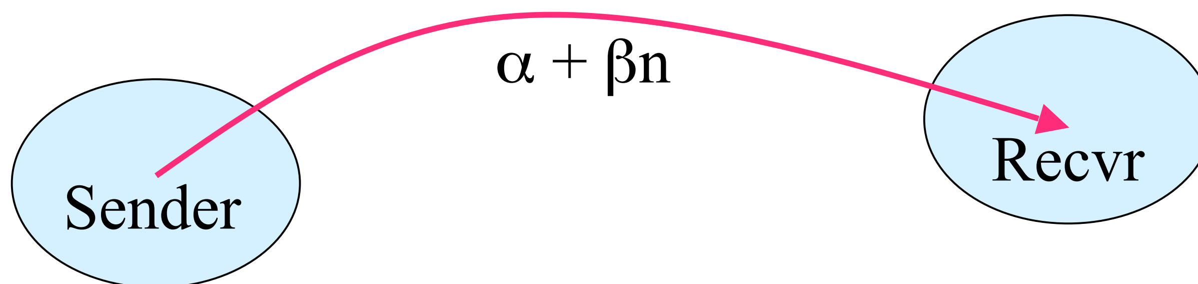
- Project proposals due on Tuesday
- Next Monday's office hour, 10/20, moved to 5pm to 6pm (note change)
- Will resume usual schedule the following Monday (3pm to 4pm)

Ring microbenchmark to determine communication cost model parameters

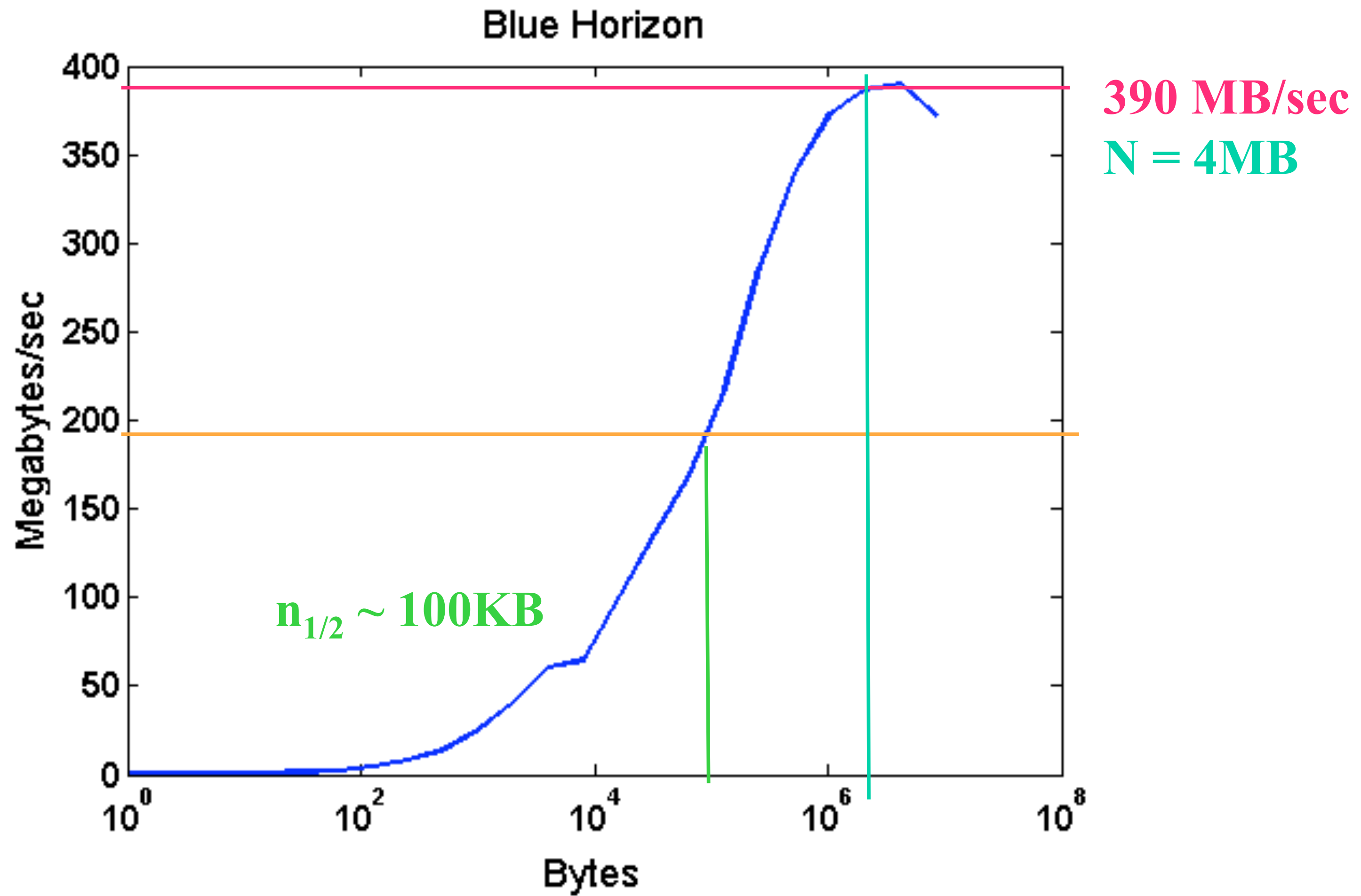
- Recall the communication cost model: transfer time = $\alpha + \beta n$

