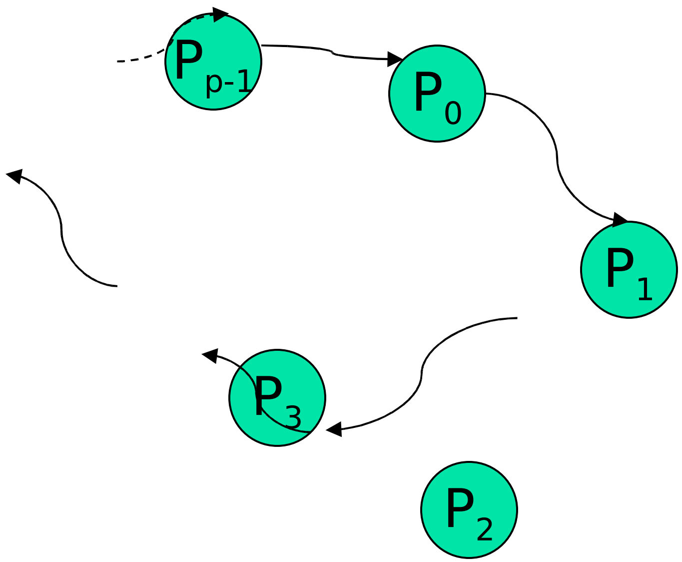
Principles of High Performance Computing (ICS 632)

Communication in a
Ring Topology

Ring Topology (Section 3.3)



- Each processor is identified by a rank
  - MY_NUM()
- There is a way to find the total number of processors
  - NUM_PROCS()
- Each processor can send a message to its successor
  - SEND(*addr*, L)
- And receive a message from its predecessor
  - RECV(*addr*, L)

- We'll just use the above pseudo-code rather than MPI
- Note that this is much simpler than the example tree topology we saw in the previous set of slides

Virtual vs. Physical Topology

- Now that we have chosen to consider a Ring topology we "pretend" our physical topology is a ring topology
- We can always implement a virtual ring topology (see previous set of slides)
  - And read Section 4.6
- So we can write many "ring algorithms"
- It may be that a better virtual topology is better suited to our physical topology
- But the ring topology makes for very simple programs and is known to be reasonably good in practice
- So it's a good candidate for our first look at parallel algorithms

Cost of communication (Sect. 3.2.1)

- It is actually difficult to precisely model the cost of communication
  - E.g., MPI implementations do various optimizations given the message sizes
- We will be using a simple model

  Time = L + m/B

  L: start-up cost or *latency*

  B: bandwidth (b = 1/B)

  m: message size

- We assume that if a message of length m is sent from P_0 to P_q, then the communication cost is q(L + m b)
- There are many assumptions in our model, some not very realistic, but we'll discuss them later

Assumptions about Communications

- Several Options
  - Both Send() and Recv() are blocking
    - Called “rendez-vous”
    - Very old-fashioned systems
  - Recv() is blocking, but Send() is not
    - Pretty standard
    - MPI supports it
  - Both Recv() and Send() are non-blocking
    - Pretty standard as well
    - MPI supports it

Assumptions about Concurrency

- One question that's important is: can the processor do multiple things at the same time?
- Typically we will assume that the processor can send, receive, **and** compute at the same time
  - Call MPI_IRecv() Call MPI_ISend()
  - Compute something
- This of course implies that the three operations are independent
  - E.g., you don't want to send the result of the computation
  - E.g., you don't want to send what you're receiving (forwarding)
- When writing parallel algorithms (in pseudo-code), we'll simply indicate concurrent activities with a || sign

Collective Communications

- To write a parallel algorithm, we will need collective operations
  - Broadcasts, etc.
- Now MPI provide those, and they likely:
  - Do not use the ring logical topology
  - Utilize the physical resources well
- Let's still go through the exercise of writing some collective communication algorithms
- We will see that for some algorithms we really want to do these communications "by hand" on our virtual topology rather than using the MPI collective

