spcl.inf.ethz.ch
@spcl_eth
DINFK

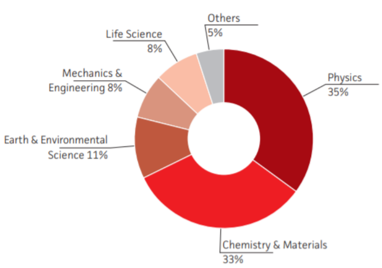
TORSTEN HOEFLER

Parallel Programming

Message Passing & Parallel Sorting (A taste of parallel algorithms!)

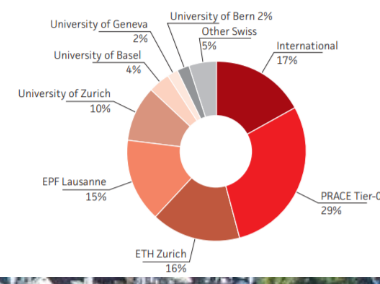
User Lab Usage by Research Field

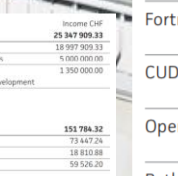
Research Field	Node h	%
Physics	15 537 867	35
Chemistry & Materials	14 370 198	33
Earth & Environmental Science	4 736 388	11
Mechanics & Engineering	3 650 101	8
Life Science	3 308 827	8
Others	2 175 327	5
Total Usage	**43 778 708**	**100**



User Lab Usage by Institution

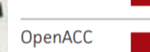
Institution	Node h	%
International	7 327 101	17
PRACE Tier-0	12 598 014	29
ETH Zurich	7 201 448	16
EPF Lausanne	6 532 008	15
University of Zurich	4 229 333	10
University of Basel	1 840 126	4
University of Geneva	941 616	2
University of Bern	749 782	2
Other Swiss	2 359 280	5
Total Usage	**43 778 708**	**100**

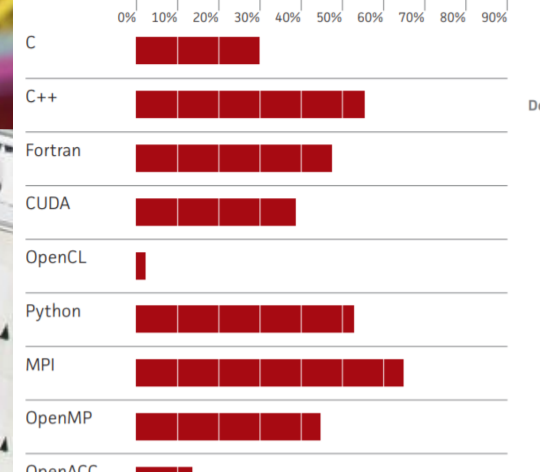




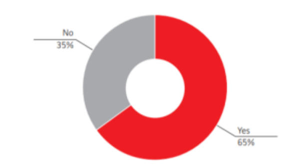
	Expenses CHF
Investments	**2 472 847.05**
Equipment and Furniture	**36 602.15**
Personnel	**10 479 422.83**
Payroll	8 082 858.03
Employer's Contributions	1 452 619.20
Further Education, Travel, Recruitment	943 945.60
Other Material Expenses	**8 162 250.28**
Maintenance Building & Technical Infrastructure	431 896.60
Energy	2 383 238.73
Administrative Expenses	7 438.26
Hardware, Software, Services	4 323 130.14
Remunerations, Marketing Workshops, Services	511 615.24
Other	504 931.31
Extraordinary Expenditures	**1 215 548.59**
Membership Fees	58 461.46
PASC Initiative Contribution External Projects	1 157 087.13
Total Expenses	**22 366 670.90**
Balance	

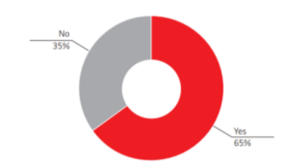
	Income CHF
Basic Budget Income	**25 347 909.33**
ETH Zurich Operations	18 997 909.33
ETH-Rat – HPCN Investments	5 000 000.00
ETH-Rat – PASC Initiative Software Development	1 350 000.00
Other Income	**151 784.32**
Services / Courses	73 447.24
Reimbursements	18 810.88
Other	59 526.20
Total Income	**25 499 693.65**
Balance	**3 133 022.75**





Do you develop and maintain application codes?





Last week

- **Transactional memory**
  - Motivation (locks are bad, wait-/lock-free is hard)
  - Concepts (Atomicity, Consistency, Isolation – ACI(D))
  - Implementation options (keep track of read and write sets)
  - Example: dining philosophers

- **Distributed memory**
  - Isolation of state – big simplification
  - Event-driven messaging/Actors (example: Erlang)
  - CSP-style (example: Go)

Learning goals for today

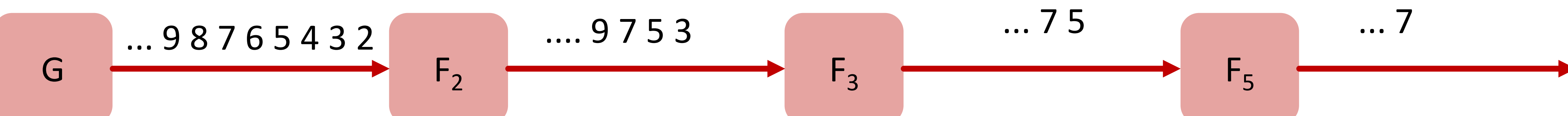
- **Finish Go**

- **Message Passing Interface**
  - Standard library for high-performance parallel programming
  - Processes, communicators, collectives – concepts of distributed memory programming
  - Matching, deadlocks – potential pitfalls

- **A primer on parallel algorithms**
  - Parallel sorting
  - Sorting with fixed structures – sorting networks

Example: Concurrent prime sieve

Each station removes multiples of the first element received and passes on the remaining elements to the next station



Concurrent prime sieve

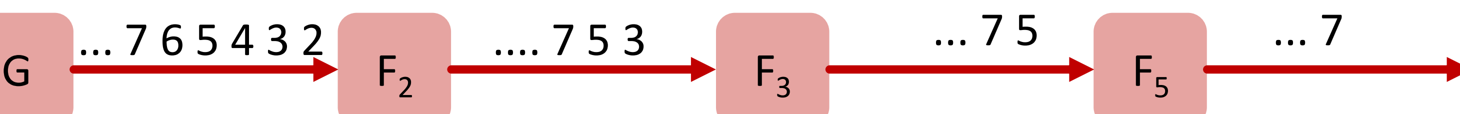
```
func Generate(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i
    }
}
```

G

```
func Filter(in <-chan int, out chan<- int, prime int) {
    for {
        i := <-in  // Receive value from 'in'.
        if i%prime != 0 {
            out <- i // Send 'i' to 'out'.
        }
    }
}
```

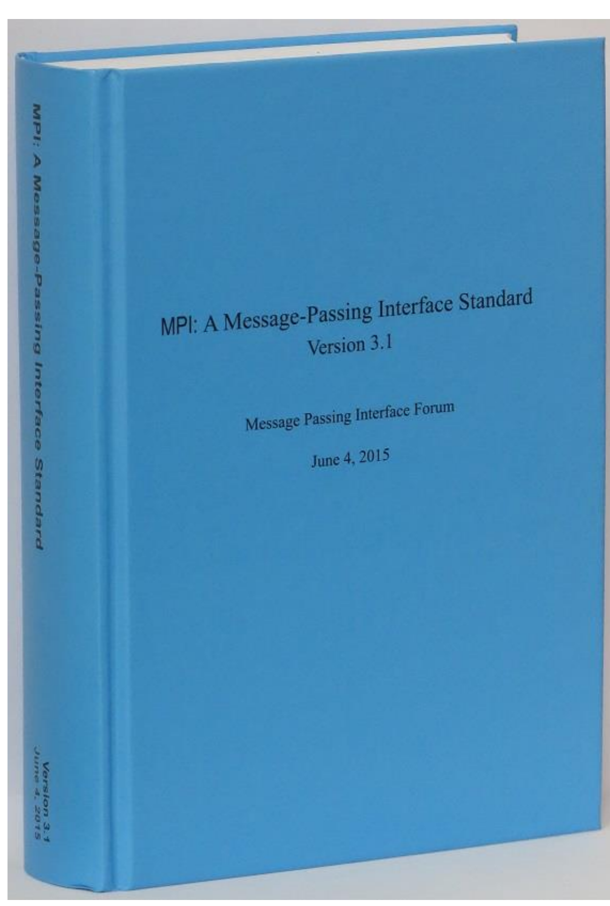
F_{prime}

```
func main() {
    ch := make(chan int)
    go Generate(ch)
    for i := 0; i < 10; i++ {
        prime := <-ch
        fmt.Println(prime)
        ch1 := make(chan int)
        go Filter(ch, ch1, prime)
        ch = ch1
    }
}
```



source code from golang.org

Message Passing Interface (MPI)



Parallel Processing Letters
© World Scientific Publishing Company

MPI ON MILLIONS OF CORES*

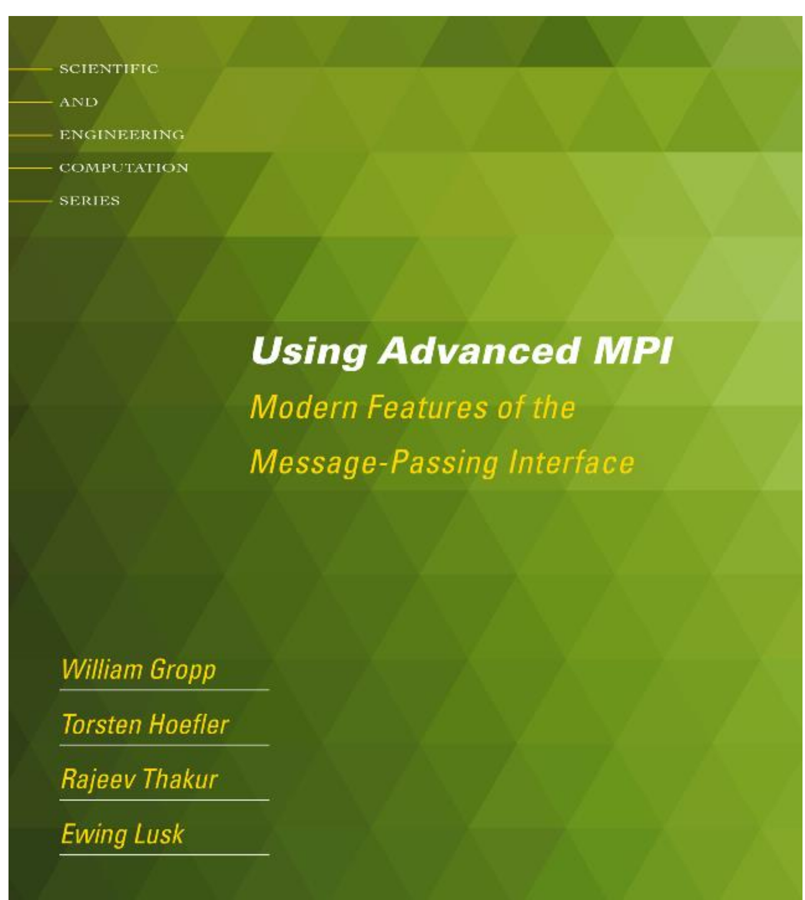
PAVAN BALAJI,[1] DARIUS BUNTINAS,[1] DAVID GOODELL,[1]
WILLIAM GROPP,[2] TORSTEN HOEFLER,[2] SAMEER KUMAR,[3]
EWING LUSK,[1] RAJEEV THAKUR,[1] JESPER LARSSON TRÄFF[4†]

[1] *Argonne National Laboratory, Argonne, IL 60439, USA*
[2] *University of Illinois, Urbana, IL 61801, USA*
[3] *IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA*
[4] *Dept. of Scientific Computing, Univ. of Vienna, Austria*

Received December 2009
Revised August 2010
Communicated by J. Dongarra

ABSTRACT

Petascale parallel computers with more than a million processing cores are expected to be available in a couple of years. Although MPI is the dominant programming interface today for large-scale systems that at the highest end already have close to 300,000 processors, a challenging question to both researchers and users is whether MPI will scale to processor and core counts in the millions. In this paper, we examine the issue of scalability of MPI to very large systems. We first examine the MPI specification itself and discuss areas with scalability concerns and how they can be overcome. We then investigate issues that an MPI implementation must address in order to be scalable. To illustrate the issues, we ran a number of simple experiments to measure MPI memory consumption at scale up to 131,072 processes, or 80%, of the IBM Blue Gene/P system at Argonne National Laboratory. Based on the results, we identified nonscalable aspects of the MPI implementation and found ways to tune it to reduce its memory footprint. We also briefly discuss issues in application scalability to large process counts and fea-



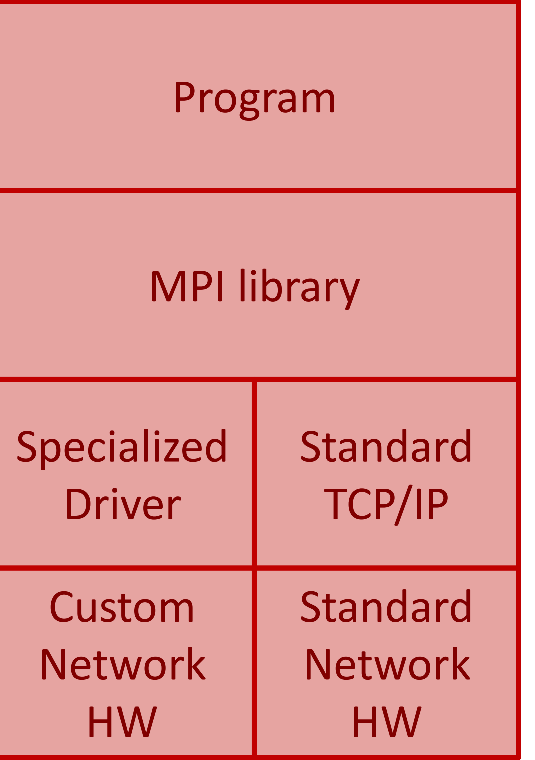
Message Passing Interface (MPI)

Message passing libraries:

- PVM (Parallel Virtual Machines) 1980s
- MPI (Message Passing Interface) 1990s

MPI = Standard API

- Hides Software/Hardware details
- Portable, flexible
- Implemented as a library

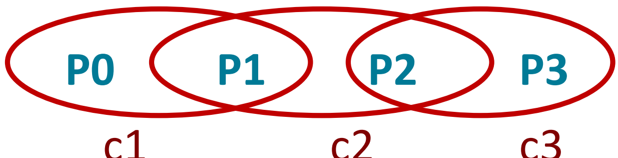


Process Identification

- **MPI processes can be collected into groups**
  - Each group can have multiple colors (some times called context)
  - *Group + color == communicator (it is like a name for the group)*
  - When an MPI application starts, the group of all processes is initially given a predefined name called **`MPI_COMM_WORLD`**
    - *The same group can have many names, but simple programs do not have to worry about multiple names*
- **A process is identified by a unique number within each communicator, called *rank***
  - For two different communicators, the same process can have two different ranks: so the meaning of a “rank” is only defined when you specify the communicator

MPI Communicators

- **Defines the communication domain of a communication operation: set of processes that are allowed to communicate with each other.**



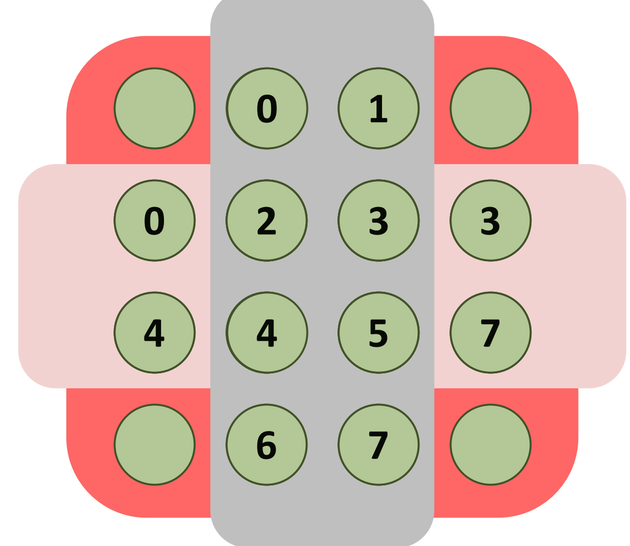
- **Initially all processes are in the communicator MPI_COMM_WORLD.**

- **The rank of processes are associated with (and unique within) a communicator, numbered from 0 to n-1**

Communicators

Communicators do not need to contain all processes in the system

Every process in a communicator has an ID called as “rank”



When you start an MPI program, there is one predefined communicator **MPI_COMM_WORLD**

Can make copies of this communicator (same group of processes, same ranks, but different “aliases”)

The same process might have different ranks in different communicators

Communicators can be created “by hand” or using tools

Simple programs typically only use the predefined communicator **MPI_COMM_WORLD**

(which is sometimes considered bad practice because of modularity issues)

Process Ranks

Processes are identified by nonnegative integers, called *ranks*

p* processes are numbered *0, 1, 2, .. p-1

```
public static void main(String args []) throws Exception {
    MPI.Init(args);
    // Get total number of processes (p)
    int size = MPI.COMM_WORLD.Size();
    // Get rank of current process (in [0..p-1])
    int rank = MPI.COMM_WORLD.Rank();
    MPI.Finalize();
}
```

SPMD

Single Program

Multiple Data
(Multiple Instances)

```
if (rank == 0)
      do this
else
      do that
```

we compile one program

the if-else makes it SPMD

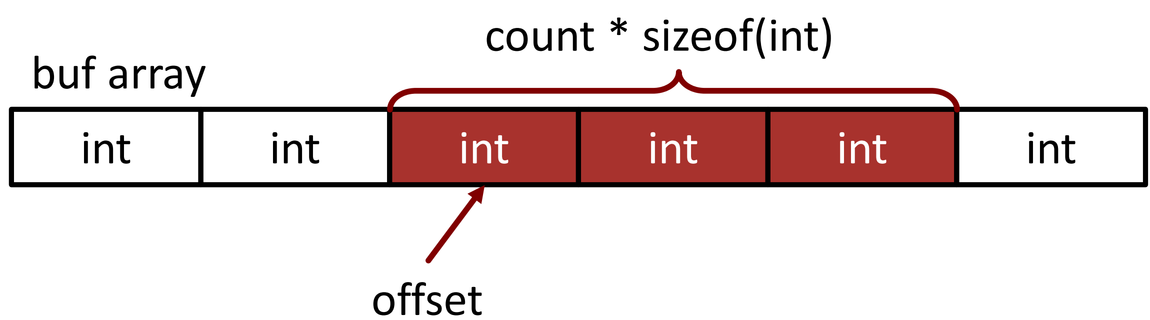
P0	P1	P2	P3

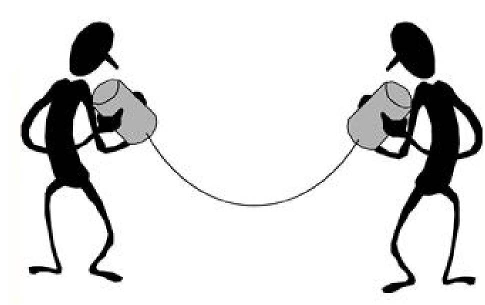
if (rank == 0)
do this
else
do that
	if (rank == 0)
	do this
	else
	do that
		if (rank == 0)
		do this
		else
		do that
			if (rank == 0)
			do this
			else
			do that

Communication

`void Comm.Send(`	communicator
`Object buf,`	pointer to data to be sent
`int offset,`	
`int count,`	number of items to be sent
`Datatype datatype,`	data type of items, must be explicitely specified
`int dest,`	destination process id
`int tag`	data id tag
`)`	

from MPJ Spec





Parallel Sort using MPI Send/Recv

Rank 0

8	23	19	67	45	35	1	24	13	30	3	5

send in O(N)

Rank 0

sort in parallel
~2 (N/2 log N/2)*

8	19	23	35	45	67

Rank 1

1	3	5	13	24	30

send in O(N)