  α = message startup time

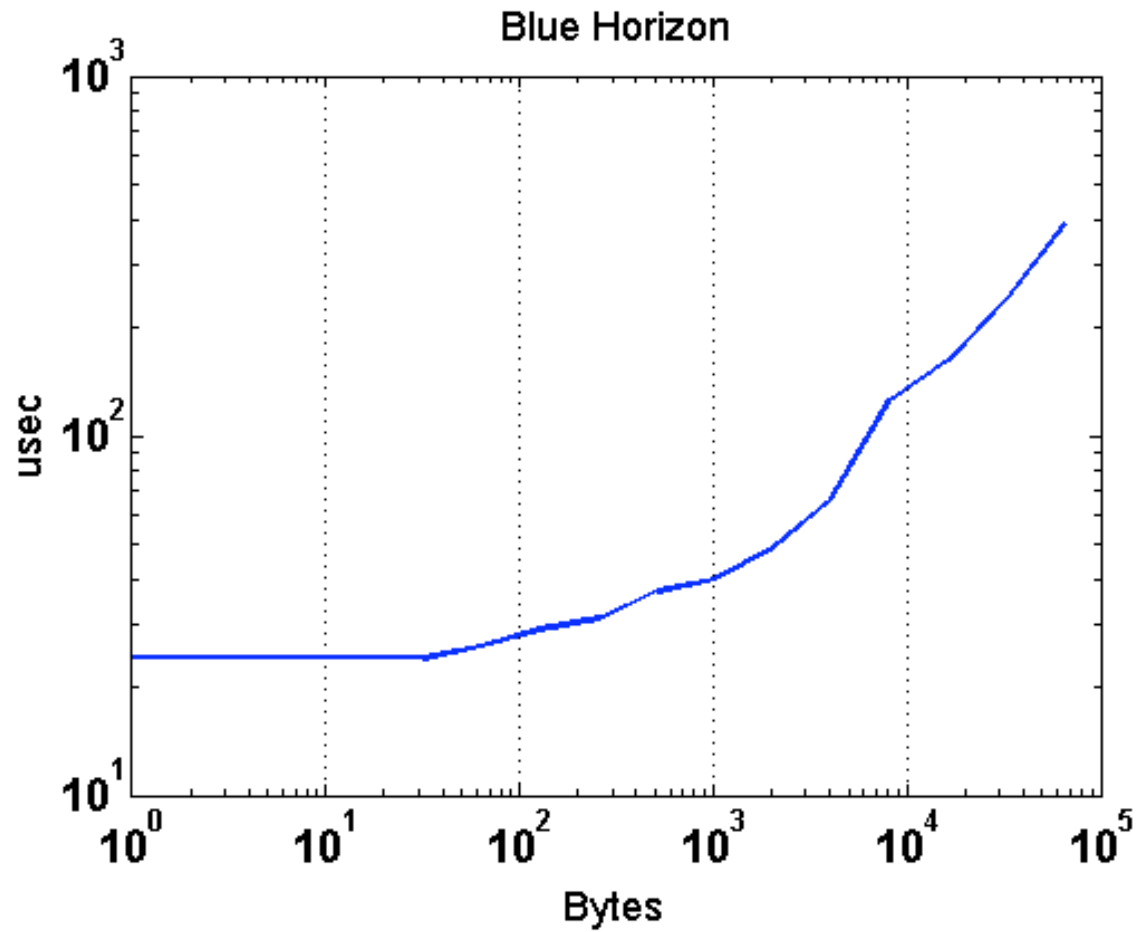
  β = inverse peak bandwidth



Communication Bandwidth (off node) on Blue Horizon

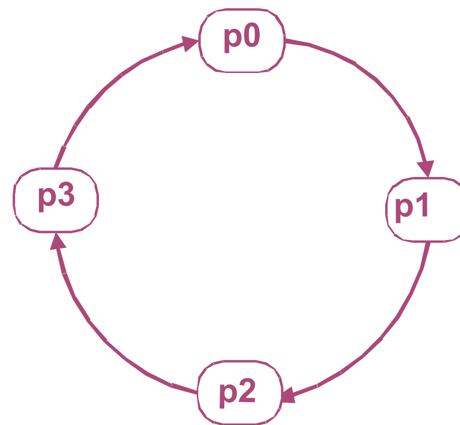


Communication times



The Ring program

- Configure the processors logically in a ring and pass messages around the ring multiple times
- Assume there are p processors
- Neighbors of processor k are
  - (k + 1) **mod** p
  - (k + p -1) **mod** p



Measurement technique with Ring

```
for (int len = 1, l=0; len <= maxSize; len *= 2, l++)
if (myid == 0) {
// (WARM UP CODE)
    const double start = MPI_Wtime( );
    for (int i = 0; i < trips; i++) {
        PROCESSOR 0 CODE
    }
const double delta = MPI_Wtime( ) - start;
Bandwidth = (long)((trips*len*nodes)/ delta /1000.0);
} else {   // myid != 0