Broadcast (Section 3.3.1)

- We want to write a program that has P_k send the same message of length m to all other processors

  Broadcast(k,*addr*,m)
- On the ring, we just send to the next processor, and so on, with no parallel communications whatsoever
- This is of course not the way one should implement a broadcast in practice if the physical topology is not merely a ring
  - MPI uses some type of tree topology

Broadcast (Section 3.3.1)

```
Brodcast(k,addr,m)
   q = MY_NUM()
   p = NUM_PROCS()
   if (q == k)
      SEND(addr,m)
   else
      if (q == k-1 mod p)
         RECV(addr,m)
      else
         RECV(addr,m)
         SEND(addr,m)
      endif
   endif
```

- Assumes a blocking receive
- Sending may be non-blocking
- The broadcast time is

  (p-1)(L+m b)

Scatter (Section 3.2.2)

- Processor k sends a different message to all other processors (and to itself)
  - P_k stores the message destined to P_q at address addr[q], including a message at addr[k]
- At the end of the execution, each processor holds the message it had received in msg
- The principle is just to pipeline communication by starting to send the message destined to P_{k-1}, the most distant processor

Scatter (Section 3.3.2)

Same execution time as the broadcast

(p-1)(L + m b)

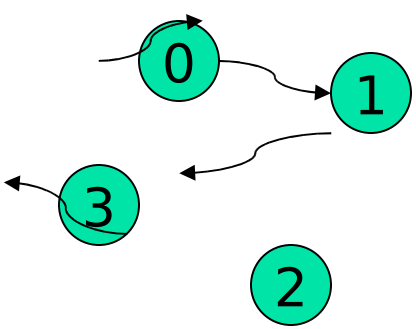
```
Scatter(k,msg,addr,m)
  q = MY_NUM()
  p = NUM_PROCS()
  if (q == k)
   for i = 0 to p-2
      SEND(addr[k+p-1-i mod p],m)
   msg ← addr[k]
  else
   RECV(tempR,L)
   for i = 1 to k-1-q mod p
      tempS ↔ tempR
      SEND(tempS,m) || RECV(tempR,m)
   msg ← tempR
```

Swapping of send buffer
and receive buffer (pointer)

Sending and
Receiving
in Parallel, with a
non blocking Send

Scatter (Section 3.3.2)

```
Scatter(k,msg,addr,m)
  q = MY_NUM()
  p = NUM_PROCS()
  if (q == k)
     for i = 0 to p-2
        SEND(addr[k+p-1-i mod p],m)
     msg ← addr[k]
  else
     RECV(tempR,L)
     for i = 1 to k-1-q mod p
        tempS ↔ tempR
        SEND(tempS,m)  || RECV(tempR,m)
     msg ← tempR
```



k = 2, p = 4

Proc q=2

send addr[2+4-1-0 % 4 = 1]
send addr[2+4-1-1 % 4 = 0]
send addr[2+4-1-2 % 4 = 3]
msg = addr[2]

Proc q=3

recv (addr[1])
// loop 2-1-3 % 4 = 2 times
send (addr[1]) || recv (addr[0])
send (addr[0]) || recv (addr[3])

msg = addr[3]

Proc q=0

recv (addr[1])
// loop 2-1-2 % 4 = 1 time
send (addr[1]) || recv (addr[0])

msg = addr[0]

Proc q=1

// loop 2-1-1 % 4 = 0 time
recv (addr[1])

