SOFT354: PARALLEL COMPUTATION AND DISTRIBUTED SYSTEMS

Week 9

Dr Robert Merrison-Hort

TODAY'S LECTURE

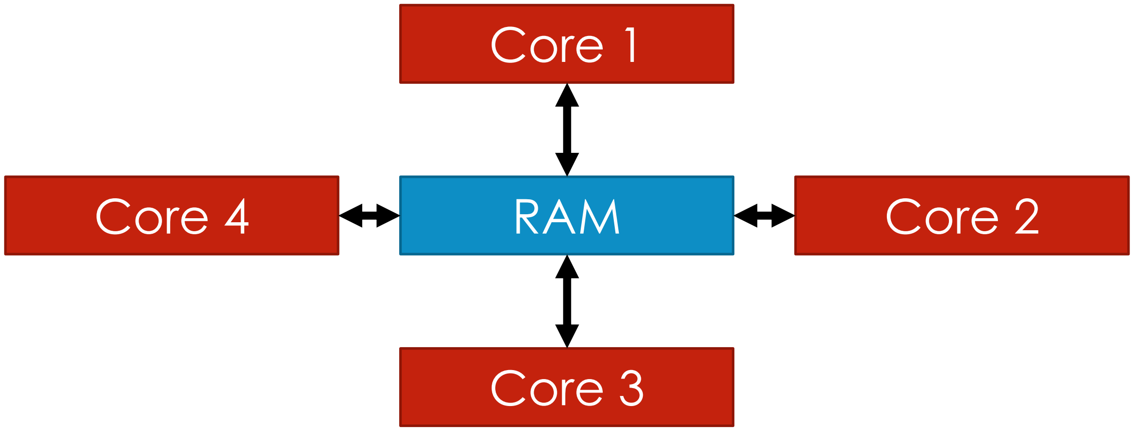
- Connection topologies for high performance computing.
- MPI collective communications.

TODAY'S LECTURE

- Connection topologies for high performance computing.
- MPI collective communications.

CONNECTION TOPOLOGIES

- If we have multiple processing cores, how should they be connected together?
- If they're within the same machine then they can communicate using shared memory (e.g. via the main memory bus).



Effectively every core is connected directly to every other core.

NETWORKING HARDWARE

- Once you have computations running in parallel across multiple machines, you need a network.

- Many options for high-performance computing, e.g:

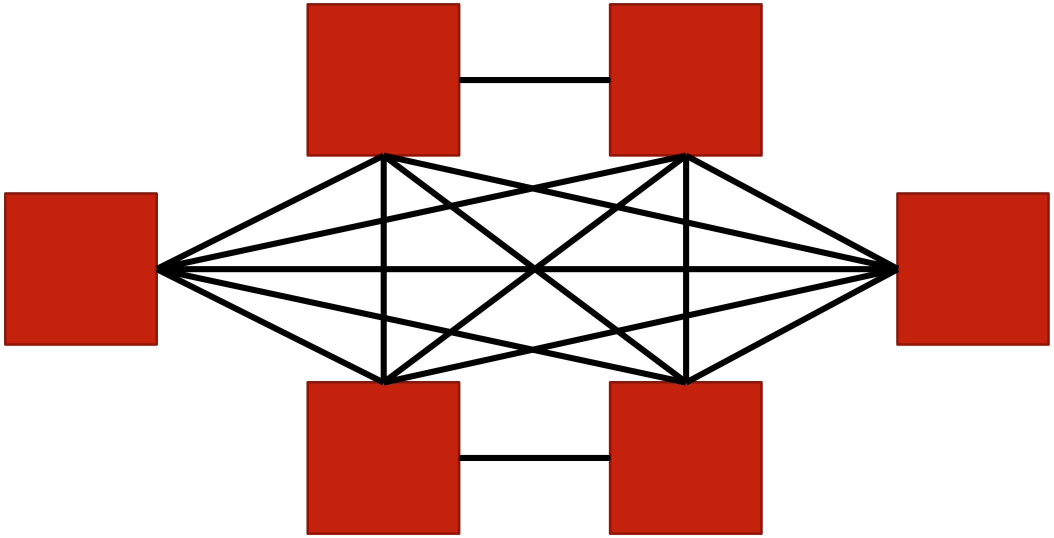
Ethernet:
- 40GB/s with wires, 100GB/s fibre

Infiniband:
- ~25GB/s, wires or fibre
- Links can be “aggregated” (duplicated), e.g. 4x aggregation = 100GB/s
- Supports “Remote DMA” to simulate shared memory.

NETWORK TOPOLOGIES

- Let's assume connections are direct – i.e. there is a physical wire (or fibre) between any two connected machines.
- How should these connections be organised?
- Simplest option is a fully connected network: connect every node to every other node.

FULLY CONNECTED NETWORK

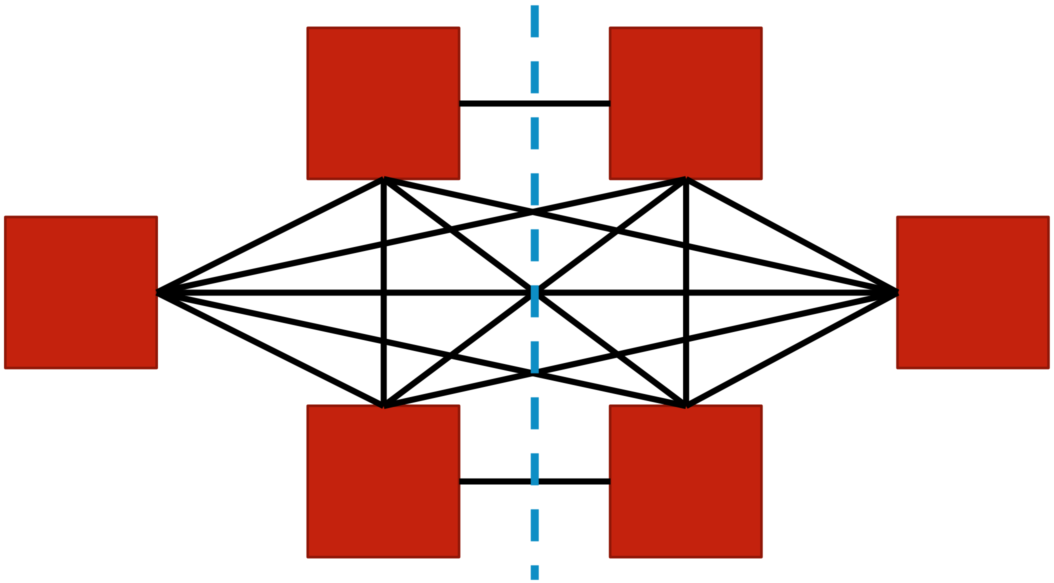


- Let's calculate some properties of this network.

- The diameter of the network is the maximum path length between a pair of nodes.
  - Here all nodes are directly connected, so diameter = 1.

- This is the best possible diameter! Higher values can cause latency.

FULLY CONNECTED NETWORK



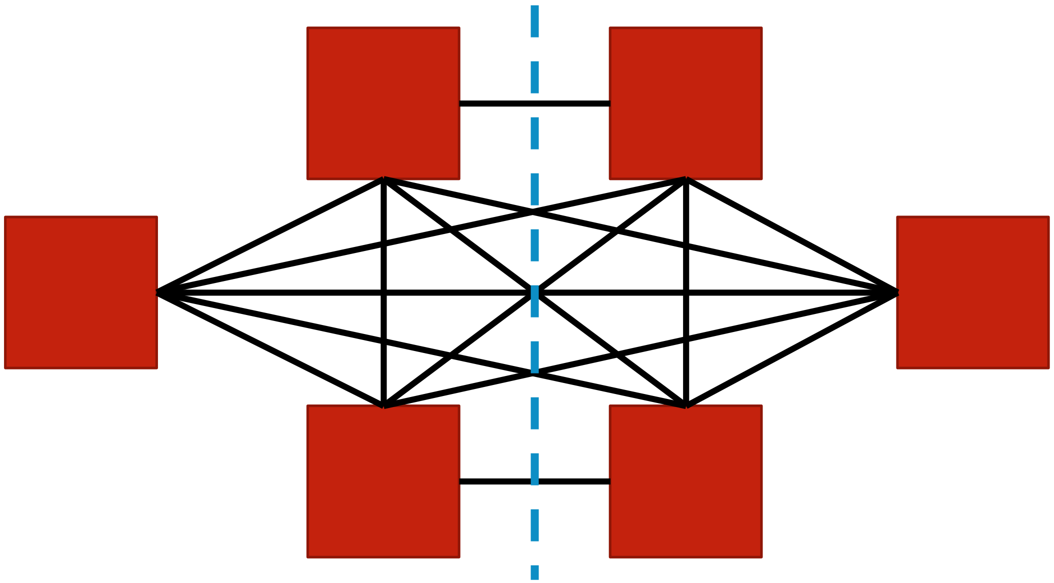
- Let’s calculate some properties of this network.

- The bisection width of the network is the minimum number of links you need to cut to divide the network into two equal halves.

