Lesson 2-1 Basic Model

Simulating the Brain
Quiz demonstrating the amount of computing power needed for some problems. This means we need to be able to efficiently use distributed memory.

A Basic Model of Distributed Memory
To design algorithms for a super computer - a machine model is necessary.

Private Memory Model - a system as individual nodes with their own memory that other nodes cannot directly access.

To get information from the private memory -- message passing must be used. The message will have to find a path from the source to the destination.

Rules of the Model:
0 Rule -- don't talk about the model. You must internalize these rules.

1 Rule -- the network is fully connected, there is always a path from any node to any other node in the network.

2 Rule -- the links are bidirectional. The links can carry a message between nodes in both directions at the same time.

3 Rule -- the nodes can send and receive one message at a time. This means if a node wants to send messages to other nodes - it must do it one at a time.

4 Rule -- the cost to send and receive 'n' words. To send a message from one node to another could go by different paths. At this time we will say the cost of communication is not affected by the path length. (Later in lesson this will be discussed).

This rule only applies when there are no messages competing for links.

This equation does not account for the path length.

$\alpha = latency, units\ of\ time, it\ is\ a\ fixed\ cost\ no\ matter\ how\ large\ the\ message.$
$\beta = inverse\ bandwidth, units\ of\ time/word.$
n = number of words

$$Tmsg(n) = \alpha + \beta n$$

5 Rule -- when messages are trying to go over the same link at the same time, this is called congestion.
K-way congestion reduces bandwidth

k= the number of messages competing for the same link
$Tmsg(n) = \alpha + \beta n$ k

This means that two messages (going in the same direction) that are competing for the same link will have a time that is closer to the serial time, than the parallel time for the two messages.

MEMORIZE THESE RULES!

Pipelined Message Delivery
Assume: there is a message that must traverse P nodes, it is one word long. The message is at P = 0 and needs to go to P-1.
So:

$$n = 1 : \quad a + t(P-1)$$

Now, lets send a message two words long. The words are pipelined through the path
So:

$$n = 2 : \quad a + t(P-1) + t$$

(the added t is for the second word exiting the pipeline)

So for a message of n words:

$$n = a + t(P-1) + t(n-1)$$

Simplify to:
Time for message of size n:

$$n : a + t(P-2) + tn$$

consider :

$t(P-2)$ to be the wire delay of the message traveling across the network.

tn is the time related to the message size.

a is the software overhead for preparing messages for delivery.

a is much larger than $t(P-2)$ → so they can be treated as one constant.

So … $a + t(P-2)$ is proportional to α and β is proportional to 1/t.

Getting the Feel for the Alpha-Beta Model

τ = compute [time/operation]
$\alpha \ll \beta \ll \alpha$

Computation requires much less time than communication, so try to have as little communication as possible.

Also, you can send ~1000 bytes in the same amount of time that it takes to prepare a message. It is better to send fewer larger messages than frequent short ones.

Applying the Rules
If two messages sent at the same time do NOT overlap - then the transmission time is *not* affected. But if the two messages overlap (travel in the same direction) at any link - then this overlap must be serialized and the transmission time is longer.

Collective Operations

All-to-One Reduce : all nodes participate to produce a final result on one node.

1. Have all odd numbered nodes send their data to the lower even number.
   Only one word is sent in each message - so the communication time is $\alpha + \beta$

2. After the first step there will only be even ranked nodes left. So now send the odd partner to it's even partner (the odd numbered node wil send to the lower even numbered node).
   This communication time is also $\alpha + \beta$.

3. Repeat until only one result is left. The communication time is:
   (# of steps) * ($\alpha + \beta$)

Point to Point

1. Assume a SPMD style.
2. Assume the pseudo code algorithm has access to the variables rank (rank is the id of the process and it is unique) and 'p' (p = number of processes).

   In practice there maybe more than one process assigned to a node.
3. A primitive called

handle ← sendAsync(buf[1:n] → dest).

It does an asynchronous send using a buffer of size ‘n’ and a destination for the message.

A return from sendAsync means … a message has been sent, not that it has been received. To find out what has happened, test the handle that is returned.

4. Asynchronous receive. This is posted by the destination.

   handle ← recvAsync(buf[1:n] ← source)

   A return from recvAsync means … the message was received. To find out if the data is available, the handle must be tested.

5. Blocking. This primitive will pause until the corresponding operations are complete. The primitive is:

   wait(handle1, handle2, …) or wait(*)

Point to Point Completion Semantics

wait(handle1, …) it is safe to use the buffer when the primitive returns.

asyncRecv - a return means the message was delivered.
asyncSend - the return does not tell you very much. This is done to allow the programmer the ability to decide what to do about a send message: wait for ‘message received’ or make a copy and continue to work.