    // (WARM UP CODE)
     for (int i = 0; i < trips; i++) {
         ALL OTHER PROCESSORS
    }
}
```

The Ring program

Processor 0:

```
MPI_Request req;
MPI_Irecv(buffer, len, MPI_CHAR, (rank + p - 1)%p,
                   tag, MPI_COMM_WORLD, &req);
MPI_Send(buffer, len, MPI_CHAR, (rank + 1) % p,
                  tag, MPI_COMM_WORLD);
MPI_Status status;
MPI_Wait(&req,&status);
```

All others:

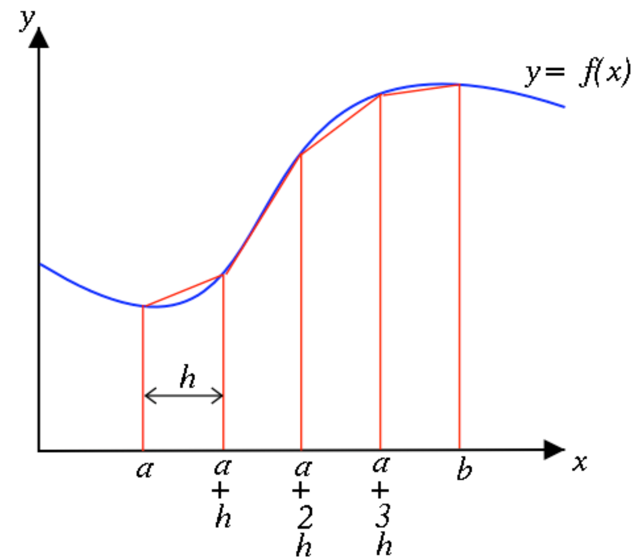
```
MPI_Status status1;
MPI_Recv(buffer, len, MPI_CHAR, (rank + p - 1)%p,
                  tag, MPI_COMM_WORLD, &status1);
MPI_Send(buffer, len, MPI_CHAR, (rank+1)%p,
                  tag, MPI_COMM_WORLD);
```

A first MPI Application

The trapezoidal rule

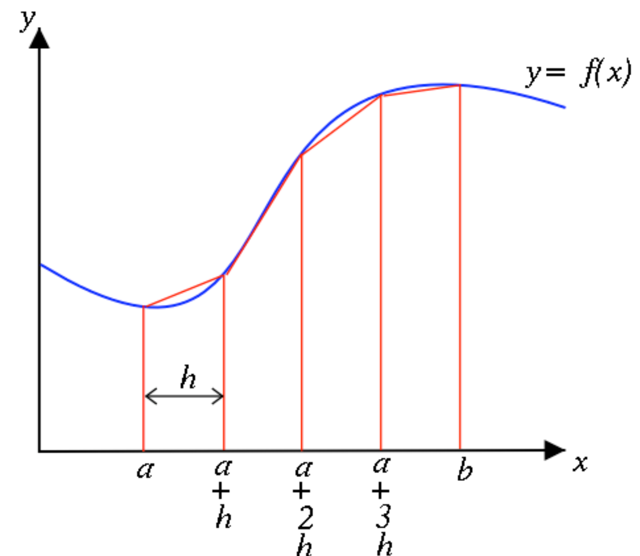
- Use the trapezoidal rule to numerically approximate the definite integral