- Here we have to cut 9 connections, so bisection width = 9.
  - In general for a fully-connected network: $\frac{N^2}{4}$

- A high value like this is better: more links → more resilience and…

FULLY CONNECTED NETWORK

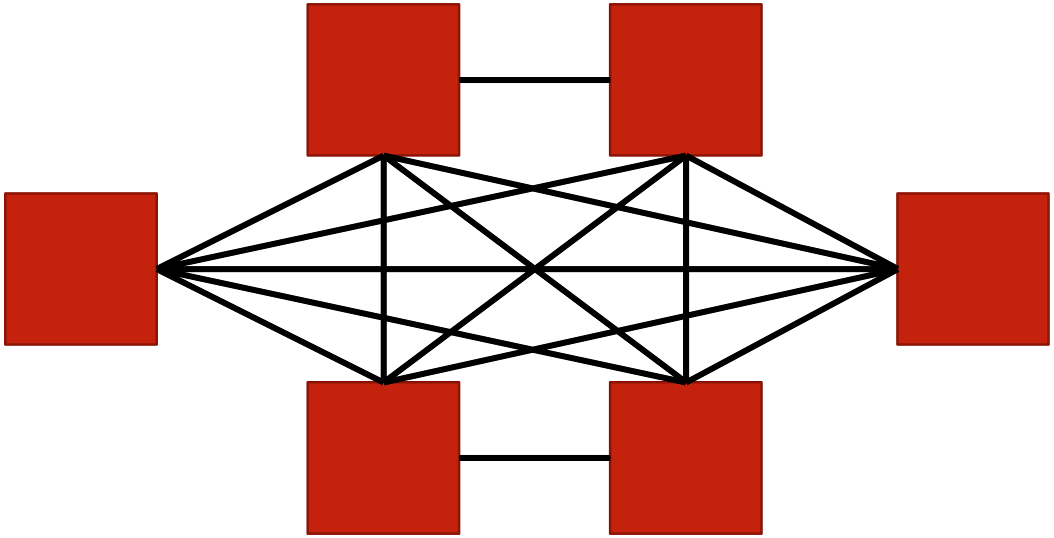


- Let's calculate some properties of this network.

- The bisection bandwidth is the bisection width multiplied by the bandwidth of a link.
  - So in our network, if links were 10GB/s then bisection bandwidth would be 9 * 10GB/s = 90GB/s.

- This is relatively high, which is good! Affects the performance of many parallel algorithms.

FULLY CONNECTED NETWORK

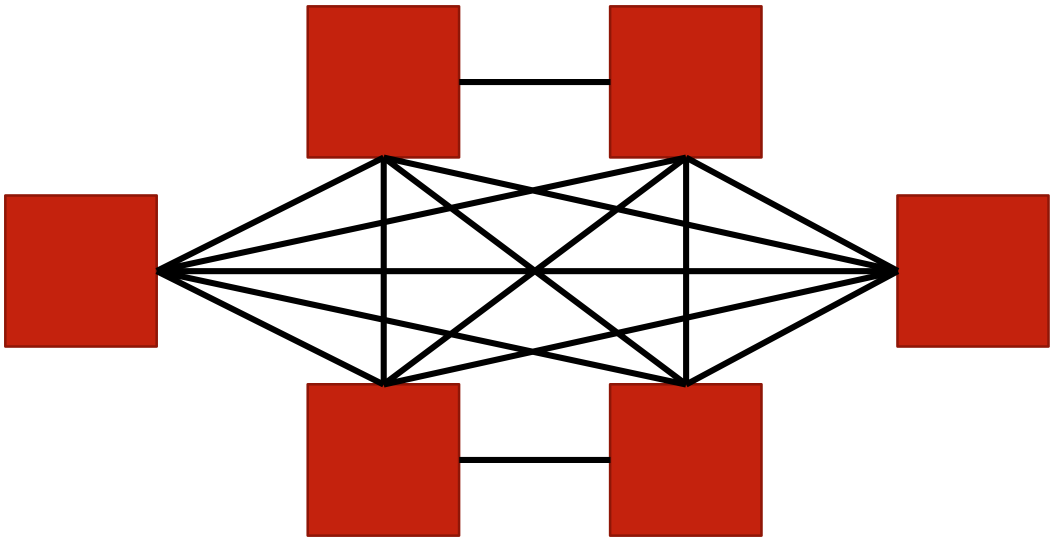


- Let’s calculate some properties of this network.

- The valency is how many connections each node makes.
  - In this case, each node connects to 5 others, so valency = 5.
  - In general, for fully connected network, valency = N-1.

- A high valency is generally bad – if machines are directly connected then you need this many network ports / cables for each machine.
  - Maximum (reasonable) number of ports is 6 or 7.

FULLY CONNECTED NETWORK



- Let's calculate some properties of this network.

- The link count is how many connections the network has in total.
  - In this case, link count = 15.
  - In general, link count = $\frac{Valency \times Number\ of\ Nodes}{2}$

- The fully connected network is also very bad in this area.
  - E.G. with 1,000 nodes, cables required = 499,500 (not practical!)

ALTERNATIVES

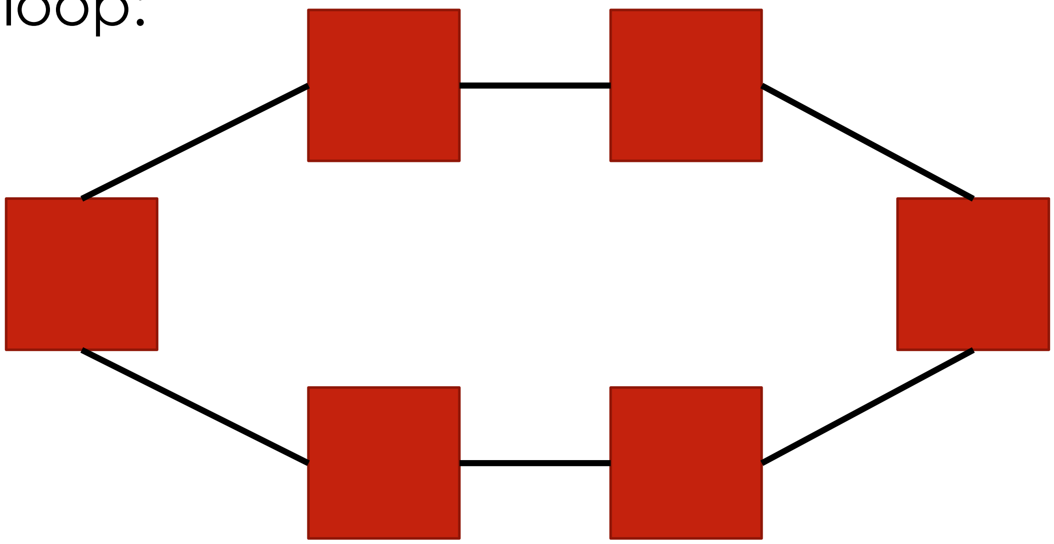
- A fully connected network is simply not practical in reality for more than a very small number of nodes.

- Let's look at two alternatives:
  - Ring
  - Thin tree

- Will see some more complicated options next week:
  - Fat tree
  - Torus

RING TOPOLOGY

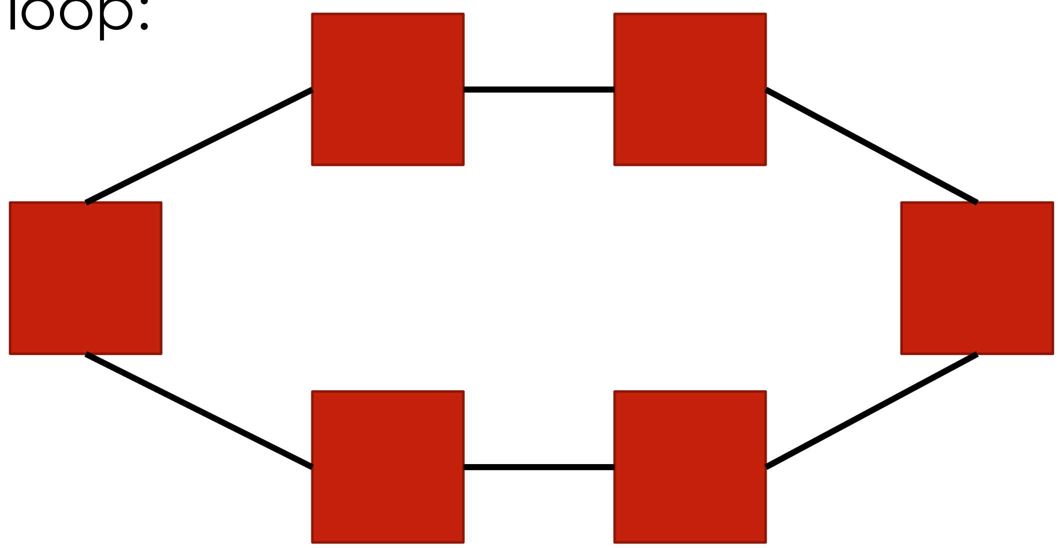
- Each node is connected to two neighbours, forming a closed loop:



- What is the diameter (max path length)?
  - In this case: 3
  - In general: N/2 (have to traverse half the network)
- Not great! For a large network, messages may have high latency.