Rank 0

8	19	23	35	45	67	1	3	5	13	24	30

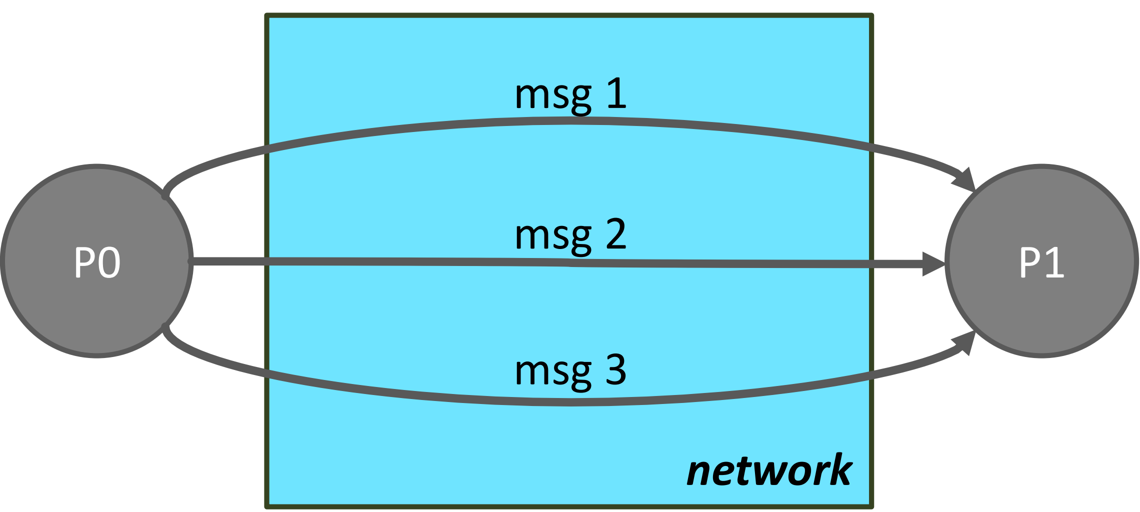
merge in O(N)

Rank 0

1	3	5	8	13	19	23	24	30	35	45	67

Message tags

- **Communicating processes may need to send several messages between each other.**
- **Message tag: differentiate between different messages being sent.**



Message matching

Send	sender communicator	sender tag	dest = r
Receive	receiver communicator	receiver tag	source = q

q

r

Receiving messages

`void Comm.Recv(`	communicator
`Object buf,`	pointer to the buffer to receive to
`int offset,`	
`int count,`	number of items to be received
`Datatype datatype,`	data type of items, must be explicitely specified
`int src,`	source process id or MPI_ANY_SOURCE
`int tag`	data id tag or MPI_ANY_TAG
`)`	

A receiver can get a message without knowing:

- the amount of data in the message,
- the sender of the message, → MPI_ANY_SOURCE
- or the tag of the message. → MPI_ANY_TAG

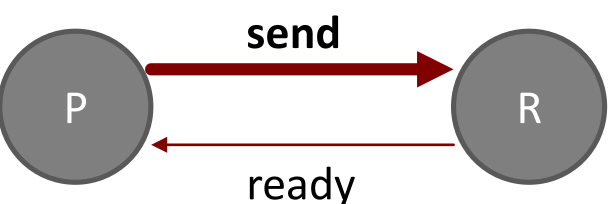
Synchronous Message Passing

Synchronous send (Ssend)

- waits until complete message can be accepted by the receiving process before completing the send

Synchronous receive (Recv)

- waits until expected message arrives



Synchronous routines can perform two actions

- transfer data
- synchronize processes

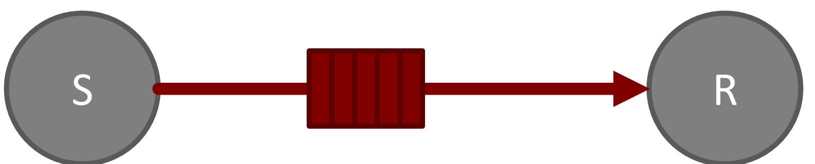
Asynchronous Message Passing

Send does not wait for actions to complete before returning

- **requires local storage for messages**

    sometimes explicit (programmer needs to care)

    sometimes implicit (transparent to the programmer)



In general

- **no synchronisation**
- **allows local progress**

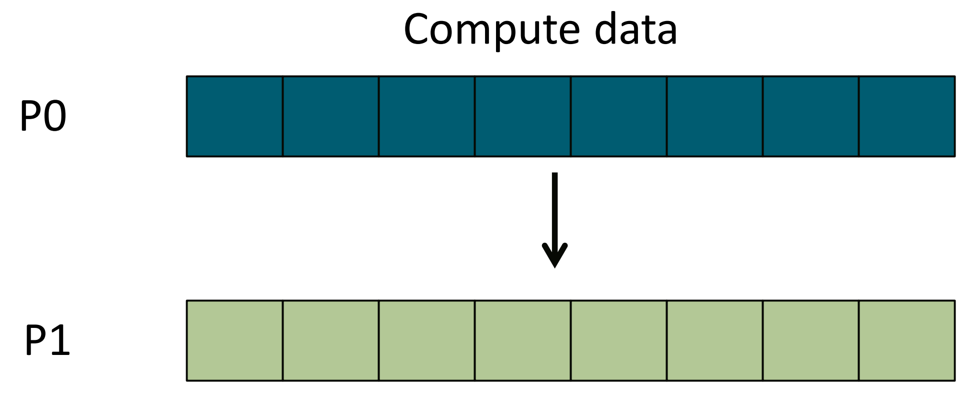
Blocking / nonblocking

Blocking: return after *local actions* are complete, though the message transfer may not have been completed

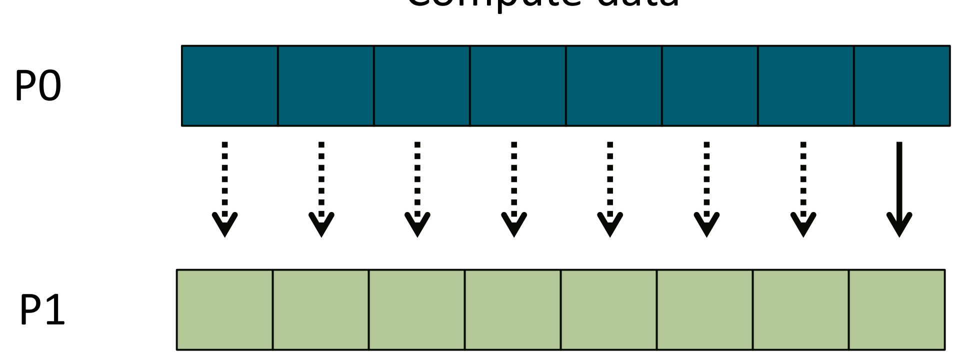
Non-blocking: return immediately

- **assumes that data storage to be used for transfer is not modified by subsequent statements until transfer complete**

A nonblocking communication example



(Full) Blocking Communication



(Streaming) Non-blocking Communication

Synchronous / asynchronous vs. blocking / nonblocking

Synchronous / Asynchronous

- about communication between sender and receiver

Blocking / Nonblocking

- about local handling of data to be sent / received

MPI Send and receive defaults

Send

- blocking,
- synchrony implementation dependent
  - depends on existence of buffering, performance considerations etc

Danger of Deadlocks.
Don't make any assumptions!

Recv

- blocking

There are a lot of different variations of this in MPI.

Sources of deadlocks

- **Send a large message from process 0 to process 1**
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- **What happens with this code?**

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This is called "unsafe" because it depends on the availability of system buffers in which to store the data sent until it can be received

Some Solutions to the "unsafe" Problem

- **Order the operations more carefully:**

Process 0	Process 1
`Send(1)`	**`Recv(0)`**
`Recv(1)`	**`Send(0)`**

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	**`Sendrecv(0)`**

More Solutions to the “unsafe” Problem

- **Supply own space as buffer for send**

Process 0	Process 1
`Bsend(1)`	**`Bsend(0)`**
`Recv(1)`	**`Recv(0)`**

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	**`Isend(0)`**
`Irecv(1)`	**`Irecv(0)`**
`Waitall`	**`Waitall`**

MPI is Simple

- **Many parallel programs can be written using just these six functions, only two of which are non-trivial:**
  - `MPI_INIT` – initialize the MPI library (must be the first routine called)
  - `MPI_COMM_SIZE` - get the size of a communicator
  - `MPI_COMM_RANK` – get the rank of the calling process in the communicator
  - `MPI_SEND` – send a message to another process
  - `MPI_RECV` – send a message to another process
  - `MPI_FINALIZE` – clean up all MPI state (must be the last MPI function called by a process)
- **For performance, however, you need to use other MPI features**

Example: compute Pi

- **The irrational number Pi has many digits**
  - And it's not clear if they're randomly distributed!
- **But they can be computed**

$$\pi \approx h \sum_{i=0}^{N-1} \frac{4}{1 + (h(i + \frac{1}{2}))^2}$$

```
for(int i=0; i<numSteps; i++) {
   double x=(i + 0.5) * h;
   sum += 4.0/(1.0 + x*x);
}
double pi=h * sum ;
```

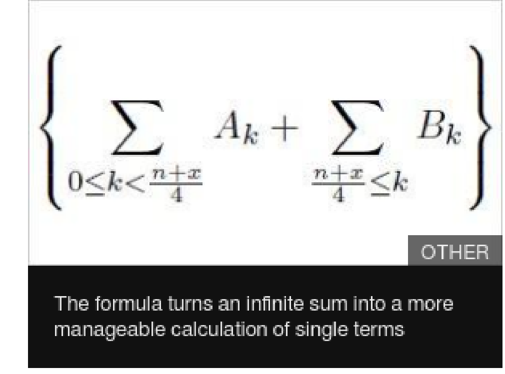
Pi record smashed as team finds two-quadrillionth digit

By Jason Palmer
Science and technology reporter, BBC News

16 September 2010 | Technology

Share

A researcher has calculated the 2,000,000,000,000,000th digit of the mathematical constant pi - and a few digits either side of it.



The formula turns an infinite sum into a more manageable calculation of single terms

Nicholas Sze, of tech firm Yahoo, said that when pi is expressed in binary, the two quadrillionth "bit" is 0.

Mr Sze used Yahoo's Hadoop cloud computing technology to more than double the previous record.

It took 23 days on 1,000 of Yahoo's computers - on a standard PC, the calculation would have taken 500 years.

Pi's parallel version

```
MPI.Init(args);
… // declare and initialize variables (sum=0 etc.)
int size = MPI.COMM_WORLD.Size();
int rank = MPI.COMM_WORLD.Rank();

for(int i=rank; i<numSteps; i=i+size) {
   double x=(i + 0.5) * h;
   sum += 4.0/(1.0 + x*x);
}

if (rank != 0) {
   double [] sendBuf = new double []{sum};
   // 1-element array containing sum
   MPI.COMM_WORLD.Send(sendBuf, 0, 1, MPI.DOUBLE, 0, 10);
}
else { // rank == 0
   double [] recvBuf = new double [1] ;
   for (int src=1 ; src<P; src++) {
      MPI.COMM_WORLD.Recv(recvBuf, 0, 1, MPI.DOUBLE, src, 10);
      sum += recvBuf[0];
   }
}
double pi = h * sum; // output pi at rank 0 only!
MPI.Finalize();
```

COLLECTIVE COMMUNICATION

Group Communication

Up to here: point-to-point communication

MPI also supports communications among groups of processors

- **not absolutely necessary for programming (but very nice!)**
- **but essential for performance**

Examples: broadcast, gather, scatter, reduce, barrier, …

Collective Computation - Reduce

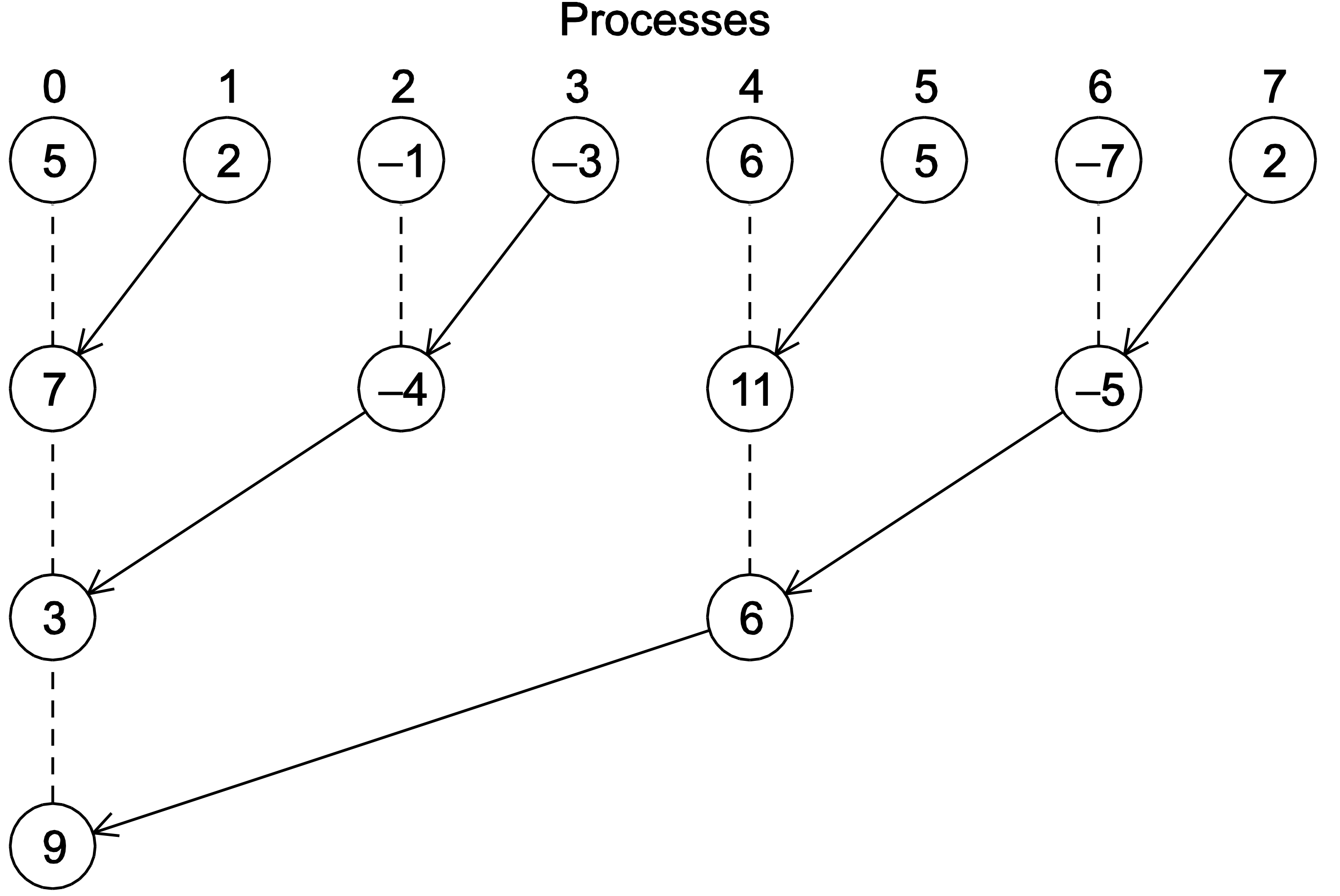
```
public void Reduce(java.lang.Object sendbuf,
            int sendoffset,
            java.lang.Object recvbuf,
            int recvoffset,
            int count,
            Datatype datatype,
            Op op,
            int root)
```

root = rank 0

		Reduce	
P0	A	→	A+B+C+D
P1	B		
P2	C		
P3	D		

		Scan	
P0	A	→	A
P1	B		A+B
P2	C		A+B+C
P3	D		A+B+C+D

Reduce implementation: a tree-structured global sum

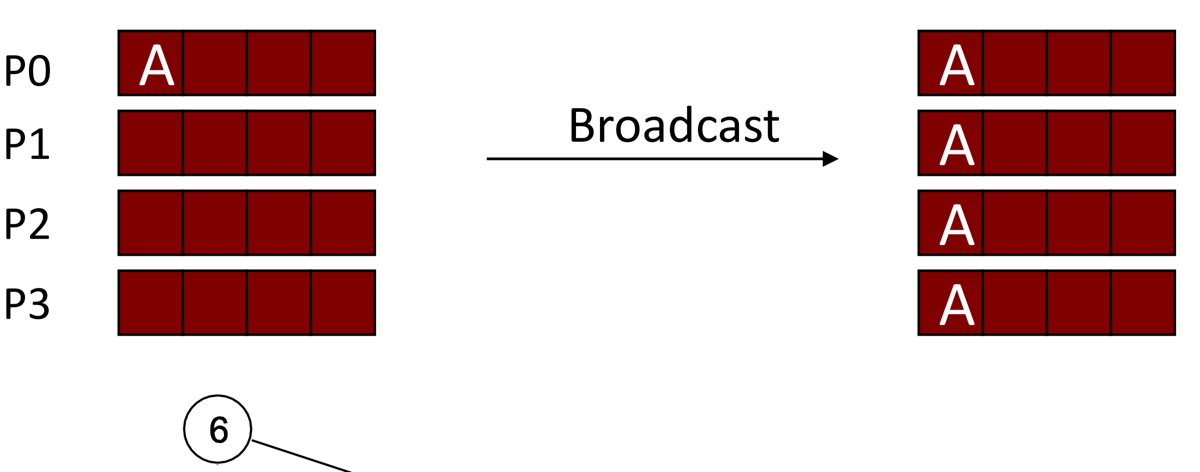


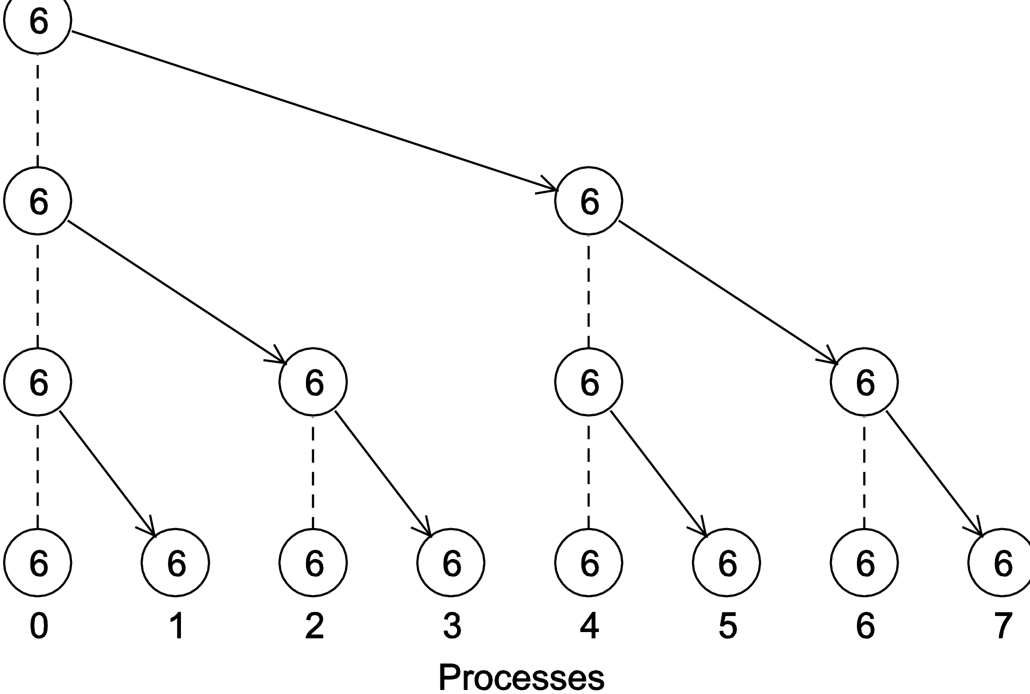
1. In the first phase:
   (a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.
   (b) Processes 0, 2, 4, and 6 add in the received values.

2. Second phase:
   (c) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   (d) Processes 0 and 4 add the received values into their new values.

3. Finally:
   (a) Process 4 sends its newest value to process 0.
   (b) Process 0 adds the received value to its newest value.