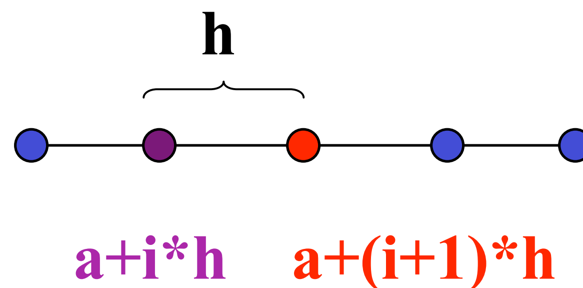
$$\int_a^b f(x)dx$$



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 ith trapezoid
  $\frac{1}{2}(f(a+i\times h)+f(a+(i+1)\times h))\times h$
- Area under the entire curve
  ≈ 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)

  $PUB = /home/cs260f/public

  Serial Code

  $PUB/Pacheco/ppmpi_c/chap04/serial.c

  Parallel Code

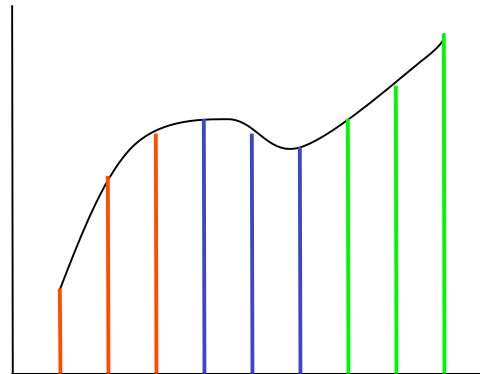
  $PUB/Pacheco/ppmpi_c/chap04/trap.c

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

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



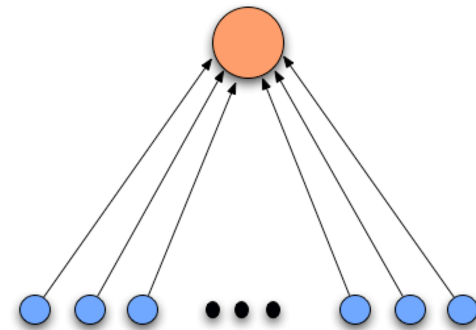
First version of the parallel code

```
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 deisgated root process

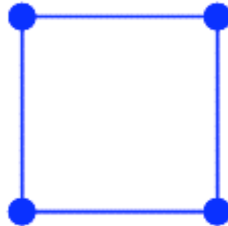