RING TOPOLOGY

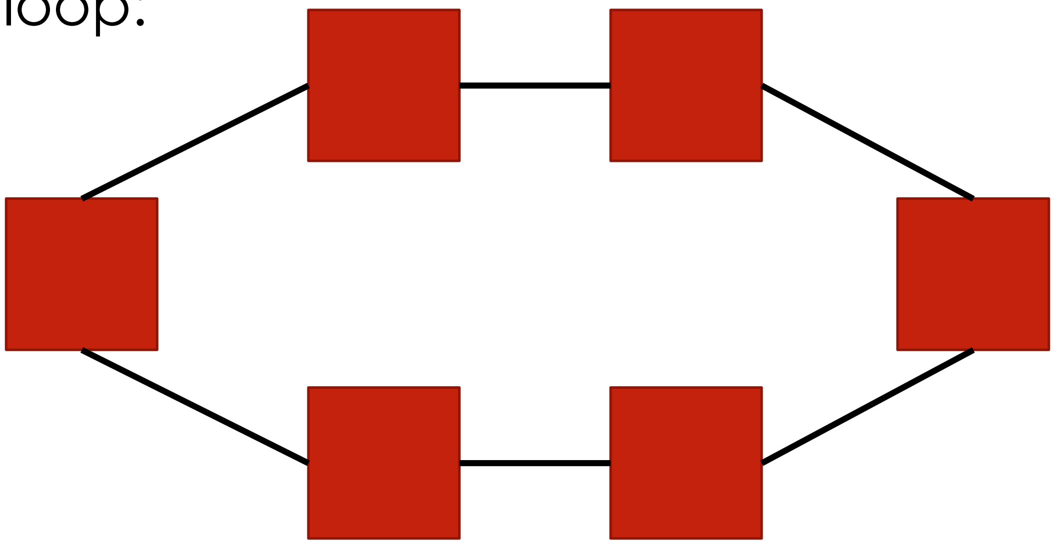
- Each node is connected to two neighbours, forming a closed loop:



- What is the bisection width, bandwidth?
  - In this case: 2, 2*Link bandwidth
  - In general: 2, 2*Link bandwidth (imagine extending both sides).
- Not great! Not resilient to broken links, low bandwidth.

RING TOPOLOGY

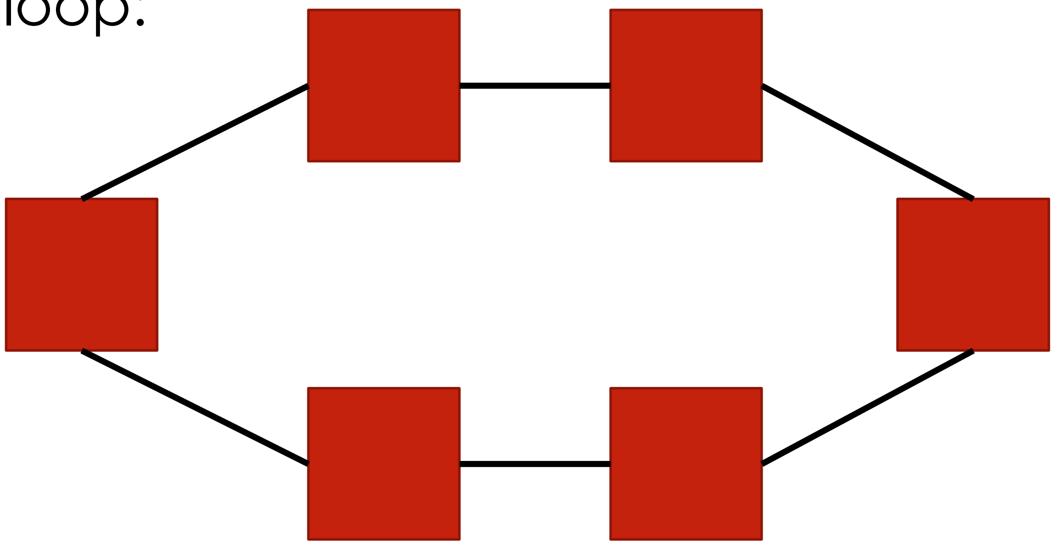
- Each node is connected to two neighbours, forming a closed loop:



- What is the valency (number of links for each node)?
  - In this case: 2
  - In general: 2
- Great! Only need two network ports per machine.

RING TOPOLOGY

- Each node is connected to two neighbours, forming a closed loop:



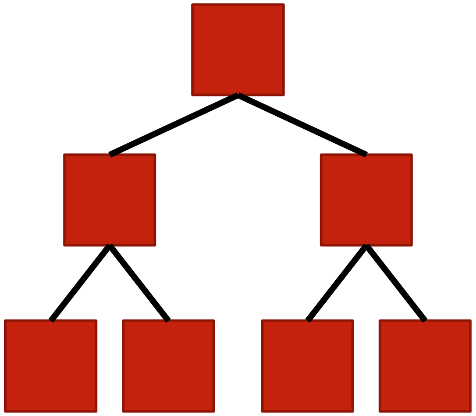
- What is the link count?
    - In this case: 6
    - In general: N
- Great! Only need as many cables as there are machines.

RING TOPOLOGY: SUMMARY

- A ring topology has poor performance:
  - Low bisection width and bisection bandwidth.
  - High diameter.
- But it's very cheap to implement:
  - Only need two network ports per machine (low valence).
  - Only need one cable per machine (low link count).
- In other words, the **exact opposite** of a fully-connected network!

THIN TREE TOPOLOGY

- Arrange nodes into a tree, where each parent has M children. Let's take M=2 (binary tree).



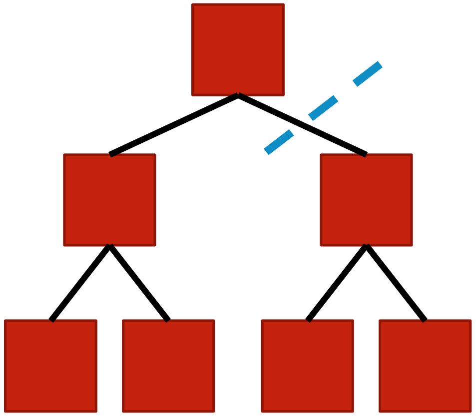
N=7

Note: ~~N=8~~ now!

- What is the diameter (max path length)?
  - In this case: 4
  - In general: $2\log_2 \boxed{\frac{N+1}{2}}$ (Number of leaf nodes)
- Not bad, grows less than linearly with N.

THIN TREE TOPOLOGY

- Arrange nodes into a tree, where each parent has M children. Let's take M=2 (binary tree).



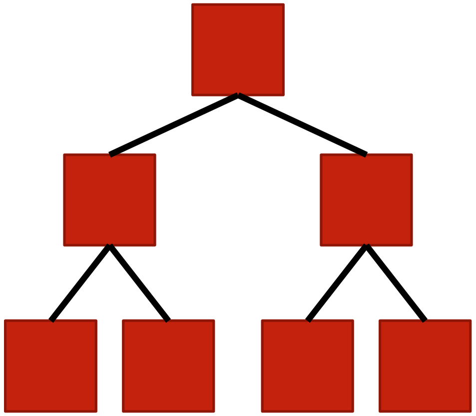
N=7

Note: ~~N=8~~ now!

- What is the bisection width, bandwidth?
  - **Note:** can't exactly cut in half here (odd N)
  - In this case: 1, Link bandwidth
  - In general (for M=2): 1, Link bandwidth
- Not great! Low resilience / bandwidth.

THIN TREE TOPOLOGY

- Arrange nodes into a tree, where each parent has M children. Let's take M=2 (binary tree).



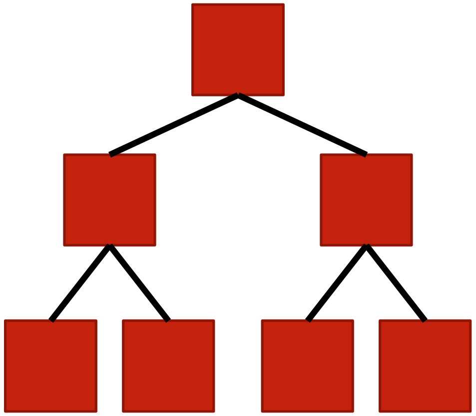
N=7

Note: ~~N=8~~ now!

- What is the valency (number links per node)?
  - In this case: 3
  - In general: M for root, 1 for leaves, M+1 for others.
- Not bad! For a binary tree only need 3 network ports max.

THIN TREE TOPOLOGY

- Arrange nodes into a tree, where each parent has M children. Let's take M=2 (binary tree).



N=7

Note: ~~N=8~~ now!

- What is the link count?
  - In this case: 6
  - In general: homework ☺
- Not bad!

THIN TREE: SUMMARY

- A thin tree is a compromise between a fully connected network and a ring.

- **Good:** low diameter, low valency and low link count.
- **Bad:** low bisection width and bisection bandwidth.

- This means it will be fairly good for latency and quite low cost to implement, but the bandwidth and resilence are limited.

- Next week, more complicated structures: fat trees and N-dimensional toruses.