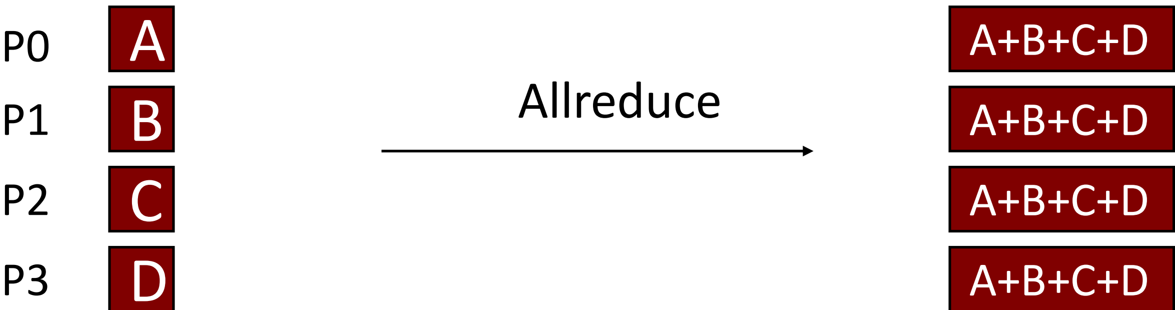
Collective Data Movement - Broadcast





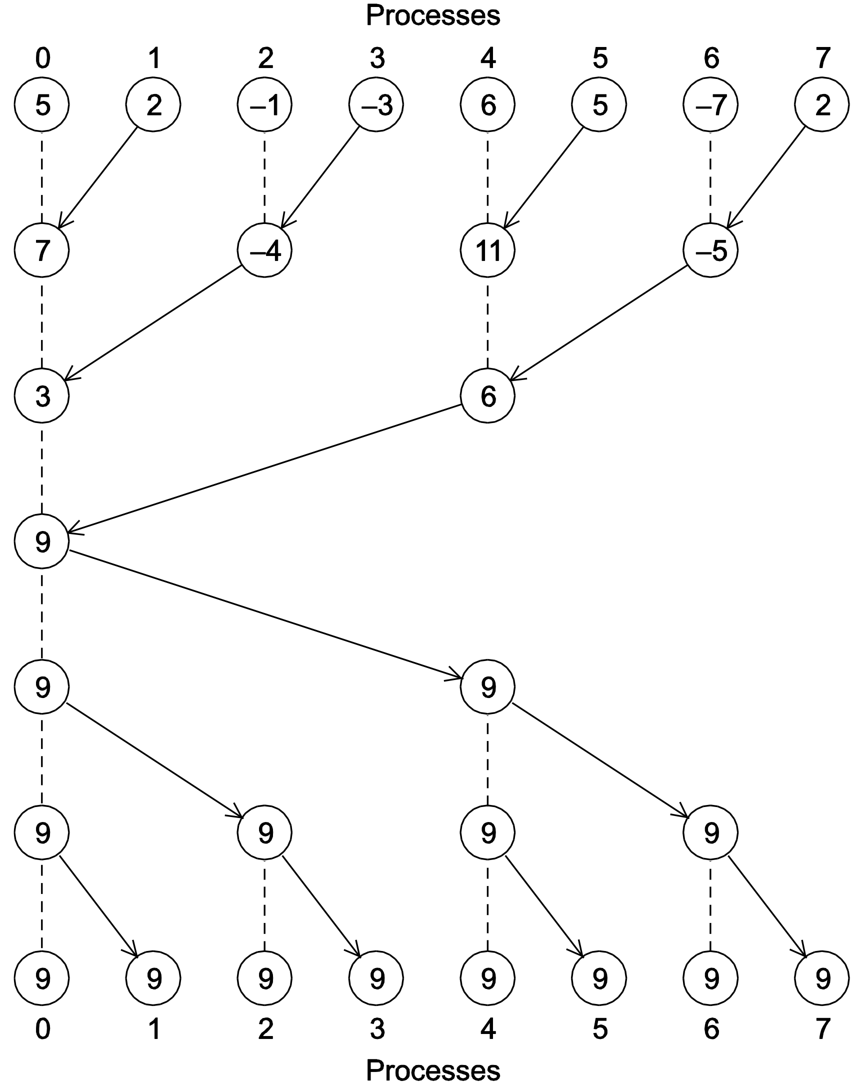
Collective Computation - Allreduce

```
public void Allreduce(java.lang.Object sendbuf,
                      int sendoffset,
                      java.lang.Object recvbuf,
                      int recvoffset,
                      int count,
                      Datatype datatype,
                      Op op)
```



Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

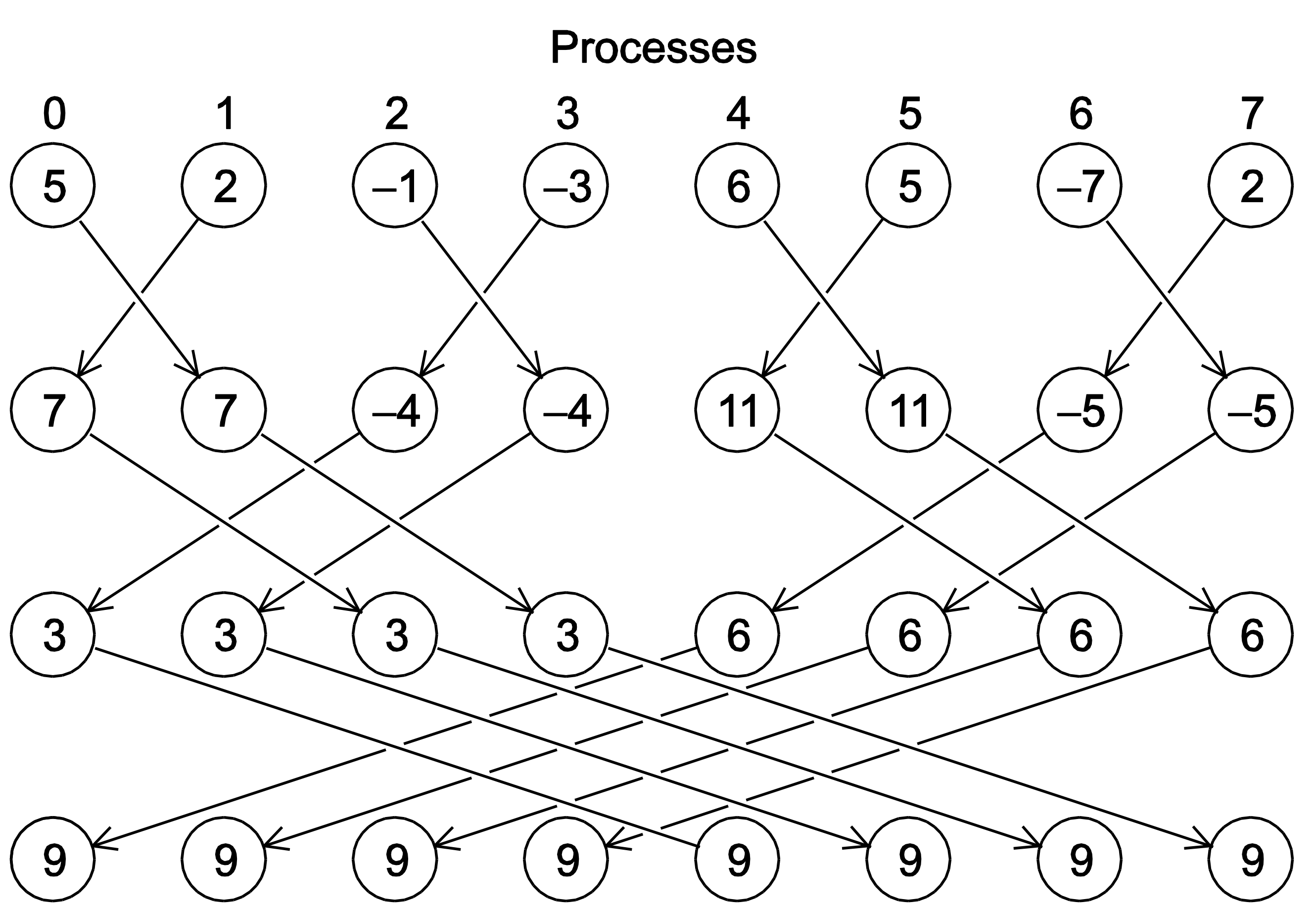
Allreduce = Reduce + Broadcast?



Q: What is the number of steps needed?

A global sum followed by distribution of the result.

Allreduce ≠ Reduce + Broadcast

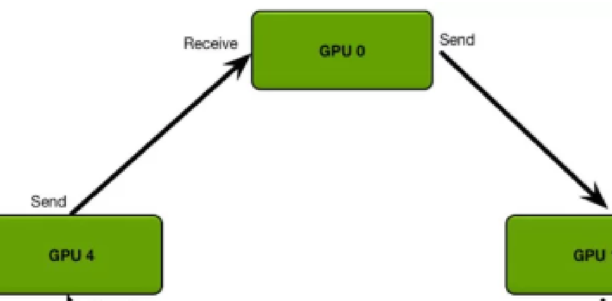


A butterfly-structured global sum.

Q: What is the number of steps needed?

Baidu's 'Ring Allreduce' Library Increases Machine Learning Efficiency Across Many GPU Nodes

by Lucian Armasu February 21, 2017 at 9:10 AM



Baidu's ring allreduce algorithm

Baidu's Silicon Valley AI Lab (SVAIL) announced an implementation of the ring allreduce algorithm for the deep learning community, which will enable significantly faster training of neural networks across GPU models.

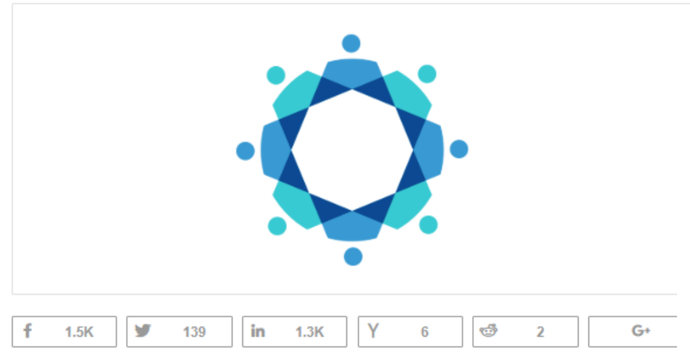
Need For Efficient Parallel Training

As neural networks have grown to include hundreds of millions or even over a billion parameters, the number of GPU nodes needed to do the training has also increased. However, the higher the number of nodes grows, the less efficient the system becomes in terms of how much computation is done by each node. Therefore, the need for algorithms that maximize the performance across the highly parallel system has also increased.

Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow

By Alex Sergeev and Mike Del Balso

October 17, 2017



1.5K | 139 | 1.3K | 6 | 2

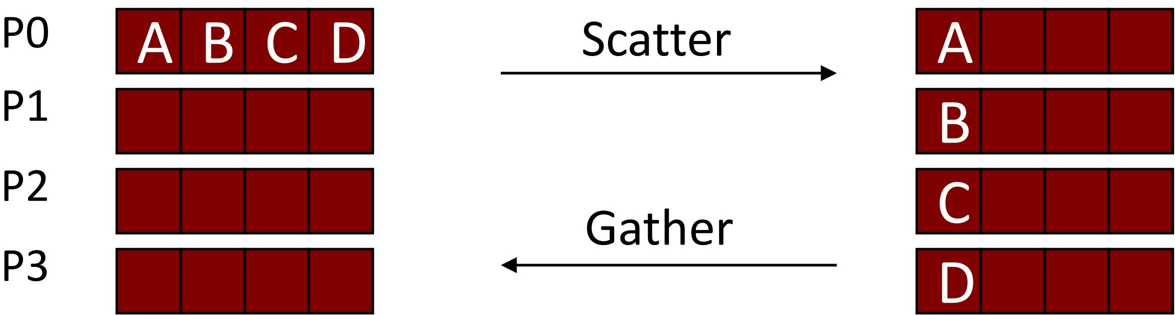
Over the past few years, advances in deep learning have driven tremendous progress in image processing, speech recognition, and forecasting. At Uber, we apply deep learning across our business; from self-driving research to trip forecasting and fraud prevention, deep learning enables our engineers and data scientists to create better experiences for our users.

Introducing Horovod

The realization that a ring-allreduce approach can improve both usability and performance motivated us to work on our own implementation to address Uber's TensorFlow needs. We adopted Baidu's draft implementation of the TensorFlow ring-allreduce algorithm and built upon it. We outline our process below:

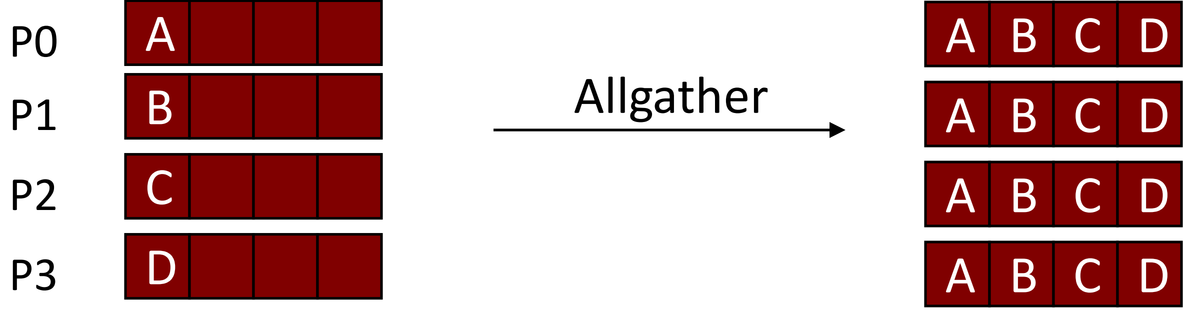
1. We converted the code into a stand-alone Python package called Horovod, named after a traditional Russian folk dance in which performers dance with linked arms in a circle, much like how distributed TensorFlow processes use Horovod to communicate with each other. At any point in time, various teams at Uber may be using different releases of TensorFlow. We wanted all teams to be able to leverage the ring-allreduce algorithm without needing to upgrade to the latest version of TensorFlow, apply patches to their versions, or even spend time building out the framework. Having a stand-alone package allowed us to cut the time required to install Horovod from about an hour to a few minutes, depending on the hardware.
2. We replaced the Baidu ring-allreduce implementation with NCCL. NCCL is NVIDIA's library for collective communication that provides a highly optimized version of ring-allreduce. NCCL 2 introduced the ability to run ring-allreduce across multiple machines, enabling us to take advantage of its many performance boosting optimizations.

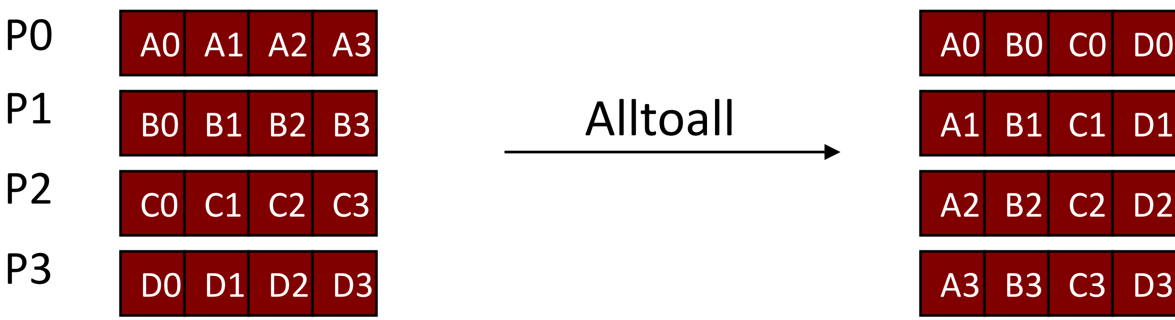
Collective Data Movement – Scatter/Gather



- **Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.**
- **Gather collects all of the components of the vector onto destination process, then destination process can process all of the components.**

More Collective Data Movement – some more (16 functions total!)



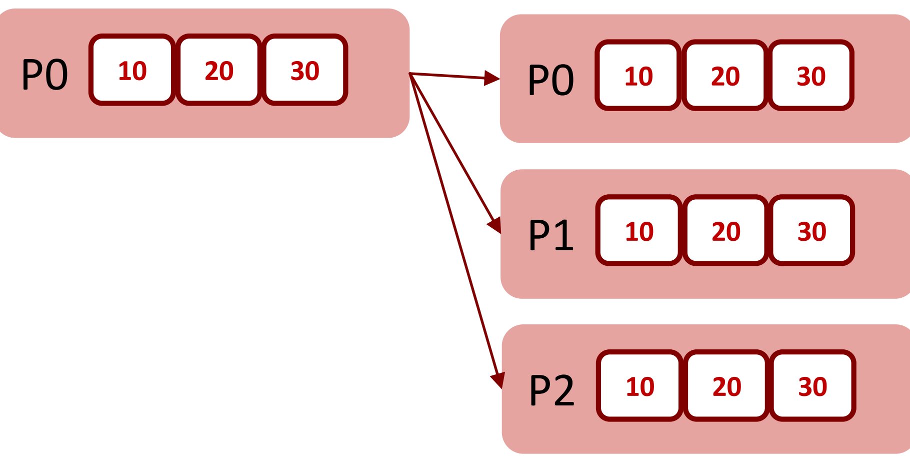


Matrix-Vector-Multiply

Compute $y = A \cdot x$, *e.g.,* $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $x = \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$ $y = \begin{bmatrix} A_{1.} \cdot x \\ A_{2.} \cdot x \\ A_{3.} \cdot x \end{bmatrix}$

Assume A and x are available only at rank 0!

1. Broadcast x

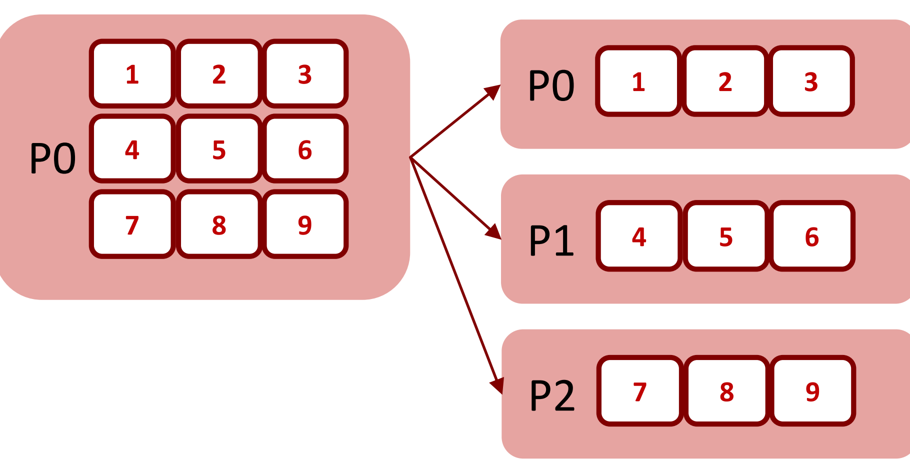


Matrix-Vector-Multiply

Compute $y = A \cdot x$, *e.g.,* $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $x = \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$ $y = \begin{bmatrix} A_{1\cdot} \cdot x \\ A_{2\cdot} \cdot x \\ A_{3\cdot} \cdot x \end{bmatrix}$

Assume A and x are available only at rank 0!