  MPI_Reduce(in, out, count, type, op, root, comm)

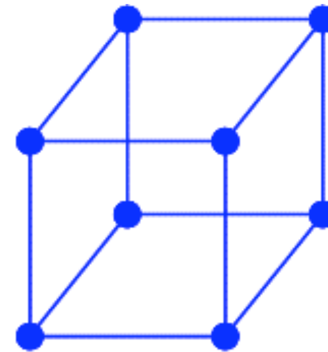
Broadcast

- 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

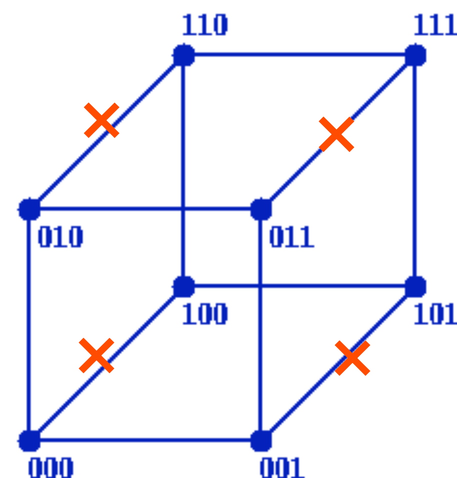


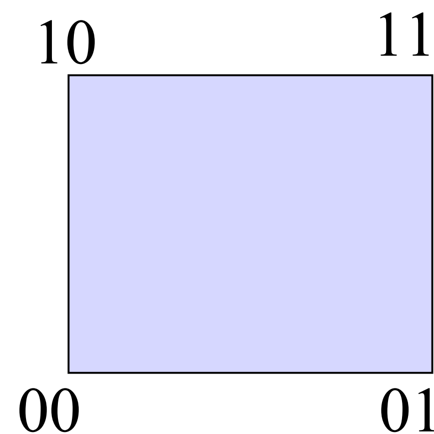


1D 2D 3D

Properties of hypercubes

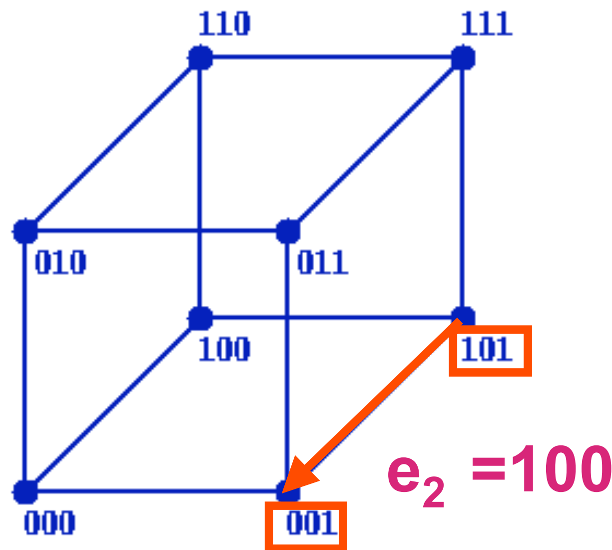
- A hypercube with p nodes has lg(p) dimensions
- *Inductive construction*: we may construct a d-cube from two (d-1) dimensional cubes
- **Diameter:** What is the maximum distance between any 2 nodes?
- **Bisection bandwidth:** How many cut edges (mincut)





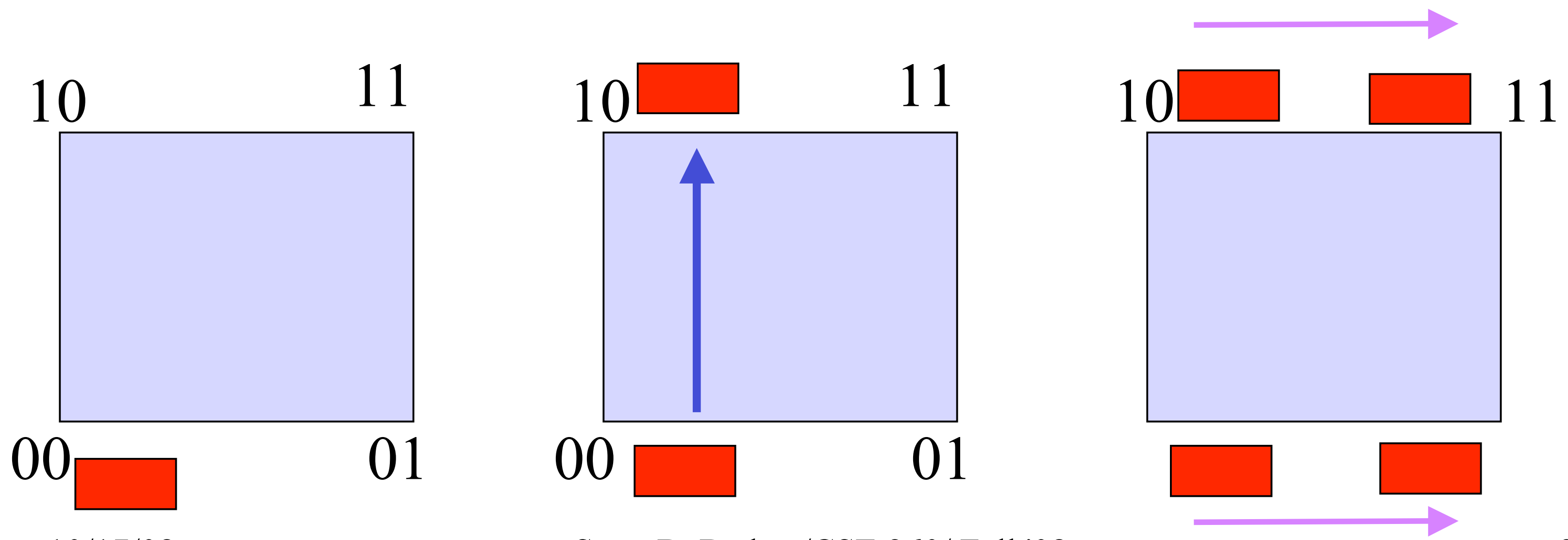
Bookkeeping

- 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 ⊗ 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)
```