TODAY'S LECTURE

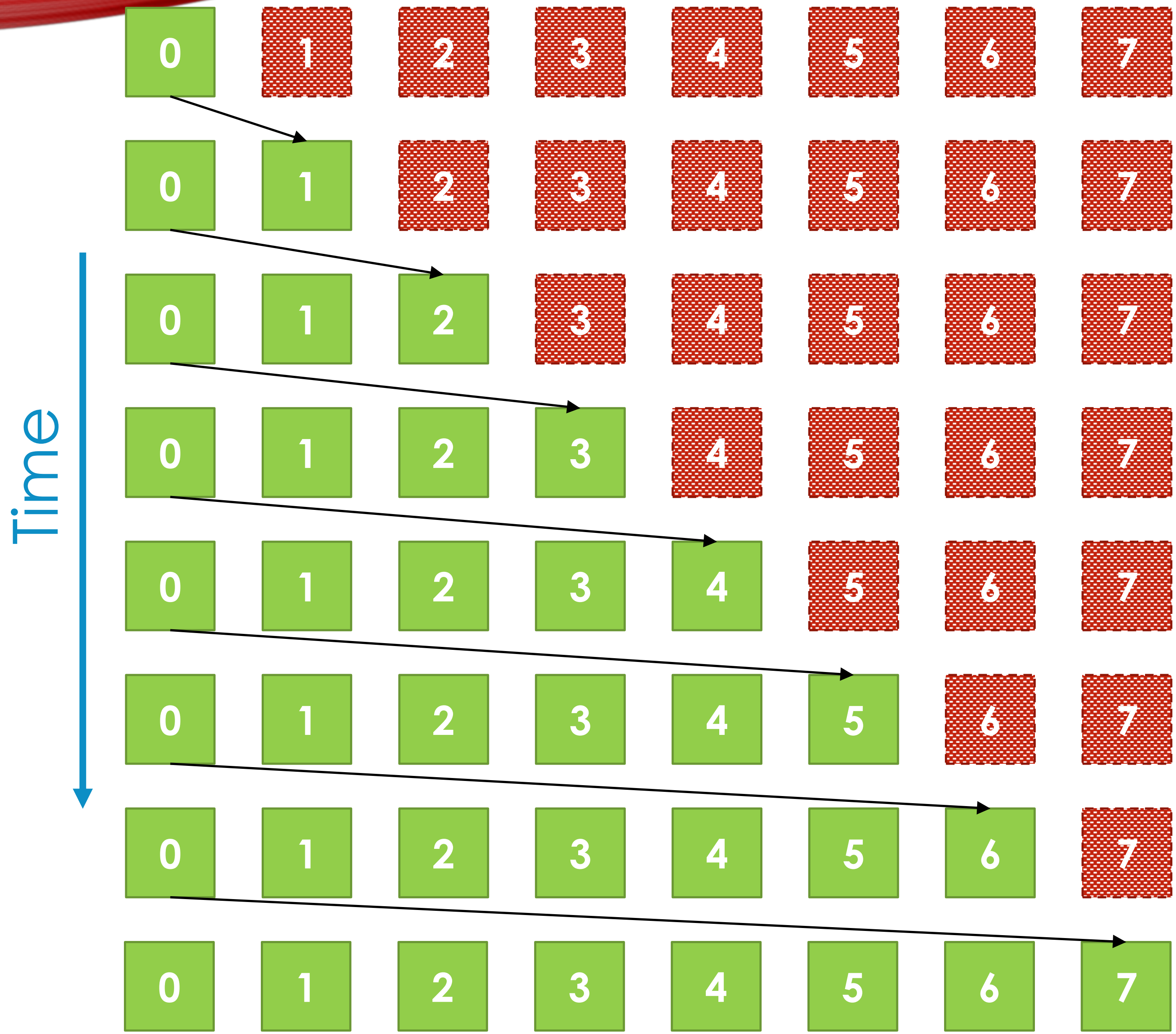
- Connection topologies for high performance computing.
- MPI collective communications.

DISTRIBUTING DATA

- Process 0 needs to transmit some data to every other process – what messages does it send?
- Simplest algorithm: send the data one by one to every process in turn.

```
for (int i=1; i < commSize; i++) {
  MPI_Send(data, dataCount, MPI_FLOAT,
           i, msgTag, MPI_COMM_WORLD);
}
```

DISTRIBUTING DATA



- Takes N = commSize-1 steps to fully distribute the data.
- Process 0 is doing all of the work!

DISTRIBUTING DATA

- More efficient alternative: MPI_Bcast.

```
MPI_Bcast(data, dataCount,
          MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Rank of the "root" process
(the one that has the data)

- **Critical:** MPI_Bcast must to be called by all threads in the communicator, otherwise it'll block and hang!

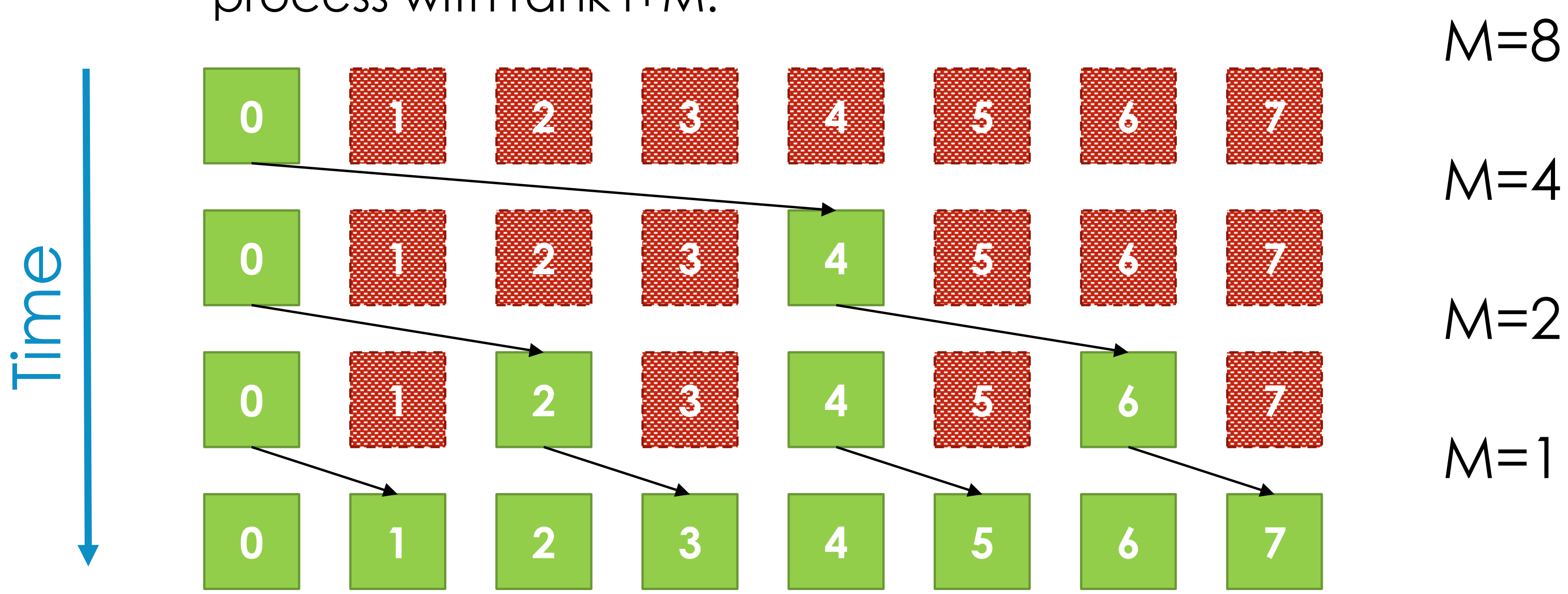
DISTRIBUTING DATA