2. Scatter A



Matrix-Vector-Multiply

Compute $y = A \cdot x$, *e.g.,* $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $x = \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$ $y = \begin{bmatrix} A_{1\cdot} \cdot x \\ A_{2\cdot} \cdot x \\ A_{3\cdot} \cdot x \end{bmatrix}$

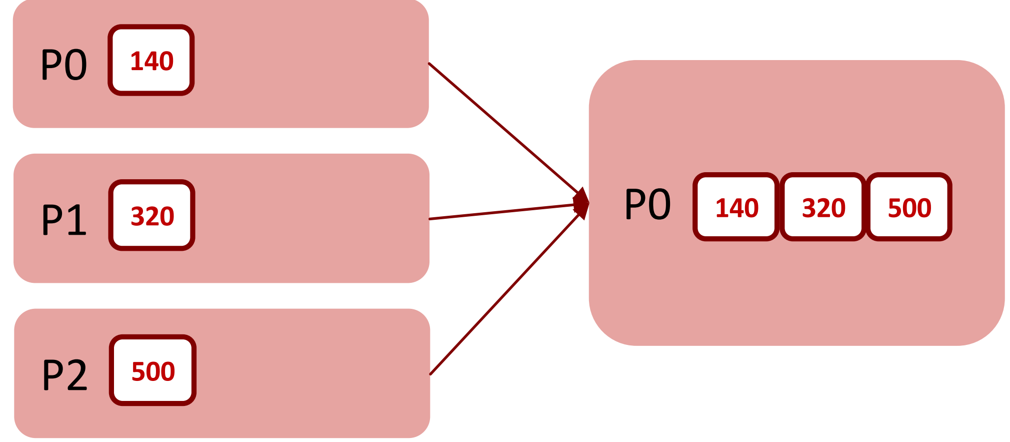
3. Compute locally

P0	1	2	3	•	10	20	30	=	140
P1	4	5	6	•	10	20	30	=	320
P2	7	8	9	•	10	20	30	=	500

Matrix-Vector-Multiply

Compute $y = A \cdot x$, *e.g.,* $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $x = \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$ $y = \begin{bmatrix} A_{1.} \cdot x \\ A_{2.} \cdot x \\ A_{3.} \cdot x \end{bmatrix}$

4. Gather result y



Iterations

Assume we want to apply the matrix-vector product iteratively

$$y_n = A\, y_{n-1}$$

Example Application:
Eigenvalue Problem for Probability Matrix, as used in Google's Pagerank algorithm.

Then each process needs the results of other processes after one step.
→Need for Gather + Broadcast in one go.
→If you're clever, you find out how to use reduce_scatter for this ☺!

Visualizing Program Behavior

MPI conclusion

- **The de-facto interface for distributed parallel computing (nearly 100% market share in HPC)**
- **Elegant and simple interface**
  - Definitely simpler than shared memory (no races, limited conflicts, avoid deadlocks with nonblocking communication)
- **We only covered the basics here, MPI-3.1 (2015) has 600+ functions**
  - More concepts:

    Derived datatypes

    Process topologies

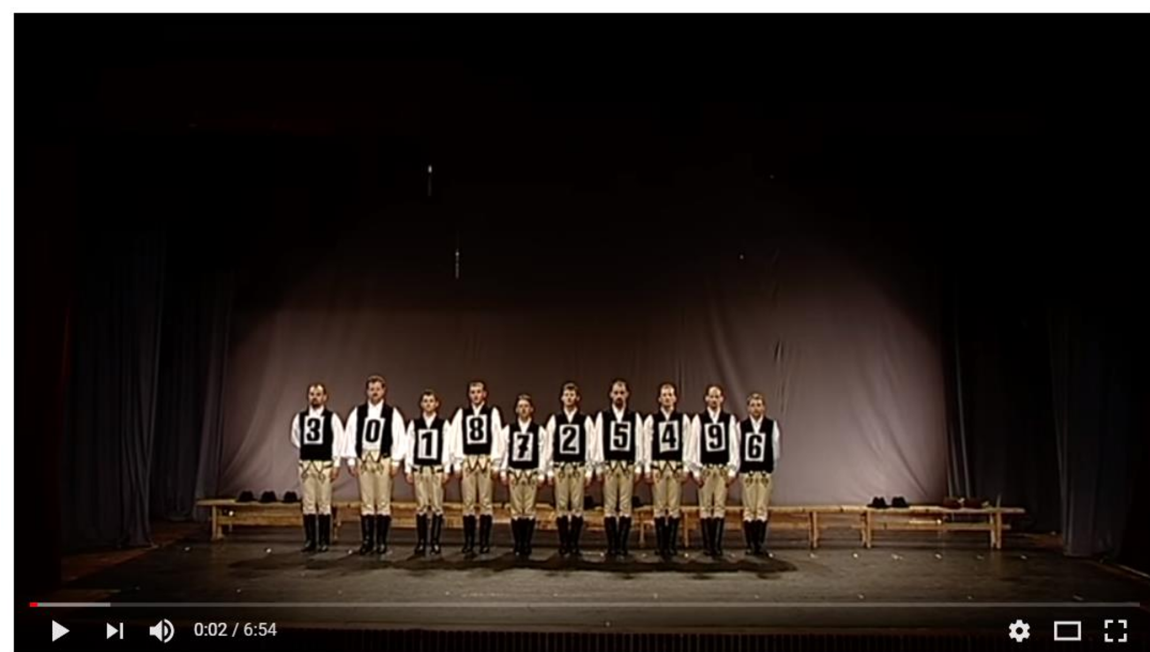
    Nonblocking and neighborhood collectives

    One-sided accesses (getting the fun of shared memory back …)

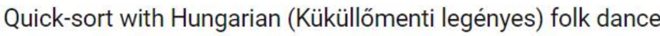
    Profiling interfaces

    …

Sorting
(one of the most fun problems in CS)



Quick-sort with Hungarian (Küküllőmenti legényes) folk dance



Insert-sort with Romanian folk dance

595,979 views 3.8K 57 SHARE

Literature

- D.E. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 5.3.4: Networks for Sorting, pp. 219–247.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 1990. ISBN 0-262-03293-7. Chapter 27: Sorting Networks, pp.704–724.

google

"chapter 27 sorting networks"

How Fast can we Sort?

Heapsort & Mergesort have $O(n \log n)$ worst-case run time

Quicksort has $O(n \log n)$ average-case run time

These bounds are all tight, actually $\Theta(n \log n)$

So maybe we can dream up another algorithm with a lower asymptotic complexity, such as $O(n)$ or $O(n \log \log n)$

This is unfortunately ***IMPOSSIBLE!***

But why?

Permutations

Assume we have n elements to sort

For simplicity, also assume none are equal (i.e., no duplicates)

How many permutations of the elements (possible orderings)?

Example, n=3

a[0]<a[1]<a[2]   a[0]<a[2]<a[1]   a[1]<a[0]<a[2]

a[1]<a[2]<a[0]   a[2]<a[0]<a[1]   a[2]<a[1]<a[0]

In general, n choices for first, n-1 for next, n-2 for next, etc. → n(n-1)(n-2)…(1) = n! possible orderings

Representing Every Comparison Sort

Algorithm must “find” the right answer among n! possible answers

Starts “knowing nothing” and gains information with each comparison

Intuition is that each comparison can, at best,
eliminate half of the remaining possibilities

Can represent this process as a decision tree

- Nodes contain “remaining possibilities”
- Edges are “answers from a comparison”
- This is not a data structure but what our proof uses to represent “the most any algorithm could know”

Decision Tree for n = 3

a < b < c, b < c < a,
a < c < b, c < a < b,
b < a < c, c < b < a

a ? b

a < b

a < b < c
a < c < b
c < a < b

a > b

b < a < c
b < c < a
c < b < a

possible orders

a < c

a < b < c
a < c < b

a > c

c < a < b

b < c

b < a < c
b < c < a

b > c

c < b < a

b < c

a < b < c

b > c

a < c < b

actual order

c < a

b < c < a

c > a

b < a < c

The leaves contain all possible orderings of a, b, c

What the decision tree tells us

Binary tree because

- Each comparison has binary outcome
- Assumes algorithm does not ask redundant questions

Because any data is possible, any algorithm needs to ask enough questions to decide among all n! answers

- Every answer is a leaf (no more questions to ask)
- So the tree must be big enough to have n! leaves
- Running any algorithm on any input will at best correspond to one root-to-leaf path in the decision tree

So no algorithm can have worst-case running time better than the height of the decision tree

Where are we

Proven: No comparison sort can have worst-case better than the height of a binary tree with n! leaves

- Turns out average-case is same asymptotically
- So how tall is a binary tree with n! leaves?

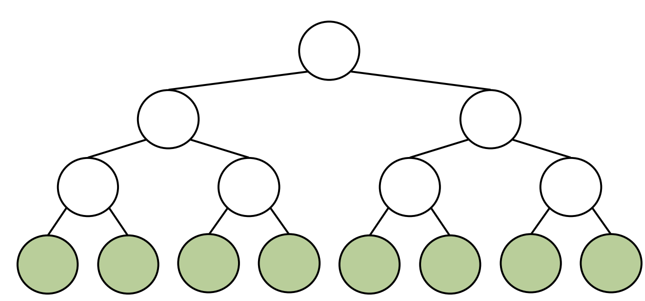
Now: Show a binary tree with n! leaves has height $\Omega(n \log n)$

- *n log n* is the lower bound, the height must be at least this
- It could be more (in other words, a comparison sorting algorithm could take longer but can not be faster)

Conclude that: (Comparison) Sorting is $\Omega(n \log n)$

Lower Bound on Height

The height of a binary tree with L leaves is at least $\log_2 L$



So the height of our decision tree, h:

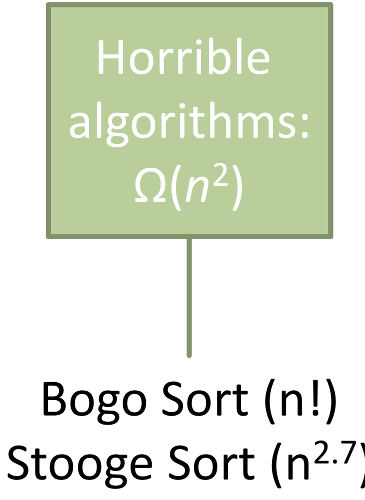
$h \geq \log_2 (n!)$	property of binary trees
$= \log_2 (n*(n-1)*(n-2)\ldots(2)(1))$	definition of factorial
$= \log_2 n + \log_2 (n-1) + \ldots + \log_2 1$	property of logarithms
$\geq \log_2 n + \log_2 (n-1) + \ldots + \log_2 (n/2)$	keep first n/2 terms
$\geq (n/2) \log_2 (n/2)$	each of the n/2 terms left is $\geq \log_2 (n/2)$
$\geq (n/2)(\log_2 n - \log_2 2)$	property of logarithms
$\geq (1/2) n \log_2 n - (1/2) n$	arithmetic
"=" $\Omega(n \log n)$	

Breaking the lower bound on sorting

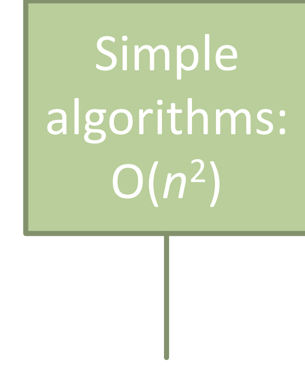
Assume 32/64-bit Integer:

2^{32} = 4294967296
13! = 6227020800

2^{64} = 18446744073709551616
21! = 51090942171709440000



Bogo Sort (n!)
Stooge Sort ($n^{2.7}$)

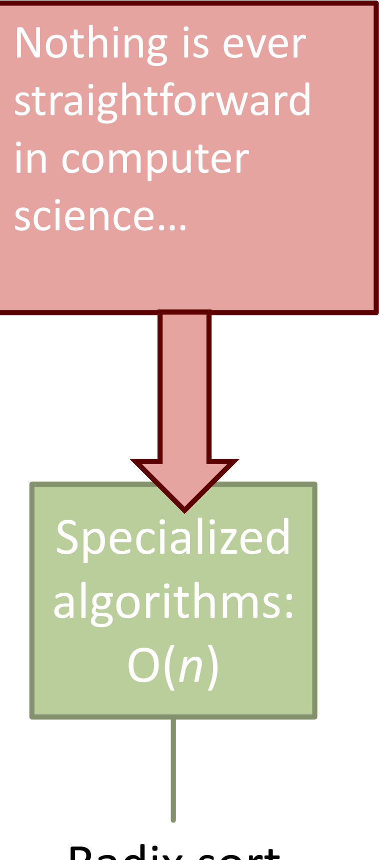


Insertion sort
Selection sort
Bubble Sort
Shell sort
…

Fancier algorithms: $O(n \log n)$

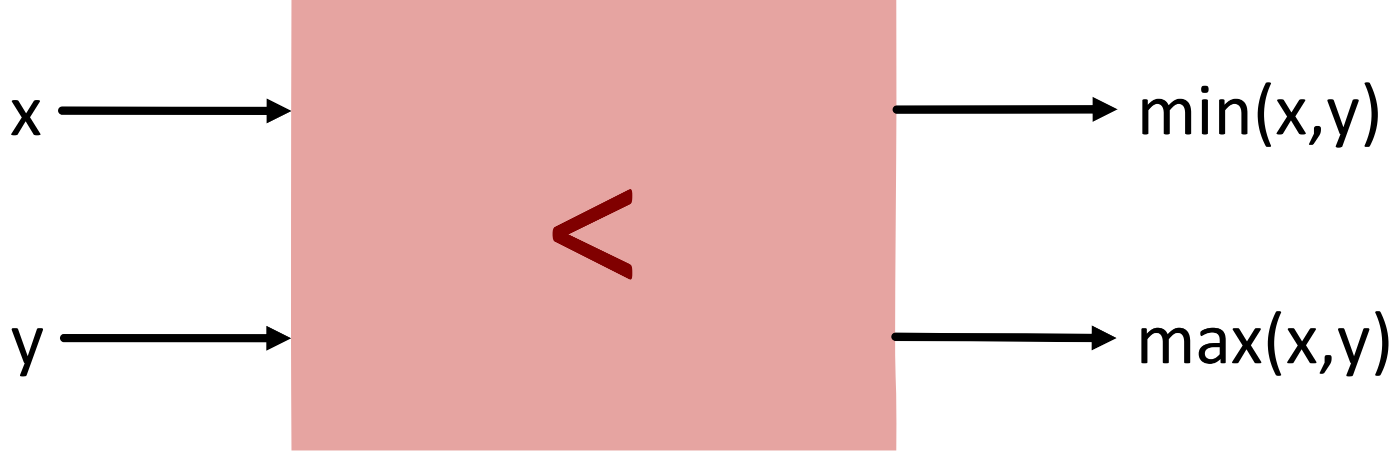
Heap sort
Merge sort
Quick sort (avg)
…

Comparison lower bound: $\Omega(n \log n)$

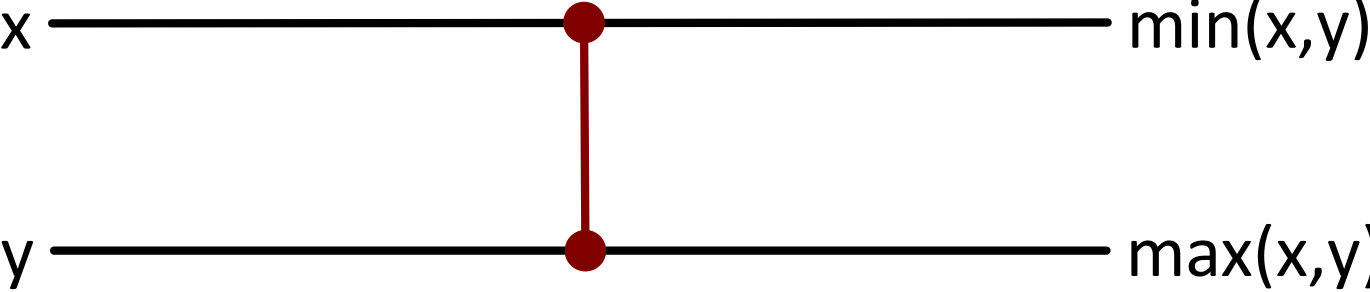


SORTING NETWORKS

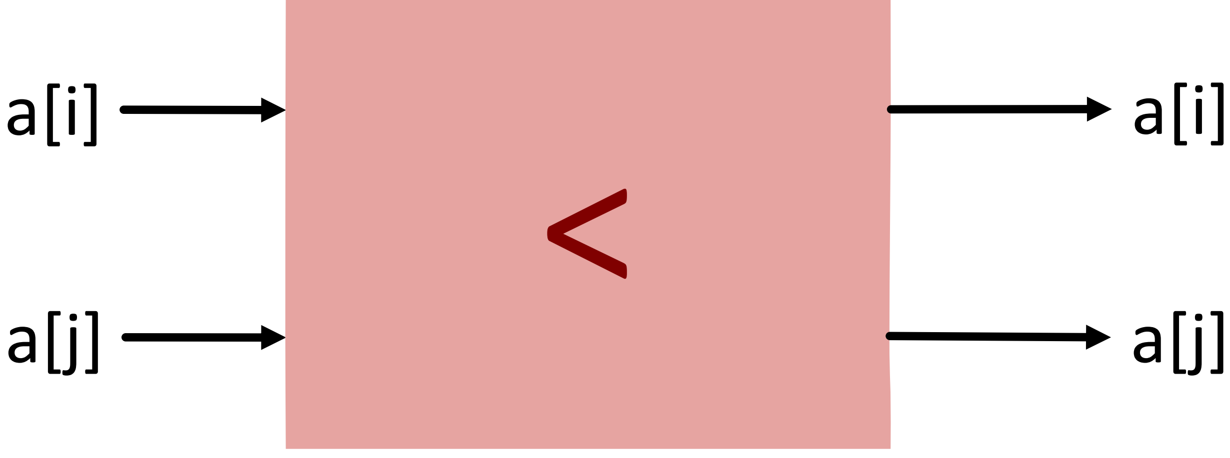
Comparator



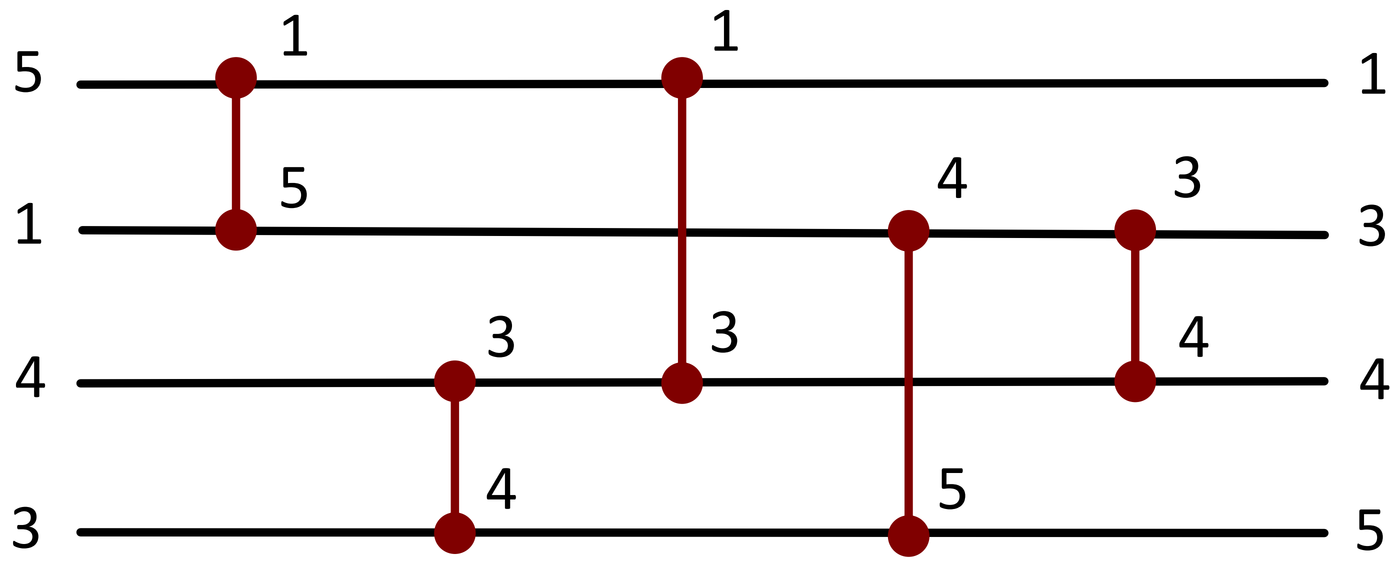
shorter notation:



```
void compare(int[] a, int i, int j, boolean dir) {
   if (dir==(a[i]>a[j])){
      int t=a[i];
      a[i]=a[j];
      a[j]=t;
   }
}
```