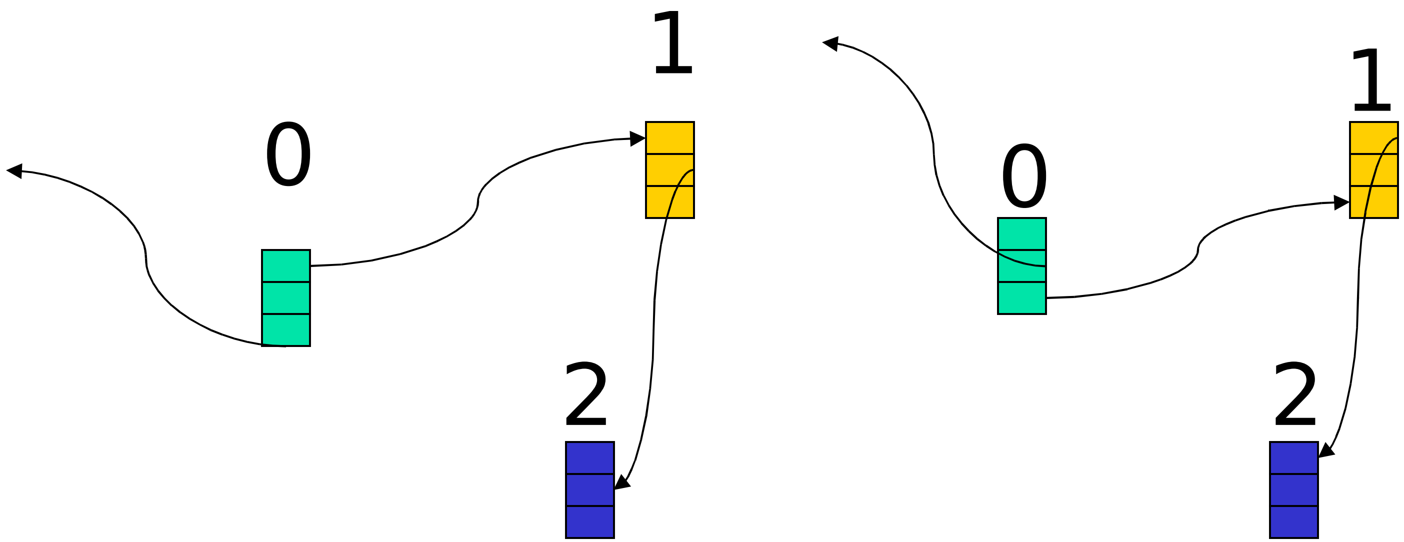
msg = addr[1]

All-to-all (Section 3.3.3)

```
All2All(my_addr, addr, m)
   q = MY_NUM()
   p = NUM_PROCS()
   addr[q] ← my_addr
   for i = 1 to p-1
         SEND(addr[q-i+1 mod p],m)
    || RECV(addr[q-i mod p],m)
```

Same execution time as the scatter

(p-1)(L + m b)



A faster broadcast?

- How can we improve performance?
- One can cut the message in many small pieces, say in r pieces where m is divisible by r.
- The root processor just sends r messages
- The performance is as follows
  - Consider the last processor to get the last piece of the message
  - There need to be p-1 steps for the first piece to arrive, which takes $(p-1)(L + m b / r)$
  - Then the remaining r-1 pieces arrive one after another, which takes $(r-1)(L + m b / r)$
  - For a total of: $(p - 2 + r) (L + mb / r)$

A faster broadcast?

- The question is, what is the value of r that minimizes

$$(p - 2 + r)\,(L + m\,b / r)\ ?$$

- One can view the above expression as (c+ar)(d+b/r), with four constants a, b, c, d
- The non-constant part of the expression is then ad.r + cb/r, which must be minimized
- It is known that this value is minimized for

$$\mathrm{sqrt}(cb / ad)$$

  and we have

$$r_{opt} = \mathrm{sqrt}(m(p-2)\,b / L)$$

  with the optimal time

$$(\mathrm{sqrt}((p-2)\,L) + \mathrm{sqrt}(m\,b))^2$$

  which tends to mb when m is large, which is independent of p!

Well-known Network Principle

- We have seen that if we cut a (large) message in many (small) messages, then we can send the message over multiple hops (in our case p-1) almost as fast as we can send it over a single hop
- This is a fundamental principle of IP networks
  - We cut messages into IP frames
  - Send them over many routers
  - But really go as fast as the slowest router