```
MPI_Bcast(data, dataCount,
          MPI_FLOAT, 0, MPI_COMM_WORLD);
```

- What does this actually do?
- It depends on what MPI implementation you're using, and how your processes are connected.
- Some examples:
  - If processes are on the same PC (and OS supports it), could be inter-process shared memory.
  - If using a LAN that supports it: Ethernet multicast.
  - If using a cluster where every node is connected to every other node: binary tree. **Let's see an example of this…**
  - Otherwise, some algorithm that is optimized for the specific connection structure – active area of research!

BINARY TREE

- Set M = the number of processes.
- At every time step:
  - Divide M by 2.
  - If a process with rank i has the data, it sends it to the process with rank i+M.



BINARY TREE

- This method of distributing data is <u>much</u> more efficient than the simple method.

- With N processes that need the data, if one transmit = one time step:
  - Simple method takes N-1 time steps, root process does all the work.
  - Binary tree takes $\log_2 N$ time steps, work is distributed amongst processes.

- **E.g. N=1 million:** simple method takes 999,999 time steps, binary tree takes 20.

BINARY TREE BROADCAST
(WITH SWEETS!)

- Algorithm:
  - Calculate M = half the number of sweets you've got left to distribute.
  - Give M sweets to the person M places to your right.
  - Repeat until you've only got one sweet left.

MPI_SCATTER

- MPI_Scatter is similar to MPI_Bcast, but instead of sending the same data to every process it distributes the values in an array amongst the processes.

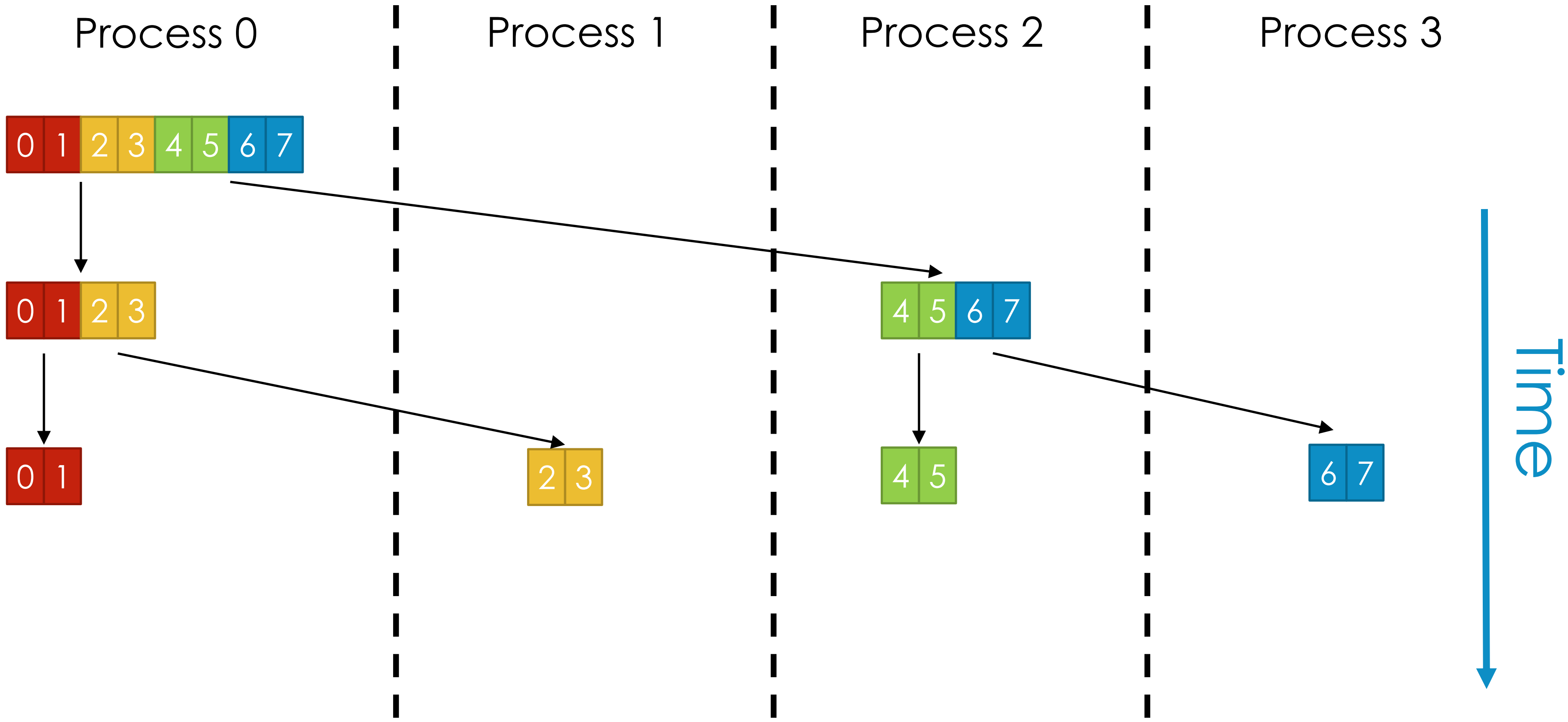
- Example:
  - Process 0 (the root) has an array of eight values:

0	1	2	3	4	5	6	7

  - Use MPI_Scatter to distribute these values to four processes:

0	1	2	3	4	5	6	7
Process 0		Process 1		Process 2		Process 3	

BINARY TREE FOR MPI_SCATTER



MPI_SCATTER

```
int myData[2];

if (myRank == 0) // Root node
{
  int allData[8] = {0, 1, 2, 3, 4, 5, 6, 7};
  MPI_Scatter(allData, 2, MPI_INT,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Scatter(NULL, -1, -1,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
```

Array to be sent.

MPI_SCATTER

```
int myData[2];

if (myRank == 0) // Root node
{
  int allData[8] = {0, 1, 2, 3, 4, 5, 6, 7};
  MPI_Scatter(allData, 2, MPI_INT,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Scatter(NULL, -1, -1,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
```

Number of values to send to each process.

MPI_SCATTER

```
int myData[2];

if (myRank == 0) // Root node
{
  int allData[8] = {0, 1, 2, 3, 4, 5, 6, 7};
  MPI_Scatter(allData, 2, MPI_INT,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Scatter(NULL, -1, -1,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
```

Type of values to send.

MPI_SCATTER

```
int myData[2];

if (myRank == 0) // Root node
{
  int allData[8] = {0, 1, 2, 3, 4, 5, 6, 7};
  MPI_Scatter(allData, 2, MPI_INT,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Scatter(NULL, -1, -1,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
```

Array to store received values in.

MPI_SCATTER

```
int myData[2];

if (myRank == 0) // Root node
{
  int allData[8] = {0, 1, 2, 3, 4, 5, 6, 7};
  MPI_Scatter(allData, 2, MPI_INT,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Scatter(NULL, -1, -1,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
```

Number of values to receive.

MPI_SCATTER

```
int myData[2];

if (myRank == 0) // Root node
{
  int allData[8] = {0, 1, 2, 3, 4, 5, 6, 7};
  MPI_Scatter(allData, 2, MPI_INT,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Scatter(NULL, -1, -1,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
```

Type of values to receive.

MPI_SCATTER

```
int myData[2];

if (myRank == 0) // Root node
{
  int allData[8] = {0, 1, 2, 3, 4, 5, 6, 7};
  MPI_Scatter(allData, 2, MPI_INT,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Scatter(NULL, -1, -1,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
```

Rank of root process (the one with the data).

MPI_SCATTER

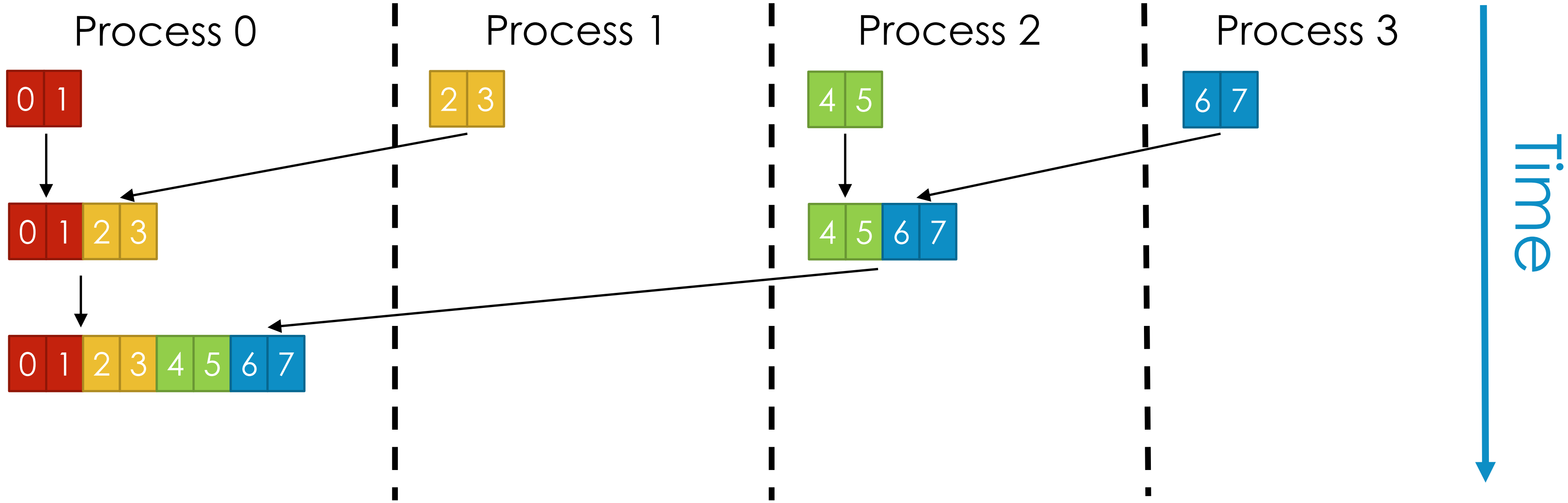
```
int myData[2];