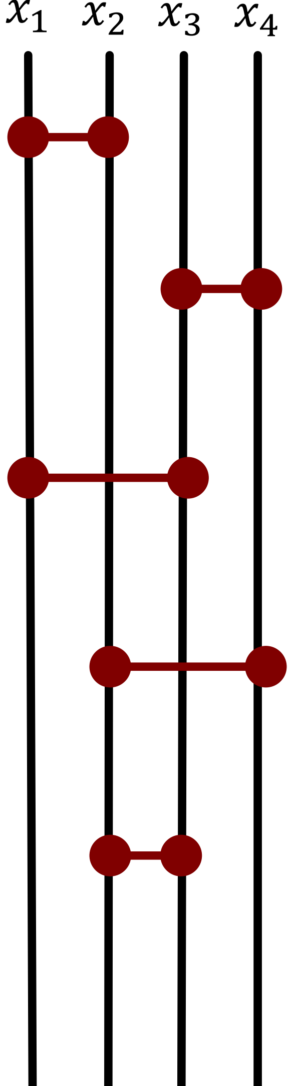


Sorting Networks

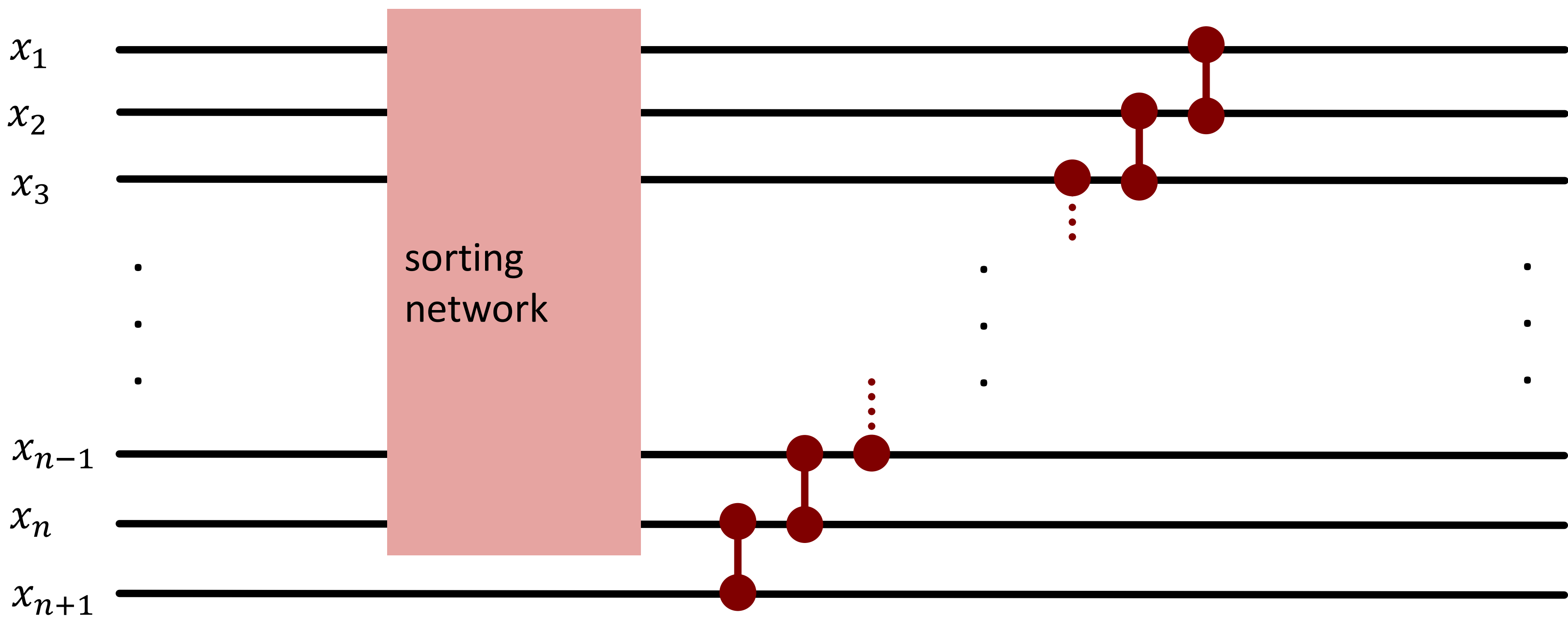


Sorting networks are data-oblivious (and redundant)

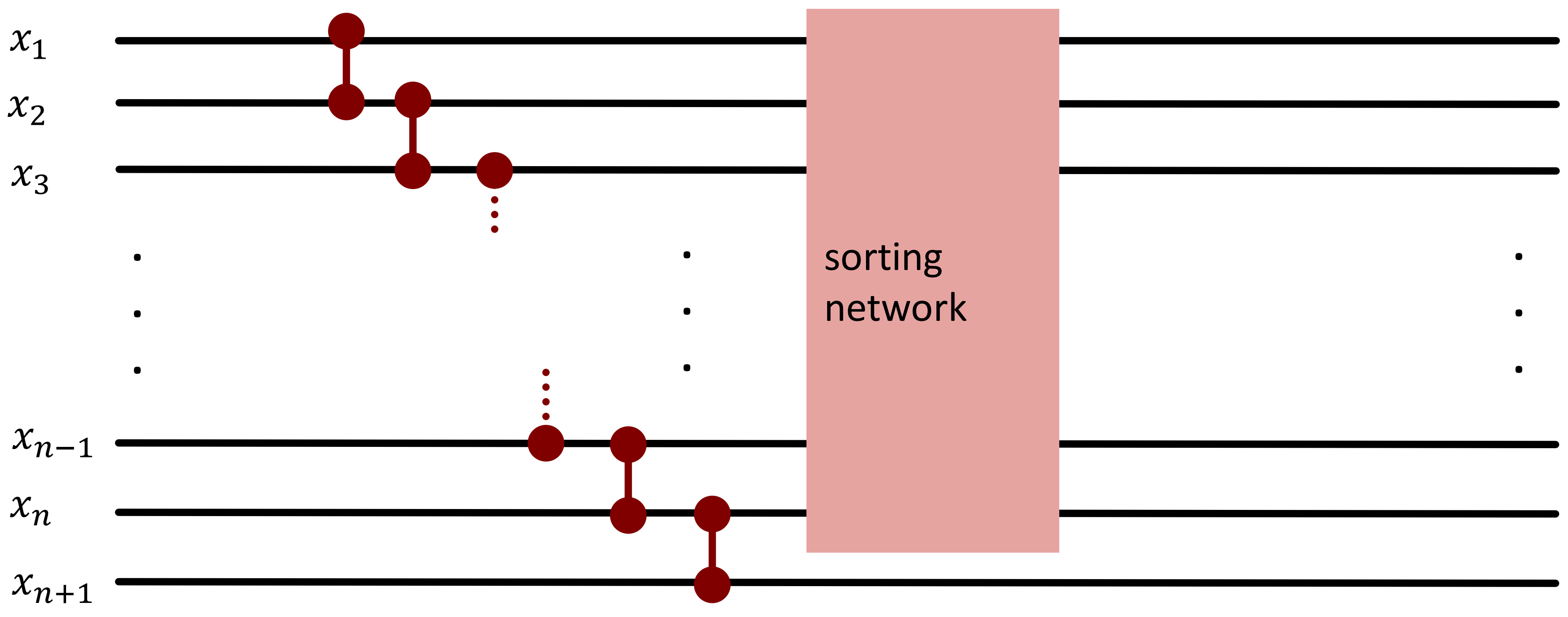
Data-oblivious comparison tree



Recursive Construction : Insertion



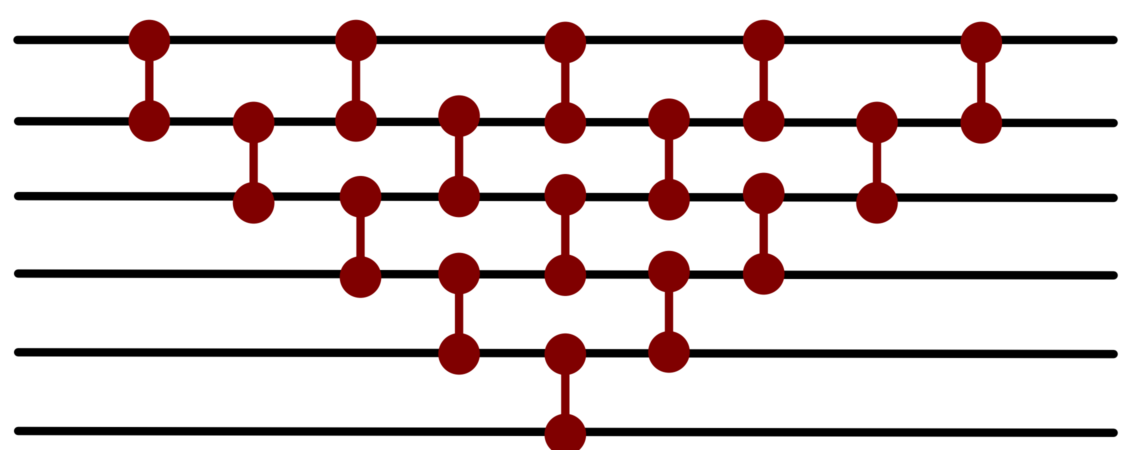
Recursive Construction: Selection



Applied recursively..

insertion sort

bubble sort

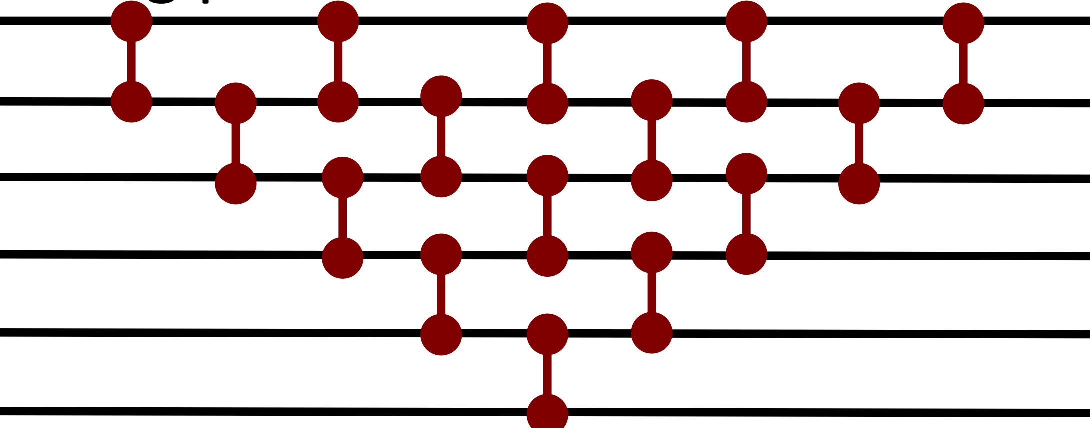


with parallelism: insertion sort = bubble sort !

Question

How many steps does a computer with infinite number of processors (comparators) require in order to sort using parallel bubble sort?

Answer: 2n – 3
Can this be improved ?



How many comparisons ?

Answer: (n-1) n/2

How many comparators are required (at a time)?

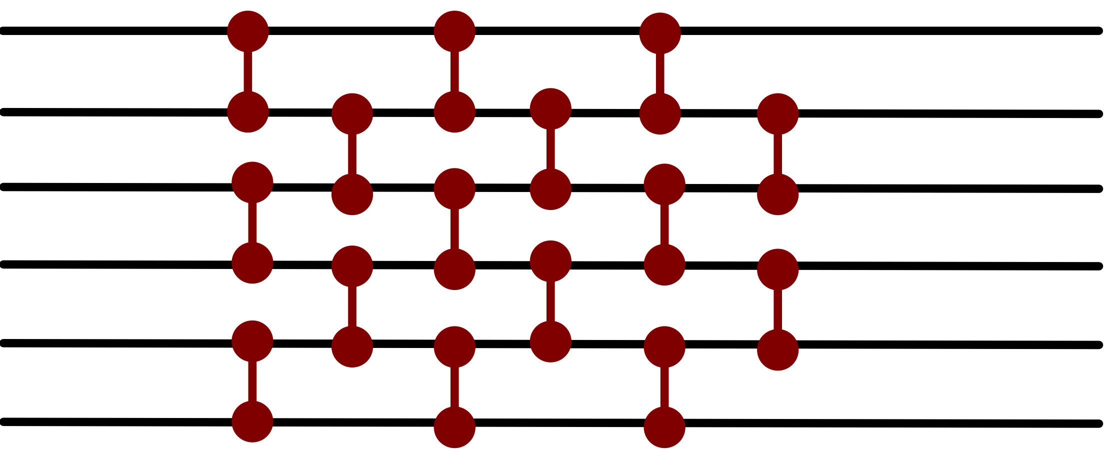
Answer: n/2
Reusable comparators: n-1

Improving parallel Bubble Sort

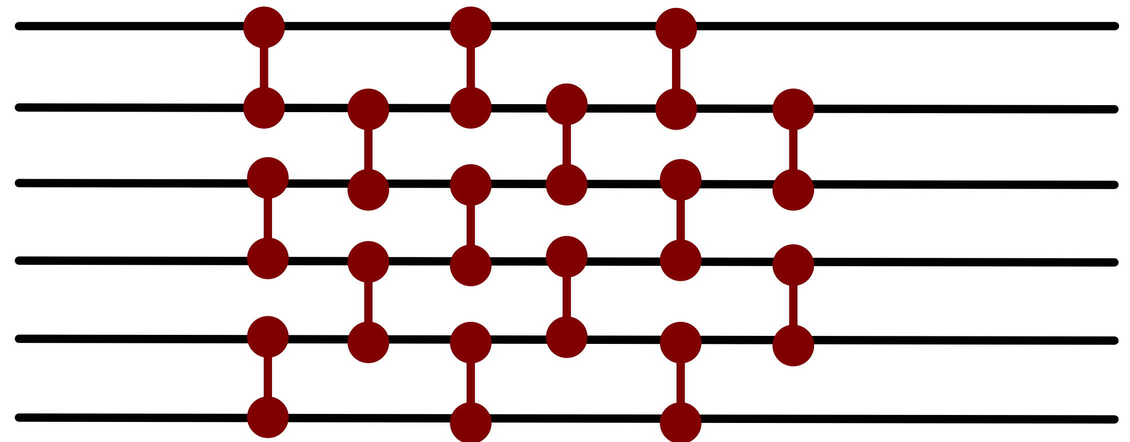
Odd-Even Transposition Sort:

0	9	↔	8		2	↔	7		3	↔	1		5	↔	6		4
1	8		9	↔	2		7	↔	1		3	↔	5		6	↔	4
2	8	↔	2		9	↔	1		7	↔	3		5	↔	4		6
3	2		8	↔	1		9	↔	3		7	↔	4		5	↔	6
4	2	↔	1		8	↔	3		9	↔	4		7	↔	5		6
5	1		2	↔	3		8	↔	4		9	↔	5		7	↔	6
6	1	↔	2		3	↔	4		8	↔	5		9	↔	6		7
7	1		2	↔	3		4	↔	5		8	↔	6		9	↔	7
8	1	↔	2		3	↔	4		5	↔	6		8	↔	7		9
	1		2		3		4		5		6		7		8		9

```
void oddEvenTranspositionSort(int[] a, boolean dir) {
   int n = a.length;
   for (int i = 0; i<n; ++i) {
      for (int j = i % 2; j+1<n; j+=2)
         compare(a,j,j+1,dir);
   }
}
```



Improvement?



Same number of comparators (at a time)

Same number of comparisons

But less parallel steps (depth): n

> In a massively parallel setup, bubble sort is thus not too bad.
>
> But it can go better...

How to get to a sorting network?

- **It's complicated ☺**
  - In fact, some structures are clear but there is a lot still to be discovered!
- **For example:**
  - what is the minimum number of comparators?
  - What is the minimum size?
  - Tradeoffs between these two?

Source: wikipedia

Optimal sorting networks [edit]

For small, fixed numbers of inputs n, *optimal* sorting networks can be constructed, with either minimal depth (for maximally parallel execution) or minimal size (number of comparators). These networks can be used to increase the performance of larger sorting networks resulting from the recursive constructions of, e.g., Batcher, by halting the recursion early and inserting optimal nets as base cases.[9] The following table summarizes the known optimality results:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Depth[10]	0	1	3	3	5	5	6	6	7	7	8	8	9	9	9	9	10
Size, upper bound[11]	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60	71
Size, lower bound (if different)[11]											33	37	41	45	49	53	58

The first sixteen depth-optimal networks are listed in Knuth's *Art of Computer Programming*,[1] and have been since the 1973 edition; however, while the optimality of the first eight was established by Floyd and Knuth in the 1960s, this property wasn't proven for the final six until 2014[12] (the cases nine and ten having been decided in 1991[9]).

For one to ten inputs, minimal (i.e. size-optimal) sorting networks are known, and for higher values, lower bounds on their sizes $S(n)$ can be derived inductively using a lemma due to Van Voorhis: $S(n+1) \geq S(n) + \lceil \log_2(n) \rceil$. All ten optimal networks have been known since 1969, with the first eight again being known as optimal since the work of Floyd and Knuth, but optimality of the cases $n = 9$ and $n = 10$ took until 2014 to be resolved.[11]

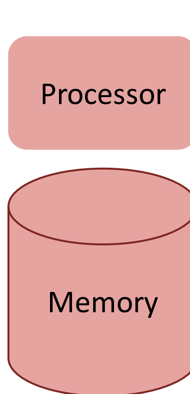
Interlude: Machine Models

RAM : Random Access Machine

- Unbounded local memory
- Each memory has unbounded capacity
- Simple operations: data, comparison, branches
- All operations take unit time

Time complexity: number of steps executed

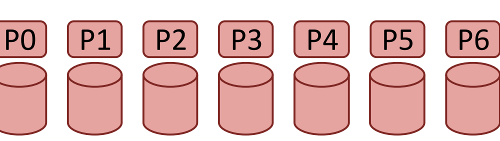
Space complexity: (maximum) number of memory cells used

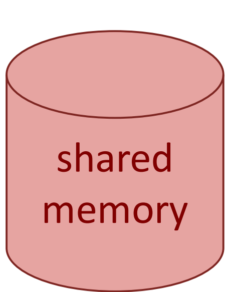


Machine Models

PRAM : *Parallel* Random Access Machine

- Abstract machine for designing algorithms applicable for parallel computers
- Unbounded collection of RAM processors $P_0, P_1, \ldots$
- Each processor has unbounded registers
- Unbounded shared memory
- All processors can access all memory in unit time
- All communication via shared memory





Shared Memory Access Model

ER: processors can simultaneously read from distinct memory locations

EW: processors can simultaneously write to distinct memory locations

CR: processors can simultanously read from any memory location

CW: processors can simultaneously write to any memory location

Specification of the machine model as one of EREW, CREW, CRCW

Example: Why the machine model can be important

Find maximum of n elements in an array A

Assume $O(n^2)$ processors and the **CRCW model**

For all $i \in \{0,1,\ldots,n-1\}$ in parallel do

$$P_{i0}: m_i \leftarrow true$$

For all $i,j \in \{0,1,\ldots,n-1\}, i \neq j$ in parallel do

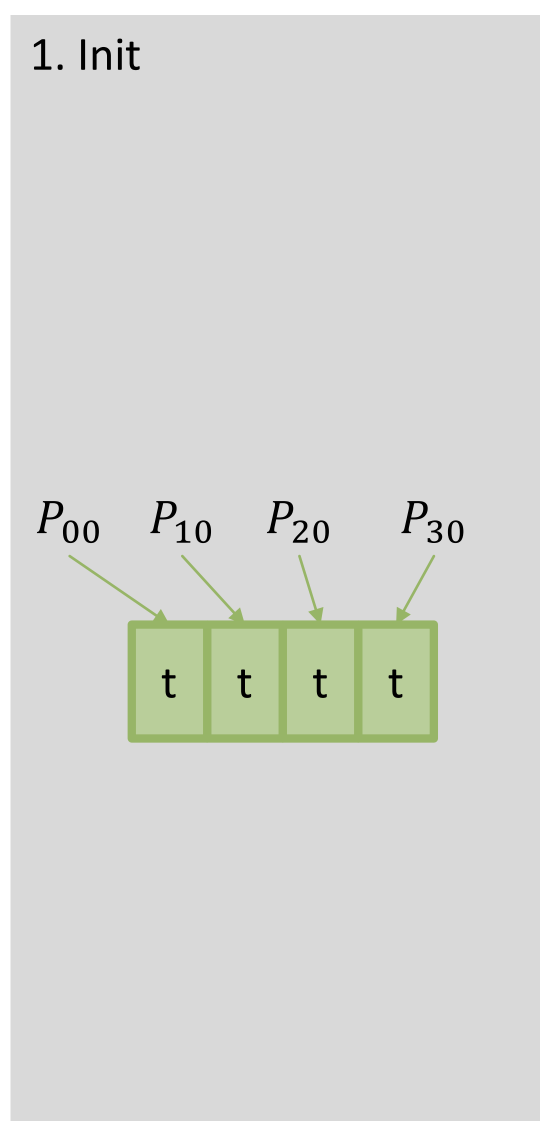
$$P_{ij}: if\ A_i < A_j\ then\ m_i \leftarrow false$$

For all $i \in \{0,1,\ldots,n-1\}$ in parallel do

$$P_{i0}: if\ m_i = true\ \ then\ max \leftarrow A_i$$

O(1) time complexity!