When writing algorithms …… *Remember - every send must have a matching receive*.

Send and Receive in Action

sendAsync and recvAsync can get trapped in a deadlock depending upon how the wait is implemented.

All-to-One Reduce Pseudocode

```
let s = local value
bitmask ← 1
while bitmask < P do
        PARTNER ← RANK ^ bitmask
        if RANK & bitmask then
                sendAsync (s → PARTNER)
                wait(*)
                break //one sent, the process can drop out
        else if (PARTNER < P)
```

```
            recvAsync(t ← PARTNER)
            wait(*)
            S ← S + t
        bitmask ← (bitmask << 1)
if RANK = 0 then print(s)
```

Note: Only senders drop out
Senders have a 1 at the bitmask position

RANK(Receiver) < RANK(Sender) < P

Vector Reductions

In the vector version:
sendAsync(S[1:n] ⇒ PARTNER) or sendAsync(S[:])

Vector Reductions Quiz
Time to do a vector reduction = $(\alpha + \beta * n) log\ P$

More Collectives
All-to-One Reduce ← → One-to-All Broadcast These are duals of one another

One-to-All Broadcast - one node has information and all other nodes get a copy of it.

Scatter → one process starts with the data, then sends a piece of it to all other processes.
The dual of a scatter is gather.

All-gather → each process has a piece of the data. The data is gathered and every process gets a copy of all the data.
The dual of all-gather is reduce-scatter.

Reduce-scatter : the processes contain data that is reduced to one vector. This vector has pieces that are then scattered to all other processes.

A Pseudo Code API for Collectives

All-to-One Reduce: each process needs to call ….. reduce(A_{local}[1:n], root).
A collective is an operation that must be executed by all processes.

The following is invalid because it does not call all processes, and results in a deadlock.

```
if RANK = 0 then
        reduce(X[:], 0)
```

else

://twiddle thumbs

broadcast(A_{local}[1:n], root) → when the broadcast is complete every process will have the same data in it's buffer.

gather(In[1:m], Out[1:m][1:P], root) → every process has a little bit of data. All the data is to be collected onto one process.

NOTE: 'm' is used instead of 'n'. The reason for this is n= mP → where mP is the size of the combined output.

scatter(In[1:m][1:P], root, Out[1:m]) ---> the input is size m x P.

allGather(In[1:m], Out[1:m][1:P]) → There is no root rank

reduceScatter(In[1:m][1:P], Out[1:m])

reshape(A[1:m][1:n]) → A^[1:m:n] goes from 2D to 1D
reshape(A[1:m:n]) → A^[1:m][1:n] goes from 1D to 2D

It goes from a logical 2D representation to a 1D representation.

AllGather from Building Blocks
AllGather Pseudo Code:

gather(IN, Out, root)
broadcast(reshape(Out), root)

Collectives Lower Bounds

What should the cost goals be?

reduce has a cost of $T(n) = (\alpha + \beta n) log\, P$ … Is this good or bad?

The tree based algorithm may be sending too much data - by a factor of P.

AllGather Quiz
If two primitives attain the lower bounds, will the combination of the two also achieve the lower bounds? The combination of optimal primitives will produce optimal results.

Implement Scatter Quiz
Recall a process can only do one simultaneous send and receive. If the root is doing all of the sends...then they execute sequentially.

Implementing Scatter and Gather
Goal: find an $\alpha logP$ algorithm.

A different way to implement scatter: Instead of one process doing all the scattering, divide and conquer the problem.

Divide the network into two and send half the data to other half of the network. Now each of these can do the same thing, divide their part of the network and data.

What is the cost of the new method?
The new method has retained the lower bound with respect to latency and bandwidth.

$T(n) = \alpha logP + \beta n\,((P-1)/P)$

When to Use Tree Based Reduction
When is the tree based algorithm still a good algorithm?
Ask yourself the question … When does the alpha term dominate the beta term?

When : $\beta n \ll \alpha$

When: n is small.

What's Wrong with Tree Based Algorithms?
There is too much redundant communication.

Bucketing Algorithms for Collectives
Using a gather - broadcast scheme can be communication intensive.
Instead have each process send to a neighbor. All processes can do this in parallel.

This will result in P-1 communication steps.
$T(n = mP) = (\alpha + \beta n/P)(P-1) \approx \alpha P + \beta n$

$\beta n \rightarrow$ is optimal
$\alpha P \rightarrow$ is sub-optimal

overall this algorithm is good.

A Bandwidth-optimal Broadcast
To devise a bandwidth-optimal broadcast, use the bandwidth friendly primitives.
First scatter, then all-gather. Reshapes are also necessary.

All-Reduce
All processes have a copy of the data.
What should be combined to make a bandwidth optimal algorithm?

Use reduce-scatter and all-gather