if (myRank == 0) // Root node
{
  int allData[8] = {0, 1, 2, 3, 4, 5, 6, 7};
  MPI_Scatter(allData, 2, MPI_INT,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Scatter(NULL, -1, -1,
              myData, 2, MPI_INT,
              0, MPI_COMM_WORLD);
}
```

Communicator to use.

MPI_GATHER

- Basically does the exact opposite of MPI_Scatter.

- Collects data from arrays distributed across all the processes into one big array on the root process.



MPI_GATHER
(WITH SWEETS!)

- Algorithm:
  - Divide everyone who still has sweets into pairs.
  - If you are the right-most person in your pair, give the other person your sweets.
  - Repeat until all the sweets have got to the root process (**me** :D)

MPI_REDUCE

- You hopefully remember the reduce operation from practical workshop 7 (Thrust).

- Take an array of values and reduce it to a single value.

- Examples (**A=[0.5, 2.0, 1.5. 1.0]**):
  - Sum(A) = 0.5 + 2.0 + 1.5 + 1.0 = 5.0
  - Prod(A) = 0.5 * 2.0 * 1.5 * 1.0 = 1.5
  - Max(A) = 2.0
  - ArgMax(A) = 1 (index of maximum element)

MPI_REDUCE

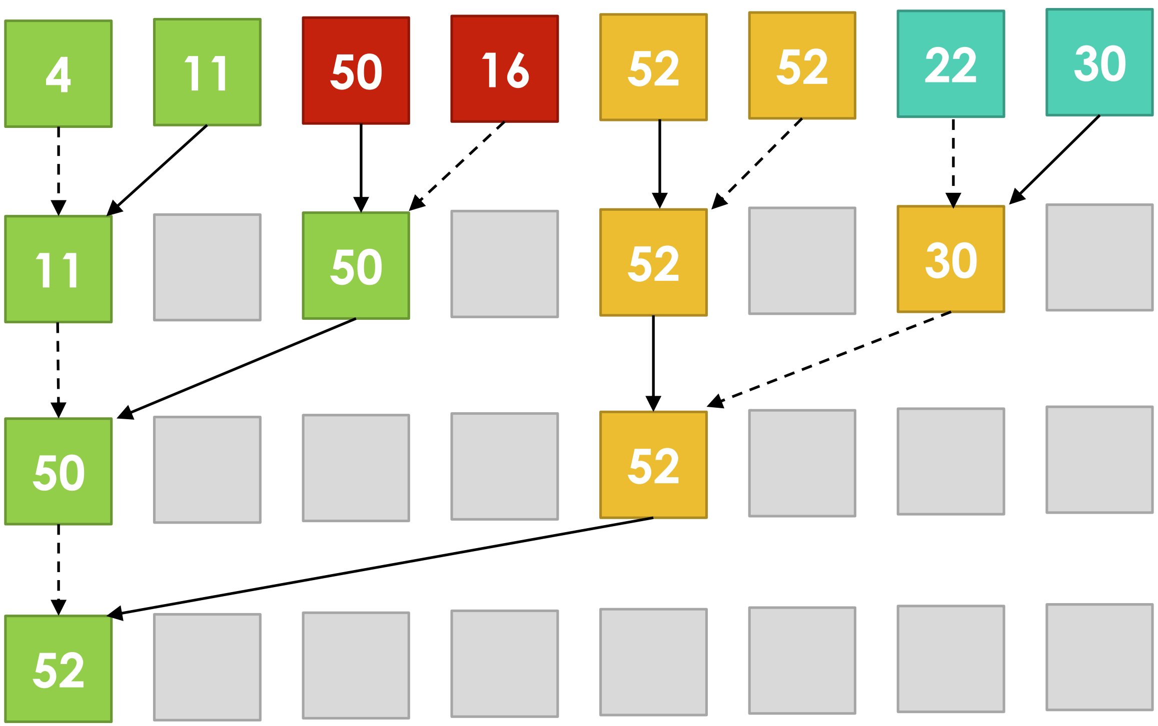
- MPI_Reduce can use a similar binary tree structure for communication.

- For example, if the operation is Max then repeatedly:
  - Divide remaining processes into pairs.
  - 2nd process in each pair gives its value to the 1st one.
  - 1st process keeps the maximum of the two values.
  - 2nd process doesn't do anything else.
  - Repeat.

MPI_REDUCE (MAX)

- Divide (active) processes into pairs.
- 2nd process in pair gives its value to 1st process, goes inactive.
- 1st process in pair takes the bigger of the two values.
- Repeat.

MPI_REDUCE (MAX)



- Divide (active) processes into pairs.
- 2^{nd} process in pair gives its value to 1^{st} process, goes inactive.
- 1^{st} process in pair takes the bigger of the two values.
- Repeat.

MPI_REDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

if (myRank == 0) // Root node
{
  float maxValue;
  MPI_Reduce(&myValue, &maxValue, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Reduce(&myValue, NULL, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
```

Local value for this process.

MPI_REDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

if (myRank == 0) // Root node
{
  float maxValue;
  MPI_Reduce(&myValue, &maxValue, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Reduce(&myValue, NULL, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
```

Where to store the result of the reduction (only used on the root process)

MPI_REDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

if (myRank == 0) // Root node
{
  float maxValue;
  MPI_Reduce(&myValue, &maxValue, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Reduce(&myValue, NULL, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
```

How many elements in the input arrays.

If > 1, reduction is done for each array position independently.

MPI_REDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

if (myRank == 0) // Root node
{
  float maxValue;
  MPI_Reduce(&myValue, &maxValue, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Reduce(&myValue, NULL, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
```

Data type of each element.

MPI_REDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

if (myRank == 0) // Root node
{
  float maxValue;
  MPI_Reduce(&myValue, &maxValue, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Reduce(&myValue, NULL, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
```

Reduction operation (list in a minute…)

MPI_REDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

if (myRank == 0) // Root node
{
  float maxValue;
  MPI_Reduce(&myValue, &maxValue, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Reduce(&myValue, NULL, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
```

Rank of root process (the one that will receive the result).

MPI_REDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

if (myRank == 0) // Root node
{
  float maxValue;
  MPI_Reduce(&myValue, &maxValue, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
else { // Non-root node
  MPI_Reduce(&myValue, NULL, 1,
             MPI_FLOAT, MPI_MAX, 0,
             MPI_COMM_WORLD);
}
```

Communicator to use.

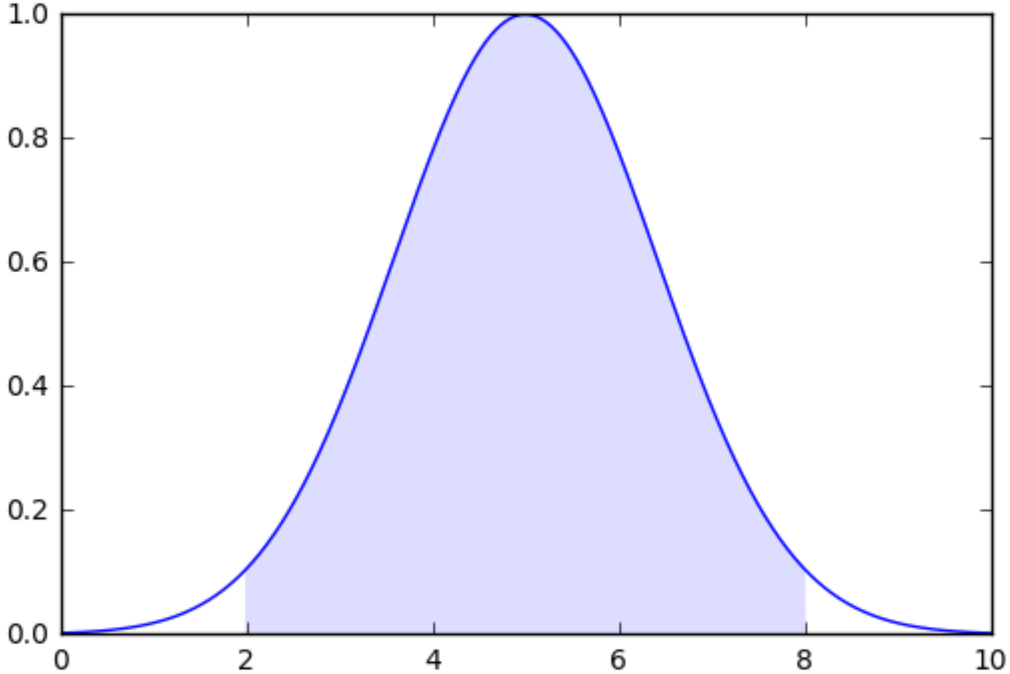
POSSIBLE REDUCTION OPERATIONS

- Extrema:
  - MPI_MAX, MPI_MIN
  - MPI_MAXLOC, MPI_MINLOC (returns min/max + index; see docs)
- Maths:
  - MPI_SUM, MPI_PROD (product)
- Logical operators:
  - MPI_LAND (are all values true?)
  - MPI_LOR (is at least one value true?)
  - MPI_LXOR (is exactly one value true?)
- Bitwise operators:
  - MPI_BAND (bitwise AND all values together)
  - MPI_BOR (bitwise OR all values together)
  - MPI_BXOR (bitwise XOR all values together)

Can also define your own operations, see documentation!