Illustration



1 4 2 9

P_{01} P_{02} P_{03}

P_{10} P_{20} P_{30}

P_{12} P_{13}

P_{21} P_{31}

P_{23}

P_{32}

f f f t

concurrent writes!

P_{00} P_{10} P_{20} P_{30}

max

CREW

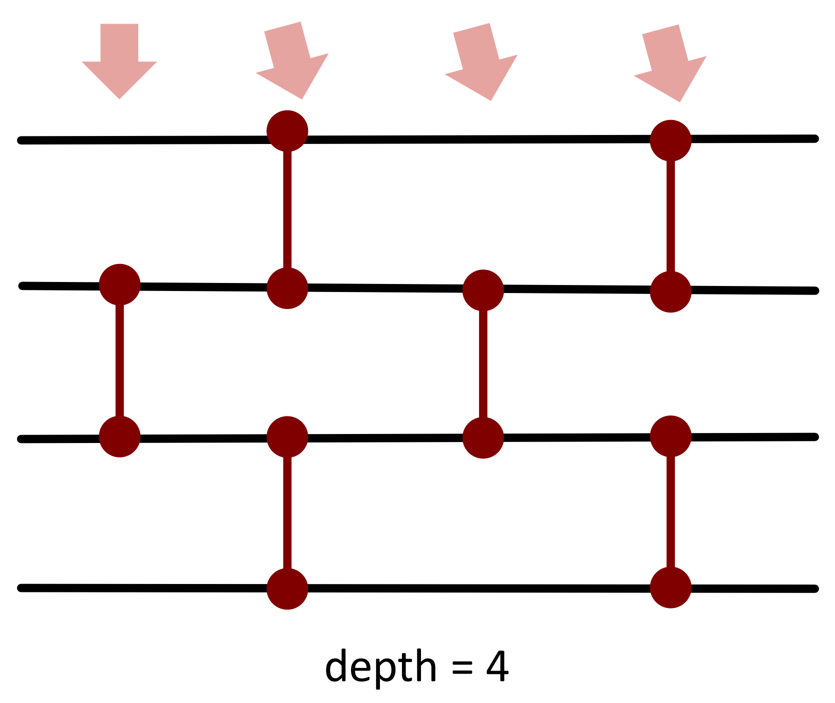
Q: How many steps does max-find require with CREW?

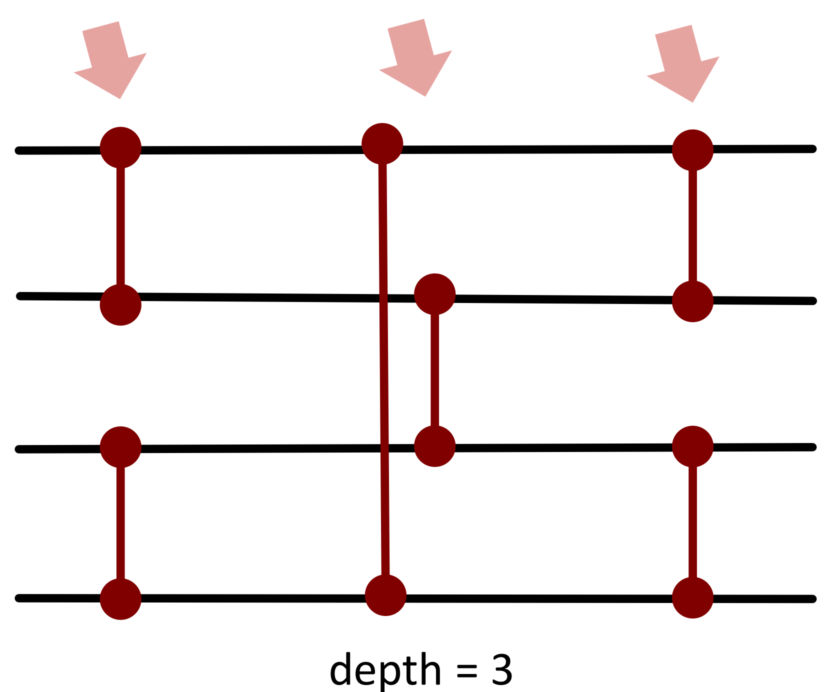
Using CREW only two values can be merged into a single value by one processor at a time step: number of values that need to be merged can be halved at each step → Requires $\Omega(\log n)$ steps

There are a lot of interesting theoretical results for PRAM machine models (e.g., CRCW simulatable with EREW) and for PRAM based algorithms (e.g., cost optimality / time optimality proofs etc). We will not go into more details here.

In the following we assume a CREW PRAM model -- and receive in retrospect a justification for the results stated above on parallel bubble sorting.

Parallel sorting





Prove that the two networks above sort four numbers. Easy?

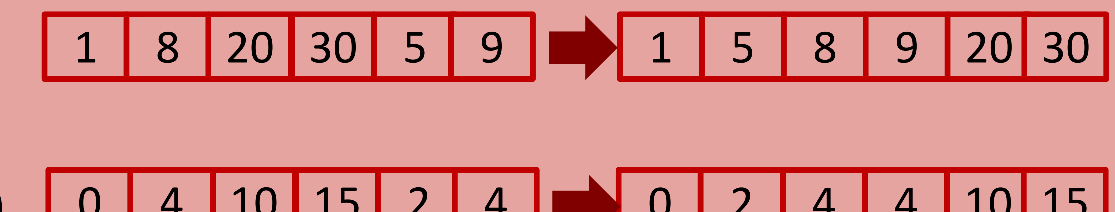
Zero-one-principle

Theorem: If a network with n input lines sorts all 2^n sequences of 0s and 1s into non-decreasing order, it will sort any arbitrary sequence of n numbers in nondecreasing order.

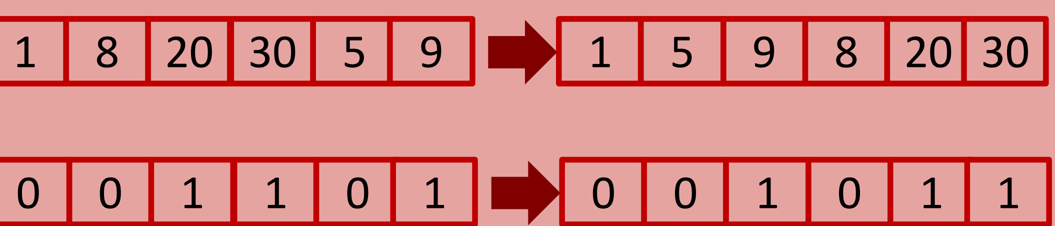
Proof

Argue: If x is sorted by a network N then also any monotonic function of x.

e.g., floor(x/2)



Show: If x is **not** sorted by the network, **then** there is a monotonic function f that maps x to 0s and 1s and **f(x) is not sorted** by the network



$$x \text{ not sorted by } N \;\Rightarrow \text{ there is an } f(x) \in \{0,1\}^n \text{ not sorted by N}$$

$$\Leftrightarrow$$

$$f \text{ sorted by N for all } f \in \{0,1\}^n \Rightarrow x \text{ sorted by N for all x}$$

Proof

Assume a monotonic function $f(x)$ with $f(x) \leq f(y)$ whenever $x \leq y$ and a network N that sorts. Let N transform $(x_1, x_2, \ldots, x_n)$ into $(y_1, y_2, \ldots, y_n)$, then it also transforms $(f(x_1), f(x_2), \ldots, f(x_n))$ into $(f(y_1), f(y_2), \ldots, f(y_n))$.

All comparators must act in the same way for the $f(x_i)$ as they do for the x_i

Assume $y_i > y_{i+1}$ for some i, then consider the monotonic function

$$f(x) = \begin{cases} 0, if\ x < y_i \\ 1, if\ x \geq y_i \end{cases}$$

→N converts

$(f(x_1), f(x_2), \ldots, f(x_n))$ **into** $(f(y_1), f(y_2), \ldots f(y_i), f(y_{i+1}), \ldots, f(y_n))$

1

0

Bitonic Sort

Bitonic (Merge) Sort is a parallel algorithm for sorting

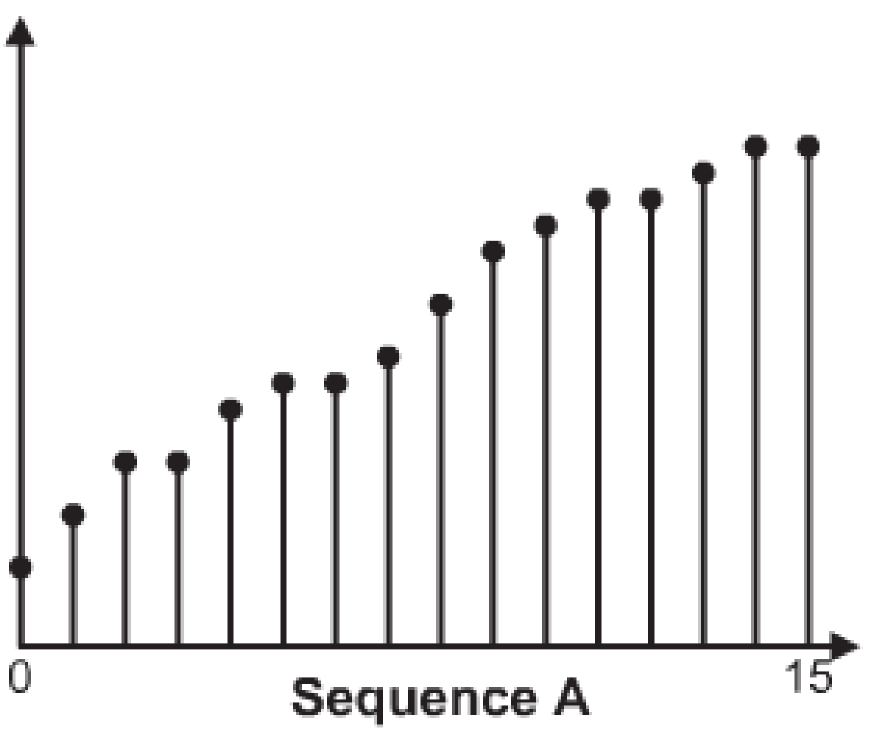
If enough processors are available, bitonic sort breaks the lower bound on sorting for comparison sort algorithm

Time complexity of $O(n\ \log^2 n)$ (sequential execution)

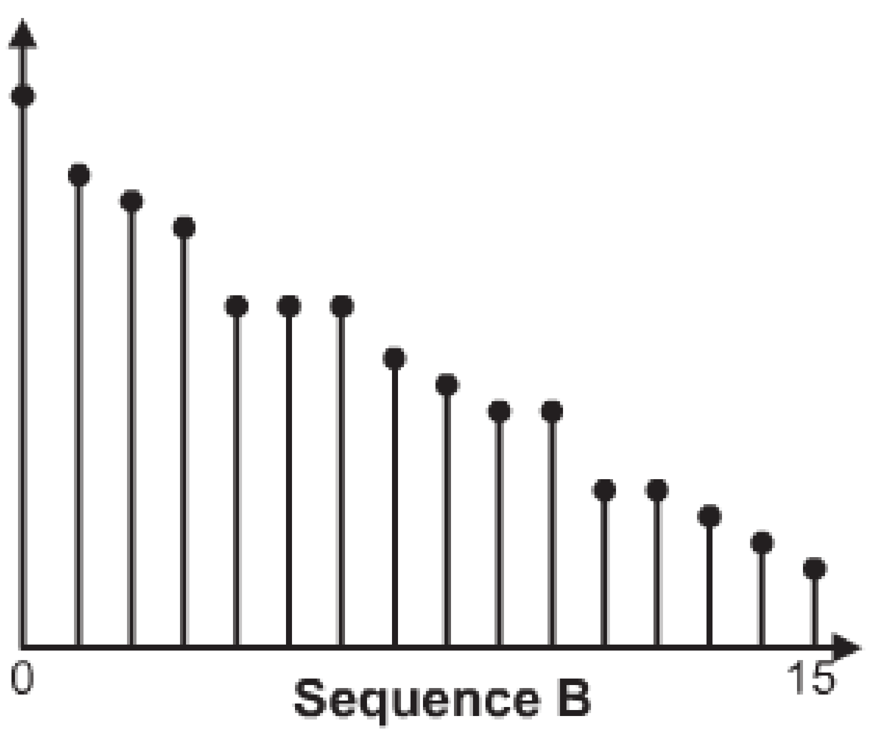
Time complexity of $O(\log^2 n)$ (parallel time)

Worst = Average = Best case

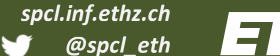
What is a Bitonic Sequence?



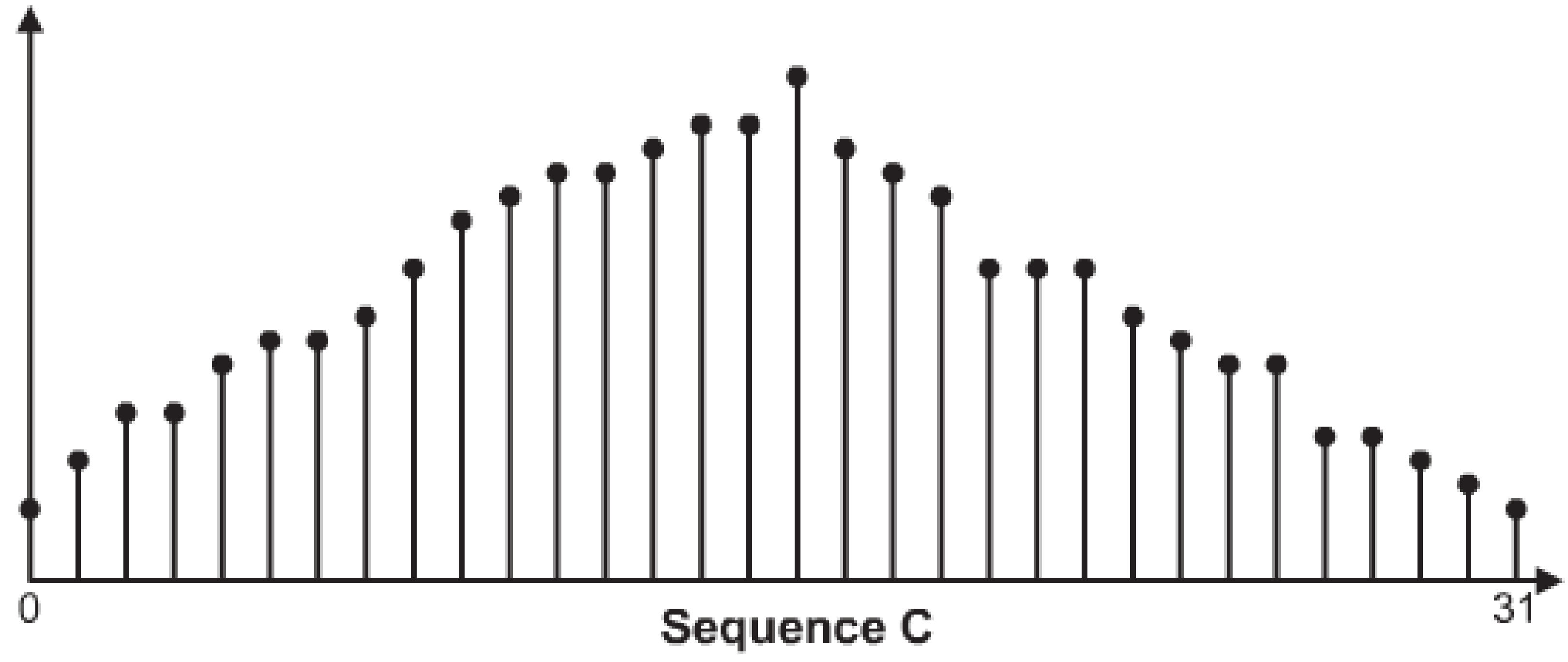
Monotonic ascending sequence



Monotonic descending sequence



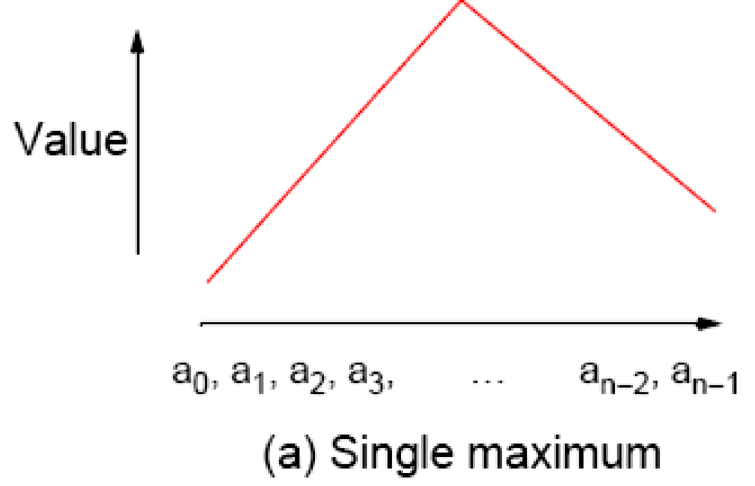
Bitonic Sets



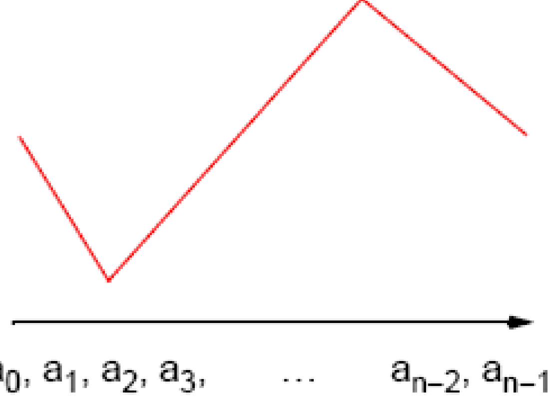
A *bitonic set* is defined as a set in which the sign of the gradient changes once at most.

So that $x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$**, for some** $k, 0 \leq k < n$

Bitonic sets - Wraparound



(a) Single maximum



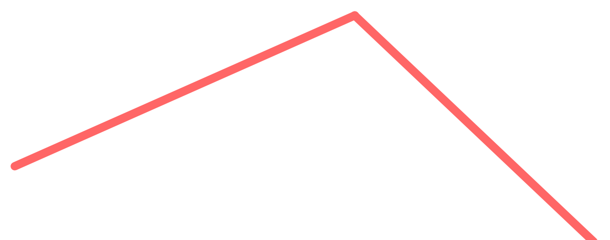
(b) Single maximum and single minimum

A *bitonic sequence* is defined as a list with no more than one **Local maximum** and no more than one **Local minimum**.

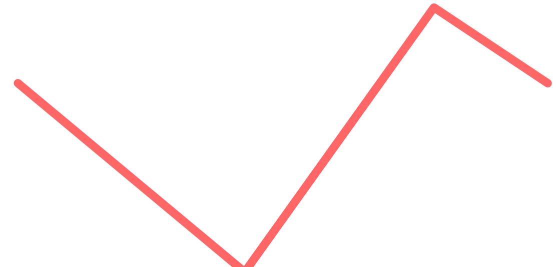
Bitonic (again)

Sequence $(x_1, x_2, \dots, x_n)$ is bitonic, if it can be circularly shifted such that it is first monotonically increasing and then monontonically decreasing.

(1, 2, 3, 4, 5, 3, 1, 0)



(4, 3, 2, 1, 2, 4, 6, 5)



Bitonic 0-1 Sequences

$$0^i 1^j 0^k$$

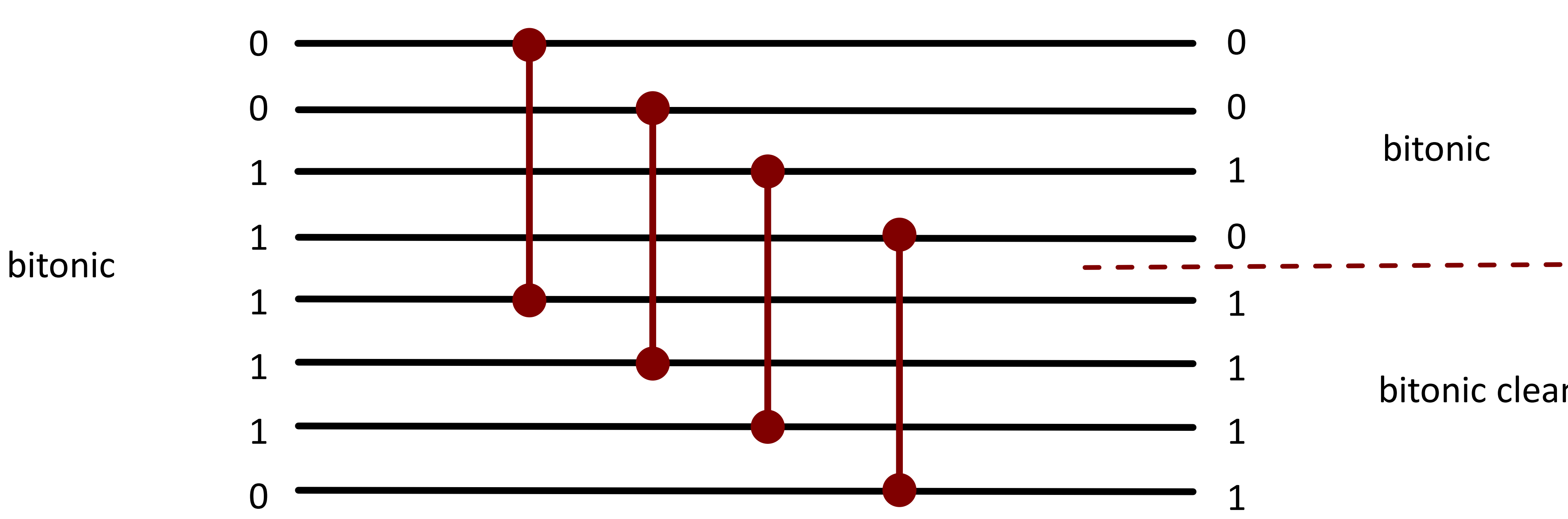
$$1^i 0^j 1^k$$

Properties

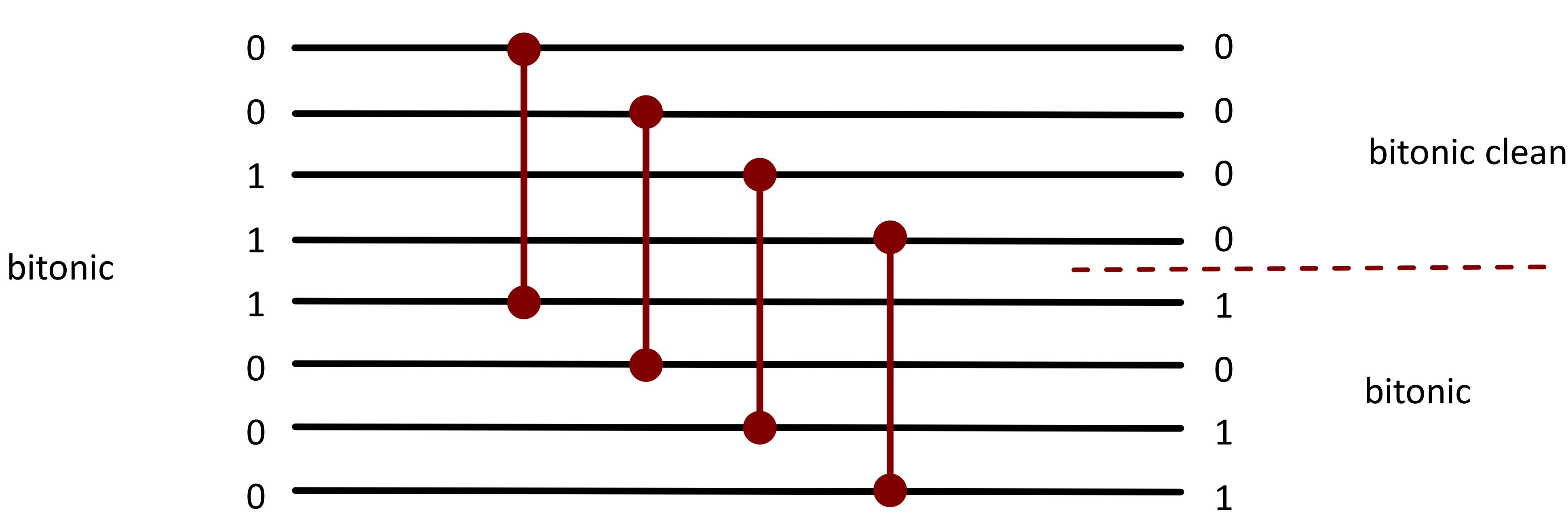
If $(x_1, x_2, \ldots, x_n)$ is monotonically increasing (decreasing) and then monotonically decreasing (increasing), then it is bitonic

If $(x_1, x_2, \ldots, x_n)$ is bitonic, then $(x_1, x_2, \ldots, x_n)^R := (x_n, x_{n-1}, \ldots, x_1)$ is also bitonic

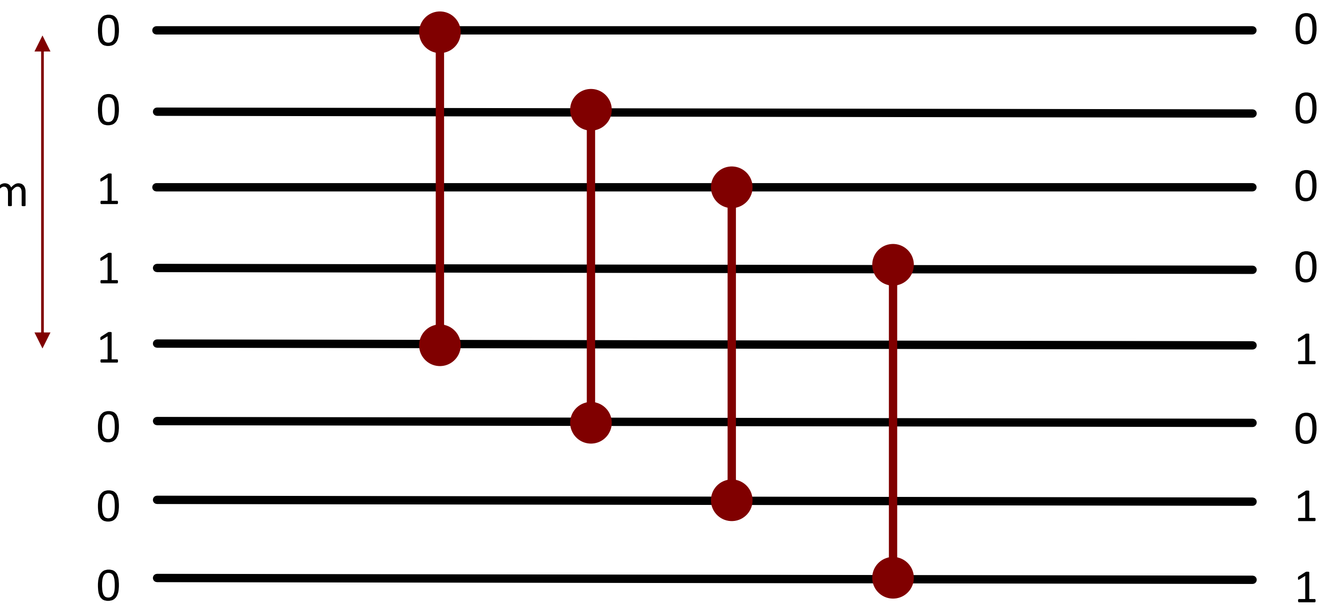
The Half-Cleaner



The Half-Cleaner

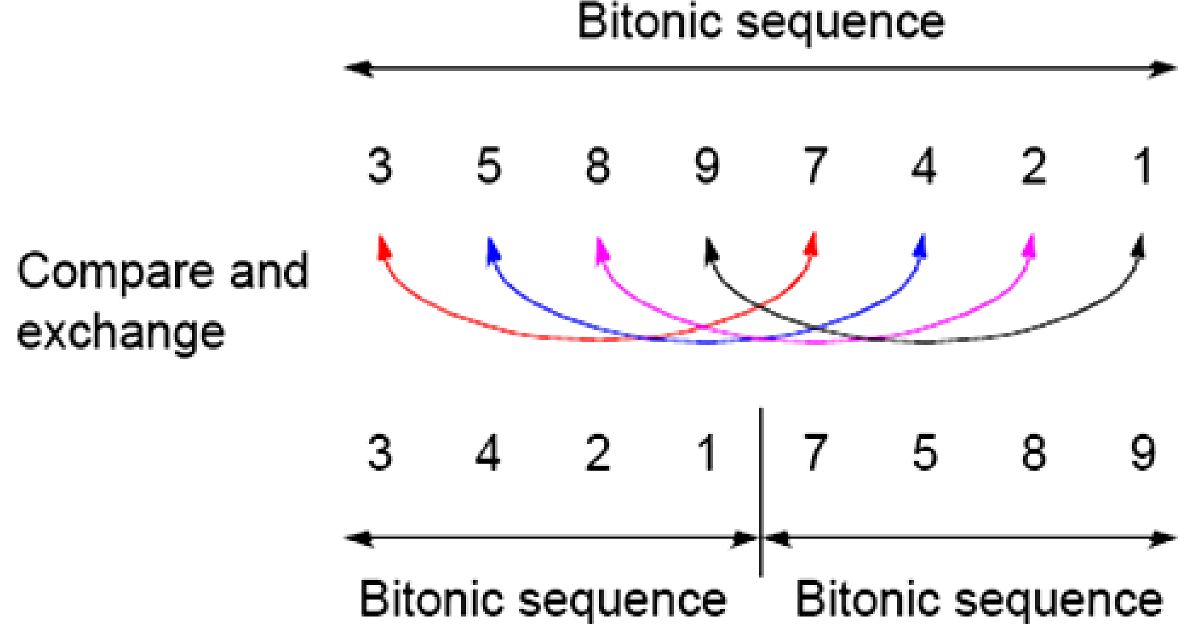


```
void halfClean(int[] a, int lo, int m, boolean dir)
{
   for (int i=lo; i<lo+m; i++)
      compare(a, i, i+m, dir);
}
```



Binary Split: Application of the Half-Cleaner

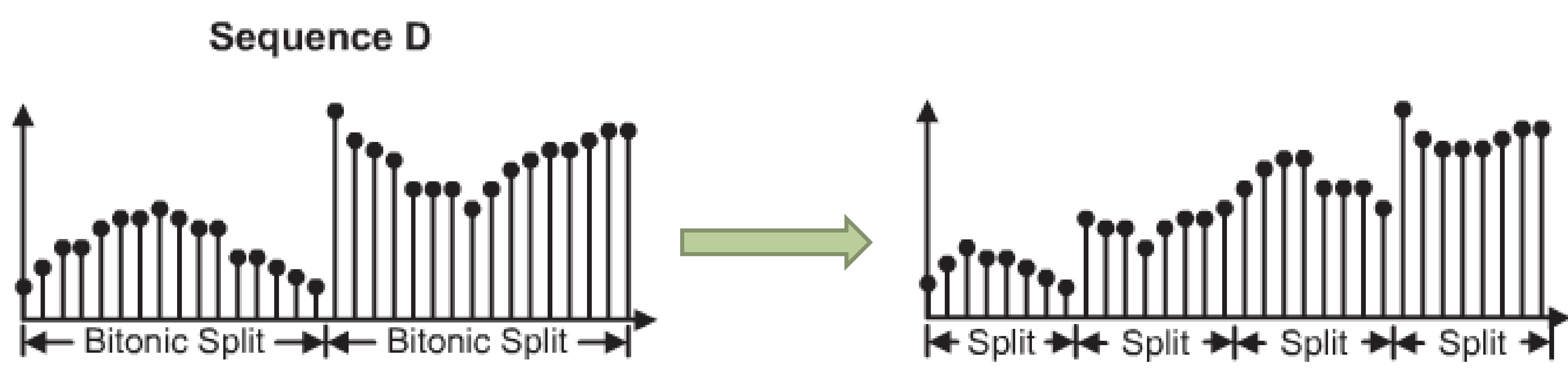
1. **Divide the bitonic list into two equal halves.**
2. **Compare-Exchange each item on the first half with the corresponding item in the second half.**



Binary splits - Result

Two *bitonic* sequences where the numbers in one sequence are all less than the numbers in the other sequence.

Because the original sequence was *bitonic*, every element in the lower half of new sequence is less than or equal to the elements in its upper half.



Bitonic Split Example

bitonic

bitonic

+

bitonic

Lemma

Input bitonic sequence of 0s and 1s, then for the output of the half-cleaner it holds that

- Upper and lower half is bitonic
- One of the two halfs is bitonic clean
- Every number in upper half $\leq$ every number in the lower half

Proof: All cases

bitonic

0
1
0

top

0
1

1
0

bottom

top

0

1
0
1

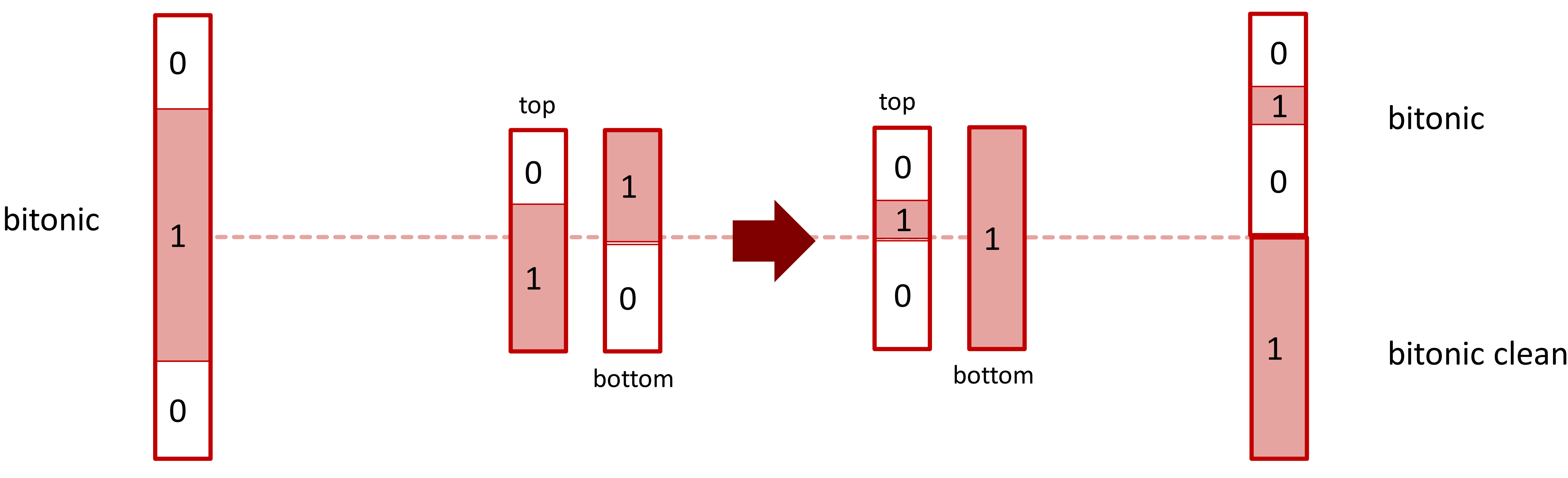
bottom

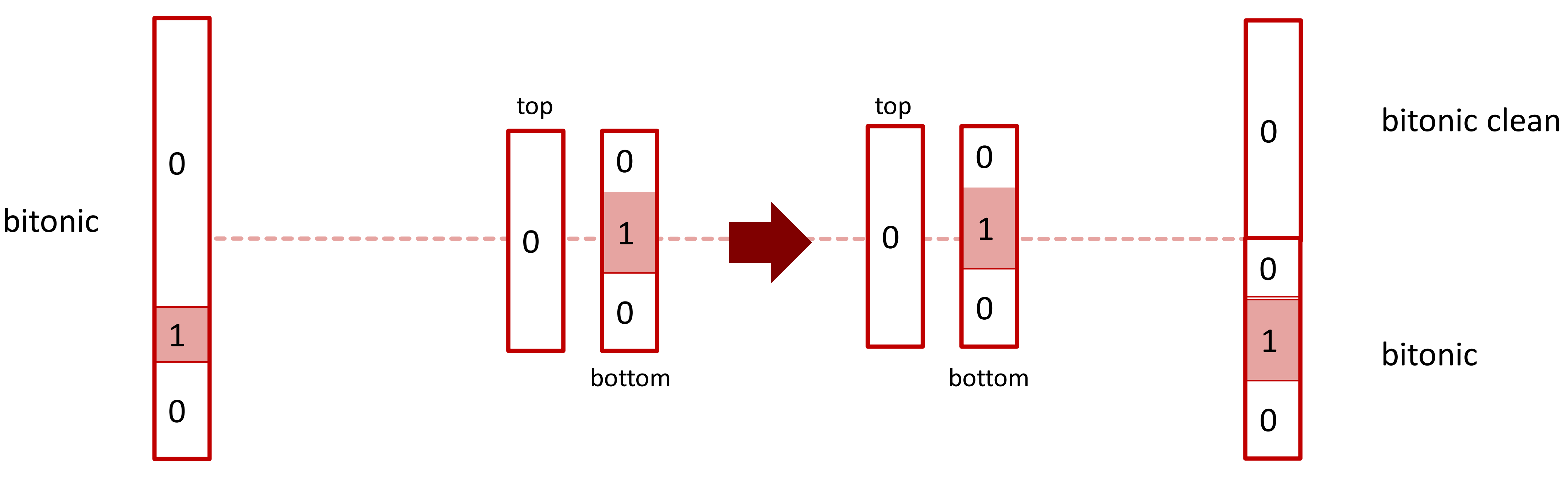
0

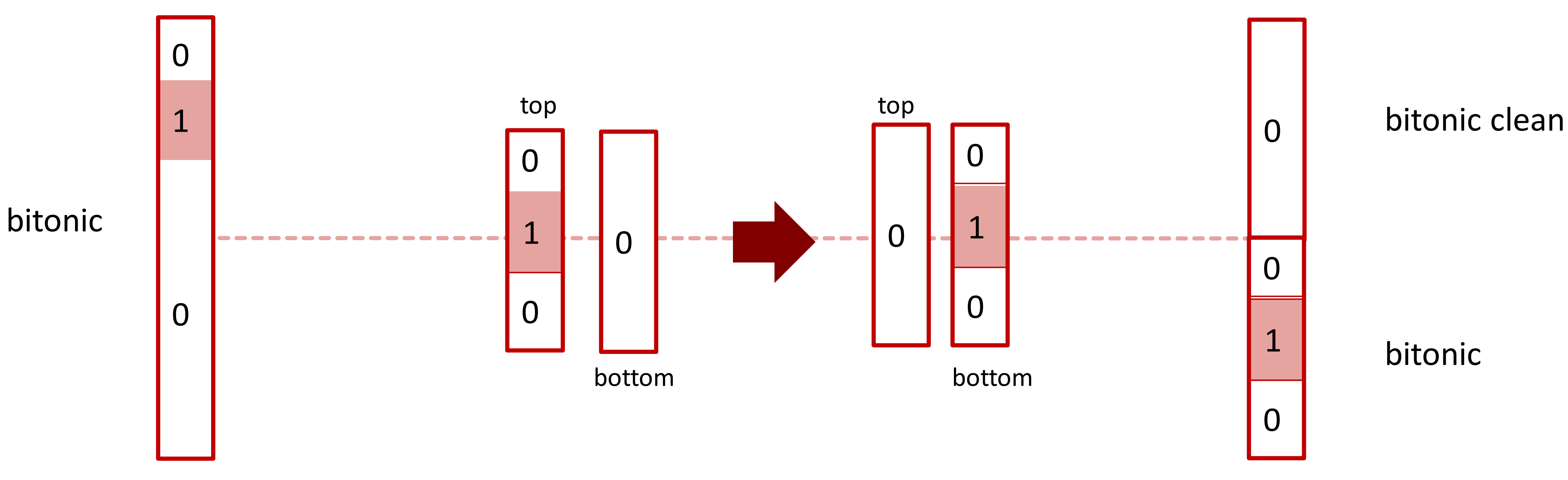
1
0
1

bitonic clean

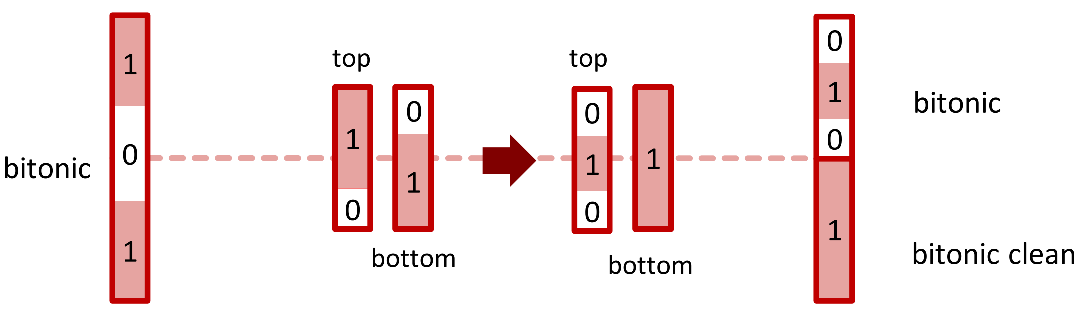
bitonic

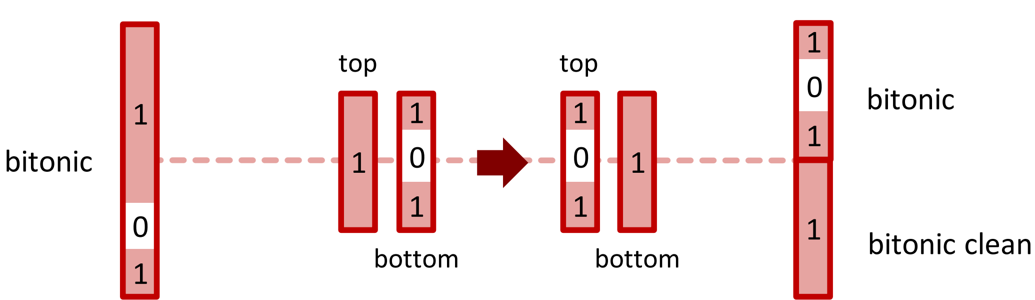

bitonic
0
1
0
top
0
1
1
0
bottom
top
0
1
0
1
bottom
0
1
0
1
bitonic
bitonic clean

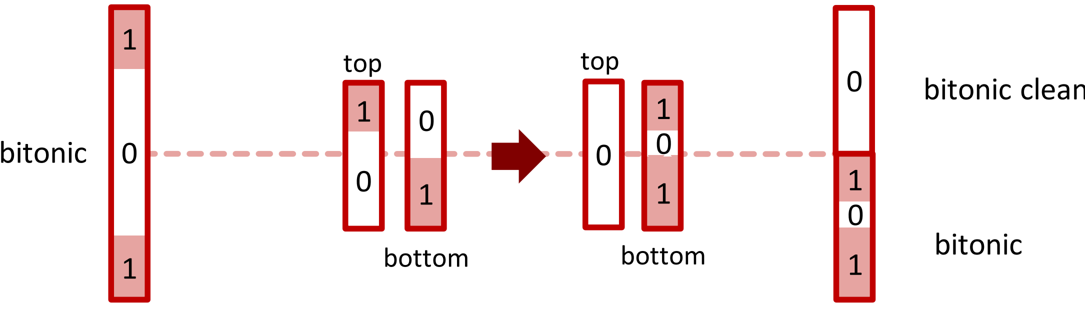

bitonic
0
1
0
top
0
0
1
0
bottom
top
0
0
1
0
bottom
0
0
1
0
bitonic clean
bitonic

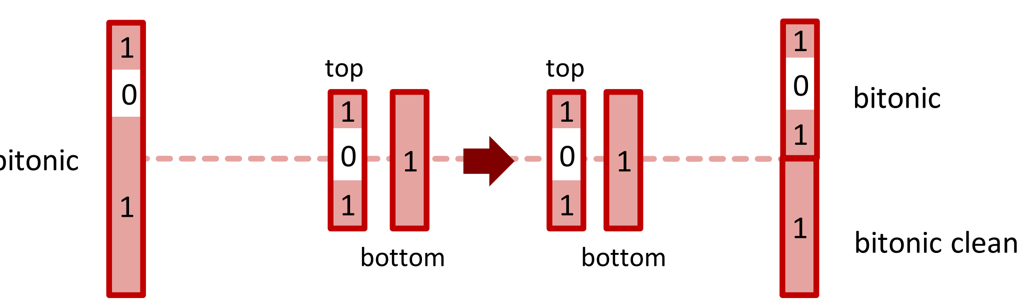

bitonic
0
1
0
top
0
1
0
0
bottom
top
0
0
1
0
bottom
0
0
1
0
bitonic clean
bitonic

The four remaining cases (010 → 101)









Construction of a Bitonic Sorting Network

bitonic

0
0
1
1
1
0
0
0

half cleaner

0
0
0
0
1
0
1
1

half cleaner

half cleaner

0
0
0
0
1
0
1
1

half cleaner

half cleaner

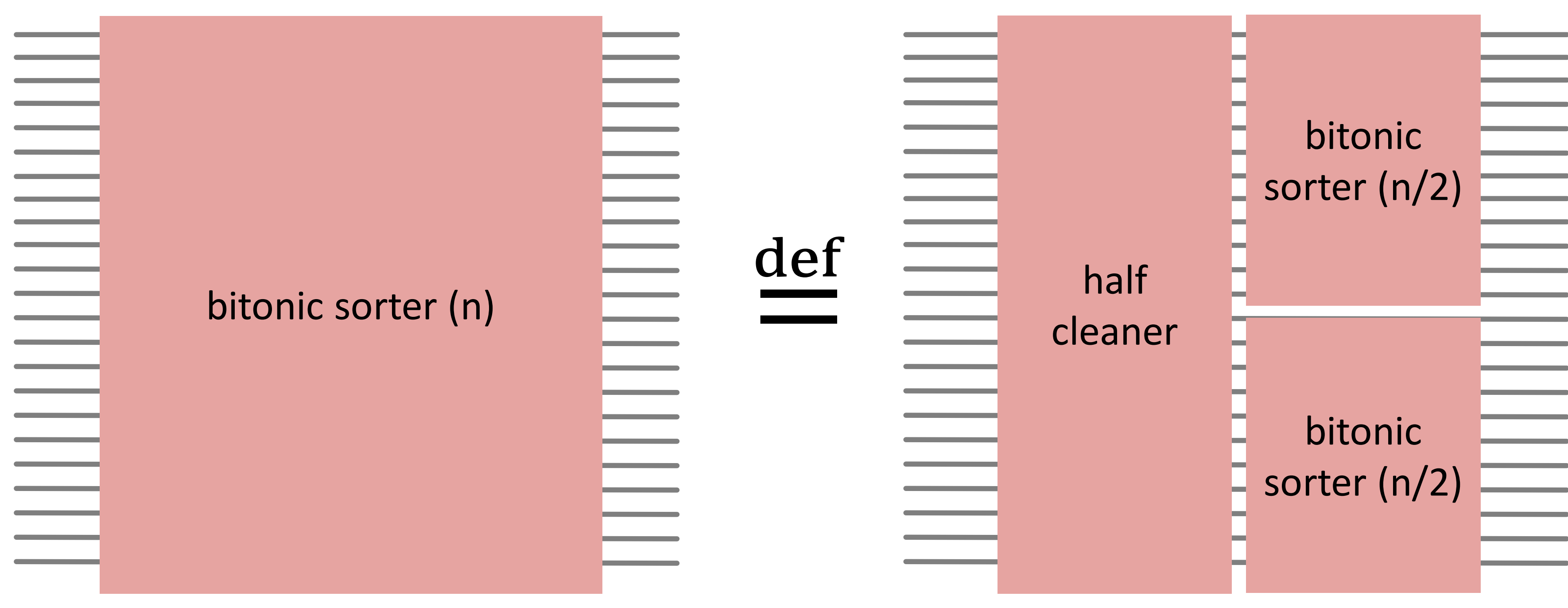
half cleaner

half cleaner

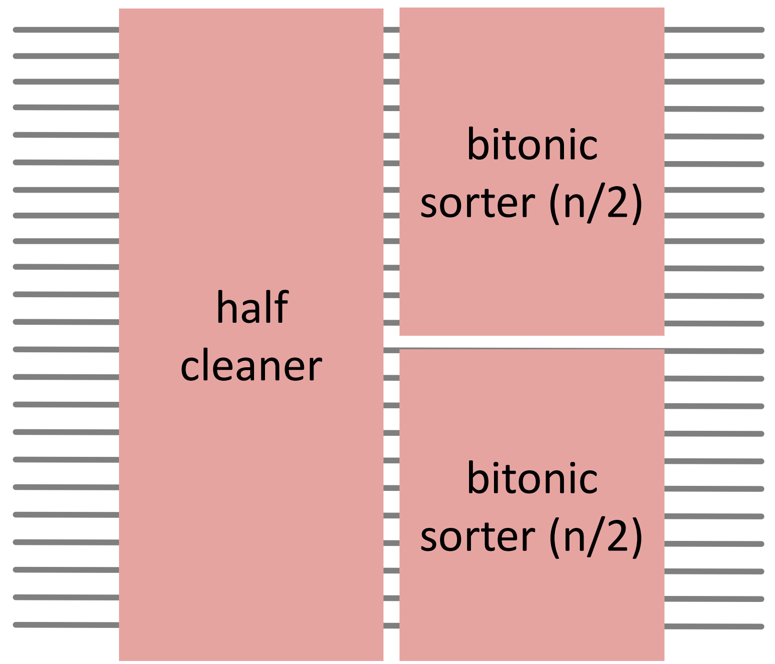
0
0
0
0
0
1
1
1

sorted

Recursive Construction

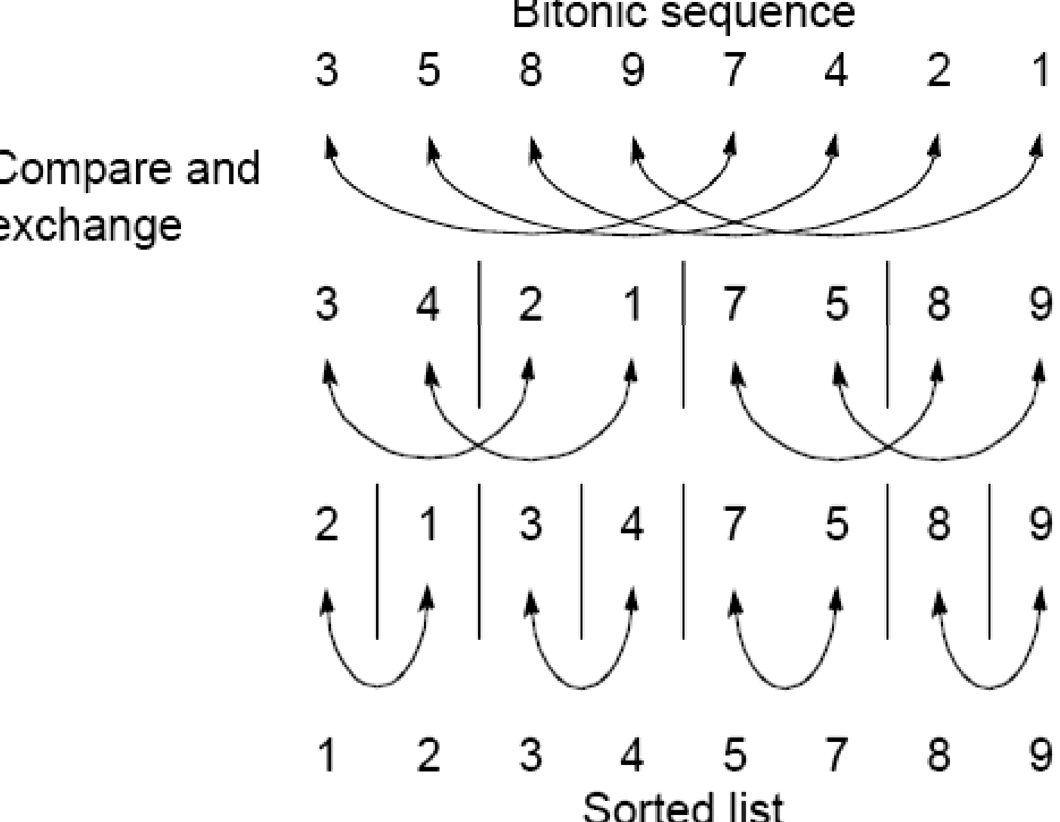


```
void bitonicMerge(int[] a, int lo, int n, boolean dir)
{
   if (n>1){
      int m=n/2;
      halfClean(a, lo, m, dir);
      bitonicMerge(a, lo, m, dir);
      bitonicMerge(a, lo+m, m, dir);
   }
}
```



Bitonic Merge

- Compare-and-exchange moves smaller numbers of each pair to left and larger numbers of pair to right.
- Given a *bitonic* sequence, recursively performing '*binary split*' will sort the list.



Bi-Merger

sorted: 0 0 1 1 → bitonic: 0 0 0 0

sorted: 0 0 0 1 → bitonic: 1 1 0 1

bi-merger

≜

bitonic: sorted 0 0 1 1, reverse sorted 1 0 0 0

half-cleaner

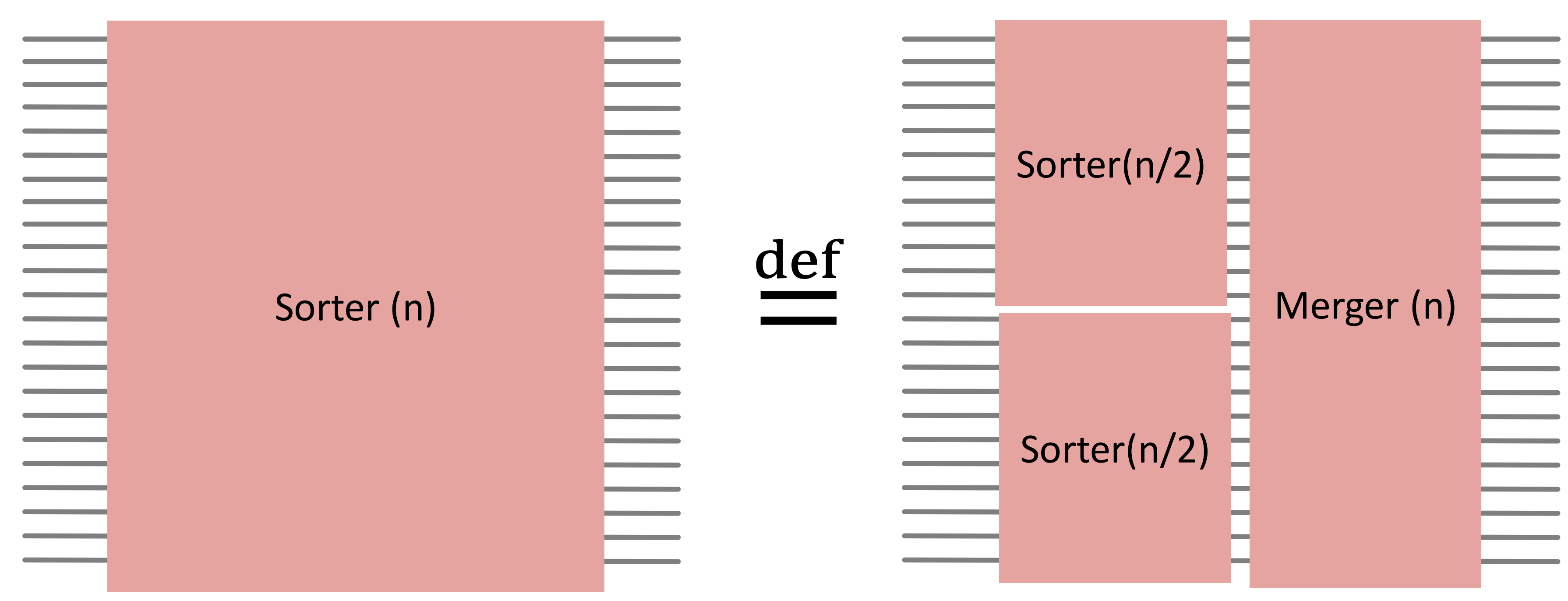
bitonic: 0 0 0 0

bitonic: 1 0 1 1

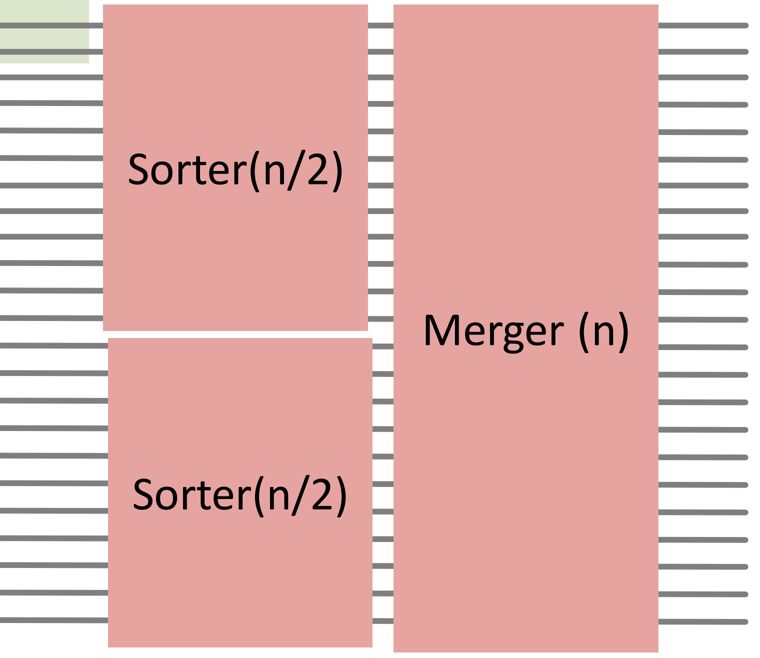
Bi-Merger on two sorted sequences acts like a half-cleaner on a bitonic sequence (when one of the sequences is reversed)

Merger

Recursive Construction of a Sorter



```
private void bitonicSort(int a[], int lo, int n, boolean dir) {
    if (n>1){
        int m=n/2;
        bitonicSort(a, lo, m, ASCENDING);
        bitonicSort(a, lo+m, n, DESCENDING);
        bitonicMerge(a, lo, n, dir);
    }
}
```



Example

half cleaner
half cleaner
half cleaner
half cleaner
Merger (2)

bi-merger
half cleaner
half cleaner
bi-merger
half cleaner
half cleaner
Merger(4)

bi-merger
half cleaner
half cleaner
half cleaner
half cleaner
half cleaner
half cleaner
Merger (8)

Example

half cleaner
half cleaner
half cleaner
half cleaner
Merger (2)

bi-merger
bi-merger
half cleaner
half cleaner
half cleaner
half cleaner
Merger(4)

bi-merger
half cleaner
half cleaner
half cleaner
half cleaner
half cleaner
half cleaner
Merger (8)

Bitonic Merge Sort

How many steps?

$$\sum_{i=1}^{\log n} \log 2^i = \sum_{i=1}^{\log n} i \log 2 = \frac{\log n \cdot (\log n + 1)}{2} = O(\log^2 n)$$

#steps / merger