REDUCTION EXAMPLE: TRAPEZOID RULE

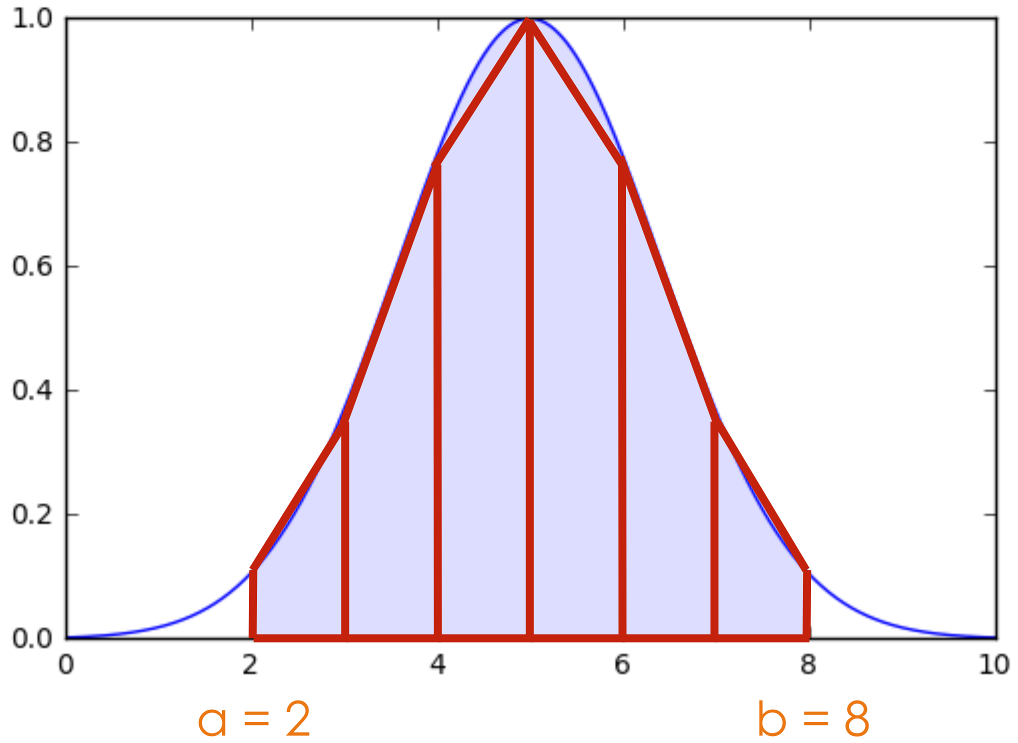
- Aim: calculate the area under a curve described by some mathematical function, between points a and b.
- E.g.:



a = 2 b = 8

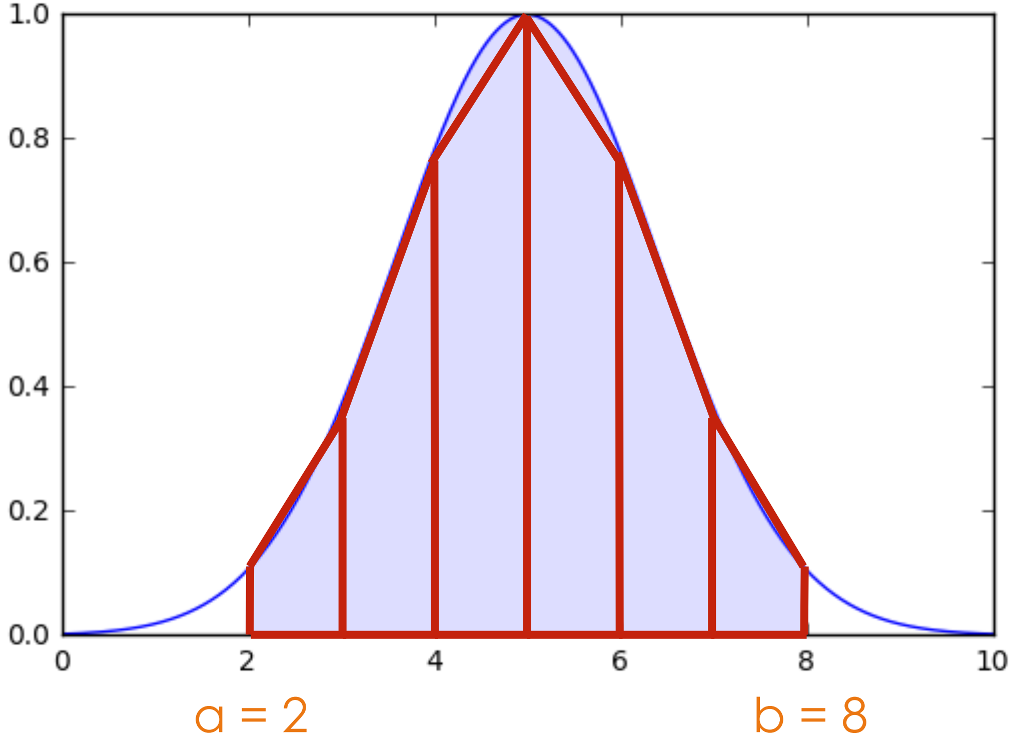
REDUCTION EXAMPLE: TRAPEZOID RULE

- Strategy: divide the area up into trapezoids with equal width. Num. trapezoids = Num. processes = N.
- E.g.:
- N=6



Note: overly simple – would probably want multiple trapezoids per process!

REDUCTION EXAMPLE: TRAPEZOID RULE



- If the x position of the start and end points of a trapezoid are x_1 and x_2…
- … the area of the trapezoid is:

$$(x_2 - x_1)\frac{f(x_1) + f(x_2)}{2}$$

- Each process calculates this for one trapezoid.
- Use MPI_Reduce to collect the sum of them all.
- Next week's workshop!

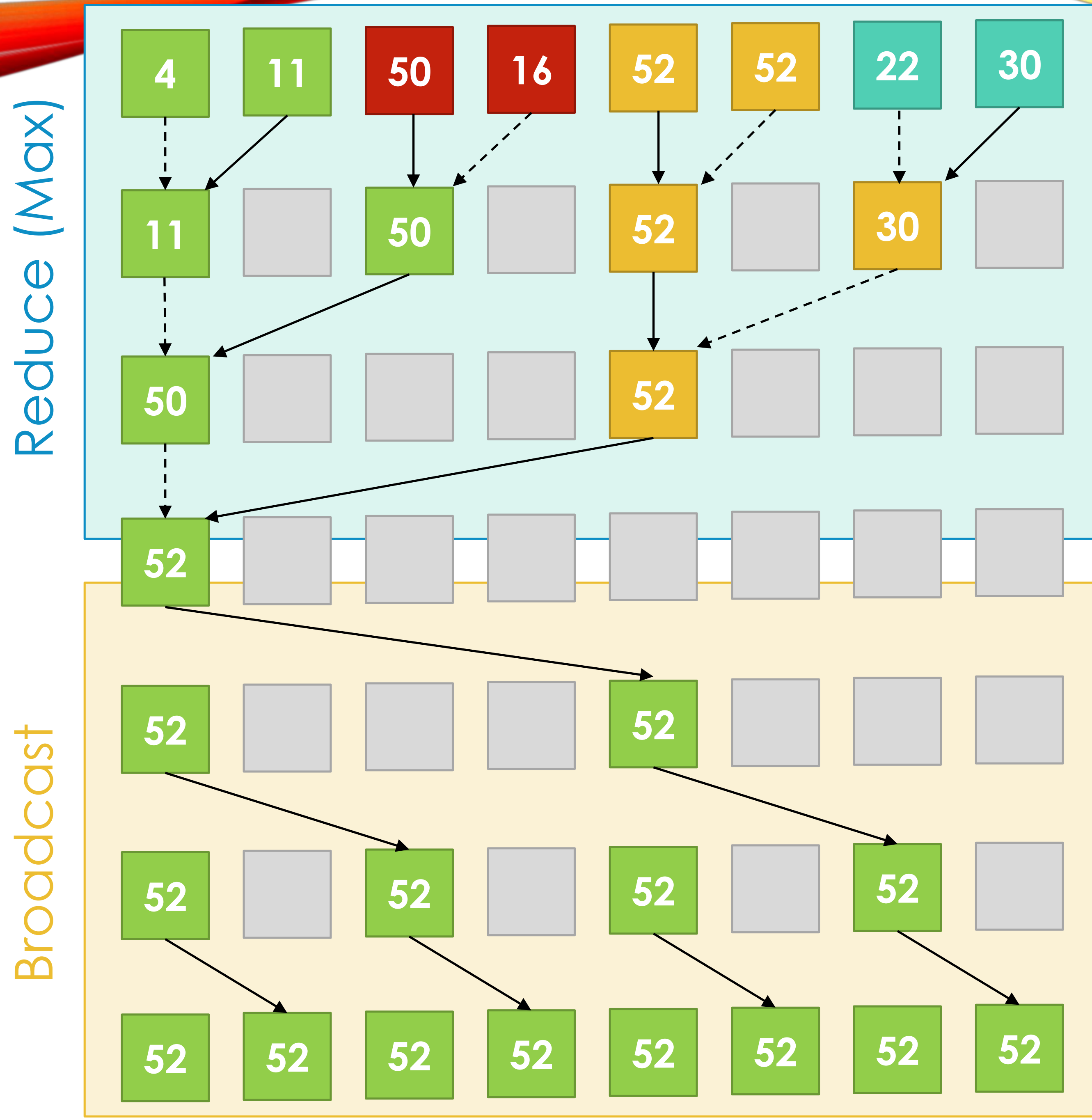
MPI_ALLREDUCE

- What if calculating a reduction is only part of the algorithm you want to run?

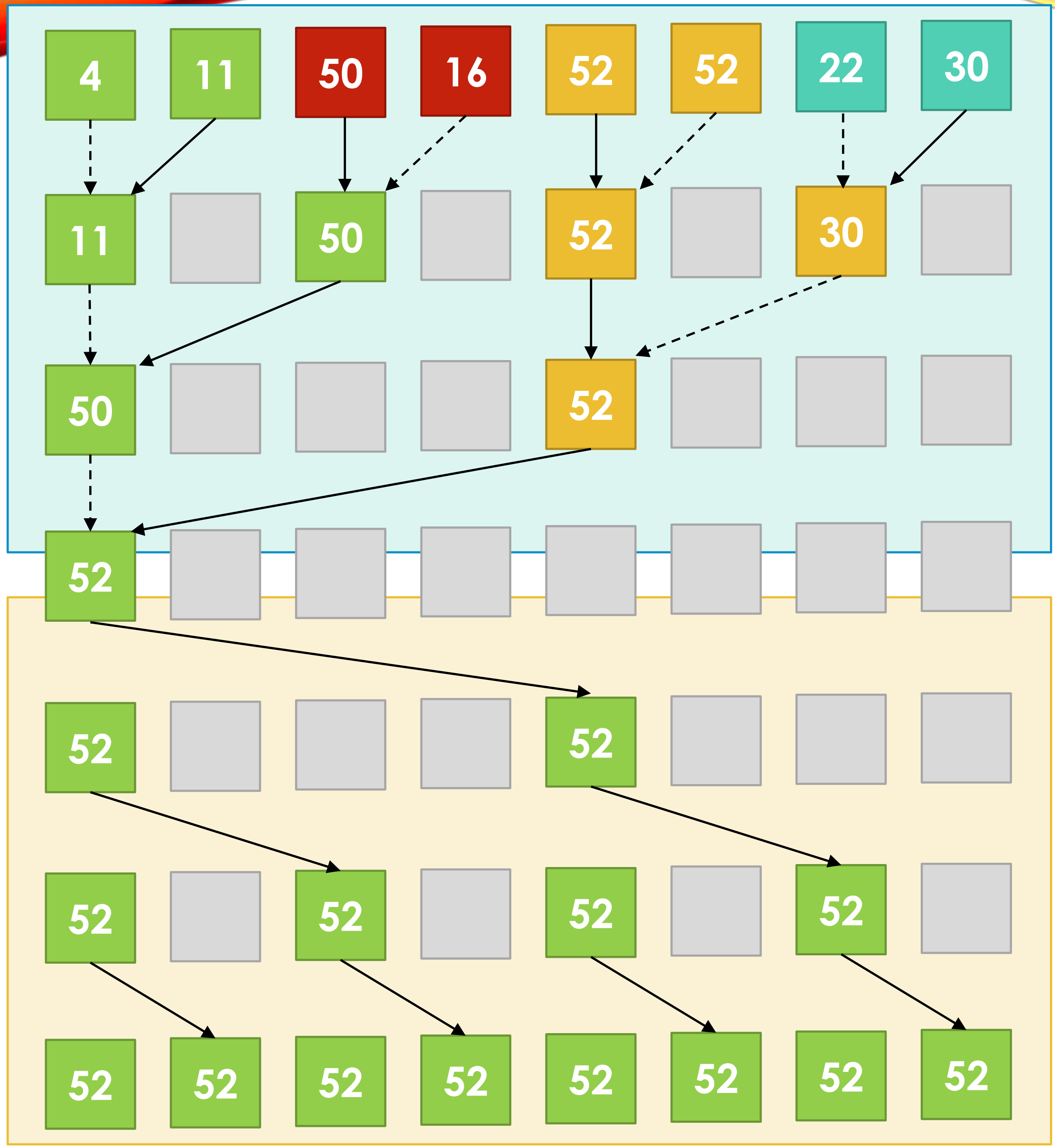
- E.g. calculate the area under a curve using Trapezoid Rule, and then use that result to do further parallel computations.

- In this case, every process needs to get the result of the reduction.

- One possibility: MPI_Reduce and then MPI_Bcast…


Reduce (Max)
Broadcast
4
11
50
16
52
52
22
30

- Reduction takes $\log_2 N$ steps.
- Broadcast also takes $\log_2 N$ steps.
- So total number of steps is $2\log_2 N$.
- Can do better!

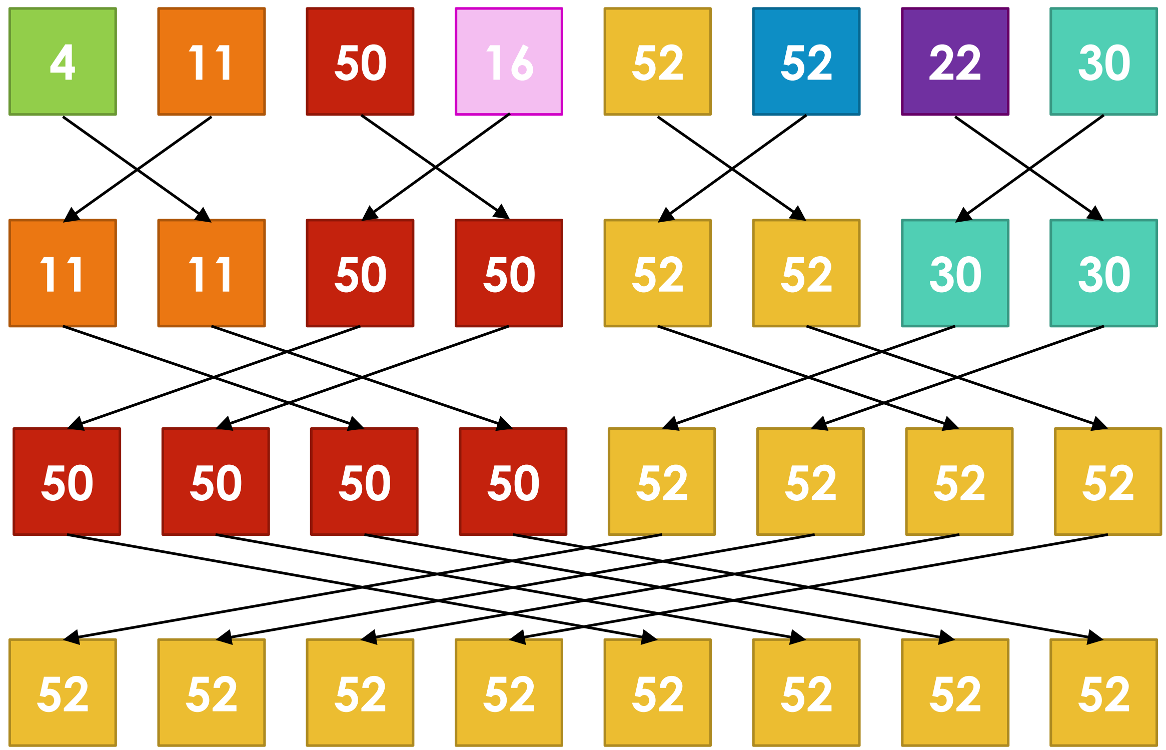


BUTTERFLY

Pairs of individual processes compare values…

Pairs of blocks of 2 processes compare values…

Pairs of blocks of 4 processes compare values…



Requires $\log_2 N$ steps – same as reduce or broadcast alone!

(But does require more data transfers…)

BUTTERFLY (SUM)
(NO SWEETS ☹)

- Algorithm:
  - Divide into pairs – exchange numbers and both replace your number with the sum.
  - Join up with the pair next to you. The two left-side partners exchange numbers and sum, and the two right-side partners do the same.
  - Join up with the other group of four. Exchange numbers and sum with your corresponding person in the other group.

MPI_ALLREDUCE

- The same as MPI_Reduce but the result is distributed to all of the processes.

- May be implemented using a butterfly-type algorithm, but depends on environment.
  - If all processes on the same machine and OS supports it, could use shared memory.
  - If on a LAN with multicast then a normal reduce followed by a multicast network message might be faster.

MPI_ALLREDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

float maxValue;
MPI_Allreduce(&myValue, &maxValue, 1,
              MPI_FLOAT, MPI_MAX, 0,
              MPI_COMM_WORLD);
```

Local value for this process.

MPI_ALLREDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

float maxValue;
MPI_Allreduce(&myValue, &maxValue, 1,
              MPI_FLOAT, MPI_MAX, 0,
              MPI_COMM_WORLD);
```

Where to store the result.

MPI_ALLREDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

float maxValue;
MPI_Allreduce(&myValue, &maxValue, 1,
              MPI_FLOAT, MPI_MAX,
              MPI_COMM_WORLD);
```

How many elements in the input arrays.

If > 1, reduction is done for each array position independently.

MPI_ALLREDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

float maxValue;
MPI_Allreduce(&myValue, &maxValue, 1,
              MPI_FLOAT, MPI_MAX,
              MPI_COMM_WORLD);
```

Type of elements.

MPI_ALLREDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

float maxValue;
MPI_Allreduce(&myValue, &maxValue, 1,
              MPI_FLOAT, MPI_MAX,
              MPI_COMM_WORLD);
```

Reduction operation.

MPI_ALLREDUCE

```
// Assume each process has a float variable
// called myValue containing a value.

float maxValue;
MPI_Allreduce(&myValue, &maxValue, 1,
              MPI_FLOAT, MPI_MAX,
              MPI_COMM_WORLD);
```

Communicator.

Note: Same code for all processes! No root node.

MPI_ALLGATHER

- Collects data from arrays distributed across all the processes into one big array that all processes have a copy of.
- Could be implemented using a butterfly arrangement: