# Concepts of Parallel and Distributed Processing

**COP 4520 – Fall 2005**

**University of Central Florida**

**Charles E. Hughes**

**ceh@cs.ucf.edu**
**http://www.cs.ucf.edu/~ceh**
**Professor, School of Computer Science**

## Basic Information

**Meeting Times:** TR 15:00 - 16:15**, CS 221**

**Instructor**: Charles E. Hughes, ceh@cs.ucf.edu, CSB-206, 823-2762

**TA**: None

**Home Pages**:
Instructor http://www.cs.ucf.edu/~ceh/;
Course http://www.cs.ucf.edu/courses/cop4520/NotesFall2005.html

**Office Hours:** TR 13:15-14:30

**References:**
Gregory R. Andrews, *Multithreaded, Parallel and Distributed Programming*, Addison-Wesley, 2000.
Ananth Grama et al, *Introduction to Parallel Computing*, Addison-Wesley, 2003.
Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2003.
Tarek El-Ghazawi, *UPC: Distributed Shared Memory Programming*, John Wiley and Sons, 2005.

**Prerequisites:** COP3530 (CS3), COT3100 (Discrete Structures), CDA3103 (Computer Organization), COP3402 (Computer System Concepts).

**Implementation Environments**: Java – Eclipse. MPI. UPC.

**Assignments**: 4 to 6 small to moderate programming assignments (some are multi-part) using a variety of parallel and distributed programming paradigms; 4 to 6 non-programming assignments; one project.
**Exams**: Two midterms and a final.

## Evaluation

**Evaluation:**
This is an approximation, subject to restructuring based on increases or decreases in assignments or in complexity of assignments over what I am currently considering.
Quiz – 60 points; Mid Term – 90 points; Final Exam – 150 points
Actually the weight of a quiz and its corresponding exam varies to your benefit.
Assignments – 200 points; Available Points – 500

A – ≥90%
B+ – 87%-89%
B – 80%-86%
C+ – 77%-79%
C – 70%-76%
D – 50%-69%
F – <50%

**Important Dates (Quiz Date is Subject to Change):**
Quiz – MidTerm1 – October 6; MidTerm2 – November 3; Withdraw Deadline – October 14;
Thanksgiving – November 24; Final – December 8, 13:00-15:50

## Concept and Some Analogies

- Mowing and Edging Grass Takes Time
- Can Share the Work in Several Ways
    - Generalists: Many People have Mowers and Edgers
    - Specialists: Half Have Mowers, Half Have Edgers
    - Agenda-based: Set an Agenda for the Whole Gang
- There are Problems with Relative Speeds
- There are Problems with Shared Resources
    - Brooms, Trash Cans, …

- There are similar problems when many threads running on one or more computers, each with one of more processors, is attacking a single problem

## Terminology

- Concurrent programming
- Multithreaded programming
- Multiprogramming
- Multiprocessing
- Distributed processing

## Topics of Course

- Architectures (hardware and software)
- Protocols (hardware and software)
- Communication and Coordination Primitives
- Algorithms
- Algorithm Complexity Analysis
- Reasoning about algorithms
    - Correctness, Safety, Liveness
    - Meta results
- Java Distributed Computing Paradigms
- MPI Message-Based Computing
- UPC SPMD Paradigm
- and other paradigms

## A Model

**Fixed Connection Network**

- Processors Labeled $P_1, P_2, \ldots, P_N$
- Each Processor knows its Unique *ID*
- Local Control
- Local Memory
- Fixed Bi-directional Connections
- Synchronous

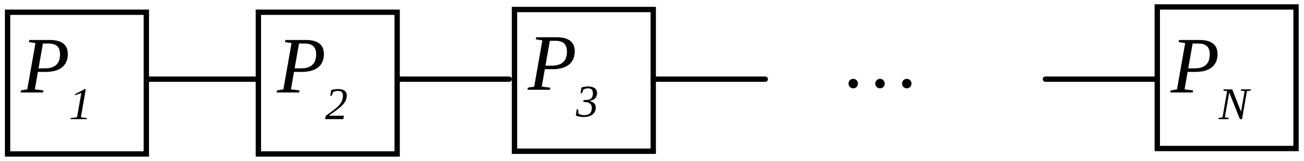
  Global Clock Signals Next Phase

## Operations at Each Phase

**Each Time the Global Clock Ticks**

- Receive Input from Neighbors
- Inspect Local Memory
- Perform Computation
- Generate Output for Neighbors
- Update Local Memory

## A Model of Cooperation: Bucket Brigades

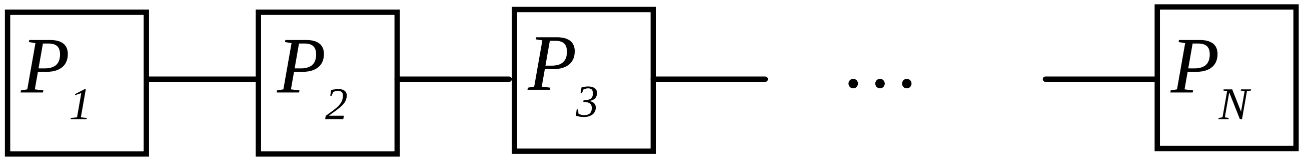


- $N$ Processors, Labeled $P_1$ to $P_N$

- Processor $P_i$ is connected to $P_{i+1}$, $i<N$

## A Sort Algorithm

**Odd-Even Transposition on Linear Array**



- The Array is *X[1 : N]*
- *$P_i$*'s Local Variable *X* is *X[i]*
- *$P_i$*'s have a Local Variables *Y* and a Global/Singular variable *Step*
- *Step* is initialized to Zero (0) at all *Pi*
- Compares and Exchanges are done alternately at Odd/Even - Even/Odd Pairs

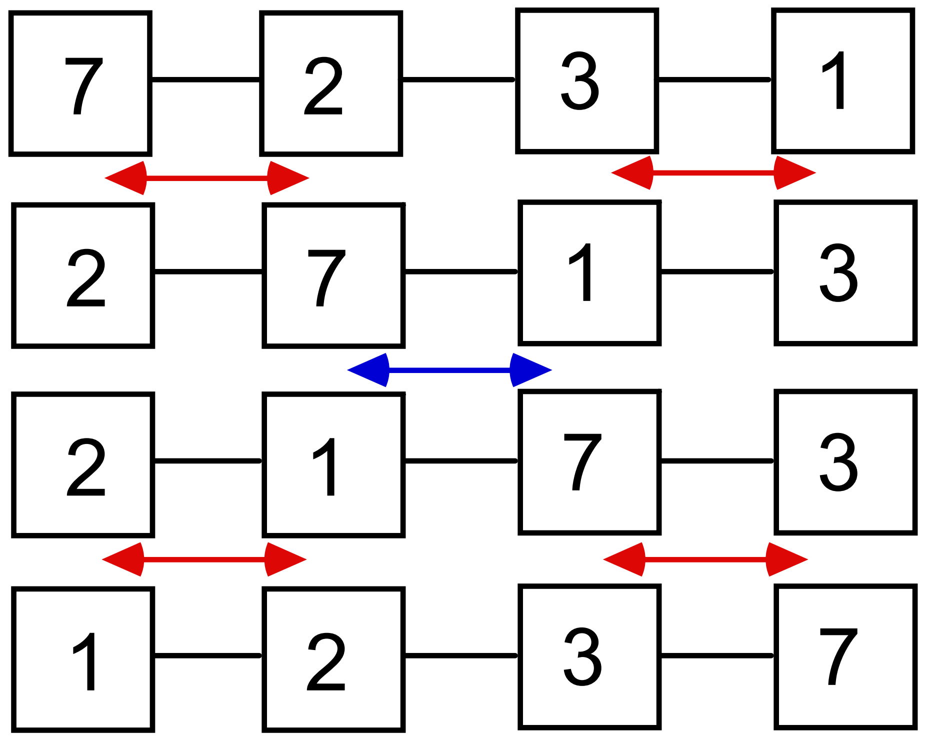
# Odd-Even Transposition

**Algorithmic Description of Parallel Bubble Sort**

```
At Each Clock Tick and For Each $P_i$ do {
  $Step$ ++;
  if parity($i$) = = parity($Step$) & $i < N$ then
        Read from $P_{i+1}$ to $Y$;
        $X$ = min($X$,$Y$)
  else if $i > 1$ then
        Read from $P_{i-1}$ to $Y$;
        $X$ = max($X$,$Y$);
}
```

## Example of Parallel Bubble Sort

**Sort 4 Numbers 7, 2, 3, 1 on an Array of 4 Processors**



Case of 4, 3, 2, 1 Takes 4 Steps

## Measuring Benefits

### How Do We Measure What We Have Gained?

- Let $T_1(N)$ be the Best Sequential Algorithm
- Let $T_P(N)$ be the Time for Parallel Algorithm (P processors)
- The Speedup $S_P(N)$ is $T_1(N)/T_P(N)$
- The Cost $C_P(N)$ is $P \times T_P(N)$, assuming $P$ processors
- The Work $W_P(N)$ is the summation of the number of steps taken by each of the processors. It is often, but not always, the same as Cost.
- The Cost Efficiency $CE_P(N)$ (often called efficiency $Ep(N)$) is
  $$S_P(N)/P = C_1(N) / C_P(N) = T_1(N) / (P \times T_P(N))$$
- The Work Efficiency $WE_P(N)$ is
  $$W_1(N) / W_P(N) = T_1(N) / W_P(N)$$

## Napkin Analysis of Parallel Bubble

**How'd We Do ? - Well, Not Great !**

| | |
|---|---|
| • $T_1(N) = N \lg N$ | Optimal Sequential |
| • $T_N(N) = N$ | Parallel Bubble |
| • $S_N(N) = \lg N$ | Speedup |
| • $C_N(N) = W_N(N) = N^2$ | Cost and Work |
| • $E_N(N) = \lg N / N$ | Cost and Work Efficiency |

**But Good Relative to Sequential Bubble**

$S_N(N) = N^2/N = N$ ; $E_N(N) = S_N(N) /N = 1$ !

## Non-Scalability of Odd-Even Sort

Assume we start with 1 processor sorting 64 values, and then try to scale up by doubling number of values (N), each time we double number of processors (P) in a ring. The cost of the parallel sort requires each processor to sort its share of values (N/P), and then do P swaps and merges. Since P processors are busy, the cost is N lg N/P. After the local sort, sets are exchanged, merged, and parts thrown away. The merge costs N/P on each of P processors, for a Cost of N, and P-1 such merges occur, for a total cost of N×(P-1).
Efficiency is then

**E = N lg N / (N lg N/P + N×(P-1)) = lg N / (P - 1 + lg N - lgP)**

First 2 columns double N as P doubles. Second three try to increase N to keep efficiency when P doubles.

| N | P | E | N | P | E |
|---|---|---|---|---|---|
| 64 | 1 | 1.0000 | 64 | 1 | 1.0000 |
| 128 | 2 | 1.0000 | 4096 | 2 | 1.0000 |
| 256 | 4 | 0.8889 | 16777216 | 4 | 0.9600 |
| 512 | 8 | 0.6923 | 2.81475E+14 | 8 | 0.9231 |
| 1024 | 16 | 0.4762 | 7.92282E+28 | 16 | 0.8972 |
| 2048 | 32 | 0.2973 | 6.2771E+57 | 32 | 0.8807 |
| 4096 | 64 | 0.1739 | 3.9402E+115 | 64 | 0.8707 |
| 8192 | 128 | 0.0977 | 1.5525E+231 | 128 | 0.8649 |

## Cost for Finding Max Value in a List

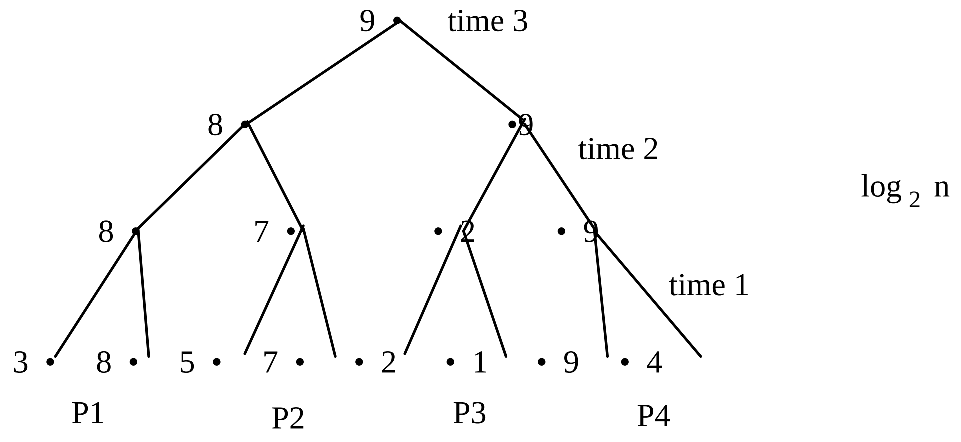
Given a sequence A of n elements find the largest of these elements.

**Serial Algorithm.**

```
    Largest = A [0]
    For i = 1 to n-1 do { if A [i] > Largest then Largest = A [i] }
```

n - 1 comparison.

**A Parallel Algorithm**



## Efficiency of Binary Tree Max

**Assume Full Binary Tree**

- $T_{N/2}(N) = T_{N/4}(N/2) + 1, N > 1$

  $T_1(2) = 1$

  $T_{N/2}(N) = lg\ N = O(lg\ N)$

- $C_N(N) = N\ lg\ N = O(N\ lg\ N)$

  $\mathrm{E}_N(N) = N / N\ lg\ N = O( 1 / lg\ N)$

- $W_{N/2}(N) = W_{N/4}(N/2) + N/2, N > 2$

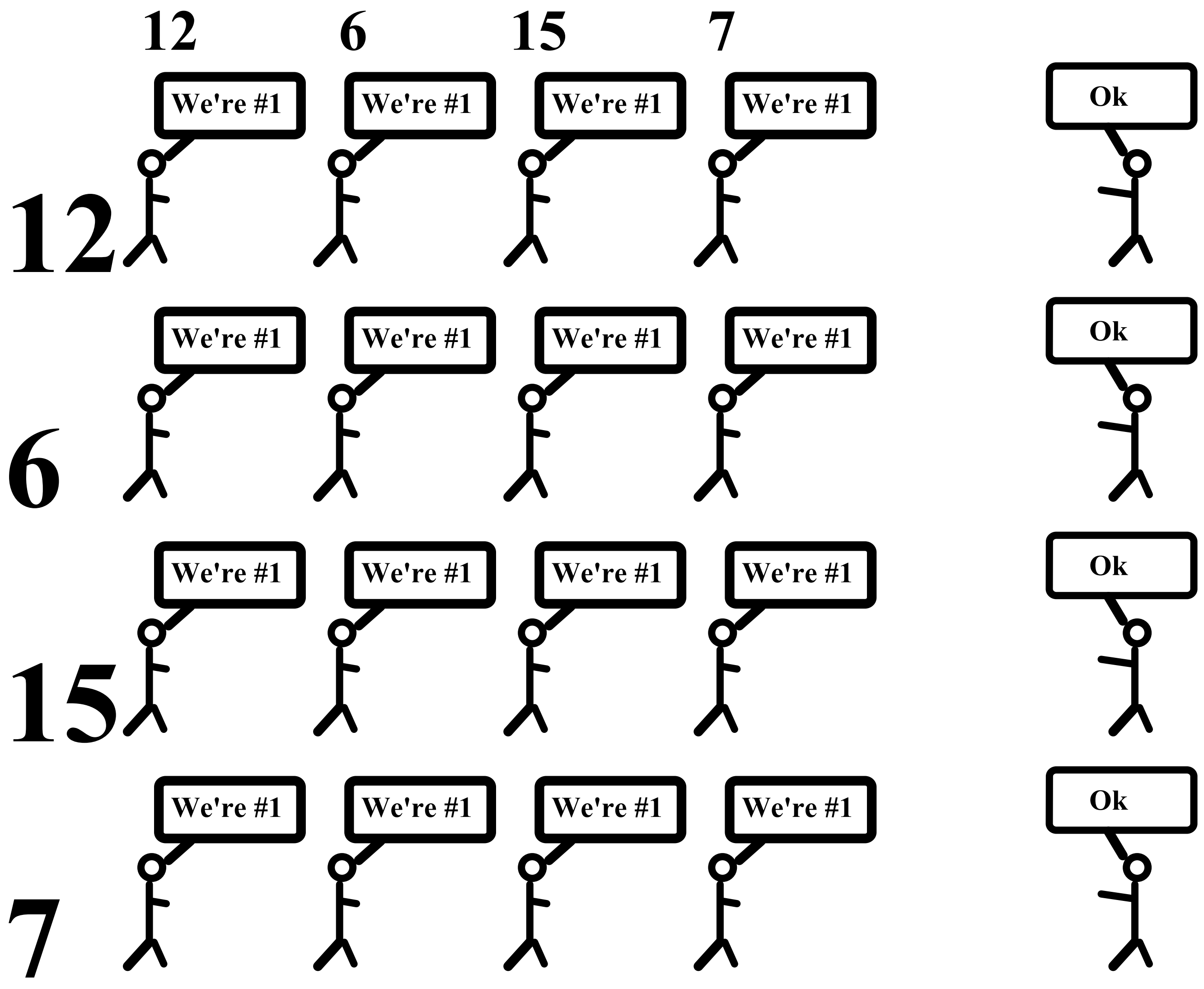
  $W_1(2) = 1$

  $W_{N/2}(N) = N - 1 = O(N)$

- This is optimally work efficient.
- But it is not optimally cost efficient.

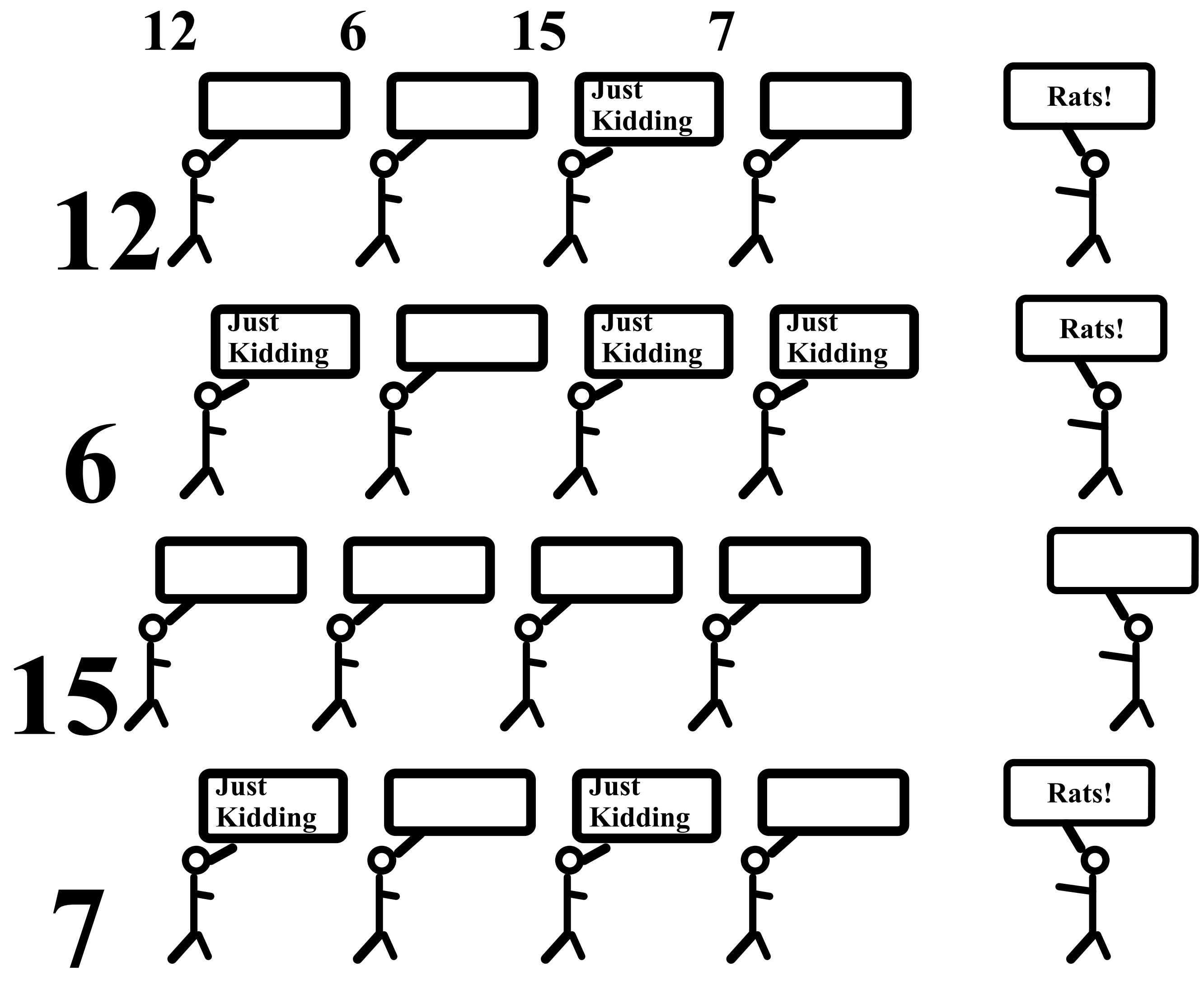
## Finding the Maximum by Controlled Anarchy

Step#1: Everyone's an Optimist



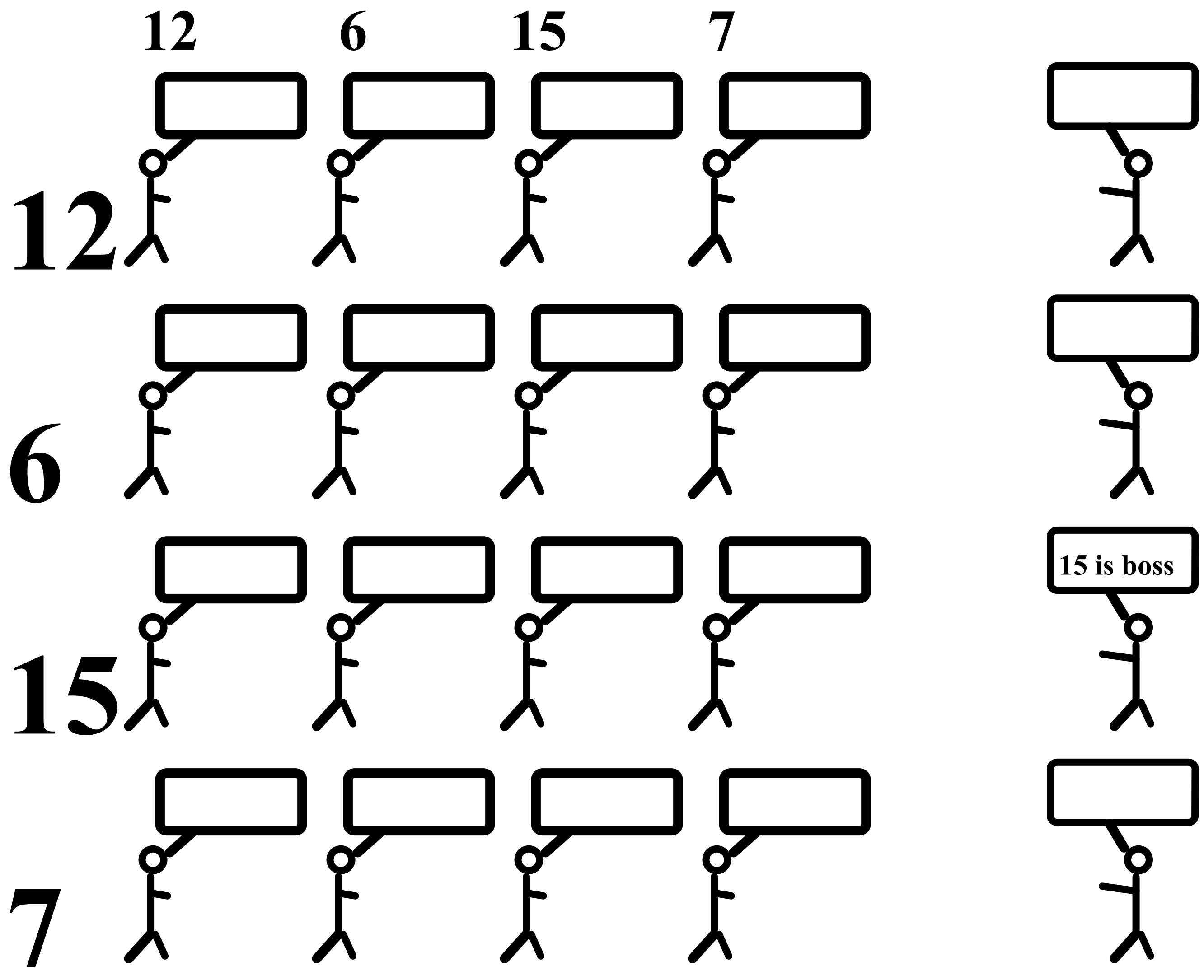
## This is the Meatiest Part

Step#2:  Realism Sets In



## That’s All Folks

Step#3: Reporting the Answer



## Analysis of Very Fast Max

Optimal in Time, Not Work on CRCW (Concurrent Read Concurrent Write) PRAM (Parallel Random Access Machine)

- Assign **N** processors to initialize **M** in 1 step.
- Assign all **N**$^{\mathbf{2}}$ processors to first statement to fill **B** in 1 step.
- Assign all **N**$^{\mathbf{2}}$ processors to 2nd statement to fill **M** in 1 step.
- Assign **N** processors to 3rd statement to select **maxVal** in 1 step.

## That Was Inefficient but Real Fast

- Can Solve Any Size Problem in 3 Steps
  But we need to make unreasonable assumptions about memory (CRCW)

- Use Lots of Processors
  Over a Million to Find Max of 1000

- We Want Fast but Not Too Expensive

## Sense of Optimality of Max

### It Depends on Model and Goals

- Can use **$N^2$** processors to find max of **N** elements on **O(1)** time.
- Work is **$O(N^2)$** on CRCW PRAM.
- Minimal work on EREW or CREW PRAM requires **O(lg N)** time.
- Can achieve **O(lg lg N)** time on CRCW doing minimal work.

## Fast, Inefficient Max in Unity Notation
## || is parallel composition

```
Program max

declare     B : array [1..N, 1..N] of boolean
            M : array [1..N] of boolean
            maxVal : integer

initially < || i : 1≤i≤N :: M[i] = false>

assign

  < || i ,j : 1≤i≤N & 1≤j≤N :: B[i,j] = A[i] ≥ A[j] ) >
  ||
  < || i : 1≤i≤N :: M[i] = < & j : 1≤j≤N :: B[i,j] > >
  ||
  < || i : 1≤i≤N :: maxVal = A[i]  if M[i]  >

end { max }
```

## Synchronous Parallel Max on Balanced Tree

Make ***A[i]*** the parent of nodes ***A[2×i]*** and ***A[2×i+1]***. In this case ***A[0]*** is the parent of ***A[0]*** and ***A[1]***! But that's good – think about it.
Hint: if this weren't so and ***A[0]***'s value was the max, it would be lost.

```
Program max
    declare  t : integer
    initially  t = N
    assign
         < || i : 0≤i<t/2 ::
              A[i]  = max(A[2*i], A[2*i+1] )      if 2*i+1<N
                   ~ A[2*i]                       if 2*i+1=N
         >
         ||  t = t/2
end { max }
```

## Efficiency of Balanced Tree Max

**Study Work Efficiency**

- $T_{N/2+1}(N) = T_{N/4+1}(N/2) + 1, N > 1$

  $T_2(2) = 1$

  $T_{N/2+1}(N) = lg\ N = O(lg\ N)$

- $W_{N/2+1}(N) = W_{N/2}(N/2) + N/2 + 1 , N > 2$

  $W_2(2) = 2$

  $W_{N/2+1}(N) = N - 1 + lg\ N = O(N)$

- ***Tree Max*** is optimally work efficient within a constant factor

## Assignment # 1

1. Consider the Max Tree we described earlier, but now use p processors to sort N values, where N may be greater than p. In this case, each processor uses a sequential algorithm to find its local max and then proceeds with the standard tree approach. Analyze the Time, Cost and Work for
   a. p = lg N
   b. p = lg lg N
   c. p = N / lg N
   d. p = N / lg lg N
2. Consider the problem of sorting a deck of cards into Spades, Hearts, Diamonds, Clubs, and in order Ace down to 2 within each suit.
   a. What is the best way for an individual to do this? Describe the approach and analyze the number of comparisons and inspections (e.g., what suit is this) done.
   b. Redo this but this time with five people. One of the five starts with all 52 cards. You will need to analyze the additional number of times a card is passed from one person to another. We assume random access; that is, any person can pass a card to any other at the same cost. The ending state of the system is that all 52 cards must be in one person's hand, sorted.

Due: Week#2 (9/1)

## Taxonomies

**Control**

**Communication Model / Address space**

**Interconnection network**

**Granularity**

## Taxonomies -- Control

**SISD (typical single instruction on single data stream)**

**SIMD (single instruction applied to many data streams)**

**MISD (multiple instructions on single data stream – pipeline)**

**MIMD (multiple instructions on multiple data streams – typical multiprocessing)**

**Programming models**

1. Control Parallel assumes separate independent functions that can be solved simultaneously. These separate functions are then assigned to separate cpu's.
2. Data Parallel assumes there is a large data set that needs to be processed and that there is single processor for each data element in set. The same set of instructions is applied to all elements in the data set. (Aggregation is needed if actual number of elements exceeds available processors.)

**Control (task)-parallel**

Assign one (or more) processors to each function. The information from one function is passed to the next like on an assembly line. Often the communication mechanism can do double duty as the coordination mechanism. Alternatively, we can use a form of “barrier synchronization.” In either case, we need to design our solution so subtasks take about the same amount of time.

**Data-parallel**

The same function is applied to each page simultaneously. Hence we process the pages independently in parallel. The processors need to be synchronized in reporting the results. SIMD architectures imply data parallelism, although MIMD can also be used.

## Even-Odd Transposition on a SIMD Machine

```
plural int value; // need to set to some value on each processor
…
int stage = 0;
if (iproc %2 == 0) // let even numbered do the work
    while (stage < N-1) { // do two stages
        if (xnetE[1].value < value) {
            int temp = value;
            value = xnetE[1].value;
            xnetE[1].value = temp;
        }
        if (xnetW[1].value > value) {
            int temp = value;
            value = xnetW[1].value;
            xnetW[1].value = temp;
        }
        stage += 2;
    }
```

## Taxonomies – Communication Model / Address Space

Private memory (separate address spaces, also called distributed memory)
Shared address space (often called shared memory).

UMA (uniform / symmetric multiprocessors (SMP))
NUMA (non-uniform) memory access.

Cache and the cache coherence (consistency) problem.

A multicomputer is a distributed memory multiprocessor in which the nodes and network are in a single cabinet. Such a system is tightly coupled and communication is over a dedicated high-speed interconnection network.

A network of workstations is a form of distributed memory multiprocessor. Such a system is loosely coupled and communication is usually through message passing.

Distributed, shared memory refers to a distributed implementation of the shared memory model.

## Taxonomies -- Interconnection Network

Dynamic interconnections, e.g., bus and crossbar
Dynamic interconnection networks use switches (indirect), rather than direct connects.
Network is dynamic (bus like).
Crossbar performance scales well (no blocking), but cost is a problem. $N^2$
Bus scales poorly on performance but nicely on cost.
Multistage compromises are usually used. (Butterfly is nice example -- n lg n switches, lg n set for any communication, blocking occurs)

Static interconnections, e.g., linear, completely connected and hypercube.
Some examples are completely-connected, star connected, linear array, ring, 2-d mesh, torus, 3-d mesh and torus, tree, hypercube.
Note that the central node in a star is the bottleneck, just as the bus is in a bus scheme.
This is also true of the root of a tree.

**BLACKBOARD TIME**

## Metrics for Static Networks

- Diameter – Maximum distance
  - Routing algorithms
    - Ring (shortest distance left or right)
    - 2D mesh (XY dimensional routing)
    - Hypercube (E dimensional routing)
      - Note Hamming distance and Hypercube
- Connectivity – Number of paths between nodes
  - Arc connectivity is minimum number of edges that can be removed before network is disconnected
- Bisection Width – Minimum number of edges removed to partition network into “equal” halves
- Channel Width – simultaneous bits that travel over each link
- Channel Rate – Peak data rate over a single link
- Channel Bandwidth – Product of Channel Width and Channel Rate
- Bisection Bandwidth – Product of Bisection Width and Channel Bandwidth
  - Measure of how much can be pushed between halves of network

## Characteristics of Topologies

| Network | Diameter | Arc Connectivity | Bisection Width | Cost (# links) |
|---|---|---|---|---|
| Complete | 1 | $p - 1$ | $p^2 / 4$ | $p (p - 1) / 2$ |
| Star | 2 | 1 | 1 | $p - 1$ |
| Binary tree | $2 \lg((p + 1)/2)$ | 1 | 1 | $p - 1$ |
| Linear array | $p - 1$ | 1 | 1 | $p - 1$ |
| Ring | $\lfloor p / 2 \rfloor$ | 2 | 2 | $p$ |
| 2d mesh | $2 (\sqrt{p} - 1)$ | 2 | $\sqrt{p}$ | $2 (p - \sqrt{p})$ |
| 2d torus | $2 \lfloor \sqrt{p} / 2 \rfloor$ | 4 | $2 \sqrt{p}$ | $2 p$ |
| Hypercube | $\lg p$ | $\lg p$ | $p / 2$ | $(p \lg p) / 2$ |
| k-ary d-cube | $d \lfloor k / 2 \rfloor$ | $2 d$ | $2 k^{d - 1}$ | $d p$ |

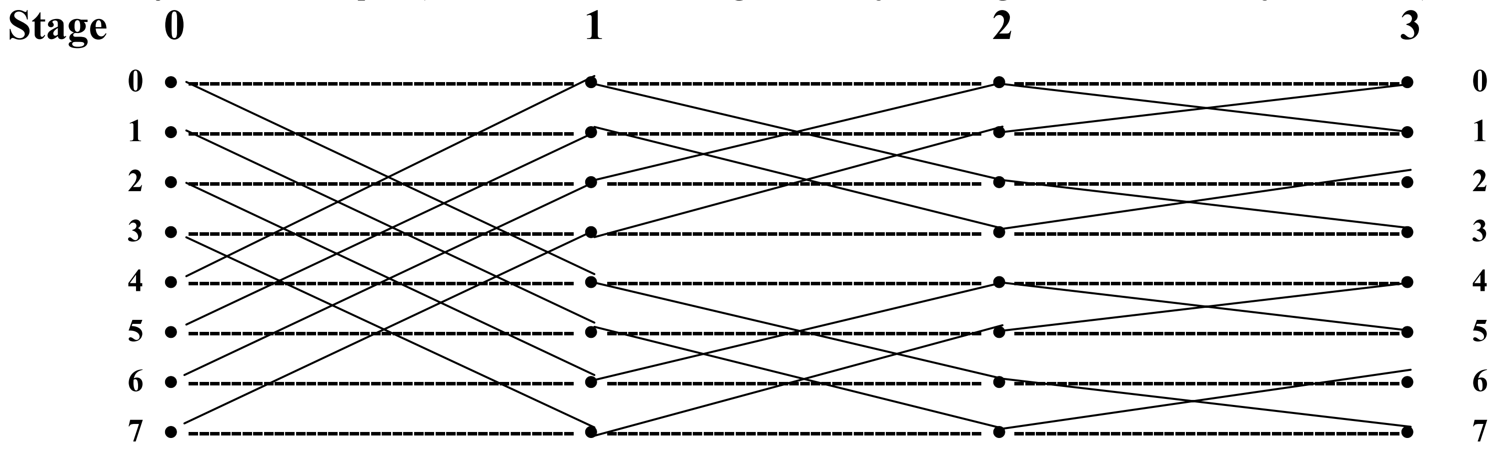
- Note the hypercube is a 2-ary, d-cube, having $2^d$ processors. A ring is a p-ary, 1-cube. A 2d torus of p processors is a $\sqrt{p}$-ary, 2-cube. A k-ary, d-cube can be created from k k-ary (d-1) cubes by connecting identical positions.

# Assignment # 2

Due: Week#3 (9/8)

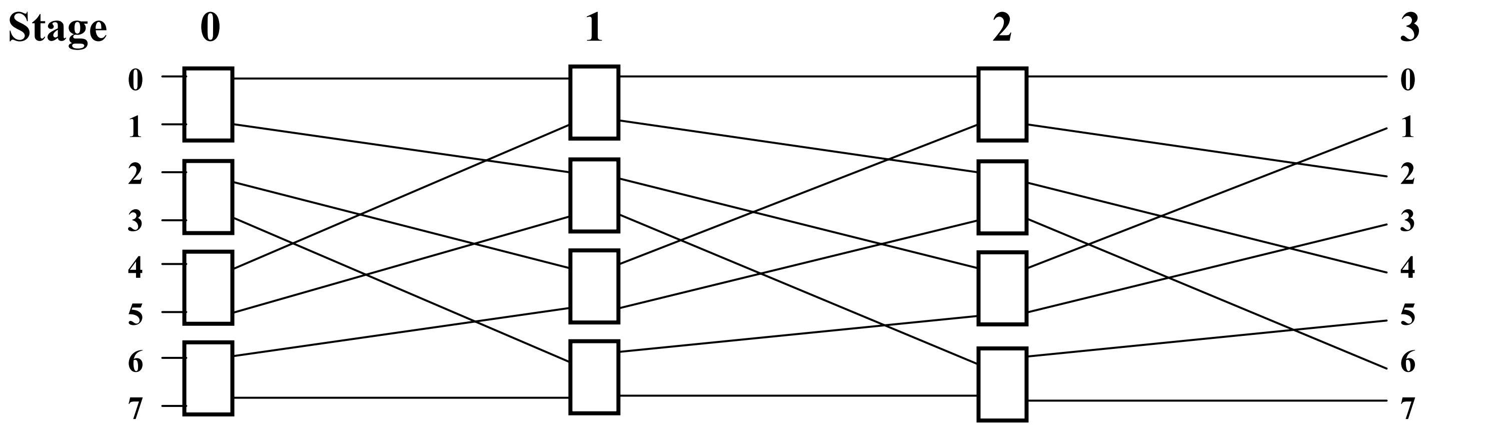
Consider the Omega and Butterfly versions of multistage dynamic interconnection networks. Each is specified on networks with p processors, where p is some power of 2.

Butterfly network with p=8 (connect crossover at stage i of line j with stage i+1 switch at line $j \oplus 2^{\lg p - i - 1}$).



Omega network with p=8 (connect output of line j at stage i to line (j shift left 1 circular) at stage i+1).

Note: You can read or write this network backwards from what I show below.



a. Prove that each of these connects node x to node y when the circuits are switched by choosing crossover at the i-th switch whenever the i-th most significant bit of $x \oplus y$ is 1.

b. How many distinct potential communication pairs (a to b, c to d; $a \neq c$, $b \neq d$) exist for p=8? Of these, how many can occur in parallel without a conflict occurring at some switch?

## Embedding Lower Order Networks into Hypercubes

- Reflected Grey Code
- Applying Code to Rings and Meshes
- Using Code to map onto Hypercube

## Reduction and Broadcast

- Reduction (all to one)
    - Tree Reduction Algorithms
    - Mapping onto a Hypercube
    - Ring version
    - 2d torus version
- Broadcast (one to all)
    - Hypercube
    - Ring
    - 2d torus
- Broadcast (all to all)
    - Hypercube
    - Ring
    - 2d torus

## Routing on Static Networks

- Communication costs associated with static networks.
  Parameters are Startup Time (*ts*), Per-Hop Time (*th*), Per-Word Transfer Time (*tw*).
- Switching Techniques:
  - Store-and-forward cost for *m* words traversing *l* links is $t_{comm} = t_s + (mt_w + t_h)\, l$. Since $t_h$ is usually quite small, we simplify this to $t_{comm} = t_s + mt_w l$.
  - Cut-through routing advances the message as soon as it arrives at a node. Wormhole routing is a specific type of cut-through in which the message is divided into small parts called flits (flow-control digits). Flits are pipelined through the network with an intermediate node needing to store the flit, but not the whole message. Since flits are of fixed size, the communication cost is $t_{comm} = t_s + lt_h + mt_w$ .
  - Thus, store-and-forward is O(*ml*), whereas cut-through is O(*m*+*l*).
- Deadlocking can occur in wormhole routing

## Granularity

- Fine grain (often data parallel)
- Coarse grain (often control / function parallel)

- One measure is time for each computation step versus communication required before next step

- BSP Model
    - Bulk Synchronous Parallel
        - Compute then communicate
        - Loop {Superstep; Barrier Synch; communicate;}
    - Granularity is ratio of size of superstep to communication time

## Programming Styles

Iterative parallelism: co // and process notation

Recursive parallelism (divide and conquer)

Producer / Consumer

Client / Server

Peers: worker, send and receive notation

Common orthogonal ways to attack are:

  Functional Decomposition

  Data Decomposition

## Foster's Design Methodology – Partitioning

- Divide computation and data into many pieces
  - Often this is done by dividing data first and then determine what computations are associated with each piece of data; it is typical to do this with a focus on the primary data structures
  - Alternatively, we can be function driven, dividing the computation into pieces and then associating data with each computation part
- In either case, the goal is to find as many **primitive tasks** as possible.

- Desirable attributes
  - **Flexible** – There are orders of magnitude more tasks than available processors
  - **Efficient** – Redundant computations and data are minimized
  - **Balanced** – Primitive tasks are roughly the same size
  - **Scalable** – The number of tasks increases with problem size

## Foster’s Design Methodology – Communication

- Determine the communication patterns between tasks
    - **Local communication** refers to cases where a task needs to communicate with a small number of other tasks; this is often done by creating communication channels from the suppliers to the consumer
    - **Global communication** refers to cases where a significant number of primitive tasks must contribute data in order to perform some computation – MAX is a very good example; one paradigm for managing this is the use of middleware in the form of a blackboard or message queue

- Communication is parallel overhead in that this is not needed for non-parallel (single task) computation

- Desirable attributes
    - **Locality** – Tasks communicate with a small number of neighbors
    - **Balanced** – Communication is balanced among tasks
    - **Communication Concurrency** – Communications can be overlapped
    - **Computation Concurrency** – Computations can be overlapped

## Foster’s Design Methodology – Agglomeration (1)

- Determine how to group tasks to improve performance or simplify design / programming
    - o Sometimes we want more consolidated tasks than processors, putting several per node – mapping is a major issue here
    - o Sometimes we want one consolidated tasks per processor; this is especially true in SPMD environments such as clusters with message passing – mapping is trivial with one processor per task

- Reduction in communication overhead is a major goal of agglomeration
    - o Agglomerating tasks that communicate removes communication overhead – this is called “increasing locality”
    - o Combining tasks that are by their nature sequential (one await output from the other) is usually a good start
    - o Of course, combining groups of sending and receiving tasks can also be effective if the senders can group their messages together in order to reduce the accumulated latency associated with many small messages

## Foster’s Design Methodology – Agglomeration (2)

- Desirable attributes
  - **Locality** – Does agglomeration increase locality?
  - **Tradeoff** – Is redundant computation less costly than replaced communication?
  - **Scalability** – Is replication of computation or data not a hindrance when problem size grows?
    Is the number of tasks an increasing function of problem size?
  - **Balance** – Are combined tasks of similar size (computation and communication)?
  - **Matching** – Are the number of tasks as small as possible, but at least as large as the number of processors likely to be available?
  - **Economical** – Is the cost of modifying existing sequential code must be reasonable

## Foster's Design Methodology – Mapping (1)

- Assign tasks to processors
  - On a centralized multiprocessor, this is done automatically
  - Thus, we assume a distributed memory parallel computer
- Our goal is to maximize processor utilization and minimize communication overhead.
  - Processor utilization is the percentage of time the processor is actively executing tasks necessary to complete the computation – a busy wait is not an example of a necessary activity; its inclusion is to remedy a mismatch or contention induced by the chosen design
  - Mapping communicating tasks to the same processor reduces communication
  - Increasing processor utilization can conflict with minimizing communication
    - Example, if we reduce communication to 0 by mapping all tasks to 1 out of p available processors, then processor utilization is 1/p
  - Optimal mapping is NP complete (we'll study this later)
- Approaches to management can include
  - Centralized – Pool processors with one manager who assigns tasks
  - Distributed – Each peer keeps its own tasks and, when overloaded, pushes some out to be picked up by others; again a blackboard or shared queue might be used
  - Static – Assign once and be happy
  - Dynamic – Assign based on dynamically generated tasks

## Foster’s Design Methodology – Mapping (2)

- Checklist
    - Did you investigate one task versus multiple tasks per processor?
    - Did you investigate both static and dynamic mapping strategies?
    - If you chose dynamic allocation, are you sure that the manager is not a bottleneck?
    - If you chose static allocation, is the ratio of tasks to processors at least one order of magnitude?

## Traces

State, history, properties

s1 $\rightarrow$ s2 $\rightarrow$ s3 ... $\rightarrow$ sk
    trace or history states; can have many traces in concurrent system

states are altered by atomic action

safety property : never enter a bad state

liveness property : eventually enter a good state

mutual exclusion is a safety property

partial correctness is a safety property

termination is a liveness property (finite histories)

total correctness is both a safety and liveness property

## Notation

co s1; // s2; // ... // sn; oc : concurrency

co [i=low to high] s;

process name { ... } : background process

< S; > : atomic action; critical section; mutual exclusion; granularity considerations

< await(B); > : conditional synchronization; barrier synchronization

< await(B) S; > : conditional atomic action

{ precondition } actions { postcondition } : basis for axiomatic proofs of correctness

## Max (some trial runs)

```
function max1
    int m = 0;
    for i = 0 to n-1
        if (a[i] > m) m = a[i];
end { max1 }

function max2
    int m = 0;
    co [i = 0 to n-1]
        if (a[i] > m) m = a[i];
end { max2 }

function max3
    int m = 0;
    co [i = 0 to n-1]
        < if (a[i] > m) m = a[i]; >
end { max3 }

function max4
    int m = 0;
    co [i = 0 to n-1]
        if (a[i] > m) < m = a[i]; >
end { max4 }
```

## Max in Concurrent Notation

The key here is that many conditions will probably be false, and so the guarded action will never even be executed. Doing just the atomic test will destroy all concurrency. Employing no guards will lead to a random selection among the candidates for max. Guarding just the assign will have the same undesirable result.

```
function max
    int m = 0;
    co [ i = 0 to n-1 ]
        if (a[i] > m)
            < if (a[i] > m)
                m = a[i]; >
end { max }
```

## Critical References

**Critical reference is one changed by another process**

**At Most Once Property (x = e); appearance of atomicity**

> **e contains at most one critical reference**
> > **and x is not read by any other process; OR**
>
> **e contains no critical references**

## Critical References (examples)

1. **int y = 0, z = 0;**
   **co x = y + z; // y = 1; z = 2; oc;**
   **Two critical references – result is {x=[0,1,2,3], y=1, z=2} even though there is no time when the state of system could have y+z equal to 2.**

2. **int x = 0, y = 0;**
   **co x = x + 1; // y = y + 1; oc;**
   **No critical references – result is {x=1, y=1}**

3. **int x = 0, y = 0;**
   **co x = y + 1; // y = y + 1; oc;**
   **One critical reference, but x not read by other – results {x=1, y=1},{x=2, y=1}**

4. **int x = 0, y = 0;**
   **co x = y + 1; // y = x + 1; oc;**
   **One critical reference per statement, and each assigned in other**
   **– results {x=1, y=2},{x=2, y=1},{x=1, y=1}**
   **okay since there is a state in which the expressions x+1 and y+1 could simultaneously be 1, even though does not satisfy at most once property**

## Await

< S; > : atomic action; critical section; mutual exclusion; granularity considerations

< await(B); > : conditional synchronization; barrier synchronization

< await(B) S; > : conditional atomic action

Example:

```
      Producer/Consumer – one item buffer (p is producer index, c is consumer index)
      Initially p = 0, c = 0;
      P:  forever {<await (p = = c);>  buf = next_produced; p++;}
      C: forever {<await (p > c);>  next_consumed = buf; c++;}
```

If implement await as spin loop, might do above as

```
      forever {while (p > c); buf = next_produced; p++;}
      forever {while (p <= c); next_consumed = buf; c++;}
```

This is a busy wait; common on parallel machines and at lower levels of architecture

## Avoiding Interference among Concurrent Processes

**Disjoint Variables**

**Make sure reference set of each process differs from write set of others**

**Weakened assertions**

**Often just do your best at the time you make a decision**
**Analogy to greedy algorithms**

**Global invariants**

**Property of shared variables that is preserved across all assignments**
**Analogy to Domain analysis in OO**

**Synchronization**

**Making statements atomic avoids exposing inner states**

## Fairness

**Unconditional Fairness:**

**Every unconditional eligible atomic action is eventually executed**

**Weak Fairness**

**Unconditionally fair; OR**

**Every conditional eligible atomic action is eventually executed, provided the condition becomes true and stays true until the atomic action is executed**

**Strong Fairness (an impractical consideration)**

**Unconditionally fair; OR**

**Every conditional eligible atomic action is eventually executed, assuming the condition is infinitely often true – this means if the condition is always guaranteed to return to true if ever it cycles from true to false then the atomic action will eventually be executed**

## Critical Section Problem

**Mutual Exclusion**

**At most one process in a critical section**

**Absence of Deadlock (Livelock)**

**If two or more processes want a critical section, at least one will succeed**

**Absence of Unnecessary Delays**

**A process ready to use an uncontested critical section will not be delayed**

**Eventual Entry**

**Every process that wants a critical section will eventually get it**

## Spin Locks

**We now want to consider how to implement the < ... > primitive of text**

**How do we handle code like <await (!lock) lock = true;> critical; lock = false;?**

**Test and Set from IBM 360/67 2 processor machine**

```
    while (TS(lock)) skip; // returns entry value of lock (before this set)
        < boolean initial=lock; lock=true; return initial; >
```

**Problems**

**one memory cycle -- basically an atomic spin lock**

**no guarantee of fairness**

**results in serious memory contention for shared lock**

```
    while (lock) skip; while (TS(lock)) { while (lock); skip} // Test and Test and Set
```

**reduces memory contention**

## Implementing Critical Sections

**To implement unconditional atomic action < S; >**

```
    CSEnter; // CSEnter is entry protocol
    S;
    CSExit; //CSExit is exit protocol
```

**To implement conditional atomic action <await (B) S; >**

```
    CSEnter;
    while (!B) { CSExit; [delay]; CSEnter; } // delay may be omitted
    S;
    CSExit;
    If B satisfies at most once property can do < await(B);> as while(!B);
```

**Relation to Java synchronized**

```
    synchronized (lock) { S; }
        is like <S;> // every process uses same lock object
    synchronized (lock) { while (!B) try{wait();}catch(...){} S; notify(); }
        is like <await(B) S;>
```

## Assignment # 3.1

**1.** Consider the following **“solution”** to the critical section problem for **n** processes:

```
shared boolean lock=false;
shared boolean waiting[1:n] = ([n] false); // all slots are false
process p [i=1 to n] {
    while (some continuation condition is true for process i) {
        while (lock) { waiting[i] = true; while (waiting[i]) delay(); }
        lock = true;
        // critical section
        lock = false;
        // wake up at most one process
        for (j=1; j<=n; j++) if (waiting[j]) { waiting[j] = false; break; }
        // non-critical stuff
    }
}
```

Which of the following does this achieve? Answer **Yes** or **No** for each and give a one sentence justification.

Mutual exclusion ____________

Avoidance of deadlock __________

Avoidance of livelock ____________

Absence of unnecessary delays _______

Eventual entry ____________

## Assignment # 3.2

**2.** In each of the following, specify which Fairness criteria (unconditional, weak and/or strong) guarantee that the statement **S** is eventually executed? Check all applicable columns.

| **Statements** | **unconditional** | **weak** | **strong** |
|---|---|---|---|
| **int x=0; co <await(x == 5) S; > // while (true) x = x+1; oc** | | | |
| **int x=0; co <S; > // while (true) x = x+1; oc** | | | |
| **int x=0; co <await(x == 5) S; > // while (true) x = (x<6 ? x+1 : x); oc** | | | |
| **int x=0; co <await(x == 5) S; > // while (true) x = (x<10 ? x+1 : 0);oc** | | | |

## Assignment # 3.3

**3.** Consider the following program

```
int x=0;
co
    <await (x != 0) x = x – 1; >    # S1
    // <await (x == 0) x = x – 4; >        # S2
    // <await (x != 0) x = x + 3; > # S3
oc
```

Does the program meet the "At-Most-Once Property"? Explain your answer.

What are the possible final values of **x**? Show all traces.

Suppose the **await** statements are replaced by non-atomic **if** statements, but the assigns become atomic.

```
int x=0;
co
    if (x != 0)  <x = x – 1; >          # S1
    // if  (x != 0) <x = x + 3; >       # S2
    // if  (x == 0) <x = x – 4; >       # S3
oc
```

What are the possible final values of **x**? Show all traces.

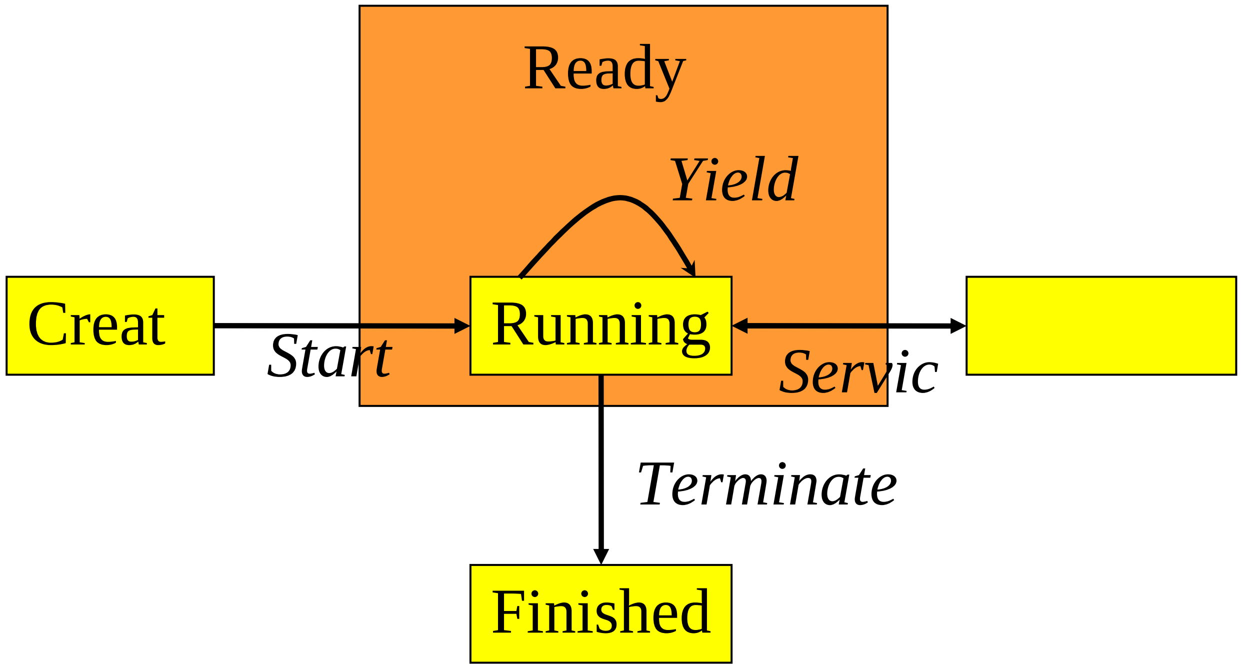
## Threads

- **Thread**
  - **A thread is a sequentially executed stream of commands**
  - **Usually lightweight**

- **A thread is not a process**
  - **Runs within the resources allocated to a process**
    - **It does have its own execution context (stack, program counter, register)**
  - **Multiple threads of a process share resources and can refer to common objects**

- **Every process has at least one thread**

## Thread Life Cycle



- **Create – new thread is allocated**
- **Running – thread is executing**
- **Ready – thread is waiting for the processor**
- **Blocked – thread is waiting on a requested service**
- **Finished – thread has been stopped and deallocated**

## Java Threads

- Like everything else, a thread is an instance of a class.
  - Thread
    - Part of the java.lang package
    - Provides basic behaviors
      - Starting
      - Stopping
      - Sleeping
      - Yielding
      - Priority management
      - A simple form of monitors
- The run method, by default is empty
- There are two ways
  - Subclass Thread and override run
  - Implement the Runnable interface

## Sample Java Threads

```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str); //sets thread’s name
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " +getName());
            try {
                sleep((int)(Math.random() * 1000)); // one second delay
            } catch (InterruptedException e) { }
        }
        System.out.println("DONE! " + getName());
    }
}

public class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("UCF").start();
        new SimpleThread("Knights").start();
    }
}
```

## Creating and Starting Threads

- **Creating a thread instances the class**
  - **No resources have been allocated to it yet**
  - **From here, the only thing you can do is call the *start* method**

  ***myThread = new Thread(this,"My Thread");***
    - **Used as part of the Runnable interface**
    - **"this" sets the context for the thread**
    - **"My Thread" sets the name**

  ***myThread = new MyThreadClass("My Thread");***
    - **Used when inheriting**
    - **Context is set to the current object**

- **Starting a thread**

  ***myThread.start();***
  - **Allocates the resource**
  - **Starts the thread running**
  - **Returns control to existing thread**

## Ways to Delay

- **Yielding the processor**
  ***myThread.sleep(1000);***
  - **Sleeps the number of milliseconds**
  - **Will not run even if processor becomes available**
  - ***sleep(0)* sees if anything else is ready to run**
  - **can also use *yield( )***

- **Blocked**
  - **Waiting on I/O**
    - **System will schedule thread when the data is available**
  - **User defined**
    - **Most often waiting on a shared piece of code or data structure**

- **Busy Wait**
  ***while (okToProceed == false) { };***
  - **Technically stays in runnable state**
  - **Program is constantly checking to see if a resource has become available**
  - **Not a good thing to do with one processor**

## Thread or Runnable?

- Thread is a class
    - Java supports only single inheritance

- Runnable is interface
    - So it can be multiply inherited

- So, use Runnable when you have to inherit from some other class
    - This is required for multithreading in applets

## Synchronization

- Mutual exclusion of threads.
    - Each synchronized method or statement is guarded by an object.
    - When entering a synchronized method or statement, the object will be locked until the block is finished.
    - When the object is locked by another thread, the current thread must wait.

## Granularity

- Synchronized method:

```
class MyClass{
   synchronized void aMethod(){
      statements
   }
}
```

- Synchronized block:

```
synchronized(exp){
   statements
}
```

## Threads and Applets

- Applets commonly use threads

- Rule of thumb, if you are going to do something that is going to take a while, spawn off a thread to do it.
  - Loading images or sound files
  - Playing sounds

- Implement the Runnable interface to use threads in an applet

- There is  a main thread, the *Event Thread*
  - This is the one that the browser talks to
  - It is the only one that will stop by default when you leave the page
  - You must stop all the others

- If you do complex actions in the Event Thread, your applet becomes totally non-responsive. – DON’T DO IT!!!!

## A “Racy” Sort

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
import java.util.*;

public class EOSort extends JApplet implements Runnable {

	private static int N = 8;
	private static int MAX_DELAY = 500; // half second
	private static int MAX_VALUE = 50; //
	private Thread[ ] threads;
	private int[ ] values;
```

## Prepare Sort Values and Threads (Start/Stop)

```
public void init() {
        Random r = new Random();
        String parm = getParameter("N");
        if (parm != null) N = Integer.parseInt(parm);
        values = new int[N];
        for (int i=0; i<N; i++)
                values[i] = r.nextInt(MAX_VALUE) + 1;
}

public void start() {
        threads = new Thread[N];
        for (int i=0; i<N; i++) {
          threads[i] = new Thread(this, Integer.toString(i));
          threads[i].start();
        }
}

public void stop() {
        for (int i=0; i<N; i++) threads[i] = null;
}
```

## Display State

```
public void paint(Graphics g) {
        Vector v = new Vector(N);
        for (int i=0; i<N; i++) v.add(new Integer(values[i]));
                g.clearRect(0, 0, getContentPane().getWidth(),
        getContentPane().getHeight());
        g.drawString(v.toString(), 0, 30);
}

private void swap(int i, int j) {
        int temp = values[i];
        values[i] = values[j];
        values[j] = temp;
}
```

## Run Each Thread

```
     public void run() {
        while (threads[0] != null) {
            try {
               Thread.sleep(
                   (int)(MAX_DELAY*Math.random()));
            } catch (InterruptedException e) {}
            int me = Integer.parseInt(Thread.currentThread().getName());
            int left = Math.max(0,me-1);
            int right = Math.min(N-1,me+1);
            boolean change = false;
            if (values[me] > values[right]) {swap(me, right);change=true;}
            if (values[me] < values[left]) {swap(me, left);change = true;}
            if (change) repaint();
        }
     }
}
```

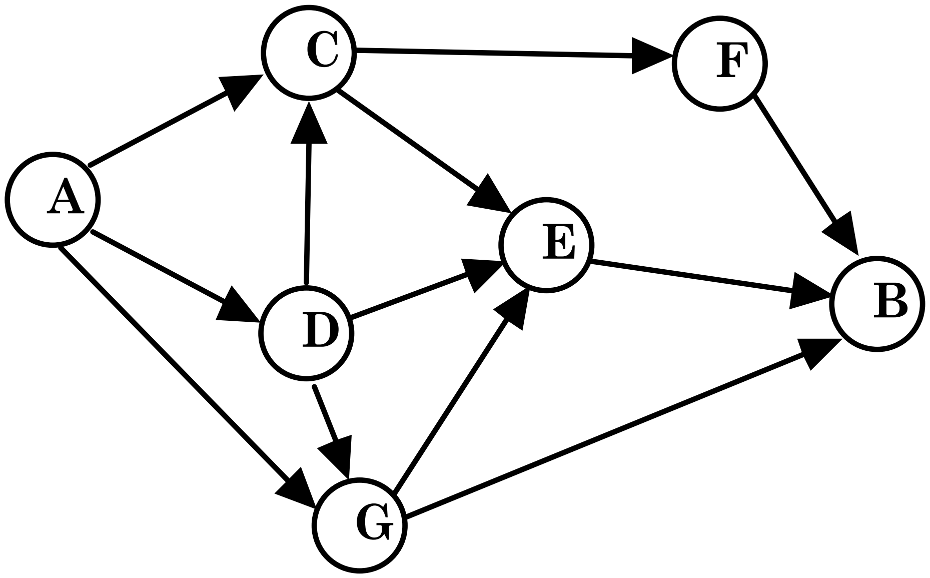
## Thoughts About “Racy” Sort

- Thought Exercise
  - Starting from the Java even/odd implementation, add delays (sleep) at various points to break the atomicity of my solution.
  - Discuss which placements of sleep cause semantic problems and which do not. Explain both as best you can.
  - Go back to EOSort. Design and re-implement it using critical section(s) – synchronized blocks. It's inherently unsafe right now!!

## Reflexive Transitive Closure

**The Problem:**

**Given a graph, G, determine for which pairs of nodes, (A,B), there is a path between A and B.**



**Array representation – 1 is True; 0 is False**

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **A** | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| **B** | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **C** | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| **D** | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| **E** | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| **F** | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| **G** | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

## Warshall’s Algorithm

```
public void warshallsAlgorithm() {
   //for each pivot try all pairs of nodes
   for (int pivot = 0; pivot < N; pivot++)
      for (int v = 0; v < N; v++)
         for (int w = 0; w < N; w++)
            // if (v != w)
            connectedMatrix[v][w] = connectedMatrix[v][w] ||
               (connectedMatrix[v][pivot] && connectedMatrix[pivot][w]);
}
```

**Analysis easily shows that this is O($N^3$).**

## Parallelizing Warshall’s Algorithm

**Can partition so we change two inner loops to**

```
    co (int v = 0; v < N; v++) (int w = 0; w < N; w++)
      connectedMatrix[v][w] = connectedMatrix[v][w] ||
        (connectedMatrix[v][pivot] && connectedMatrix[pivot][w]);
```

**This then can be carried out in O(N) time with $N^2$ processors. To do so, you would need a CREW PRAM style machine. The concurrent reads are needed to avoid contention. An alternative method, if we have just N processors is to run only the inner loop instances in parallel. This is a form of agglomeration. Here is the new algorithm.**

```
public void parallelWarshallsAlgorithm() {
  //for each pivot try all pairs of nodes
  for (int pivot = 0; pivot < N; pivot++)
    for (int v = 0; v < N; v++)
      co (int w = 0; w < N; w++)
        connectedMatrix[v][w] = connectedMatrix[v][w] ||
          (connectedMatrix[v][pivot] && connectedMatrix[pivot][w]);
}
```

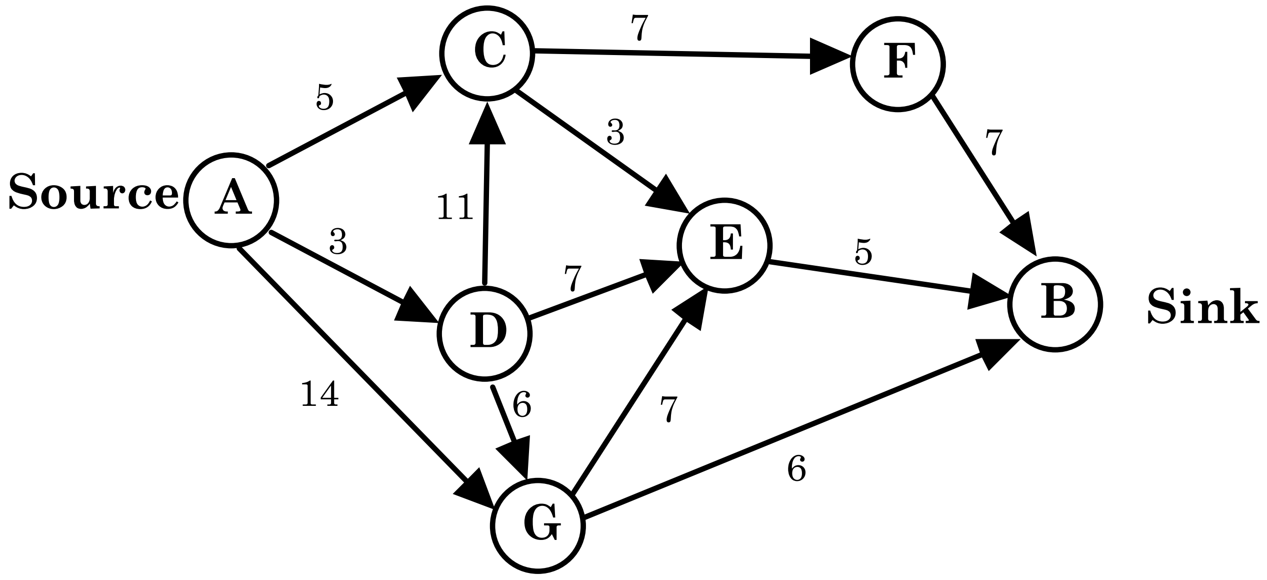
**This is $O(N^2)$. Again, we need a CREW PRAM.**

## Weary Traveler – Shortest Path

### The Problem:

**Given a graph (a dag), G, with weighted arcs, and two nodes, A and B, determine the minimum weight path from A to B.**

**Greedy fails here: Get 3 + 6 + 6 = 15; but can get 5 + 3 + 5 = 13**



### Array representation

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **A** | 0 | ∞ | 5 | 3 | ∞ | ∞ | 14 |
| **B** | ∞ | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| **C** | ∞ | ∞ | 0 | ∞ | 3 | 7 | ∞ |
| **D** | ∞ | ∞ | 11 | 0 | 7 | ∞ | 6 |
| **E** | ∞ | 5 | ∞ | ∞ | 0 | ∞ | ∞ |
| **F** | ∞ | 7 | ∞ | ∞ | ∞ | 0 | ∞ |
| **G** | ∞ | 6 | ∞ | ∞ | 7 | ∞ | 0 |

## Floyd’s All Shortest Paths Algorithm

```
final int INFINITY = Integer.MAX_VALUE; // choose value not used in weights

private boolean connected(int v, int w) {
   return adjacencyMatrix[v][w] != INFINITY)
}

public void floydsAlgorithm() {
   for (int pivot = 0; pivot < N; pivot++)
      for (int v = 0; v < N; v++)
         for (int w = 0; w < N; w++)
            if (connected(v,pivot) && connected (pivot,w))
               adjacencyMatrix[v][w] =
                  Math.min(
                        adjacencyMatrix[v][w],
                        adjacencyMatrix[v][pivot] + adjacencyMatrix[pivot][w] );
}
```

**Analysis again shows that this is $O(N^3)$.**
**Parallelization follows as with Warshall’s.**

## Multi-Threaded Implementation

**Let’s consider the case where we have N threads, each holding a row of the adjacency matrix.**

**Can the threads progress independently?**

**If not, when must they synchronize?**

**Will threads interfere with each other?**

**If so, will the interference lead to incorrect results?**

## Barrier Synchronization in Java

```
public class Barrier {
  private int count;
  // Barrier Constructors
  // Default just coordinates one thread (rather meaningless)
  public Barrier() {
    this(1);
  }
  public Barrier (int count) {
    setCount(count);
  }
  // Set count of number of threads to coordinate
  public void setCount(int n) {
    count = n;
  }
  // Block at the barrier until all workers have joined
  // Critical Region -- Must be synchronized
  synchronized public void join() {
    count--;
    while (count > 0)
      try {
        wait();
      } catch (InterruptedException e) {System.out.println(e);}
    notifyAll();
  }
}
```

## More on Parallelizing Floyd’s All Shortest Paths Algorithm

```
final int INFINITY = Integer.MAX_VALUE; // choose value not used in weights

private boolean connected(int v, int w) {
   return adjacencyMatrix[v][w] != INFINITY)
}

public void floydsAlgorithm() {
   [co | for] (int pivot = 0; pivot < N; pivot++)
      [co | for] (int v = 0; v < N; v++)
         [co | for] (int w = 0; w < N; w++)
            if (connected(v,pivot) && connected (pivot,w))
               adjacencyMatrix[v][w] =
                  Math.min(
                        adjacencyMatrix[v][w],
                        adjacencyMatrix[v][pivot] + adjacencyMatrix[pivot][w] );
}
```

## Inspecting Case # 1

```
for (int pivot = 0; pivot < N; pivot++)
   for (int v = 0; v < N; v++)
      co (int w = 0; w < N; w++)
         if (connected(v,pivot) && connected (pivot,w))
            adjacencyMatrix[v][w] =
               Math.min(
                     adjacencyMatrix[v][w],
                     adjacencyMatrix[v][pivot] + adjacencyMatrix[pivot][w] );
```

**This is best run with N processors, each being assigned an element from row w on which it works. We will need to do $N^2$ barrier synchronizations, one per pivot/row pair. There is no read contention for the [v][w] elements (except for the cases where v or w equals pivot and that can be avoided with a judicious if clause), but there is contention for the pivot elements. Fortunately, there is no write contention, which is helpful if we are depending on cache. There appears to be a problem of reading an area that is changing when either v or w is the pivot; however such an element would never get an improvement from using itself as the pivot.**

## Inspecting Case # 2

```
for (int pivot = 0; pivot < N; pivot++)
   co (int v = 0; v < N; v++)
      for (int w = 0; w < N; w++)
         if (connected(v,pivot) && connected (pivot,w))
            adjacencyMatrix[v][w] =
               Math.min(
                     adjacencyMatrix[v][w],
                     adjacencyMatrix[v][pivot] + adjacencyMatrix[pivot][w] );
```

**This is best run with N processors, each being assigned a row on which it works. We will need to do N barrier synchronizations, one per pivot value. The rest of the discussion matches that of case#1.**

## Inspecting Case # 3

```
for (int pivot = 0; pivot < N; pivot++)
   co (int v = 0; v < N; v++)
      co (int w = 0; w < N; w++)
         if (connected(v,pivot) && connected (pivot,w))
            adjacencyMatrix[v][w] =
               Math.min(
                     adjacencyMatrix[v][w],
                     adjacencyMatrix[v][pivot] + adjacencyMatrix[pivot][w] );
```

**This is best run with $N^2$ processors, each being assigned a single element from the adjacency matrix. This will take N barrier synchronizations, one per pivot. There is lots of read contention.**

## Inspecting Case # 4

```
co (int pivot = 0; pivot < N; pivot++)
   for (int v = 0; v < N; v++)
      for (int w = 0; w < N; w++)
         if (connected(v,pivot) && connected (pivot,w))
            adjacencyMatrix[v][w] =
               Math.min(
                     adjacencyMatrix[v][w],
                     adjacencyMatrix[v][pivot] + adjacencyMatrix[pivot][w] );
```

**Well, we can use N processors, but this is not valid since we do multiple pivot points in parallel. It can be made so by encompassing the nested loop with a test to see if any changes were made on the most recent iteration. You keep iterating until convergence.**

## Inspecting Case # 5

```
co (int pivot = 0; pivot < N; pivot++)
   for (int v = 0; v < N; v++)
      co (int w = 0; w < N; w++)
         if (connected(v,pivot) && connected (pivot,w))
            adjacencyMatrix[v][w] =
               Math.min(
                     adjacencyMatrix[v][w],
                     adjacencyMatrix[v][pivot] + adjacencyMatrix[pivot][w] );
```

**Well, we can use $N^2$ processors, but this again is not valid since we do multiple pivot points in parallel.**

## Inspecting Case # 6

```
co (int pivot = 0; pivot < N; pivot++)
   co (int v = 0; v < N; v++)
      for (int w = 0; w < N; w++)
         if (connected(v,pivot) && connected (pivot,w))
            adjacencyMatrix[v][w] =
               Math.min(
                     adjacencyMatrix[v][w],
                     adjacencyMatrix[v][pivot] + adjacencyMatrix[pivot][w] );
```

**Well, we can use $N^2$ processors, but this again is not valid since we do multiple pivot points in parallel.**

## Inspecting Case # 7

```
co (int pivot = 0; pivot < N; pivot++)
   co (int v = 0; v < N; v++)
      co (int w = 0; w < N; w++)
         if (connected(v,pivot) && connected (pivot,w))
            adjacencyMatrix[v][w] =
               Math.min(
                     adjacencyMatrix[v][w],
                     adjacencyMatrix[v][pivot] + adjacencyMatrix[pivot][w] );
```

**This can use $N^3$ processors, but it fails to solve the problem. This is still interesting, since it may be that convergence is quick. Here is the changed program.**

```
boolean changed = true;
while (changed) do {
   changed = false;
   co (int pivot,v,w in [<0,..,N-1>,<0,..,N-1>,<0,..,N-1>])
      if (connected(v,pivot) && connected (pivot,w))  {
         int trial = adjacencyMatrix[v][pivot] + adjacencyMatrix[pivot][w];
         if  (trial < adjacencyMatrix[v][w]) {
            changed = true; adjacencyMatrix[v][w] = trial;
         }
      }
   }
```

## Analyzing the Parallel Max

Parallel Algorithm.

```
For k = 1 to lg n do
    For i = 1 to n/2^k pardo
        A [i]  :=  max(A[2*i], A[2*i+1]);
Largest :  = A[1].
```

p = no. of processors

Time

$T_1 = n - 1$

$T_{n/2} = \log n$

$\text{Cost} = p \times T$

$\text{Cost}_1 = n - 1 \approx n = Ø(n)$

$\text{Cost}_{n/2} = n/2 \lg n = Ø(n \lg n)$

Speed up.

$Sp = T_1/Tp = (n–1)/\lg n = Ø(1 / \lg n)$

Not efficient.

## Being Efficient

Can we do better?

No. of elements = n
# of processors = p
# of elements assigned to each processor = n/p
So, $2 \le n/p \le n$
Since # of elements in each processor is n/p

$T_{seq} = (n/p–1)$ [(m - 1) comparisons for max. of m elements]

Once all the p processors have found out their respective maximums, the parallel computation takes over. With p processes in action, the time to find the maximum takes lg p time.

$T_{par} = \lg p$
$T_{tol} = T_{par} + T_{seq}$
$= \lg p + n/p – 1$
with p = 1. $T_{tol} = \lg 1 + n/1 - 1 = n - 1$
$p = n/2$ $T_{tol} = \lg n/2+n/(n/2) - 1 = \lg n/2 + 1 = \lg n – 1 + 1 = \lg n$

## The Right Number of Processors

What is a good value of p? It is one that brings down the cost to match that of the sequential algorithm and still gains on computational time complexity.

Let $p = n/\lg n$

# of elements in each processor = $n/(n/\lg n) = \lg n$

$T_{seq} = \lg n - 1$
$T_{par} = \lg p = \lg (n/\lg n) = \lg n - \lg \lg n$ (negligible)

$T_{tol} = \lg n - 1 + \lg n - \lg\lg n$

$\approx 2 \lg n - 1 = Ø ( \lg n)$

Cost = $Ø (\lg n) * n/\lg n = Ø (n)$

## Example of Brent's Scheduling

Example    n = 256    lg n = 8    p = n/lg n = 32

Each processor gets 8 elements, so

$T_{seq} = 8 - 1 = 7$

p = 32, so

$T_{par} = \lg 32 = 5$

$T_{tol} = 7 + 5 = 12.$

Cost = 32 ∗ 12 = 384

Now if p = n/2 = 256/2 = 128,

Each processor gets 2 elements, so

$T_{seq} = 1$ and

Tpar = log 128 = 7

Ttol = 7 + 1 = 8

Cost = 128 ∗ 8 = 1024

Using p = n/lg n processors for sequencing is called Brent's scheduling.

## Parallel Binary Tree Reduction Algorithm

$T_P(N) = O(N/P + \lg P)$
$C_P(N) = O(N + P \lg P)$
$W_P(N) = O(N + P) = O(N)$, provided P is O(N).
$EC_P(N) = O(1/(1 + P \lg P / N))$
$EW_P(N) = O(1/(1 + P/N) = O(1)$, provided P is O(N).

$T_N(N) = O(\lg N)$
$C_N(N) = O(N \lg N)$
$W_N(N) = O(N)$.
$EC_N(N) = O(1/( \lg N))$
$EW_N(N) = O(1)$.

$T_{N/\lg N}(N) = O(\lg N)$
$C_{N/\lg N}(N) = O(N)$
$W_{N/\lg N}(N) = O(N)$.
$EC_{N/\lg N}(N) = O(1)$
$EW_{N/\lg N}(N) = O(1)$.

## Parallel CRCW Max Algorithm

Super Fast CRCW Algorithm:

$T_{N^2}(N) = O(1)$

$C_{N^2}(N) = O(N^2)$

$W_{N^2}(N) = O(N^2)$.

$EC_{N^2}(N) = O(1/N)$

$EW_{N^2}(N) = O(1/N)$.

## How Do We Make Access Fair -- Tie Breaker

**Tie Breaker**

```
boolean in1=false, in2=false; int last=1; //
CS1: last=1; in1=true; <await(!in2 or last==2);> S; in1=false;
CS2: last=2; in2=true; <await(!in1 or last==1);> S; in2=false;

boolean in1=false, in2=false; int last=1; //
CS1: last=1; in1=true; while (in2 and last==1) delay; S; in1=false;
CS2: last=2; in2=true; while (in1 and last==2) delay; S; in2=false;
```

**This does not scale very well to n participants.**

**Need to break tie with all n-1 of them before proceeding.**

## How Do We Make Access Fair -- Ticket

**Ticket Algorithm**

```
    int number=1, next=1, turn[1:n] = ([n] 0);
    process CS[I=1 to n]
        …
        <turn[i] = number; number++;>
        <await (turn[i] == next);>
        S;
        <next++;>
        …
```

**The exit protocol can be non-atomic since only one process can execute it at a time. The entry protocol is more problematic.**

**Many machines have an atomic add (Fetch and Add / FA) that returns the old value.**

```
        <turn[i] = number; number++;>
        <await (turn[i] == next);>
    becomes
        turn[i] = FA(number, 1)
        while (turn[i] != next) delay();
```

## Barrier Synchronization

**Shared Counter**

```
    <count++;>
    <await(count==n);>
    or
    FA(count, 1);
    while (count != n) delay(small);
```

**Flags and Coordinators**

```
    int arrive[1:n]  ([n] 0), continue[1:n] = ([n] 0);
    process Worker[i=1 to n] {
        do task i;
        arrive[i] = 1;
        <await (continue[i] == 1);>
        continue[i] = 0;
    }
    process Coordinator {
        while (true) {
            for [int i=1 to n] {
                <await (arrive[i] == 1);>
                arrive[i] = 0;
            }
            for [int i=1 to n] continue[i] = 1;
        }
    }
```

## Combining Tree Barrier Synchronization

**Flags and coordinators has two drawbacks**

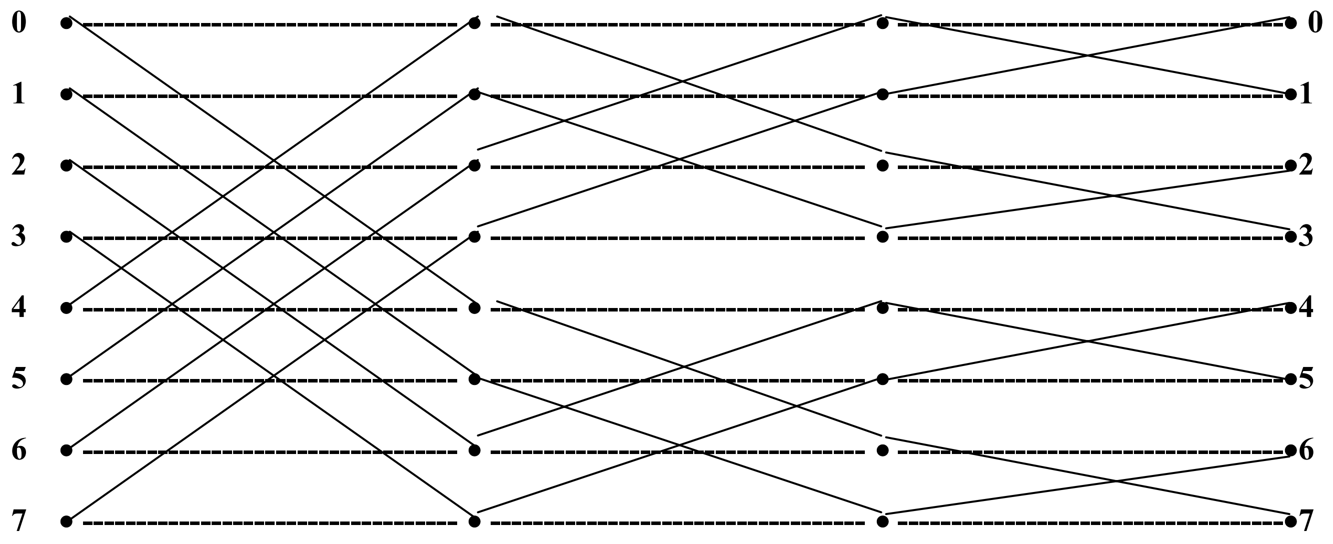
- **Extra process to coordinate (a referee of sorts)**
- **Coordinator's effort is proportional to number of workers**

**Overcome by making each worker serve as a coordinator**

- **Organize workers into a tree**
  - **some are leaves; some interior; one is a root**
- **Plus: propagation is lg N (up and down tree)**
- **Minus: this destroys symmetry of workers**

## Symmetric Butterfly Barrier Synchronization

**Can use Butterfly structure (a dynamic version of Hypercube)**



**In our problem we just think of the lines as communication between nodes, starting from right, moving left. In general this provides a lg N scheme for any processor to communicate with any other, just like a hypercube. Implementation in hardware is through switches that can be pass through or cross over.**

## Symmetric Dissemination Barrier Synchronization

**The butterfly technique just described requires $2^k$ processors, for some k.**

**Can set up a pattern that communicates to node+1, node+2, node+4, …, treating the sequence as a ring. This does not depend on a power of two.**

**Example n=5:**

**{ {0,1}, {1,2}, {2,3}, {3,4}, {4,0} }**
**{ {0,2}, {1,3}, {2,4}, {3,5}, {4,1} }**
**{ {0,4}, {1,0}, {2,1}, {3,2}, {4,3} }**

**Process sets the arrival flag of right neighbor and waits for and then clears its own arrival flag. Time is $\lceil \lg N \rceil$ (ceiling rounds up).**

**Both butterfly and dissemination barrier can lead to a race.**

**Consider butterfly. Assume process 0 arrives at its first stage and sets its flag arrive[0]. Suppose process 1 is slow. Suppose 2 and 3 arrive at barrier and set each other's flags, clear them and proceed to the next stage. In stage 2, process 2 wants to synchronize with process 0, whose arrive flag is set. So process 2 clears arrive[0] and proceeds to stage 3. Process 1 never knows that arrive[0] was set for it. Some processes will exit too soon; others will never exit. Solution is to use a non-boolean for arrive's value. This is count of number of stages at which task has arrived.**

## Data Parallel

**Data Parallel on SIMD machine**
- **MasPar Examples**
  - **Don't need barrier synchronization if on SIMD machine**

**Parallel Prefix in lg N time (N lg N cost, N work)**

```
plural int s, a;
s ← a;
for j ← 0 to ⌈log n⌉ 1 do
    if ( myProcNumber >= 2^j  )
        s ← s + proc[myProcNumber – 2^j  ].s;
```

**Parallel operations on linked lists**
- **Computing length of list headed by each element in lg N time**

```
plural int length = 1;
plural int partner = next; // linked list of processor numbers
while (partner != null) {
    length = length+ proc[partner].length;
    partner = proc[partner].partner;
}
```

## Bag of Tasks

**Bag of Tasks parallel strategy**

- **Each thread involved just grabs a task from the bag and executes it**
- **Can also have separate bags for each thread**
  - **Tasks in a single bag are run non-concurrently**

**Use in Java Event thread**

```
SwingUtilities.invokeLater (new Runnable() { // asynchronous
        public void run() { … } }
// Cannot do synchronous call from Event thread -- think about it
SwingUtilities.invokeAndWait(new Runnable() { // synchronous
        public void run() { … } }
```

## Communication and Coordination

**SIMD Machine**

**Coordinates via a single master node**

**Communicates over a high speed, low latency dedicated network**

**SIMD Model**

**Limits you to data parallelism**

**Encourages/Forces you into regular communication patterns**

**MIMD**

**Requires some means of coordination (synchronization)**

**Allows you flexibility in your communication paradigms**

**Distributed Systems**

**Involve the greatest challenges for coordination**

**Provide the most flexibility for communication paradigms**

**Place a burden due to variable speed, high latency shared network**

## Maspar X-Net Adds

```
/*************************************************************************
This program illustrates the use of DPU timing functions.
*************************************************************************/
#include <mpl.h>
#include <stdio.h>
extern void dpuTimerStart();
extern double dpuTimerElapsed();

int SlowAvg(src)
plural int src;
{
    int i;
    for (i=0; i<100; i++) {
        src += xnetN[1].src + xnetS[1].src + xnetE[1].src + xnetW[1].src +
               xnetNE[1].src +xnetNW[1].src +xnetSE[1].src +xnetSW[1].src;
        src = src/9;
    }
}
int FasterAvg(src)
plural int src;
{   int i;
    for (i=0; i<100; i++) {
        src += xnetN[1].src + xnetS[1].src;
        src += xnetE[1].src + xnetW[1].src;
        src = src/9;
    }
}
```

```
int EvenFasterAvg(src)
plural int src;
{
    register int i;
    register plural int tmp;
    for (i=0; i<100; i++) {
        tmp = src;
        tmp += xnetN[1].tmp + xnetS[1].tmp;
        tmp += xnetE[1].tmp + xnetW[1].tmp;
        src = tmp/9;
    }
}
main()
{
    plural int A;
    int sum,i;
    double time;
    /* initialize the array A */
    A = iproc;
    dpuTimerStart();
    SlowAvg(A);
    A = iproc;
    dpuTimerStart();
    FasterAvg(A);
    A = iproc;
    dpuTimerStart();
    EvenFasterAvg(A);
}
```

## Semaphores

**Abstraction with two services P (wait) and V (signal)**

```
sem s;
P(s): <await(s>0) s--;>
V(s): <s++;>
```

**Internal state is a non-negative int value -- counting or general semaphore; or**
**a binary value (0 or 1) -- binary semaphore**
**fairness can be assured with proper implementation of await.**

**Critical section problem and semaphores**

```
sem mutex = 1;
process CS[i=1 to n] {
    ...
    mutex.wait(); critical section; mutex.signal();
    ...
}
```

## Semaphores and Java synchronized

Each object can have a field called mutex (mutual exclusion)

```
sem mutex = 1;

synchronized(Object obj):
    obj.mutex.p(); // used p since wait means something else in Java
    // body of synchronized code
    obj.mutex.v();
```

This won't quite work since Java's locks are reentrant.
We should not do p() or v() if we own lock.

Java's wait()/notify() might be implemented by adding another field

```
sem waitingtask = 0;
```

and implementing the services by code like

```
wait():
    obj.mutex.v();obj.waitingTask.p(); obj.mutex.p();
notify():
    obj.waitingTask.v();
```

Note: wait() and notify() are actually done by native code in Java.

## Semaphores and Barriers -- one of many solutions

```
sem done=0, barrier=0;

process workers[i=1 to n] {
    while (true) {
        // do task i
        done.signal();
        barrier.wait();
    }
}

process coordinator {
    while (true) {
        // wait for all n tasks
        for [i=1 to n] done.wait();
        // let all n tasks through barrier
        for [i=1 to n] barrier.signal();
    }
}
```

## Semaphores and Producer/Consumer Problem

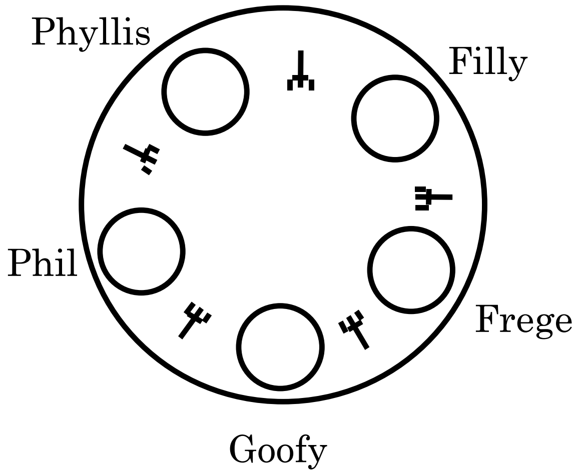
```
typeT buf[n];
int front=0, rear=0;
sem empty=n, full=0, mutexD=1, mutexF=1;

process producer[i=1 to nProducers] {
    while (true) {
        // produce data to deposit in buffer
        empty.wait(); mutexD.wait();
        buf[rear] = data; rear = (rear+1) % n;
        mutexD.signal(); full.signal();
    }
}

process consumer[i=1 to nConsumers] {
    while (true) {
        full.wait(); mutexF.wait();
        result = buf[front]; front = (front+1) % n;
        mutexF.signal(); empty.signal();
        // consume the result just fetched
    }
}
```

## Dining Philosophers

**In our variant, there will be 5 philosophers sitting at a table with dishes and 5 forks. Each philosopher performs the sequence**

```
loop
    think;
    /* somehow get forks */
    eat;
end loop;
```

**To eat, a philosopher must have two forks, one from each side of the plate. A solution must try to avoid deadlock and starvation, yet retain fairness**



## Semaphores and Dining Philosopher Problem

```
sem fork[n] = ([n] 1);

// a solution that can lead to starvation
process philosopher[i=0 to n-1] {
    while (true) {
        // think and get hungry
        fork[i].wait(); fork[(i+1)%n].wait();
        eat();
        fork[i].signal(); fork[(i+1)%n].signal();
    }
}
```

## Semaphores and Dining Philosopher Problem (2)

```
sem fork[n] = ([n] 1);

process philosopher[i=0 to n-1 by 2] { // even numbered
    while (true) {
        // think and get hungry
        fork[i].wait(); fork[(i+1)%n].wait();
        eat();
        fork[(i+1)%n].signal(); fork[i].signal();
    }
}


process philosopher[i=1 to n-1 by 2] { // odd numbered
    while (true) {
        // think and get hungry
        fork[(i+1)%n].wait(); fork[i].wait();
        eat();
        fork[i].signal(); fork[(i+1)%n].signal();
    }
}
```

## Diners Club – Probabilistic Attack

```
sem fork[n] = ([n] 1);

process philosopher[i=0 to n-1] {
    while (true) {
        // think and get hungry
        boolean success = false;
        while (!success) {
            int first = (i + randomChoice(0, 1))%n;
            fork[first].wait();
            int second = (first==i) ? (i+1)%n : i;
            success = fork[second].tryToGet(); // check and decrement if can
            if (!success) fork[first].signal();
        }
        eat();
        fork[second].signal();fork[first].signal();
    }
}
```

**It can then be proved that the probability of livelock is 0 in above.**

## Semaphores and Readers/Writers Problem – too constrained

```
sem rw = 1;

process reader[i=1 to nReaders] {
    while (true) {
        // want to read
        P(rw);
            // read database
        V(rw);
    }
}

process writer[i=1 to nWriters] {
    while (true) {
        // want to write
        P(rw);
            // write database
        V(rw);
    }
}
```

## Semaphores and Readers/Writers Problem – readers rule

```
int nr = 0; // number of active readers
sem rw = 1; // semaphore for DB
sem mutexR = 1; // semaphore for nr
process reader[i=1 to nReaders] {
    while (true) {
        // want to read
        P(mutexR);
            if (++nr==1) P(rw) // first reader through grabs DB lock
        V(mutexR);
            // read database
        P(mutexR);
            if (--nr == 0) V(rw); // last one out locks the door
        V(mutexR);
    }
}
process writer[i=1 to nWriters] {
    while (true) {
        // want to write
        P(rw);
            // write database
        V(rw);
    }
}
```

## Semaphores and Readers/Writers – pass baton (coarse grain)

```
// Invariant: (nr == 0 || nw == 0) && nw <= 1
int nr = 0, nw = 0; // number of active readers/writers
process reader[i=1 to nReaders] {
    while (true) {
        // want to read
        <await(nw == 0) nr++;>
            // read database
        <nr--;>
    }
}
process writer[i=1 to nWriters] {
    while (true) {
        // want to write
        <await(nr == 0 && nw == 0) nw++;>
            // write database
        nw--;
    }
}
```

## Semaphores and Readers/Writers– pass baton (fine grain)#1

```
// Invariant: (nr == 0 || nw == 0) && nw <= 1
int nr = 0, nw = 0; // number of active readers/writers
sem  e = 1, // entry to critical region
     r = 0, w = 0; // used to delay readers/writers
int dr = 0, dw = 0; // number of delayed readers/writers
process reader[i=1 to nReaders] {
     while (true) {
          // want to read
          // <await(nw == 0) nr++;>
          P(e);
          if ( nw > 0 ) { dr++; V(e); P(r); }
               // or if ( nw > 0 || dw > 0 ) { dr++; V(e); P(r); } // update quicker
          nr++;
          if ( dr > 0 ) {dr--; V(r);}
          else V(e)
               // read database
          // <nr--;>
          P(e);
          nr--;
          if ( nr==0 && dw > 0 ) { dw--; V(w); }
          else V(e);
     }
}
```

## Semaphores and Readers/Writers– pass baton (fine grain)#2

```
process writer[i=1 to nWriters] {
    while (true) {
        // want to write
        // <await(nr == 0 && nw == 0) nw++;>
        P(e);
        if ( nr > 0 || nw > 0 ) { dw++; v(e); P(w); }
        nw++;
        V(e);
            // write database
        P(e);
        // <nw--;>
        if ( dr > 0 ) { dr--; V(r); }
        else if ( dw > 0 ) { dw--; V(w); }
        else V(e);
        // or if ( dw > 0 ) { dw--; V(w); }
        //     else if ( dr > 0 ) { dr--; V(r); }
        //     else V(e);
    }
}
```

## Simple Monitor Overview

**Monitor is by its very nature exclusive – it is synchronized for all**

**Threads entering a monitor must check their conditions to be sure they can productively move forward in the monitor.**

**cond is a queue at which threads wait.**

- **empty(cv) – true if cv is empty**
- **wait(cv) – thread waits at rear of cv**
- **signal(cv) – awakens thread at head of cv (if non-empty)**
    - **can do as SW signal and wait (signaled process gets preference)**
    - **or as SC signal and continue (more common, where signaler goes on)**

```
monitor Semaphore {
    int s = 0; cond sQueue;
    procedure P() { while (s == 0) wait(sQueue); s--; } // use if for SW
    procedure V() { s++; signal(sQueue); }
}
```

## Monitors and Bounded Buffer

```
monitor BoundedBuffer {
    typeT buf[n];
    int front=0, rear=0, count=0;
    cond notFull, notEmpty;
    procedure deposit(typeT data) {
        while (count == n) wait(notFull);
        buf[rear] = data;
        rear = (rear+1) % n;
        count++;
        signal(notEmpty);
    }
    procedure fetch( typeT &result) {
        while (count == 0) wait(notEmpty);
        result = buf[front];
        front = (front+1) % n;
        count--;
        signal(notFull);
    }
}
```

## Monitors and Readers/Writers

```
monitor ReaderWriter {
    int nr=0, nw=0;
    cond okRead, okWrite;
    procedure requestRead() {
        while (nw > 0) wait(okRead);
        nr++;
    }
    procedure releaseRead() {
        if (--nr == 0) signal(okWrite);
    }
    procedure requestWrite() {
        while (nr >0 || nw > 0) wait(okWrite);
        nw++;
    }
    procedure releaseWrite() {
        nw--;
        signal(okWrite); signal_all(okRead);
    }
}
```

## Monitors – Queue Management

**All our prior monitors assume FIFO queue management.**

**We now consider cases where priority queues are desirable.**

**Extend services on monitor to include**

> **wait(cv, priority) – thread waits in cv based on priority**
> **minrank(cv) – return priority of top thread in cv**

**Note: signal acts appropriately on a min queue, releasing highest priority (lowest ranked) thread.**

**Can use min heap to avoid O(N) inserts or perhaps deletes. Compare:**

**Unsorted – wait O(1); signal O(N); signal_all O(N); minrank O(N)**
**Sorted – wait O(N); signal O(1); signal_all O(N); minrank O(1)**
**Heap – wait O(lg N); signal O(lg N); signal_all O(N); minrank O(1)**
**Since #wait >= #signal, heap is best**

## Monitors and Shortest Job First Scheduling

```
monitor ShortestJobFirst {
    bool free = true;
    cond turn;
    procedure request (int time) {
        if (free)
            free = false;
        else
            wait(turn, time);
    }
    procedure release () {
        if  (empty(turn))
            free = true;
        else
            signal(turn);
    }
}
```

## Monitors and Sleep Timer (Covering Condition)

```
monitor Timer {
    int tod = 0;
    cond check;
    procedure sleep (int interval) {
        int wakeup = tod + interval;
        while (wakeup > tod) wait(check);
    }
    procedure tick () {
        tod++;
        signal_all(check); // everyone must check covering condition
    }
}
```

## Monitors and Sleep Timer (Priority Wait)

```
monitor Timer {
    int tod = 0;
    cond check;
    procedure sleep (int interval) {
        int wakeup = tod + interval;
        if (wakeup > tod) wait(check, wakeup);
    }
    procedure tick () {
        tod++;
        while (!empty(check) && minrank(check) <= tod)
            signal (check); // if awakened, condition is met
    }
}
```

## Monitors and Sleeping Barber (Rendezvous Approach)

```
monitor BarberShop {
    bool barberReady = false, customerWaiting = false, doorOpened = false;
    cond barberAvail, chairOccupied, doorOpen, customerLeft;
    procedure getHaircut() {
        while (!barberReady) wait(barberAvail); barberReady = false;
        customerWaiting = true; signal(chairOccupied);
        while (!doorOpened) wait(doorOpen); doorOpened = false;
        signal(customerLeft);
    }
    procedure getNextCustomer() {
        barberReady = true; signal(barberAvail);
        while (!customerWaiting) wait(chairOccupied); customerWaiting = false;
    }
    procedure finishedHaircut {
        doorOpened = true; signal(doorOpen);
        while (doorOpened) wait(customerLeft);
    }
}
```

## Disk Scheduling Algorithms

**SST – Shortest Seek Time First**

**Very unfair**

**What is queue management strategy?**

**SCAN – Elevator Algorithm (also called LOOK)**

**Fair but can be have large variances**

**What does queue data structure look like for this?**

**CSCAN – Like elevator but takes no one down (also called CLOOK)**

**Fair and does not have large variances**

## Scan Disk Scheduling – Separate Scheduler Monitor

```
monitor DiskScheduler { // CSCAN
    int position = -1, c = 0, n = 1;
    cond scan[2]; // queues for each direction
    procedure request(int cyl) {
        if (position == -1) position =cyl;
        else if (cyl > position) wait(scan[c], cyl);
        else wait(scan[n], cyl);
    }
    procedure release() {
        int temp;
        if (!empty(scan[c])  position – minrank(scan[c]);
        else if (!empty(scan[n]) {
            temp = c; c = n; n = temp; position = minrank(scan[c]);
        }
        else position = -1;
        signal(scan[c]);
    }
}
```

## Monitors and One-Way Bridges -- !!non-exclusion!!

```
monitor Bridge {
    cond northbound, southbound; // conds are waiting stations
    int northOnBridge, southOnBridge;
    procedure enterSouthbound( ) {
        if (northOnBridge>0) wait(southbound);
        southOnBridge++;
    }
    procedure leaveSouthbound( ) {
        southOnBridge--;
        if (southOnBridge==0) signal_all(northbound);
    }
    // + northbound versions
    // note: the bridge crossing is not in monitor
}
```

## Monitors and One-Way Bridges -- !!impolite!!

```
monitor Bridge {
    cond northbound, southbound;
    int northOnBridge, southOnBridge;
    procedure enterSouthbound( ) {
        while (northOnBridge>0) wait(southbound);
        southOnBridge++;
    }
    procedure leaveSouthbound( ) {
        southOnBridge--;
        if (southOnBridge==0) signal_all(northbound);
    }
    // + northbound versions
    // note: the bridge crossing is not in monitor
}
```

## Monitors and One-Way Bridges – Oops accident again!!

```
monitor Bridge {
    cond northbound, southbound;
    int northOnBridge, southOnBridge;
    procedure enterSouthbound( ) {
        if ((northOnBridge>0) || !empty(northbound)) wait(southbound);
        southOnBridge++;
    }
    procedure leaveSouthbound( ) {
        southOnBridge--;
        if (southOnBridge==0) signal_all(northbound);
    }
    // + northbound versions
    // note: the bridge crossing is not in monitor
}
```

## Monitors and One-Way Bridges -- !!too polite -- deadlock!!

```
monitor Bridge {
    cond northbound, southbound;
    int northOnBridge, southOnBridge;
    procedure enterSouthbound( ) {
        while ((northOnBridge>0) || !empty(northbound)) wait(southbound);
        southOnBridge++;
    }
    procedure leaveSouthbound( ) {
        southOnBridge--;
        if (southOnBridge==0) signal_all(northbound);
    }
    // + northbound versions
    // note: the bridge crossing is not in monitor
}
```

## Monitors and One-Way Bridges -- ??just right??

```
monitor Bridge {
    cond northbound, southbound;
    int northOnBridge, southOnBridge;
    procedure enterSouthbound( ) {
        while ((northOnBridge>0) ||
          ((southOnBridge>0) && !empty(northbound))) wait(southbound);
        southOnBridge++;
    }
    procedure leaveSouthbound( ) {
        southOnBridge--;
        if (southOnBridge==0) signal_all(northbound);
    }
    // + northbound versions
    // note: the bridge crossing is not in monitor
}
```

## Monitors and One-Way Bridges -- ??better??

```
monitor Bridge {
    cond northbound, southbound;
    int northOnBridge, southOnBridge;
    procedure enterSouthbound( ) {
        if ((northOnBridge>0) || !empty(northbound)) wait(southbound);
        while (northOnBridge>0) wait(southbound);
        southOnBridge++;
    }
    procedure leaveSouthbound( ) {
        southOnBridge--;
        if (southOnBridge==0) signal_all(northbound);
    }
    // + northbound versions
    // note: the bridge crossing is not in monitor
}
```

## Java Support for Monitors

- Synchronize : specifies critical section using an object as lock
  - can do at granularity of method
  - can do at granularity of a block
- Java synchronized, wait/notify/notify_all
- Locks are reentrant
- Locks can be temporarily given up : wait and notify

## Paths as a Declaration of Concurrency

**The granularity is at a method level. We say which methods can execute in parallel, and which are mutually exclusive.**

**Path expressions:**

**m – where m is a method is a path expression.**

**Let e1 and e2 be path expressions, then**

|  |  |
|---|---|
| **e1, e2** | **e1 and e2 can run in parallel** |
| **{ e1 }** | **0 or more of e1 can run in parallel** |
| **e1 ; e2** | **an instance of e1 must precede each e2** |
| **e1 + e2** | **e1 and e2 may not run in parallel** |
| **n: (e1)** | **up to n versions of e1 can run in parallel** |

**Consider**

```
   path (start_read; do_read) end
   path 1:( { do_read } + do_write ) end
   path 1:( start_read + { start_write ; do_write } ) end
```

**This gives exclusive write and non-exclusive read. Moreover, every do_read must be preceded by a start_read, and every do_write must be preceded by a start_write. Also, once write gets going, reads can't even start.**

**Note: plus above can be replaced by comma, since 1:( … ) controls concurrency.**

## Greedy – Basics

**Want to Max or Min some objective function. Solution must satisfy some feasibility constraint.**

**Any solution satisfying constraints is feasible.**

**A feasible solution that maxes or mins the objective is optimal.**

**Greedy solutions are often suboptimal, but always feasible.**

**For example, our First Fit never overfills a trunk, so it always return a feasible solution. Its solutions are, however, not guaranteed to be optimal.**

**General Form of Greedy Algorithm:**

```
solution := {};

FOR i:=1 to NumberOfChoices DO
    X := Select (A);  (* where Select is simple *)
    IF Feasible (Solution ∪ X) THEN
        Solution := Solution ∪ X
RETURN Solution
```

## Spanning Tree Problem

**Assume that G = (V, E), where G is an undirected graph, V is the set of vertices (nodes), and E is the set of edges.**

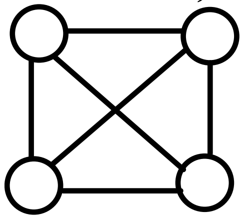
**A spanning tree of G is a subgraph which is a tree that encompasses all nodes in the original graph. Such a tree will commonly include just a subset of the original edges. Here, by tree, we mean a graph with no simple cycles. We ignore the normal designation of a root and we do not order nodes.**

**If G is a single connected component, then there is always a spanning tree.**

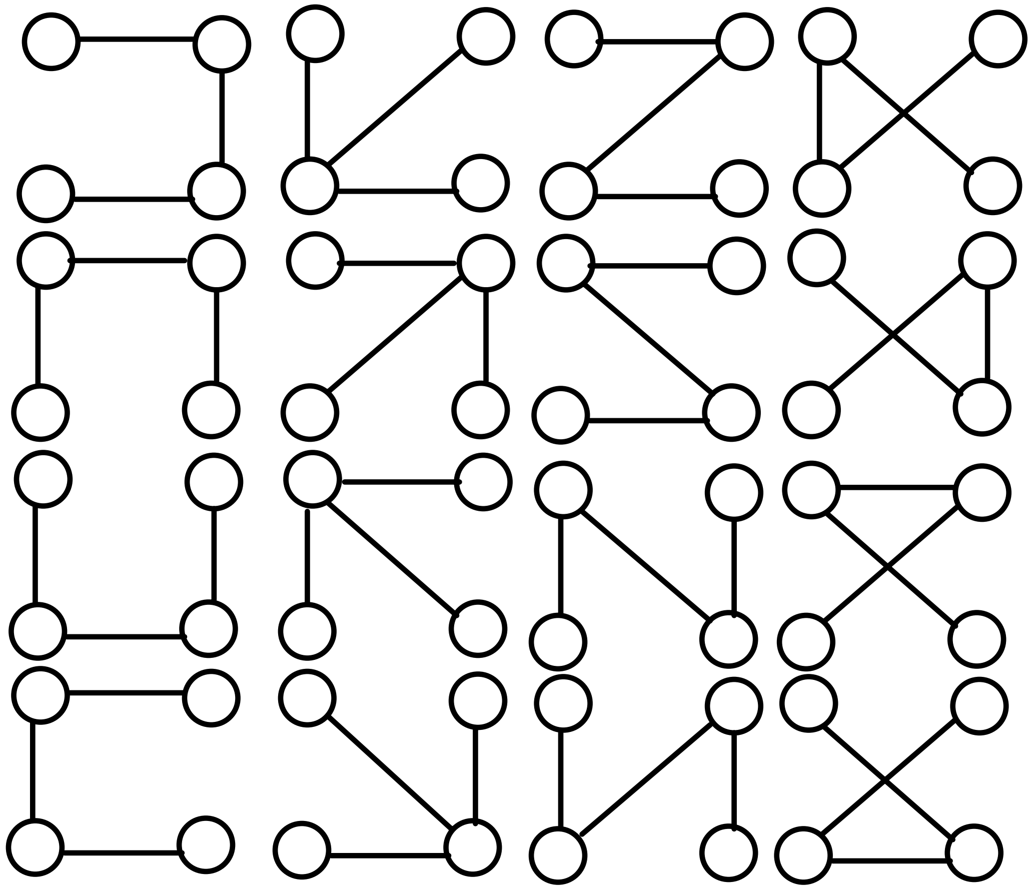
**Adding weights to edges gives us the minimum spanning tree problem, where we wish to span with edges whose sum is minimum among all spanning trees.**

## Spanning Trees are Everywhere

**Consider four nodes, fully connected as below,**



**The spanning trees are:**



## Min Spanning Tree–Prim's Algorithm

Weights could be distances, costs, signal degradation, …

Feasible – There are no simple cycles at every stage.

Greedy – We grab the closest node to one of the ones that has already been included.

There are lots of ways to implement Prim’s algorithm.

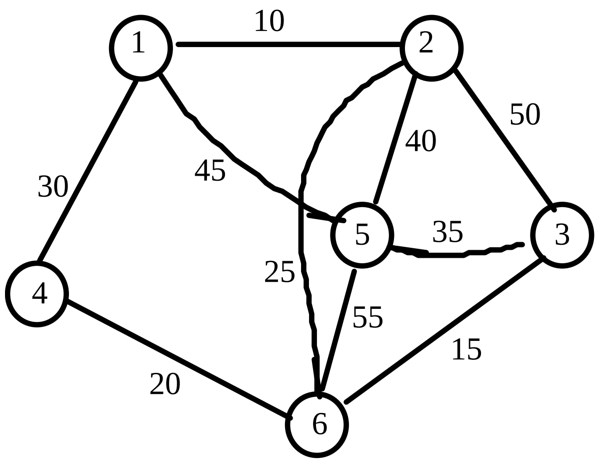
We will study an $O(N^2)$ way.

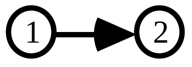
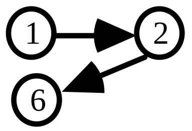
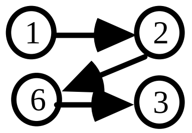
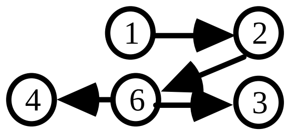
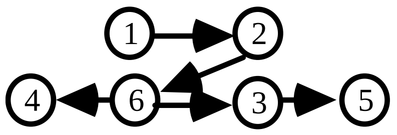
Other implementations are O(MlgN), where M = max (|E|, N)

## Min Spanning Tree–Prim's Algorithm

```
program PrimMinSpan;
var   N, j, k : Integer;
      Adjacency : AdjacencyMatrix;
      V : set of 1..MaxNodes;
      Dist, Source: Array [1..MaxNodes];
begin
   (* Assume N nodes, labeled 1 to N *)
   GetGraph(N, Adjacency);
   Dist := Adjacency[1];
   V := [2..N];
   Source[1] := 0;            { Root has no source }
   for j in V do
      Source[j] := 1;   { Distances are from root }
   while V <> [ ] do begin
      k := index in V with smallest value in Dist;
      V := V – [k];
      for j in V do
         if Dist[j] > Adjacency[k,j] then begin
            Dist[j] := Adjacency[k,j]; Source[j] := k
         end;
   end;
end.
```

# Applying Prim’s Algorithm



| Node | Dist/Source | Cost | Tree |
|---|---|---|---|
| 1 | [**0**/0,10/1,∞/1,30/1,45/1,∞/1] | | |
| 2 | [**0**/0,**10**/1,50/2,30/1,40/2,25/2] | 10 |  |
| 6 | [**0**/0,**10**/1,15/6,20/6,40/2,**25**/2] | 25 |  |
| 3 | [**0**/0,**10**/1,**15**/6,20/6,35/3,**25**/2] | 15 |  |
| 4 | [**0**/0,**10**/1,**15**/6,**20**/6,35/3,**25**/2] | 20 |  |
| 5 | [**0**/0,**10**/1,**15**/6,**20**/6,**35**/3,**25**/2] | 35 |  |

## Block-Striped Partitioning

Using p processors and N nodes.

Partition $N^2$ Adjacency matrix into p groups of N/p columns.

Partition Dist and Source into p groups of N/p elements.

Processor i, $1 \le i \le p$, must manage a block of Adjacency columns, and a block of Dist and Source elements, ranging from the (i-1)*(N/p)+1-th to the iN/p-th.

Need to initialize just N/p elements on each processor.

Min on each processor needs to be computed, and then a global min must be found (accumulation) and the index of this node reported (one to all broadcast).

After receiving index of min, each processor must update its share of Dist and Source lists.

This process continues until no more nodes are left to be selected.

## Analyzing Parallel Prim's Algorithm

Initialization time is just N/p.

The time to find a Min starts with N/p time for local mins, is followed by a single node accumulation, and then by a one-all broadcast of the selected node.

The time to update the Dist and Source lists is N/p.

The loop runs N times, and there is a **TRUE DEPENDENCY** between successive iterations of the loop.

The computation time is $O(N^2/p)$.

The communication time is dependent on the architecture. On a Hypercube, accumulation and one-all broadcast are both $O(\lg p)$. On a mesh, these times are $O(\sqrt{p})$.

$T_p$ (Hypercube) = $O(N^2/p) + O(N \lg p)$.

$T_p$ (Mesh) = $O(N^2/p) + O(N \sqrt{p})$.

E (Hypercube) = $1/(1 + p \lg p / N)$

E (Mesh) = $1/(1 + p^{1.5} / N)$

E (Hypercube) = $O(1)$ if $p = O(N/ \lg N)$

E (Mesh) = $O(1)$ if $p = O(N^{2/3})$

# Assignment # 1 Key

1. Consider the Max Tree we described earlier, but now use p processors to sort N values, where N may be greater than p. In this case, each processor uses a sequential algorithm to find its local max and then proceeds with the standard tree approach. Analyze the Time, Cost and Work for
    a. p = lg N
    b. p = lg lg N
    c. p = N / lg N
    d. p = N / lg lg N

| | **Time** $T_P(N)$ | **Cost** $C_P(N)$ | **Work** $W_P(N)$ | **Cost Efficiency** $EC_P(N)$ | **Work Efficiency** $EW_P(N)$ |
|---|---|---|---|---|---|
| **P = lg N** | **O(N/lg N)** | **O(N)** | **O(N)** | **O(1)** | **O(1)** |
| **P = lg lg N** | **O(N/lg lg N)** | **O(N)** | **O(N)** | **O(1)** | **O(1)** |
| **P = N / lg N** | **O(lg N)** | **O(N)** | **O(N)** | **O(1)** | **O(1)** |
| **P = N / lg lg N** | **O(lg N)** | **O(N lg N/ lg lg N)** | **O(N)** | **O(lg lg N/ lg N)** | **O(1)** |

2. Consider the problem of sorting a deck of cards into Spades, Hearts, Diamonds, Clubs, and in order Ace down to 2 within each suit.
    a. What is the best way for an individual to do this? There many metric and many answers.
    Pick each card and put it in a predetermined slot. Gather up. This requires 52 inspections and 52 fetches from table.
    b. Redo this but this time with five people. One of the five starts with all 52 cards.
    Original holder hands them out based on assigning one suit to each of the other 4. Each has thirteen pre-allocated slots. Original collects in batch suit. This requires 52 inspections, a parallel fetch of 13 cards and 4 fetches from partners.

# Assignment # 2 Key

a. Prove that each of the Butterfly and Omega networks connects node x to node y when the circuits are switched by choosing crossover at the i-th switch whenever the i-th most significant bit of $x \oplus y$ is 1.

   Butterfly: This starts with node x and successively, from left to right, complements (by using a crossover) the selected bit if the corresponding one in $x \oplus y$ is 1. Mathematically, we have that the result
   $x \oplus \vee\ (i = \log n - 1 \text{ to } 0)\ (2^i \wedge x \oplus y) = x \oplus (\ x \oplus y\ ) = (x \oplus x) \oplus y = 0 \oplus y = y$

   Omega: This starts with node x and successively complements the leftmost bit of (x shift left i circularly) if the left one in ($x \oplus y$ shift left i circularly) is 1. Mathematically, this is equivalent to the Butterfly, except that the potential complementing of a bit (due to crossover) only occurs when that bit becomes the leftmost one as a result of the successive left shifts at the end of each layer in the circuitry. The final shift just brings us back to the original permutation of the bits.

b. How many distinct potential communication pairs (*a* to *b*, *c* to *d*; $a \neq c$, $b \neq d$) exist for p=8? Of these, how many can occur in parallel without a conflict occurring at some switch? Answer this for both Butterfly and Omega switching networks.

   Butterfly or Omega: *a* can be any node that connect to any other node, so there are $p^2$ choices for *a*, *b*; *c* can any node but *a* and can connect to any of the remaining (p-1) destinations, so there are $(p-1)^2$ choices, but half of these were already seen (this occurs since there is not difference between the pair {(*a*,*b*}, (*c*,*d*)} and {(*c*,*d*), (*a*,*b*)}. Thus, the number of potential pairs is $p^2 \times (p-1)^2/2$, or 2 when p=2, 72 when p=4 and 1568, when p=8.

   Butterfly or Omega: The thing we cannot do is for the path from *c* to *d* to intersect the path from *a* to *b*. This is a restriction on the first part where we only limited intersection to the beginning and end nodes. One can see that a multistage network with lg p stages gives rise to lg p opportunities for intersection. Thus, the number of potential non-conflicting pairs is $p^2 \times (p-l) \times (p - \lg p)/2$, or 2 when p=2, 48 when p=4 and 1120 when p=8.

# Assignment # 3.1 Key

**1.** Consider the following **“solution”** to the critical section problem for **n** processes:

```
shared boolean lock=false;
shared boolean waiting[1:n] = ([n] false); // all slots are false
process p [i=1 to n] {
    while (some continuation condition is true for process i) {
        while (lock) { waiting[i] = true; while (waiting[i]) delay(); }
        lock = true;
        // critical section
        lock = false;
        // wake up at most one process
        for (j=1; j<=n; j++) if (waiting[j]) { waiting[j] = false; break; }
        // non-critical stuff
    }
}
```

Which of the following does this achieve? Answer **Yes** or **No** for each and give a one sentence justification.

Mutual exclusion ______NO______ Multiple threads can see !lock before one sets it true

Avoidance of deadlock ______NO______ Assume one thread sets lock and all others want to wait but are delayed at start of wait block; first resets lock but sees no one delayed, thus signaling no one; if the first thread’s condition is now false, it never reenters critical section and other threads are blocked forever.

Avoidance of livelock ______NO______ Threads can get stuck at while (waiting[i]) delay();

Absence of unnecessary delays __________NO Change deadlock case to have the first thread come back in later and actually wake some other thread; that thread could have gone earlier.

Eventual entry ______NO______ There is a biased wake up, always favoring lower numbered threads. This can lead to higher numbered ones being blocked forever,

## Assignment # 3.2 Key

**2.** In each of the following, specify which Fairness criteria (unconditional, weak and/or strong) guarantee that the statement **S** is eventually executed? Check all applicable columns.

| **Statements** | **unconditional** | **weak** | **strong** |
|---|---|---|---|
| **int x=0; co <await(x == 5) S; > // while (true) x = x+1; oc** | | | |
| **int x=0; co <S; > // while (true) x = x+1; oc** | **X** | **X** | **X** |
| **int x=0; co <await(x == 5) S; > // while (true) x = (x<5 ? x+1 : x); oc** | | **X** | **X** |
| **int x=0; co <await(x == 5) S; > // while (true) x = (x<10 ? x+1 : 0);oc** | | | **X** |

# Assignment # 3.3 Key

**3.** Consider the following program

```
int x=0;
co
    <await (x != 0) x = x – 1; >      # S1
    // <await (x == 0) x = x – 4; >  # S2
    // <await (x != 0) x = x + 3; >  # S3
oc
```

Does the program meet the "At-Most-Once Property"? Explain your answer.

**No – Just one critical reference per statement, but each assigns a value used by other**

What are the possible final values of **x**? Show all traces.

**{-2}   {S2, S1, S3} x = -2; {S2, S3, S1} x = -2;**

Suppose the **await** statements are replaced by non-atomic **if** statements, but the assigns become atomic.

```
int x=0;
co
    if (x != 0)  <x = x – 1; >          # S1
    // if (x != 0) <x = x + 3; >        # S2
    // if (x == 0) <x = x – 4; >        # S3
oc
```

What are the possible final values of **x**? Show all traces.

**{-1, -2, -4, -5}**
**{S1, S2, S3} x = -4; {S1, S3, S2} x = -1; {S1, S3a, S2, S3b} x = -4; {S2, S1, S3} x = -4; {S2, S3, S1} x = -5; {S2, S3a, S1, S3b} x = -4; {S3, S1, S2} x = -2; {S3a, S1, S3b, S2} x = -1; {S3a, S1, S2, S3b } x = -4; {S3, S2, S1} x = -2; {S3a, S2, S3b, S1} x = -5; {S3a, S2, S1, S3b } x = -4**

## Assignment # 4.2 Key

**2.** Suppose there are m producer processes and n consumer processes. The producer processes periodically call broadcast(message) to send a copy of message to all n consumers. Each consumer receives a copy of the message by calling fetch(message, myId), where message is a result argument and myId ranges from 0 to n-1. Write a monitor that implements broadcast and fetch. Use the Signal and Continue discipline. The monitor should store only one message at a time, which means that after one producer calls broadcast, any future call of broadcast has to delay until every consumer has received a copy of the previous message.

```
monitor PC {
    string next;;
    int need = 0;
    boolean ready[n]; // assume all false at start
    cond wantToBroadcast, wantToFetch;
    procedure broadcast(string msg) {
        // Must use “while” to prevent amore than one broadcaster from getting through.
        while (need > 0) wait(wantToBroadcast);
        next = msg; need = n;
        for (int i=0; i<n; i++) ready[i] = true;
        signal_all(wantToFetch);
    }
    procedure fetch(string msg, int myId) {
        // Can use “if” provided id’s are unique to threads.
        if (!ready[myId]) wait(wantToFetch);
        msg = next; ready[myId] = false; need--;
        if (need == 0) signal(wantToBroadcast);
    }
}
```

# Midterm#1 Topics and Promises

**Topics**

1. **Concurrent Programming Concepts**
2. **Introduction, even-odd transposition algorithm, analysis**
3. **Concepts of analysis of parallel algorithms**
   - Architectural considerations -- synchronous versus asynchronous; barrier synchronization; centralized control
   - Issues of communication and coordination in parallel and distributed implementations
   - Time, Cost, Speedup, Work, Cost Efficiency and Work Efficiency.
   - Virtualizing an algorithm -- focus on even-odd transposition.
4. **Taxonomies (control, address space, interconnection network, granularity)**
   - SIMD, MIMD, data versus task parallel
5. **Taxonomies (address space)**
   - private memory (separate address spaces), also called distributed memory, vs shared address space (often called shared memory). UMA (uniform / symmetric multiprocessors (SMP)) versus NUMA (non-uniform) memory access. Cache and the cache coherence (consistency) problem.
6. **Taxonomies (interconnection network)**
   - static vs dynamic interconnections.
   - Dynamic: butterfly, Omega, crossbar, bus-based networks.
   - Static: completely-connected, star connected, linear array, ring, 2-d mesh and torus, 3-d mesh and torus, tree, hypercube.
7. **Programming Styles**
   - Iterative parallelism: co // and process notation
   - Recursive parallelism
   - Producer / Consumer
   - Client / Server
   - Peers: worker, send and receive notation
8. **State, history, properties**
   - s1 -> s2 -> s3 ... ->sk : trace or history states; can have many traces in concurrent system
   - safety property : never enter a bad state
   - liveness property : eventually enter a good state

9. **Notation for concurrency**
    - co s1; // s2; // ... // sn; oc : concurrency
    - process name { ... } : background process
    - < S; > : atomic action; critical section; mutual exclusion; granularity considerations
    - < await(B) > : conditional synchronization; barrier synchronization
    - < await(B) S; > : conditional atomic action
    - { precondition } actions { postcondition } : basis for axiomatic proofs of correctness
10. **Java Support for Concurrency**
    - Threads : either inherit from Thread class or implement Runnable interface
    - Synchronized : specifies critical section using an object as lock
    - Locks are reentrant
    - Locks can be temporarily given up : wait and notify
11. **Critical References**
    - Critical reference is one changed by another process
    - At Most Once Property (x = e); appearance of atomicity
12. **Fairness**
    - Unconditional, Weak and String Fairness
13. **SpinLocks**
    - Critical section problem
        - mutual exclusion
        - absence of deadlock and livelock
        - absence of unnecessary delays
        - eventual entry (relates to fairness)
    - How do we handle code like <await (!lock) lock = true;> critical; lock = false;?
        - CSEnter: while (TS(lock)) delay; // returns entry value of lock (before this set)
    - To implement unconditional atomic action < S; >
        - CSEnter; S; CSExit; // CSEnter is entry protocol; CSExit is exit protocol
    - To implement conditional atomic action <await (B) S; >
        - CSEnter; while (!B) { CSExit; delay; CSEnter; } S; CSExit;
        - if B satisfies at most once property can do < await(B);> as while(!B);
    - Relation to Java synchronized
        - synchronized (lock) { S; } is like <S;> // in simple notation, every process uses same lock object
        - synchronized (lock) { while (!B) try{wait();}catch(...){} S; notify(); } is like <await(B) S;>
14. **Fair Solutions**
    - Tie Breaker
    - Ticket Algorithm

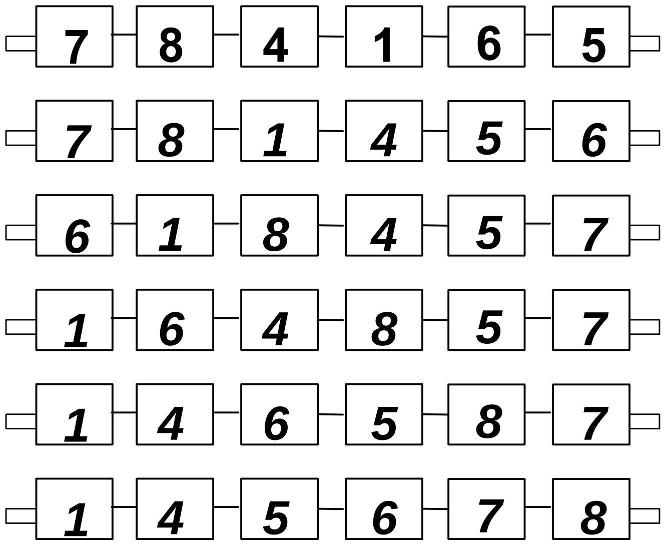
15. **Barrier Synchronization**
    - Shared Counter
    - Flags and Coordinators
    - Symmetric Barriers
16. **Data Parallel**
    - MasPar Example
    - Parallel Prefix and Parallel Linked List Length
17. **Semaphores**
    - Abstraction with two services P (wait) and V (signal)
    - Critical section problem and semaphores
    - Java synchronized and semaphores
    - Barriers and semaphores
    - Producer / Consumer Problem; Dining Philosophers Problem; Reader/Writer Problems
18. **Monitors**
    - monitors and conds
    - wait(cv), wait(cv, rank), signal(cv), signal_all(cv), empty(cv), minrank(cv)
        - signal and wait versus signal and continue
        - queues, priority queues, BPOTs, heaps and analysis
    - monitor examples
        - semaphores, bounded buffers, readers/writers, shortest-job-next, sleeping barber
        - CSCAN/SCAN disk scheduler (bitonic lists)
    - Java synchronized, wait/notify/notify_all
19. **Single lane bridge problem using semaphores and monitors**

**Promises**

1. A question on Even-Odd Transposition sort
2. A question on analysis of parallel algorithms
3. A question on taxonomies (control, address, interconnection)
4. A question on Java synchronized
5. A trace question on co s1; // s2; // ... // sn; oc
6. A question on locks
7. A question on fairness
8. A question on barriers
9. A question on semaphores (analysis, not synthesis)
10. A question on monitors

## Quiz#1 Sample#1

**1.** Easy Start ☺ Apply the even-odd parallel algorithm presented in class for sorting the **6** elements in the following ring of **6** processors. Show the results of each of the up to **5** passes that it takes to complete this ascending (low to high) sort.



**Initial Contents**

**After Pass 1**

**After Pass 2**

**After Pass 3**

**After Pass 4**

**After Pass 5**

## Quiz#1 Sample#2

**2.** In each of the following, specify which Fairness criteria (unconditional, weak and/or strong) guarantee that the statement **S** is eventually executed? Check all applicable columns. For some, the applicable criteria could include all, some or none.

| **Statements** | **unconditional** | **weak** | **strong** |
|---|---|---|---|
| **int x=0, y=0; co <await(x > y) S; > //<br>while (true) x = x+1; //<br>while (true) y = (y+1) % 10; oc** | | ***X*** | ***X*** |
| **int x=0, y=0; co <await(x == y) S; > //<br>while (true) x = x+1; //<br>while (true) y = (y+1) % 10; oc** | | | |
| **int x=0; y=0; co <await(x == y) S; > //<br>while (true) x = x+1; //<br>while (true) y = y+1; oc** | | | ***?*** |
| **int x=0, y=0; co <S; > //<br>while (true) x = x+1; //<br>while (true) y = y+1; oc** | ***X*** | ***X*** | ***X*** |

## Quiz#1 Sample#3

**3.** Briefly explain the meanings of notify() and notifyAll() in Java synchronized blocks. Differentiate one from the other.

***Each is used by a thread when it leaves a critical (synchronized) region. The purpose is to wake up thread(s) that are waiting for changes that might satisfy their conditional entries to the critical region. Notify wakes up just one thread (assuming at least one has issued a wait on the synchronization object. NotifyAll wakes up all threads waiting on this object for changes***.

## Quiz#1 Sample#4

**4.** Consider the following to solve the critical section problem:

```
var lock = 0;
process P[i=1 to n] {
    while true do {
        <await lock == 0>; lock = i;
        while (lock != i) do { <await lock == 0>; lock = i; }
        S1: // critical section
        lock = 0;
        S2; // non-critical section
    }
}
```

Does this ensure mutual exclusion? If so, why? If not, why not?

*No. P1 and P2 see lock equal to 0. P1 sets lock to 1and enters S1. P2 sets lock to 2 and enters S1.*

Does this approach avoid livelock? If so, why? If not, why not?

*Yes. When lock is not equal to 0, it is equal to the index of one of the Pi that wants the critical section. This one will pass through when lock is reset to 0. Thus, we never have a case where all who want the critical section are needlessly spinning.*

## Quiz#1 Sample#5

**5.** Fill in the following. All are about characteristics of parallel machines based on interconnection newtorks

**What is the diameter of**

A hypercube with 64 processors? ___6 = (lg 64)_

A wraparound mesh with 64 processors? ___8 = (2 * (√64)/2)__

A ring with 64 processors? ___32 = (64/2)___

A star network with 64 processors? __2 = (center then partner)__

**What is the bisection width of**

A hypercube with 64 processors? _32 = (everyone has partners)

A wraparound mesh with 64 processors? ___16 = (cut 8 + 8)___

A ring with 64 processors? ___2 = (cut linear and wrap)_

A star network with 64 processors? ___1 = (cut someone off)_

## Quiz#1 Sample#6

**6.** The following is the **Ticket Algorithm** for a Fair Critical Section solution. Add <…> where necessary to make parts of this atomic. Justify each addition. You need to have as little as possible forced to be atomic.

```
int number=1, next=1, turn[1:n] = ([n] 0);
process CS[i=1 to n] {
    while (true) {
        turn[i] = number++;
        await (turn[i] == next);
        S1; // critical section
        next++;
        S2; non-critical section
    }
}
```

***The italicized statement must be made atomic.***
*The first of the other statements satisfies the at-most-once property (turn[i] == next) and the second is in a place where only one process can be (the next++ right at the end of the critical section). However, <turn[i] = number++;> must be atomic, else the value of number could be seen as the* same by two processes (so they may enter S1 concurrently).

## Quiz#1 Sample#7

**7.** Consider the following program

```
int u=0, v=1, w=2, x;
co
        x = u + v + w;
        // <u = v + w;>
        // <v = 4;>
        // <w = 5;>
oc
```

Does the program meet the "At-Most-Once Property"? Explain your answer.

***No. x = u + v + w involves three variables modified by others processes.***

What are the possible final values of **x**? You need to be concerned about the fact that your compiler may take advantage of the commutivity of addition. Explain your answers.

***u = {0, 3, 6, 9}, v = {1, 4}, w = {2, 5}. Here u can be 0 or any of the sums of v and w, since all relative orderings are possible.***

***x = {3, 6, 9, 12, 15, 18}. Here x can be the sum of any combination of the possible values of u, v, w, since commutivity allows us to grab these in any order***.

## Quiz#1 Sample#8

**8.** We have looked at various ways to use **P** processors to quickly and efficiently find the largest element in a list **A[0…N–1]**. Regarding efficiency, we have sometimes focused on **Cost Efficiency** and other times on **Work Efficiency**. One of the early algorithms we looked at was based on **binary tree reduction**. Assuming this algorithm, fill in the following table for values of **P = 1**, **N/2** and **N/lg N**. I filled in the first row since I'm a nice guy, and it was real easy.

| | **Time** $T_P(N)$ | **Cost** $C_P(N)$ | **Work** $W_P(N)$ | **Cost Eff.** $EC_P(N)$ | **Work Eff.** $EW_P(N)$ |
|---|---|---|---|---|---|
| **P = 1** | **O(N)** | **O(N)** | **O(N)** | **O(1)** | **O(1)** |
| **P = N/2** | ***O(lg N)*** | ***O(N lg N)*** | ***O(N)*** | ***O(1/lg N)*** | ***O(1)*** |
| **P = N / lg N** | ***O(lg N)*** | ***O(N)*** | ***O(N)*** | ***O(1)*** | ***O(1)*** |

## Quiz#1 Sample#9

**9.** The **Unix** kernel provides two atomic operations similar to:

***sleep( )***: // *block the executing thread*

***wakeup( )***: // *awaken all blocked threads*

A call of **sleep** always blocks the caller. A call of **wakeup** awakens every thread that has called **sleep** since the last time that **wakeup** was called.

A call to **wakeup** should awaken all tasks that entered **sleep** prior to the time **wakeup** is started, but not any that arrive later. The following "solution" has a serious problem in that it can wake up the wrong tasks. Explain how this undesirable result can happen.

```
sem   e = 1, delay = 0; int count = 0;

sleep( ):     P(e) ;  count++;  V(e);  P(delay);
wakeup( ):  P(e);  while (count > 0) { count--; V(delay); } V(e);
```

*There is a race condition occurring in the sleep(), right after the V(e) (end of atomized) code and the P(delay) (sleep until awakened)*
*Here's the scenario:*
*P1 executes sleep() and gets just past V(e) – count == 1*
*P2 executes wakeup() performing one V(delay) – count == 0*
*P3 executes sleep() and gets o P(delay) before P1 – count == 1 and P3 awakened*
*P1 gets to P(delay) and sleeps*

*Results is P1 asleep and P3 awake, even though P1 was the only one that preceded P2's awake*

## Quiz#1 Sample#10

**10.** Write a **Barrier** monitor with two services **init(int n)** and **join**(). **Init** must be called just once, prior to any process being started that must use the barrier. The monitor must be reusable. You must state if you are using signal and wait (**SW**) or signal and continue (**SC**) semantics, and explain why you made the choice you did.

```
monitor Barrier {
    int expected, arrivals;
    cond queue;
    procedure int (int n) {
        expected = n; arrivals = 0;
    }
    procedure join () {
        if  (++arrivals == expected) {
            arrivals = 0; signal_all(queue); // SC or SW since done
        } else wait (queue)
    }
}
```

## Message Passing

**Alternative to shared memory**

**We can use messages for communication and coordination of processes, typically running on separate nodes in a network or cluster.**

**A cluster uses a dedicated network, sometimes with very low latency.**

**MPI is a standard C or C++ API. The specific messages have evolved, with lots of influence from researchers at Oak Ridge National Labs.**

- **Two key primitives:**
  - **Send**
  - **Receive**

## Key Message Passing Issues

- **Distinguishing messages**
  - **different applications**
  - **messages intended for other processes**
- **Reliability**
  - **reliability**
  - **sequencing**
- **Deadlock**

## Distinguishing Among Messages

- Criteria:
  - Communicator (application group -- a set of processes)
  - sender (process ID)
  - tag – user defined channel number

## Send and Receive

```
int MPI_Send(
      void              *data,          // obviously points to data
      int               count,          // how many units of data
      MPI_Datatype      datatype,       // e.g., MPI_INT
      int               destination,    // receiver pid
      int               tag,            // essentially a channel #
      MPI_Comm          communicator    // group
)
```

```
int MPI_Recv(
      void              *data,          // obviously points to data
      int               count,          // how many units of data
      MPI_Datatype      datatype,       // e.g., MPI_INT
      int               sender,         // sender pid
      int               tag,            // essentially a channel #
      MPI_Comm          communicator,   // group
      MPI_Status        *status         // receipt status
)
```

## Send and Receive Characteristics

- MPI Preserves Message Order
- MPI Guarantees (some) Message Integrity
- Type Conversions
  - converting between types
  - big-endian/little-endian issues
  - sending structures/classes
    - MPI_BYTE

- MPI_Send and MPI_Recv are blocking
  - block until data is available for read
  - block until it is safe to write to data
  - deadlock is possible

## Utility Services

- int MPI_Init ( int *argc, char **argv[]);
  - must be called before any other MPI function
- int MPI_Finalize ( void );
  - no MPI function can be called after MPI_Finalize
  - 
- Processes are assigned IDs (ranks)
  - consecutive integers starting with 0
- int MPI_Comm_size ( MPI_Comm communicator, int *process_count);
  - number of processes in communicator (all if MPI_COMM_WORLD)
- int MPI_Comm_rank ( MPI_Comm communicator, int *process_ID);
  - ID (rank) of the processor (in communicator)

## First MPI Program

```
#include “mpi.h”
#include <fstream.h>
void main(int argc, char *argv[]) {
   int pid;   // process ID
   int np;    // number of processes

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &pid);
   MPI_Comm_size(MPI_COMM_WORLD, &np);
   if(pid = = 1) {
      int data[3] = {1, 2, 3};
      MPI_Send(data, 3, MPI_INT, 0, 26,  MPI_COMM_WORLD);
   }
   if(pid = = 0) {
      int receive_data[100];
      MPI_Status status;
      MPI_Recv(receive_data,100,MPI_INT,1,26,MPI_COMM_WORLD,&status);
      cout<<“received data from p1.  First element is “<<receive_data[0]<<endl;
   }
   MPI_Finalize();
}
```

## MPI from Quinn's Text

**See Chapter 4 and Chapter 6 Notes (linked off of course web page).**

## Floyd’s Algorithm from Quinn (declarations)

```
#include <stdio.h>
#include <mpi.h>
#include "../MyMPI.h"
typedef int dtype;
#define MPI_TYPE MPI_INT

int main (int argc, char *argv[]) {
   dtype** a;        /* Doubly-subscripted array */
   dtype*  storage;  /* Local portion of array elements */
   int     i, j, k;
   int     id;       /* Process rank */
   int     m;        /* Rows in matrix */
   int     n;        /* Columns in matrix */
   int     p;        /* Number of processes */
   double  time, max_time;

   void compute_shortest_paths (int, int, int**, int); // should be dtype
```

## Floyd’s Algorithm from Quinn (main)

```
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &id);
  MPI_Comm_size (MPI_COMM_WORLD, &p);

  read_row_striped_matrix (argv[1], (void *) &a,
    (void *) &storage, MPI_TYPE, &m, &n, MPI_COMM_WORLD);

  if (m != n) terminate (id, "Matrix must be square\n");

  print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
    MPI_COMM_WORLD);
  MPI_Barrier (MPI_COMM_WORLD);
  time = -MPI_Wtime();
  compute_shortest_paths (id, p, (dtype **) a, n);
  time += MPI_Wtime();
  MPI_Reduce (&time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0,
    MPI_COMM_WORLD);
  if (!id) printf ("Floyd, matrix size %d, %d processes: %6.2f seconds\n",
    n, p, max_time);
  print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
    MPI_COMM_WORLD);
  MPI_Finalize();
}
```

## Floyd’s Algorithm from Quinn

```
void compute_shortest_paths (int id, int p, dtype **a, int n) {
  int  i, j, k;
  int  offset;   /* Local index of broadcast row */
  int  root;     /* Process controlling row to be bcast */
  int* tmp;      /* Holds the broadcast row */

  tmp = (dtype *) malloc (n * sizeof(dtype));
  for (k = 0; k < n; k++) {
    root = BLOCK_OWNER(k,p,n);
    if (root == id) {
      offset = k - BLOCK_LOW(id,p,n);
      for (j = 0; j < n; j++)
        tmp[j] = a[offset][j];
    }
    MPI_Bcast (tmp, n, MPI_TYPE, root, MPI_COMM_WORLD);
    for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
      for (j = 0; j < n; j++)
        a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
  }
  free (tmp);
}
```

## Adjacency Matrix Generator

```
#include <stdio.h>
int main(int argc, char *argv[]) {
        int     i,j,n, MAX, DEBUG=0;
        FILE *outfile;
        int     data;
        if (argc<4) {
                printf("Usage: generate nodes_in_graph max_value binary_output_file [DEBUG]\n");
                exit(1);
        }
        sscanf(argv[1], "%d", &n); sscanf(argv[2], "%d", &MAX);
        outfile = fopen(argv[3], "w");
        printf("… for graph with %d nodes into %s; MAX value is %d\n", n, argv[3], MAX);
        if (argc>4)
                if (strcmp(argv[4],"DEBUG") == 0) DEBUG = 1;
        fwrite(&n, sizeof(int), 1, outfile);
        fwrite(&n, sizeof(int), 1, outfile);
        for (i=0; i<n; i++) {
                if (DEBUG) printf("\n");
                for (j=0; j<n; j++) {
                        if (i == j) data = 0;
                        else data = rand() % (MAX+1);
                        fwrite(&data, sizeof(int), 1, outfile);
                        if (DEBUG) printf("%6d", data); // hopefully MAX is no greater than 99999
                }
        }
        if (DEBUG) printf("\n");
        fclose(outfile);
}
```

## Programming Assignment#2

Let $\lambda$ be latency and $\beta$ be bandwidth, then the time needed to send an n-byte message is $\lambda + n / \beta$. Write an MPI program to determine $\lambda$ and $\beta$ on the Zephyr cluster using the "ping pong" test. Design the program to run on exactly two processes. Process 0 records the time and then sends a message to process 1. After process 1 receives the message, it immediately sends the message back to process 0. Process 0 receives the message and records the time. The elapsed time divided by 2 is the average message-passing time. Create your program so it has two command line parameters, both integers. The first is the value of n, the message length, and the second is the value of m, the number of times the experiment is run before an average time is computed. Run this on sufficient sizes of n and m so that you can get an estimate of $\lambda$ and $\beta$. Now, run it a bunch more times to verify or refine your estimates.

Now, redo this experiment, but this time cycle the message from node 0 to node 1 … to node k-1 and back to 0. The time is the elapsed time divided by k. The value of k is the only parameter on your command line. Note: If k=2, then it's the same as above. If k=1, then you have no external communication, but node 0 does talk to itself. Thus, the order of sending and receiving is quite important. Reject values of k<1. Compare these results with those above, treating k=1 as a special case since no external transmission actually occurs.

You must turn in the program (both as c code and as an executable), a spreadsheet in Excel format that records all your experiments and a write-up that discusses your experiments, your hypotheses and your final conclusions. All assignments are turned in electronically to me.

**Due: October 20.**

## Programming Assignment#3

**Implement Prim's algorithm on the Zephyr cluster. You must analyze and benchmark your implementation as was done in Quinn's book for Floyd's algorithm, except that you need to run on 1 to 16 processors. We have provided you with a generator (see two slides previous for program generate) for a set of large arrays in the form of a binary file that starts with N then N again (to match Quinn's routines for reading striped matrices) followed by the $N^2$ values of the adjacency matrix. These values will all be integers (MPI_INT) and the matrix will always be in row major order. This does not inhibit you, as Quinn provides routines to distribute these in either row or column striped fashion. Your output of the tree must be optional (see how I do this for the matrix in generate.c), so a single parameter can turn this on or off. Moreover, the output should be timed (see floyd.c to do this). The argument list to your program, whose executable must be named prim is as follows AdjacencyFileName [TREE]. If the keyword TREE is omitted, the tree is not printed.**

**Due: 10/27**

## Distributed Computing Paradigms (in Java)

This material is in Power Point Slides at DistributedParadigms.ppt

## Project Suggestions

1) For Vision people
   a) Parallel or distributed edge detection
      i) Don’t do Canny, as that was already done
   b) Parallel or distributed depth calculation
      i) Might use disparity maps from stereo
2) For Graphics people
   a) Parallel tone or color mapping
      i) Map tone from one image to another
      ii) Do for image sequences
      iii) Use background color shift to alter rendering of foreground objects
      iv) Map tones from neighboring parts of image (a smoothing technique)
      v) Compare various color theories
   b) Parallel or distributed compression/decompression of HDR images
      i) Do for single images
      ii) Do for film clips rather than single images
      iii) Compare various representations
3) For Systems people
   a) RMI that supports both synchronous and asynchronous calls
      i) Might experiment with a new class called “Future”
   b) Spaces implementation in C# or C++
      i) Could do pure tuple spaces or some variant
4) For Algorithms people
   a) Parallel or distributed evaluation of constraints
      i) For instance, Gaussian elimination delayed due to non-linearity
   b) Parallel or distributed graph rewriting
      i) Perhaps synthesized attribute evaluation on a tree, e.g., constant propagation
   c) Parallel or distributed term rewriting
   d) Parallel or distributed Cocke-Kasami-Younger algorithm
   e) Parallel or distributed reconstruction of phylogenies

In all cases, you must design experiments to determine the scalability of your solution. If you have a team of two, you might have a run-off of a parallel versus a distributed solution.

## An Improved Parallel Sort

### The Shearsort

- Assume N is a Perfect Square
- Organize into a $\sqrt{N} \times \sqrt{N}$ Array of Cells
- Alternately Sort Rows and Columns
  (In the Manner of Shearing Sheep)
- Sort Odd Numbered Rows Left to Right
  Sort Even Numbered Rows Right to Left
- Conceptually Algorithm Uses Two Clocks
- Standard clock tells everyone to participate in one more step of a simple row or column sort
- Added clock tells cells to alternate between rows and columns
- IDs $\sqrt{N}(i-1)+1$ to $i\sqrt{N}$  Cooperate on Row i;
  $i, i+\sqrt{N}, .., i+N-\sqrt{N}$  Cooperate on Column i

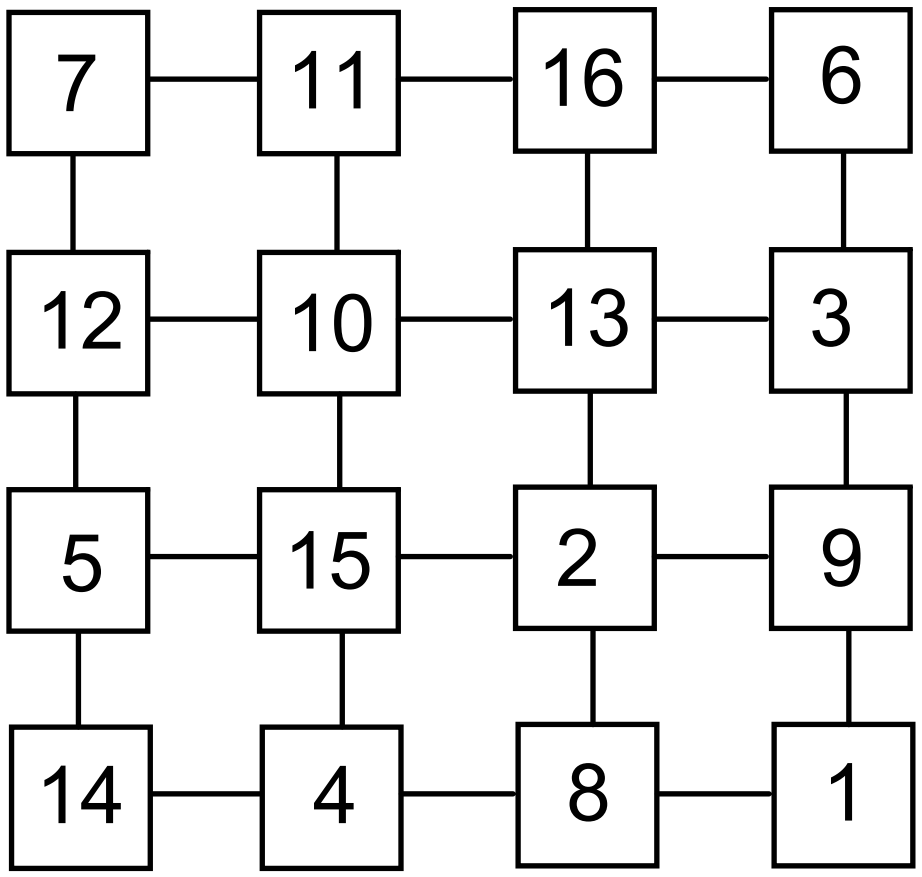
## Shearsort Algorithm

**Managing with One Clock**

```
At Each Clock Tick and For Each $P_i$ do
    Step := Step+1;
    MajorStep := (Step-1) div $\sqrt{N}$ + 1;
    if odd(MajorStep) then
        if odd((i-1) div $\sqrt{N}$ + 1) then
            SortLeftToRight     // Bubble $\sqrt{N}$
        else SortRightToLeft    // Bubble $\sqrt{N}$
    else
        SortTopToBottom  // Bubble $\sqrt{N}$
```

## An Example Shearsort

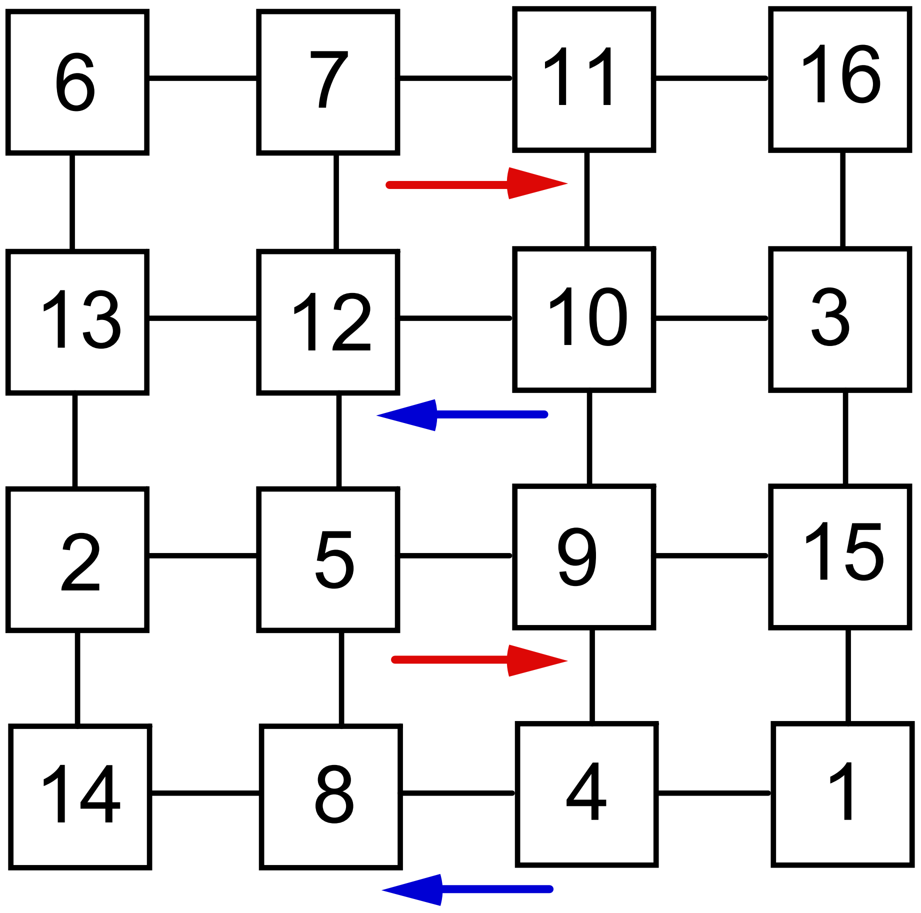
Initially



- N = 16, √N = 4: Use 4 × 4 Matrix

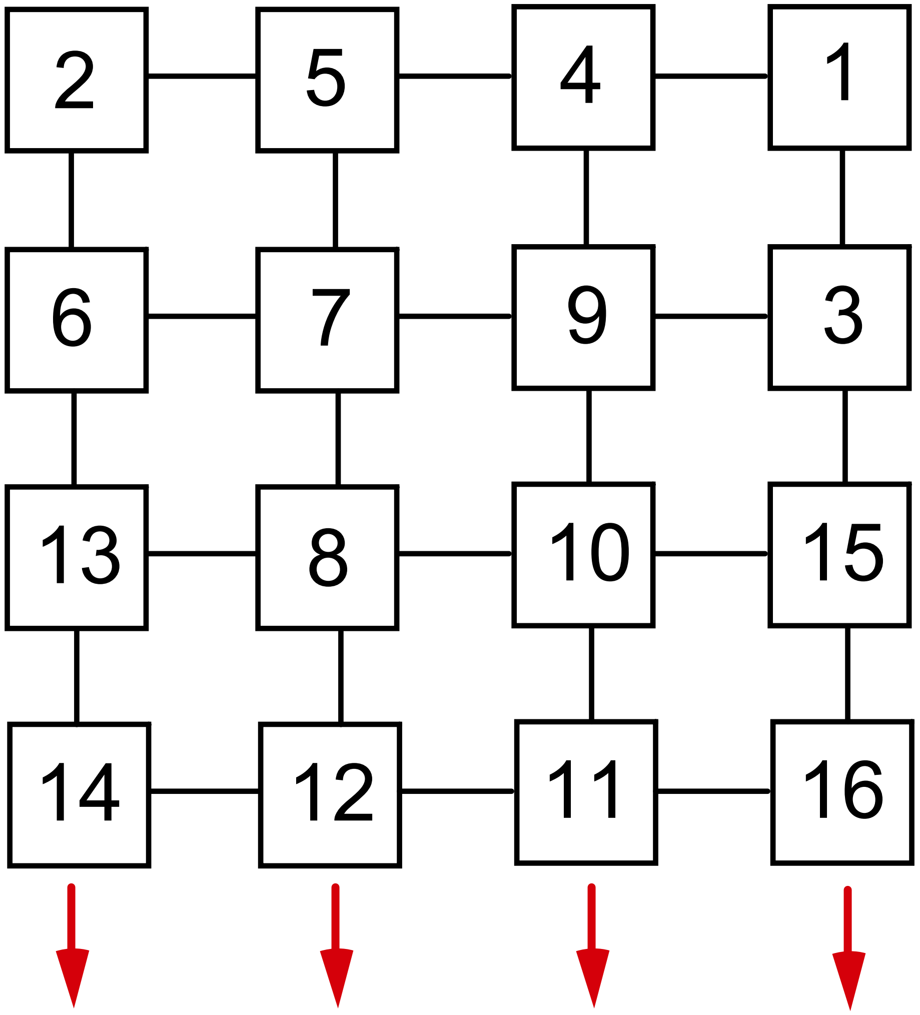
## Shearsort Pass#1

Sort Rows



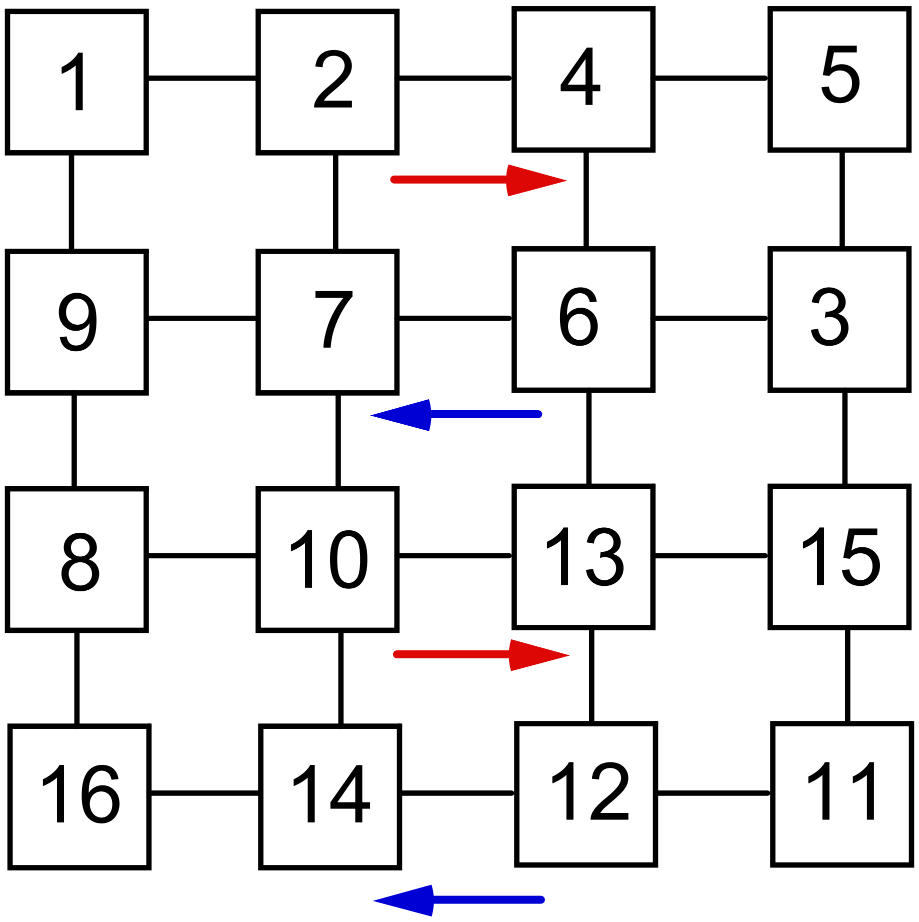
## Shearsort Pass#2

Sort Cols



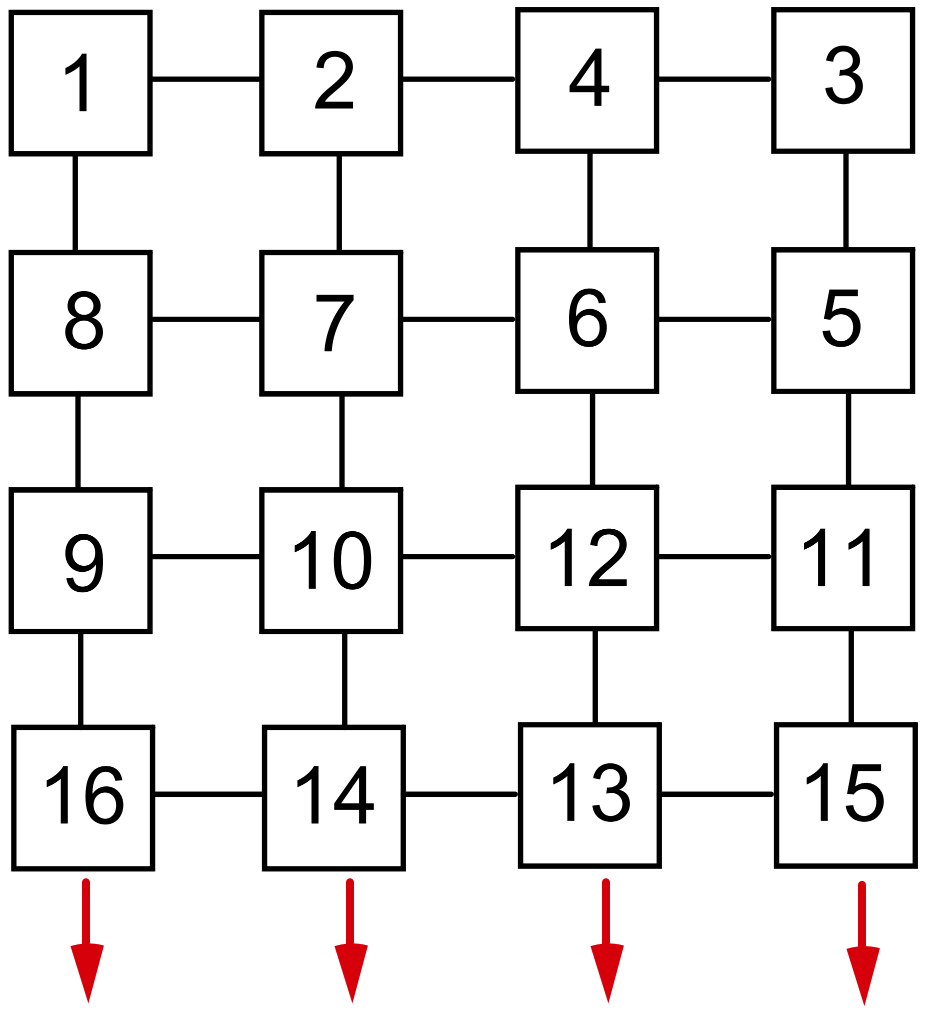
## Shearsort Pass#3

Sort Rows



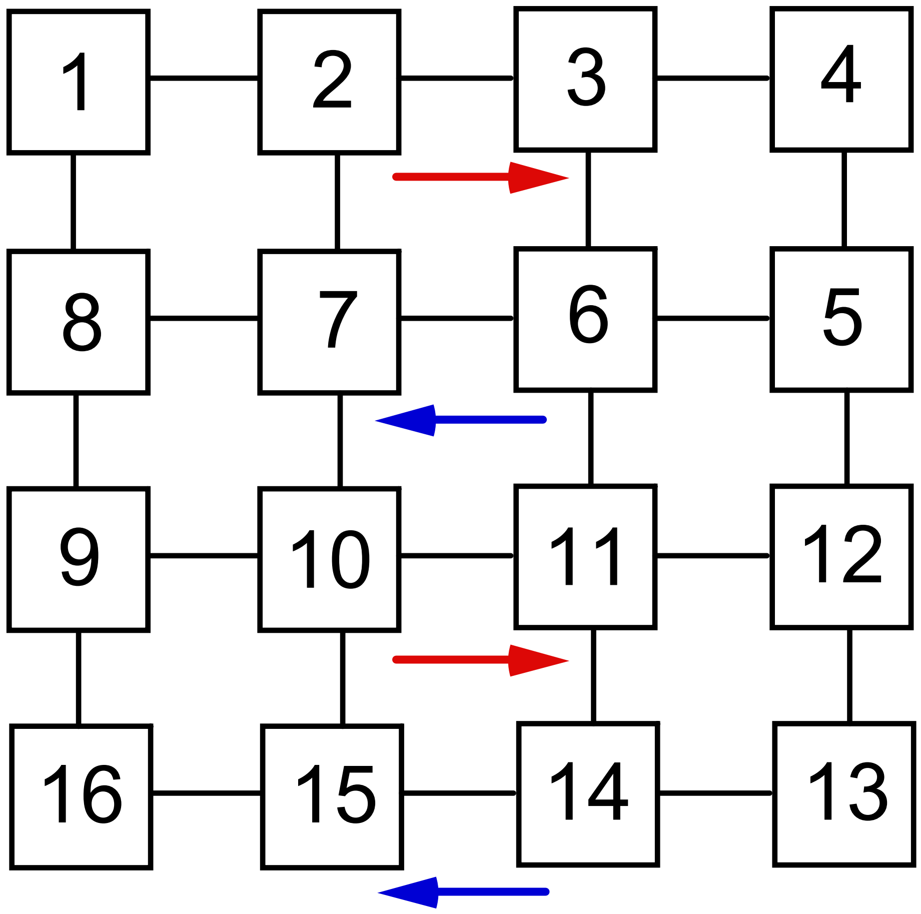
## Shearsort Pass#4

Sort Cols



## Shearsort Pass#5

Sort Rows



## Efficiency of Shearsort

**How'd We Do? Not Bad!**

| | |
|---|---|
| • $T_1(N) = N \log N$ | Optimal Sequential |
| • $T_N(N) = \sqrt{N} \times \log N$ | Parallel Shearsort |
| • $S_N(N) = \sqrt{N}$ | Speedup |
| • $C_N(N) = W_N(N) = N \times \sqrt{N} \times \log N$ | Cost |
| • $E_N(N) = 1 / \sqrt{N}$ | Efficiency |

## Properties of Odd-Even Transposition

**In Some Things Luck Shines Our Way**

- Algorithm Uses Compare / Exchange Operations
- The Algorithm is Oblivious
    - Communication Independent of Prior Results
- All Oblivious Comparison-Exchange (OCE) Algorithms are Easy to Analyze

## How Can Algorithm Fail?

**Properties of a Faulty Permutation Sort**

- Assume $X_1$, $X_2$, … , $X_n$ is to be Sorted
- Assume $X_{\pi(1)}$, … , $X_{\pi(N)}$ is a Correct Sort
- Assume $X_{\sigma(1)}$, … , $X_{\sigma(N)}$ is an Incorrect Permutation Produced by Some Faulty "Sort"
- Let k be Smallest Index Where $X_{\sigma(k)} > X_{\pi(k)}$
- Then, the Permutation is Correct up to k-1

$$X_{\sigma(1)} = X_{\pi(1)}, X_{\sigma(2)} = X_{\pi(2)}, \ldots, X_{\sigma(k-1)} = X_{\pi(k-1)}$$

## The 0-1 Sorting Lemma

**A Faulty OCE Sort also Fails on 0, 1 Data**

- Define $Y_i = 0$ if $X_i \leq X_{\pi(k)}$ ,
  $Y_i = 1$ if $X_i > X_{\pi(k)}$
- $X_{\sigma(i)} \leq X_{\sigma(j)}$ implies $Y_{\sigma(i)} \leq Y_{\sigma(j)}$, Since Oblivious
- Thus, Output on 0,1 Data is
  - 0, 0, 0, …, 0, 1, …, 1, 0, …,
  - There is a 0 after the 1 in the k-th cell
- $\therefore$ Any Faulty Sort Fails on Some 0, 1 Data

## Correctness of Odd-Even Sort

**Proof Based on 0-1 Sorting Lemma**

- Consider Rightmost Cell $P_k$ Containing a 1
  - If k is even then it won't move at step 1
    - But it will shuttle right at all subsequent steps until it reaches N-th cell
  - If k is odd, it starts moving at step 1
    - And it will shuttle right at all subsequent steps until it reaches N-th cell
- Consider the i-th Rightmost 1
  - By step i+1, no 1's block right shuttle
    - So i-th 1 starts moving by step i+1
  - i-th rightmost 1 is home (cell N-i+1) in at most N-i moves
  - i-th rightmost is home no later than by i+(N-i) = N-th step
- This Shows Correctness and Timing

## Correctness of Shearsort

**Proof Based on 0-1 Sorting Lemma**

- Each Pair of Passes Sorts at Least Half of the Unsorted Rows
- To See This, Consider Three Categories
  - All 0 rows
  - All 1 rows
  - Dirty rows - some 0's, some 1's
- Can Divide Rows into Categories
  - Upper all 0-rows
  - Lower all 1-rows
  - Dirty rows in middle

## Halving in Shearsort

**Each Pair of Passes Cuts Dirty Rows in Half**

- A Row Sorting Pass Will Leave Dirty Pairs

| | | |
|---|---|---|
| 0…0…01…1 | 0…01 …… 1 | 0 … 01 … 1 |
| 1…10 …… 0 | 1…1…10…0 | 1 … 10 … 0 |
| (more 0's) | (more 1's) | (equal 0's 1's) |

- Dirty Pairs After Column Sorting Pass

| | | |
|---|---|---|
| 0……………0 | 0…01…10…0 | 0 … …0 |
| 1…10…01…1 | 1……………1 | 1 … …1 |
| (more 0's) | (more 1's) | (equals) |

## Convergence of Shearsort

**How Much Work Before It's Sorted?**

- Number of Halvings is Bounded by $\log \sqrt{N}$
- But We Do Two Passes per Halving
- Number of Passes is $2 \times \log \sqrt{N} + 1 =$
  $\log \sqrt{N}^2 + 1 = \log N + 1$
  The + 1 is for One Dirty Row Left
- Each Pass Requires a Sort of $\sqrt{N}$ Cells
  We Can Parallel Bubble Sort in $\sqrt{N}$ Steps
- Total is $\sqrt{N} \times (\log N + 1) = O(\sqrt{N} \times \log N)$

## Shearsort Proof – Prelims.

The crux of this correctness and analysis proof for ShearSort is to show that each row/column pair of sorts reduces the number of dirty rows by ½. Moreover, after each row/column sort, all the clean 0 rows are at the top (lowered numbered rows), and all the clean 1 rows are at the bottom (higher numbered rows.)
Notation: Assume R rows and C columns.

**Lemma 1. After any row sort, each even/odd pair of rows is sorted low to high, high to low, respectively.**

Proof: This is a direct consequence of the proof that the Even-Odd Transposition algorithm works.

**Lemma 2. After the first column comparison exchange (there are number of rows of these CE's for a complete column sort), the number of dirty rows is reduced to no more than half of what there were after the preceding row sort. Moreover, each clean 0 row will be in the lower numbered row of such a pair, and each clean 1 row will be in the higher numbered row.**

Proof: This is done by showing the interaction of all possible combinations of dirty pairs (more 0's than 1's, more 1's than 0's, equal number of 0's and 1's.) There is an overhead that does this.

**Lemma 3: Each column comparison-exchange (there are number of rows of them for a complete column sort) in which there is pair of clean rows leaves each clean 0 row in the lower numbered row of such a pair (on the top), and each clean 1 row will be in the higher numbered row (on the bottom).**

Proof: This case is no different than the other case above in which we had an equal number of 0's and 1's in dirty rows.

# Shearsort Proof

**Theorem 1: Each row/column pair of sorts reduces the number of dirty rows by at least one half. Moreover, after each row/column, all the clean 0 rows are at the top (lowered numbered rows), and all the clean 1 rows are at the bottom (higher numbered rows) of the mesh.**

Proof: By Lemma 1, we know that each column sort starts with even-odd pairs of rows sorted low to high and high to low, respectively. By Lemmas 2 and 3, the first comparison-exchange operation of the column sort will result in all row pairs being of one of the forms clean 0/dirty, dirty/clean 1 or clean 0/clean 1. The remaining R-1 comparison-exchange operations of the column sort will, in effect, move each clean 0 row up, so they are all together at the top (low numbered) rows of the mesh. Similarly, the clean 1's will move down to be together at the bottom of the mesh. The reason that this is doable in the R-1 remaining passes is that the clean 0's are already on the top of the pairs from which they formed and the clean 1's are on the bottom of these pairs, leaving at most R-1 moves to find their respective destinations.

**Theorem 2: The parallel ShearSort algorithm correctly sorts an R×C mesh in O(R+C)lg R steps.**

Proof: By Theorem 1, the parallel ShearSort sorts all but, perhaps, one row of an R×C mesh containing 0-1 data in lg R row/column parallel even-odd transposition sorts. By previous analysis, we know that the row sorts take R steps and the column sorts take C steps. Thus, each row/column pair of sorts takes (R+C) steps. The final cleanup of the last unordered row takes R steps, so the total number of steps is (R+C)lg R + R. By the OCE (oblivious comparison exchange) lemma, an OCE sort that works on 0-1 data works on arbitrary data. Thus, the parallel ShearSort is correct and runs in O(R+C)lg R. When the mesh is square, this is $2\sqrt{N} \lg \sqrt{N} = \sqrt{N} \lg N$.

## Revsort: An Improvement of Shearsort

**The Revsort**

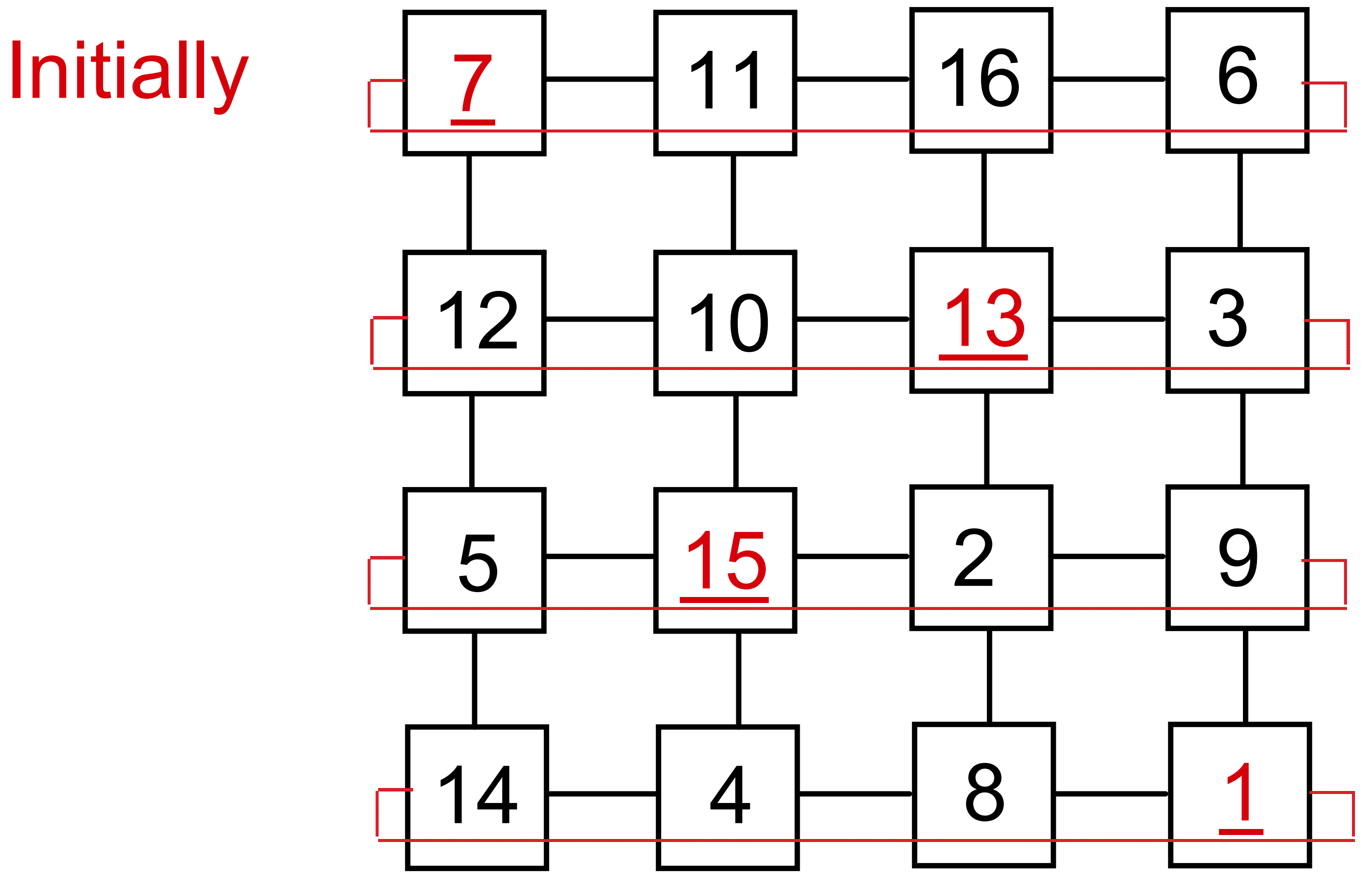
- Revsort is a Column / Row Alternating Sort
- For Convenience We Number Cells from 0
- Define rev(i) = Bit Reversal of i
- Revsort Sorts the Columns Downwards
- It Then Sorts Rows to the Right,
  Viewing Row i as Cyclically Starting at Column rev(i)
- Clearly the Wraparound Property That We Didn't Need in Shearsort is Critical Here

## Boundary Conditions and Complexity

- Revsort does Not Actually Complete a Sort
  - But It Leaves at Most 8 Dirty Rows
  - These Rows Can be Handled by Shearsort
- Let d be the Number of Dirty Rows
  - On Each Column / Row Pass With $d > 8$

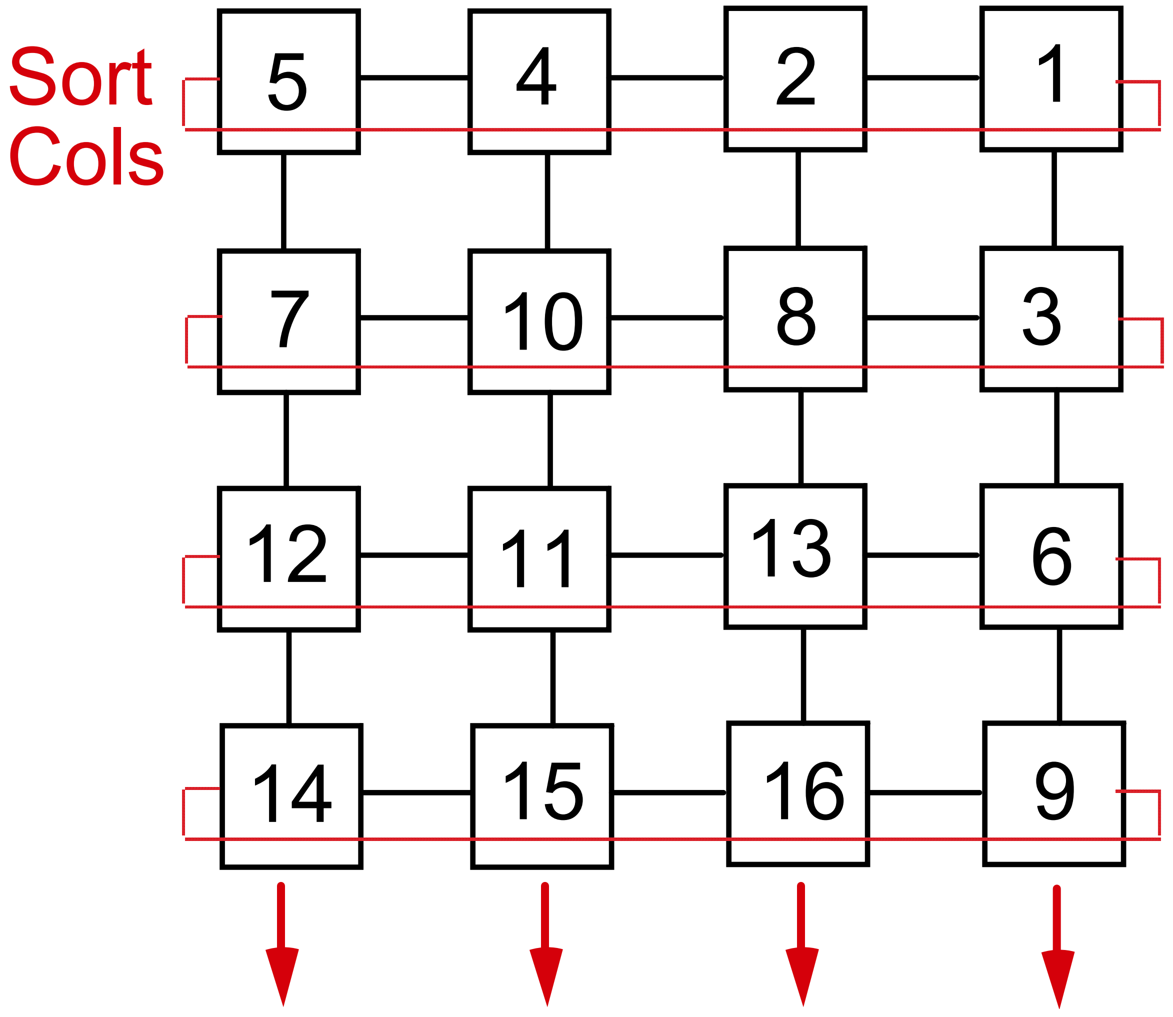
    Reduces Dirty Rows by $O(\sqrt{d})$
- Running Time is $2 \times \sqrt{N} \times (\lg \lg \sqrt{N} + 2)$

## Revsort on an Old Example

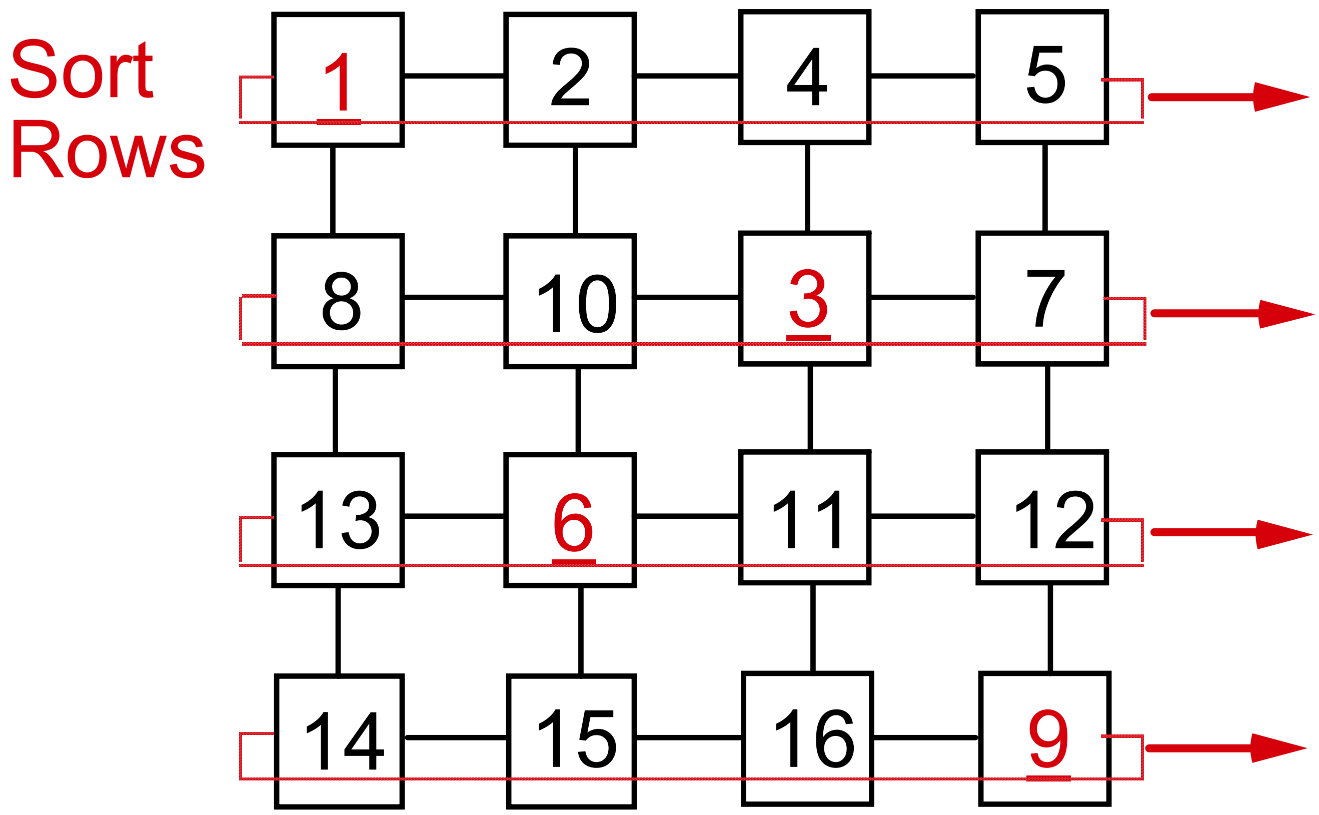


- We Underscored the Starting Column of Each Row.

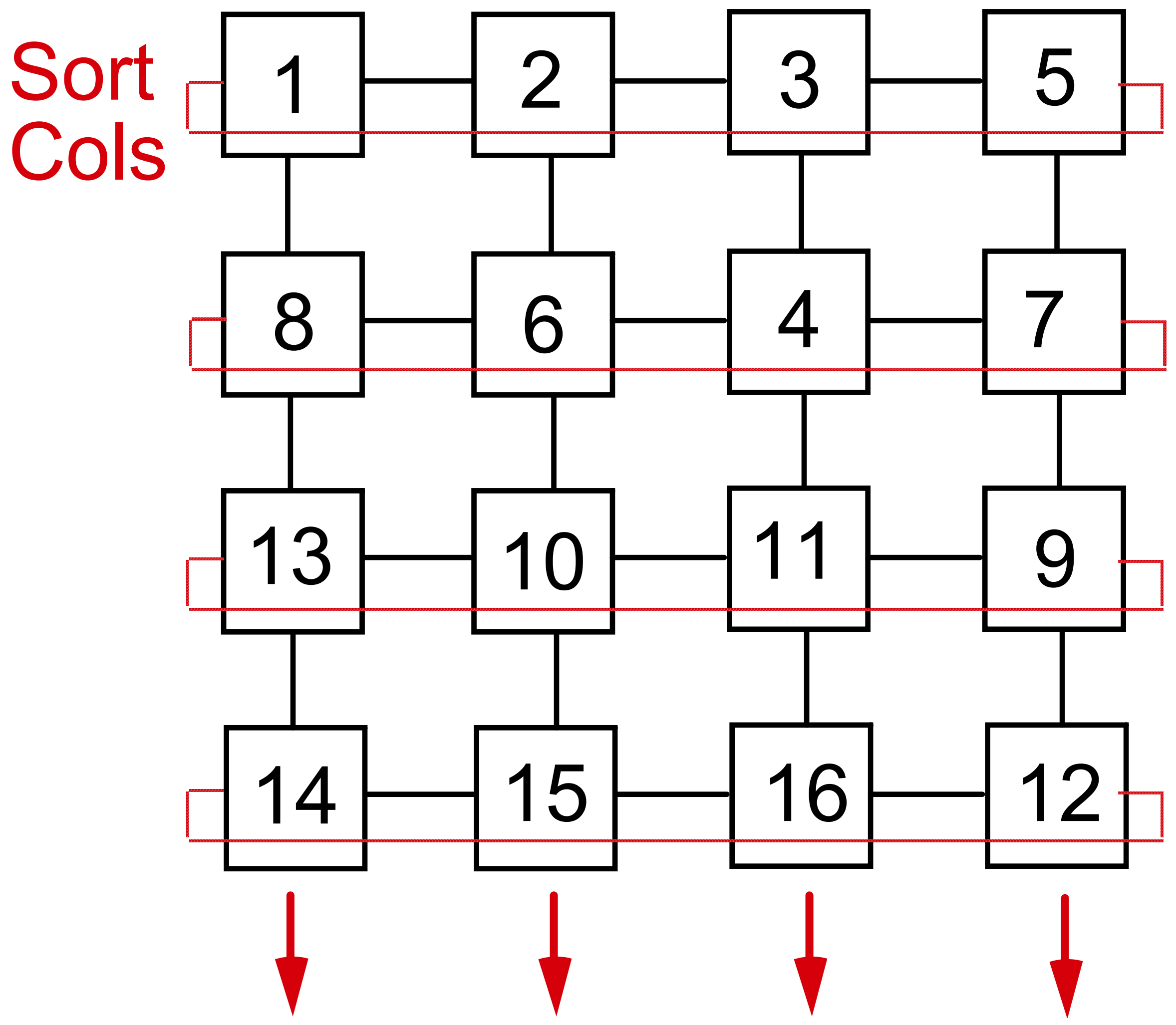
## Revsort Pass#1



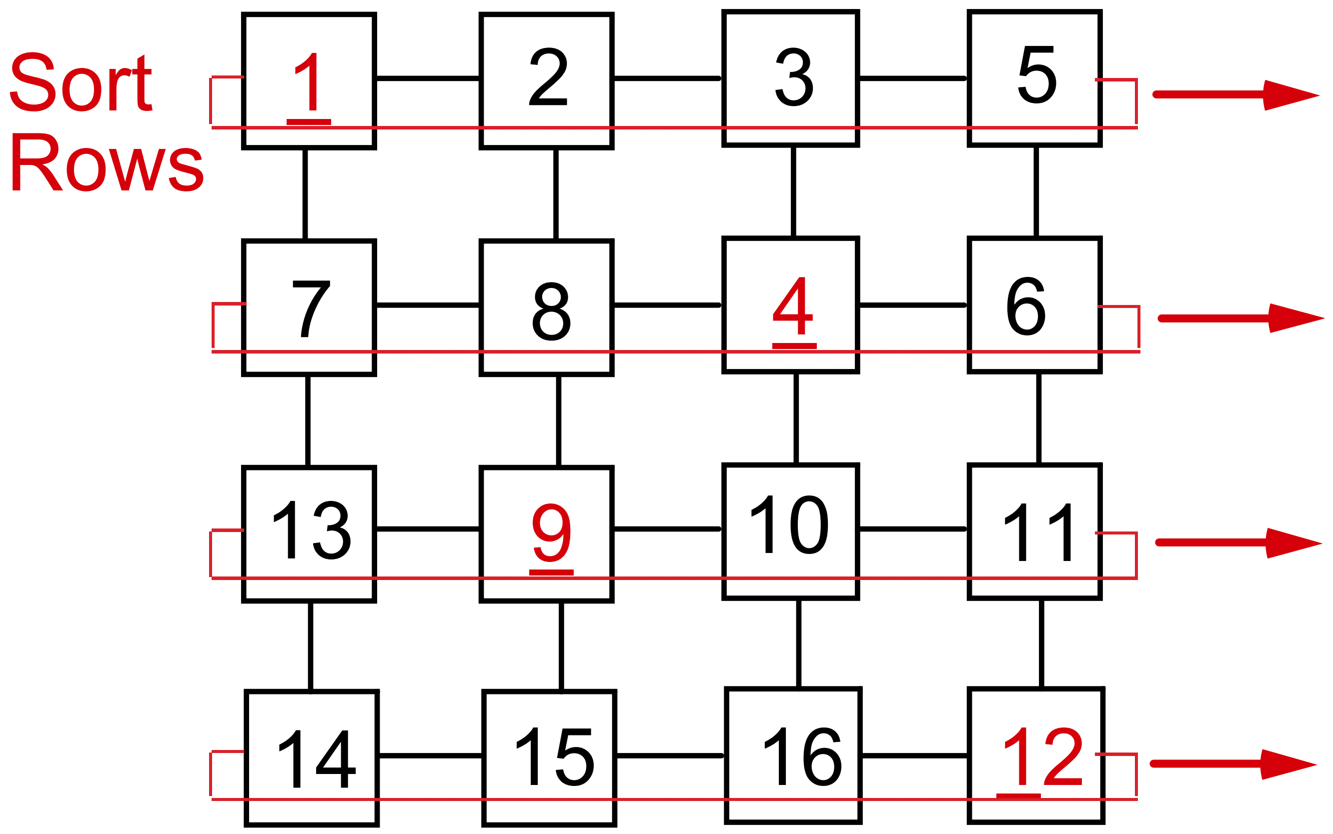
## Revsort Pass#2



## Revsort Pass#3

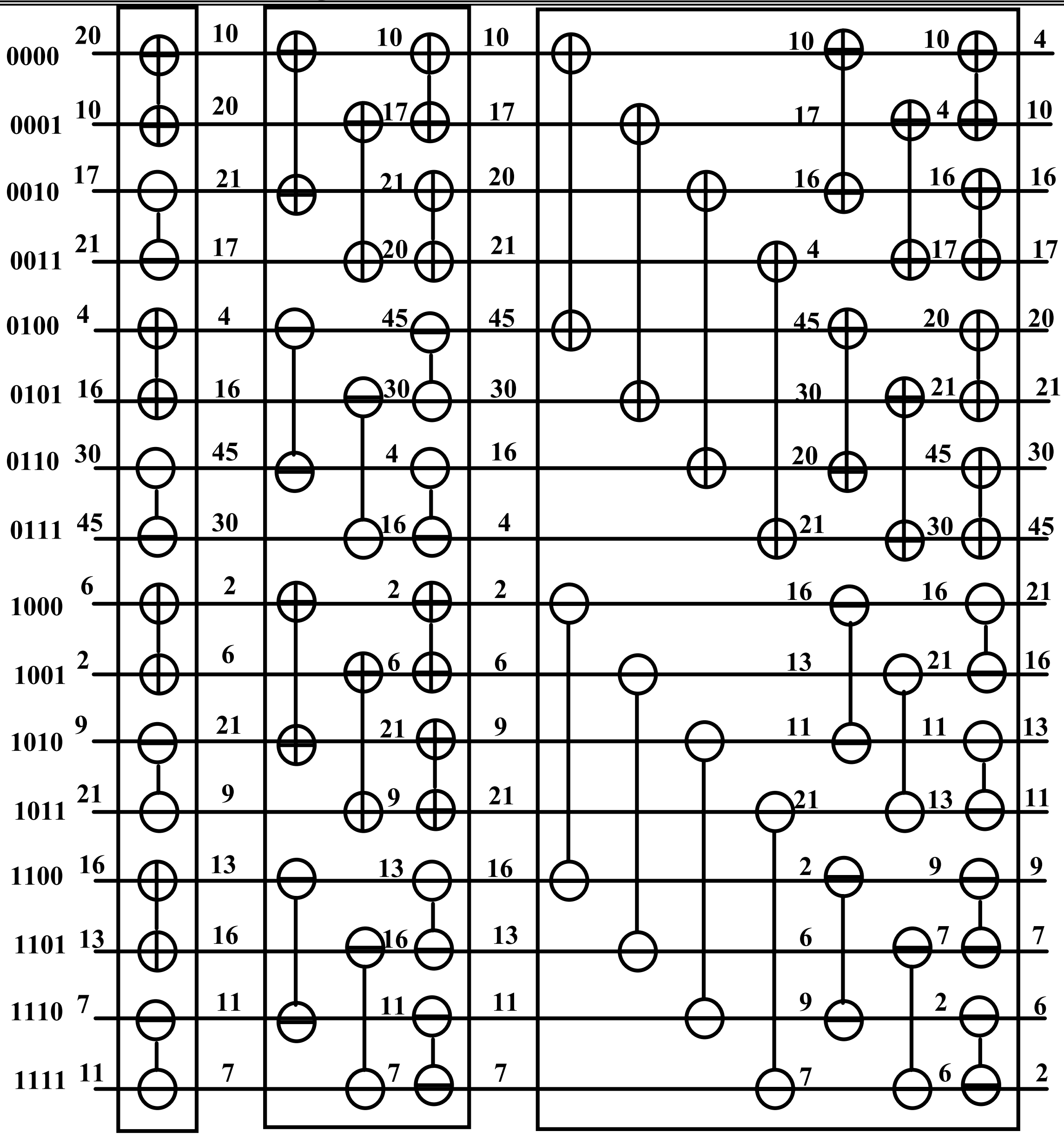


## Revsort Pass#4

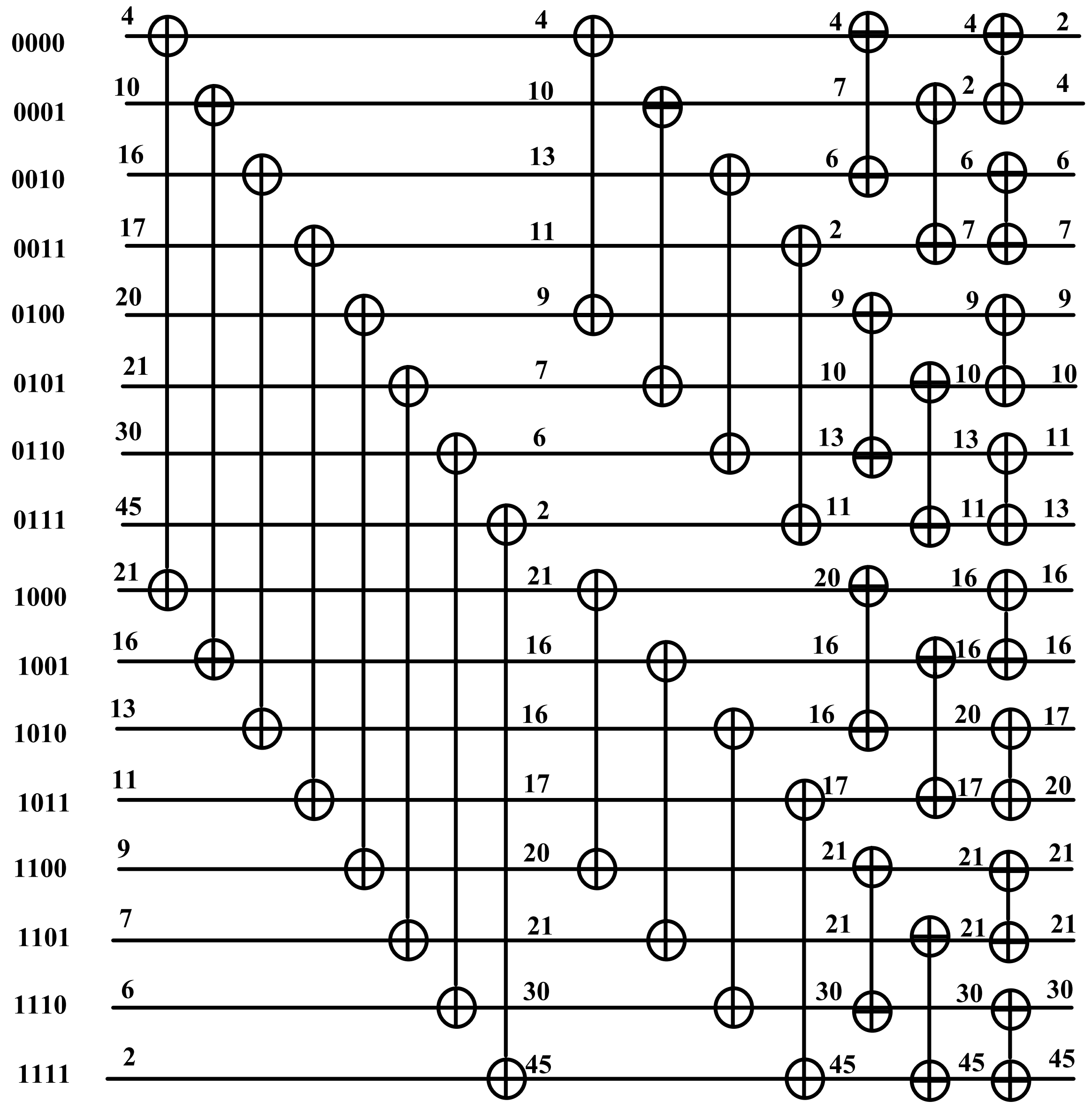


Revsort is Stuck
At This Point Can Use Shearsort

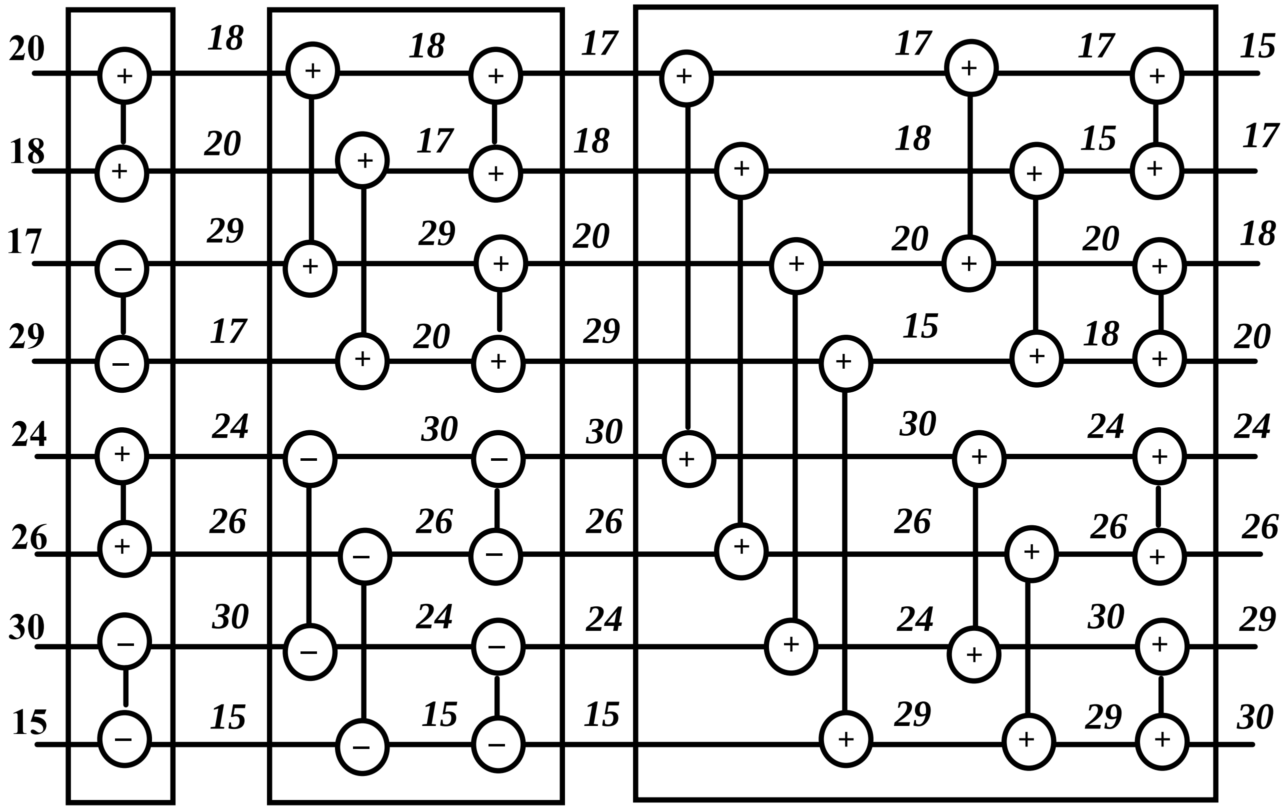
# Bitonic Sort – Making a List Bitonic



# Bitonic Merge – Finishing the Sort



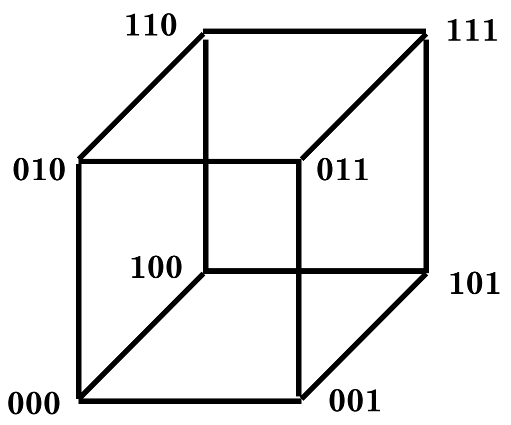
## Bitonic Sort

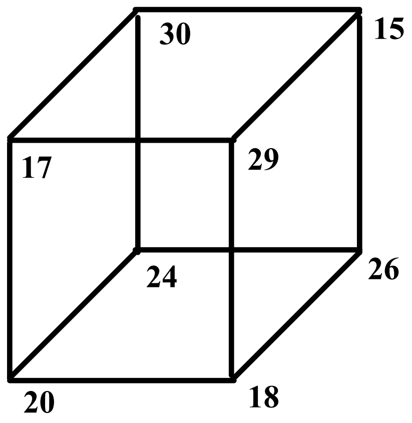


Time = $\sum_{i=1}^{\lg n} i$ = (lg n + 1) lg n / 2 = O($\lg^2$n).

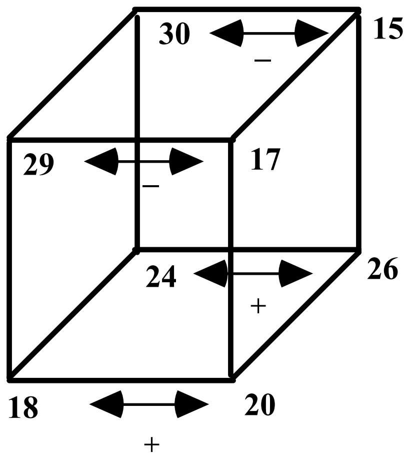
## Bitonic Sort on Hypercube

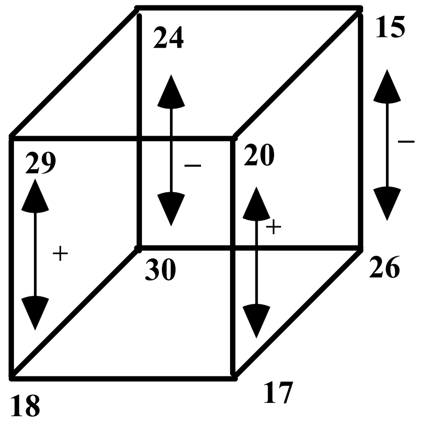
- The Mapping is Natural – Use 3-Cube for 8 Values

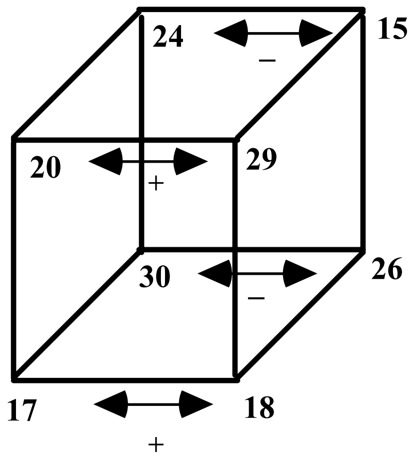




# Use Hypercube to Make List Bitonic



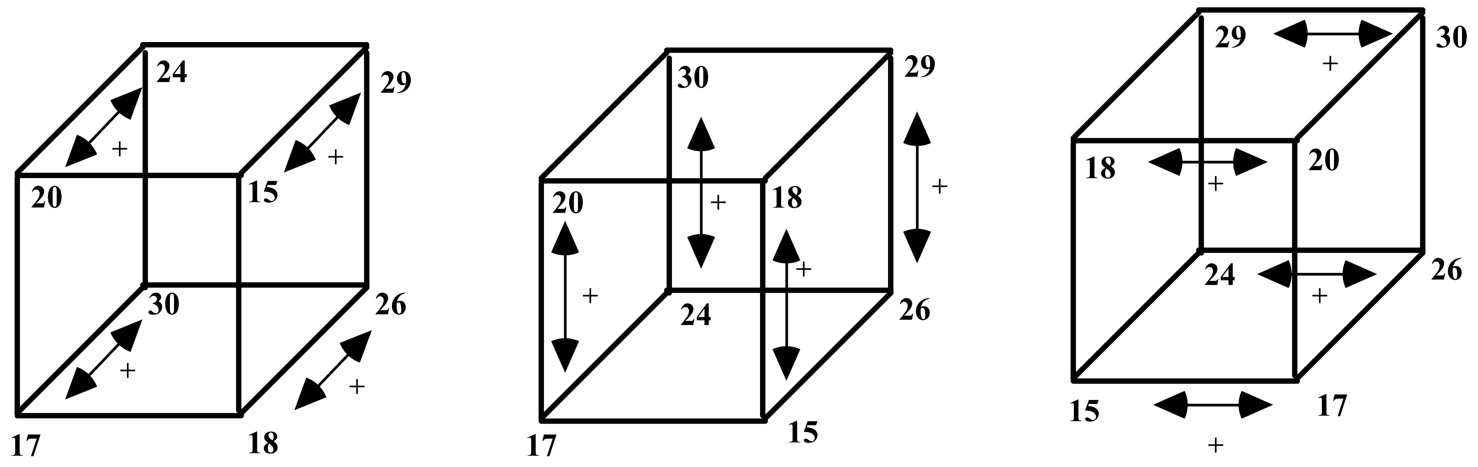




Phase 1

Phase 2, Steps 1 & 2 – Bitonic Now

# Use Hypercube to Sort Bitonic List



Phase 3, Steps 1, 2 & 3 – Sorting the Bitonic List

## A Fast, Inefficient Max

- **Quick, but Not Blindingly Fast**
- **Use a Doubly Logarithmic-Depth Tree**
  - If $N=2^{2^k}$, then root has $2^{2^{k-1}}$ children
  - At ith level, $0 \le i < k$, each node has $2^{2^{k-i-1}}$ children
  - At level k, each node has 2 leaves as children

## Example of Doubly Log Depth Tree

**If N = 64K = $2^{16} = 2^{2^4}$, then**

level 0 (root) has 256 = $2^8 = 2^{2^3}$ children

level 1 nodes have 16 = $2^4 = 2^{2^2}$ children

level 2 nodes have 4 = $2^2 = 2^{2^1}$ children

level 3 nodes have 2 = $2^1 = 2^{2^0}$ children

level 4 nodes have 2 children

Number of leaves = $2\times2\times4\times16\times256 = 2^{1+1+2+4+8} = 2^{16}$

- **Each Internal Node Gets Max of Subtree**
  - Using super fast max, each level takes O(1), so T(N) = O(lg lg n)
  - Work is O(N lg lg N), E = 1 / lg lg N – Non-Optimal

## Doubly Logarithmic Max

Doubly Logarithmic Tree Algorithms ($N = 2^{2^k}$):
we count levels from leaves up

| level | #trees | #kids /tree | time | | work/tree | | work | |
|---|---|---|---|---|---|---|---|---|
| | | | fast | lg | fast | lg | fast | lg |
| 0 | N/2 | 2 | 1 | 1 | 1 | 1 | N/2 | N/2 |
| 1 | $N/2^2$ | 2 | 1 | 1 | 1 | 1 | N/4 | N/4 |
| 2 | $N/2^4$ | $2^2$ | 1 | 2 | $2^4$ | $2^2$ | N | $N/2^2$ |
| 3 | $N/2^8$ | $2^4$ | 1 | 4 | $2^8$ | $2^4$ | N | $N/2^4$ |
| 4 | $N/2^{16}$ | $2^8$ | 1 | 8 | $2^{16}$ | $2^8$ | N | $N/2^8$ |
| • | • | • | • | • | • | • | • | • |
| • | • | • | • | • | • | • | • | • |
| k-1 | $2^{2^{k-1}}$ | $2^{2^{k-2}}$ | 1 | $2^{k-2}$ | $2^{2^{k-1}}$ | $2^{2^{k-2}}$ | N | $N/2^{2^{k-2}}$ |
| k | 1 | $2^{2^{k-1}}$ | 1 | $2^{k-1}$ | $2^{2^k}$ | $2^{2^{k-1}}$ | N | $N/2^{2^{k-1}}$ |
| **Order of Totals** | | | lglgN | lgN | | | Nlg lgN | N |

Conclusions:

- Doubly Logarithmic Max with fast algorithm is fast and reasonably efficient.
- Doubly Logarithmic Max with tree algorithm is no faster than standard binary tree reduction algorithm but is still work efficient.

## Fast, Efficient CRCW Max

- **Accelerated Cascading**
  - Use Work Optimal Binary Tree Reduction Algorithm to Get Problem Size Reduced -- Don’t go too far; don’t quit too soon
  - Finish with Work Suboptimal, Fast Algorithm

- **In Case of max**
  - Use lg N algorithm for lg lg lg N levels
    - Reduces size to N / $2^{\lg \lg \lg N}$ = N/lg lg N elements in lg lg lg N steps = O(lg lg N), taking O(N) work.
  - Next, use CRCW doubly log-depth super fast algorithm.
    - This requires no more than O(lg lg N) steps, Work is O(N/lg lg N $\times$ lg lg (N / lg lg N)) = O(N)

- **The Total is O(lg lg N) Time and O(N) Work**

## Summary of Accelerated Cascading

Accelerated Cascading:
- Use work optimal, not super fast algorithm to reduce problem size.
- Use work suboptimal, super fast algorithm on remaining subproblem.

Reduce problem using lg tree algorithm for lg lg lg N levels
Work is O(N) since work for lg N levels is O(N)
Time is number of levels = lg lg lg N = O(lg lg N)
# of nodes left is $N / 2^{\lg \lg \lg N} = N / \lg \lg N$

Attack remaining problem using CRCW doubly log algorithm
Time is clearly O(lg lg N) since it is this fast on N values
Work is $O(N/\lg \lg N \times \lg \lg (N / \lg \lg N)) = O(N)$

Conclusions:
- Accelerated Cascading Max is fast and optimally work efficient.
- This is another case of using two types of specialists.
  One is work efficient and reasonably fast
  The other is very fast, but not real work efficient

The sum total is a fast, work efficient algorithm

## PCN – Program Composition Notation

**Principle:**

- **First-Class Concurrency:** not an add-on
- **Controlled Non-Determinism:** do it out of intent
- **Compositionality:** easy to understand compositions
- **Mapping Independence:** results independent of mapping

**Realization of Principles:**

- **Definitional Variables:** an abstract machine-independent model of communication and synchronization
- **Concurrent Composition:** compose simple components into lightweight concurrent tasks that communicate and synchronize through definitional variables
- **Non-Deterministic Choice:** while predictable computation is usually desirable, reactive programs can benefit from non-determinism
- **Encapsulation of State:** disallow sharing of data structures that are subject to change in order to avoid unintended non-determinism

## Definitional versus Mutable Variables

**Definitional:**

Initially have undefined value

Can be defined just once – like in Strand

The definition operator is **=**

Can receive a tuple, int, double or char value

An attempt to read an undefined definitional variable blocks the reader.

Are recognized by being undeclared variables

Can be shared across parallel compositions

**Mutable:**

Initially are arbitrary value

Can be defined many times – like in C

The assignment operator is **:=**

Can receive an int, double or char value

An attempt to read a mutable variable always succeeds.

Are recognized by being explicitly declared variables

Can**not** be shared across parallel compositions

## Compositions (Sequential and Parallel)

**Sequential:** uses no operator or the semicolon (;)

{ ; block$_0$, block$_1$, … , block$_k$ }

blocks are executed in the given order

**Parallel:** uses the double bar ( || )

{ || block$_0$, block$_1$, … , block$_k$ }

blocks are executed concurrently with the guarantee of fairness in that each must eventually make progress

## Compositions (Choice)

**Choice:** uses the question mark ( ? )

{ ? $\text{guard}_0 \rightarrow \text{block}_0$, $\text{guard}_1 \rightarrow \text{block}_1$,… , $\text{guard}_k \rightarrow \text{block}_k$ }

guards may be evaluated in any order or in parallel; each guard’s Boolean expression is evaluated left to right; a guard blocks if it references an undefined definitional variable; if all guards fail, no action occurs; if one succeeds, its block is executed; if more than one succeeds, one of the selected blocks is non-deterministically chosen

each guard is a sequence of one or more tests using
- arithmetic comparison (a < b, a > b, a <= b, a >= b)
- equality tests (a == b, a != b)
- type tests (int(a), char(a), double(a), tuple(a))
- synchronization tests (data(a))
- tuple matches (?=)
- default action (**default**)

Note this is based on CSP notation (cooperating sequential processes)

## Tuples

**Tuple Form:**

{ $\text{term}_0$, $\text{term}_1$, … , $\text{term}_{k-1}$ }, $k \geq 0$

where each term is a definitional data structures, including _ which is an anonymous definitional variable

examples: {a, b}   {“abc”}   {}   {12, {13, {} }}  {5.2, _, _, c}

**Tuple Creation and Access:**

{ || proc(1, {x, y, {z} } ), x = “abc”, y = {123} }
passes the tuple {“abc”, {123}, {z}} as proc’s 2nd argument

the same effect can be achieved by

```
{ || make_tuple(3,tup), proc(1, tup ),
        tup[0] = “abc”, tup[1] = {123}, tup[2] = {z} }
```

from above it is clear that indices can be used to access tuple parts

A tuple guard test (?=) can be used for matching as in
tup ?= {“abc”, a, {b}} which matches the above tuple with
a = {123} and b = z.

Note tuple guards are not unification.  Defns. are passed from left to right, only.

List access can also be used where {1, {2, {3, {} } } } can be denoted as [1, 2, 3] and this is of form [h | t] where h=1 and t=[2, 3].  This is like in Prolog and LISP.

## PCN Examples

Define r as TRUE if e is a member of the list l.

```
#define  TRUE    1
#define  FALSE   0
member(e,  l, r)
{?   l ?= [v | l1], v == e → r = TRUE,
     l ?= [v | l1], v != e → member(e, l1, r),
     l ?= [ ] → r = FALSE
}
```

Compute the height z of binary tree t.
Use {left, root, right} to represent a node.

```
height(t, z)
{?   t ?= {left, _, right} →
     { ||  height(left, l), height(right, r),
           {?   l >= r     → z = l+1,
                l < r -> z = r+1 }
     } ,
     t ?= { }  → z = 0
}
```

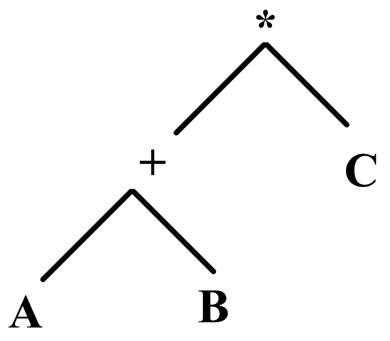
## Another PCN Example

**Compute the preorder traversal in p of binary tree t.**
**Use {left, root, right}**
**to represent a node.**

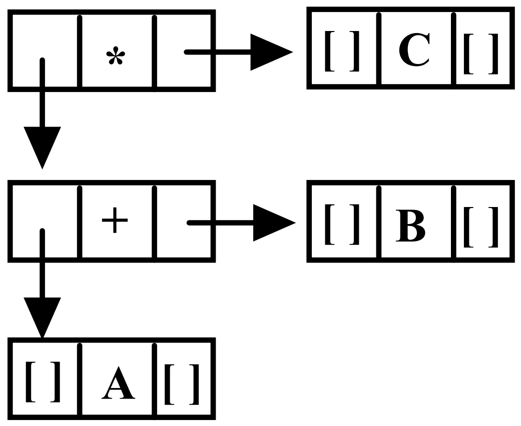
```
preorder(t, p)
{     build_pre(t, p, [ ])
}

build_pre(t, b, e)
{?    t ?= {left, val, right} →
      { ||   b = [val | m1],
             build_pre(left, m1, m2),
             build_pre(right, m2, e)
      } ,
      t ?= { }  →  b = e
}
```

## Pre-Order Traversal in PCN

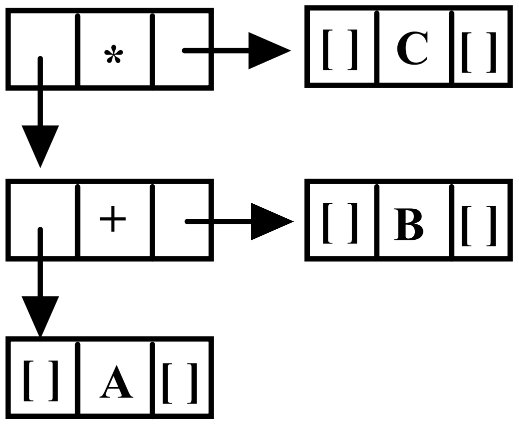
**The expression tree**



**is represented as:**

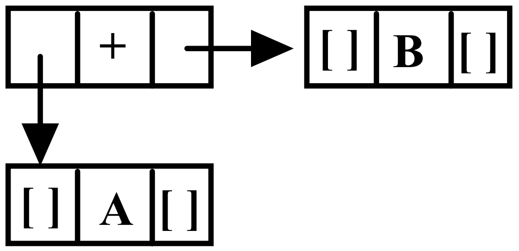


## Invoking the PCN Code

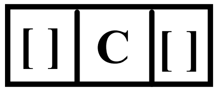
t = 

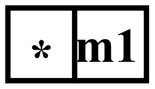
b = Undefined e = [ ]

This matches the first condition, so we bind

left = 

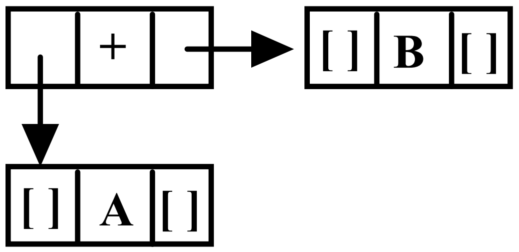
value = *

right = 

b = 

m1 = Undefined m2 = Undefined

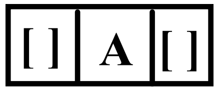
## Level 2 Invocation of build_pre

**t =** 

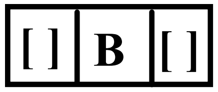
**b = Undefined**

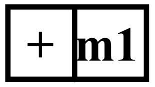
**e = Undefined (bound to parallel calls b)**

**This matches the first condition, so we bind**

**left =** 

**value = +**

**right =** 

**b =** 

**m1 = Undefined**

**m2 = Undefined**

## Parallel Level 2 Invocation of build_pre

**t =** $\boxed{[\ ] \mid C \mid [\ ]}$

**b = Undefined (bound to parallel calls e)**

**e = [ ]**

**This matches the first condition, so we bind**

**left = [ ]**

**value = C**

**right = [ ]**

**b =** $\boxed{C \mid m1}$

**Due to easy case with [ ] call we have**

**m1 = m2 (since b = e in the new call)**

**Due to easy case with [ ] call we have**

**m2 = e = [ ]**

**and, in consequence,**

**b =** $\boxed{C \mid [\ ]}$**, and this is also value of parallel e**

## Level 3 Invocation of build_pre

**t =** $\begin{array}{|c|c|c|}\hline [\ ] & A & [\ ] \\ \hline \end{array}$

**b = Undefined**

**e = Undefined (bound to parallel calls b)**

**This matches the first condition, so we bind**

**left = [ ]**

**value = A**

**right = [ ]**

**b =** $\begin{array}{|c|c|}\hline A & m1 \\ \hline \end{array}$

**Due to easy case with [ ] call we have**

**m1 = m2 (since b = e in the new call)**

**Due to easy case with [ ] call we have**

**m2 = e = parallel calls b**

## Parallel Level 3 Invocation of build_pre

**t =** $\boxed{[\,]}\boxed{B}\boxed{[\,]}$

**b = Undefined (bound to parallel calls e)**

**e =** $\boxed{C}\boxed{[\,]}$ **(bound to parallel level 2 )**

**This matches the first condition, so we bind**

**left = [ ]  value = B  right = [ ]**

**b =** $\boxed{B}\boxed{m1}$

**Due to easy case with [ ] call we have**

**m1 = m2 (since b = e in the new call)**

**Due to easy case with [ ] call we have**

**m2 = e =** $\boxed{C}\boxed{[\,]}$ **and, in consequence,**

**b =** $\boxed{B}\boxed{\;}\rightarrow\boxed{C}\boxed{[\,]}$**, and this is also value of parallel e**

**All the pieces are now defined, and original**

**b =** $\boxed{*}\boxed{\;}\rightarrow\boxed{+}\boxed{\;}\rightarrow\boxed{A}\boxed{\;}\rightarrow\boxed{B}\boxed{\;}\rightarrow\boxed{C}\boxed{[\,]}$

## Logic Programming – A Prolog Program

**First Program**

```
chase( X, Y)  :-     dog( X ), cat ( Y ).
cat( fuzzy ).
cat( pumpkin ).
dog( rover ).
```

**Query**

```
?-chase(X, fuzzy).
```

**Answer**

```
X = rover
```

**Query**

```
?-chase(fuzzy, Y).
```

**Answer**

```
No
```

**Query**

```
?-chase(X, Y).
```

**Answer**

```
X = rover, Y = fuzzy
X = rover, Y = pumpkin
```

## The Vocabulary of Logic Programming

**The basic element in a Prolog program is a term.**

**Terms can be simple – variable or constant**
**or complex – a functor and arguments or a list.**

**A variable is an upper case name.**

**A constant is a number or a lower case name.**

**A functor is also lower case.**

**A list is a predefined functor of two arguments which is written in the form [head | tail], where head is the first element of the list and tail is the remainder. The corresponding functor is just concatenation.**

**A variable can be bound only once to another term**

**Binding normally occurs through unification, where a variable must match another term**

**A clause has a head and an optional body.**

## Programs and Clauses

**Example general forms of clauses are**
**H.**
**or**
**H :– $B_1$ , $B_2$ , … , $B_n$.**
**The first clause states a fact, e.g.,**
**factorial ( 5, 120 ).**

**The second states that proposition H is supported by the truth of all of $B_1$ , … $B_n$. We read**

```
factorial ( N, Fact1) :–
     N>0, factorial ( N-1, Fact2 ), Fact1 = N * Fact2.
```

**as "the proposition that N is related to Fact1 by the functor factorial is supported if N>0, and N-1 is related to Fact2 by the functor factorial, and Fact1 is equal to N times Fact2."**

**We also say that we can achieve the goal factorial(N, Fact1), if we can achieve the other subgoals.**

**Running a Prolog program consists of posing a query, e.g.,**
**?– factorial ( 5, Fact)**

## Another Prolog Program

**Second Program**

```
append( [ ], Y, Y).
append( [H | X], Y, [H | Z] ) :- append( X, Y, Z).
```

**Query**

```
?-append( [a, b], [c, d], Z).
```

**Answer**

```
Z = [a, b, c, d]
```

**Query**

```
?-append( [a, b], Y, [a, b, c, d] ).
```

**Answer**

```
Y = [c, d]
```

**Query**

```
?-append( X, [c, d], [a, b, c, d] ).
```

**Answer**

```
X = [a, b]
```

**Query**

```
?-append( X, [c], [a, b] ).
```

**Answer**

```
No
```

## Abstract Prolog Sample

### Prolog Program

```
p(X, Y)   :-    q(X), r(X, Y).
q(4).
q(X):-    s(X), t(X).
r(X, Y)   :-    u(X), v(Y).
r(X, 3)   :-    w(X).
s(5).
s(6).
t(6).
u(1).
v(3).
w(6).
```

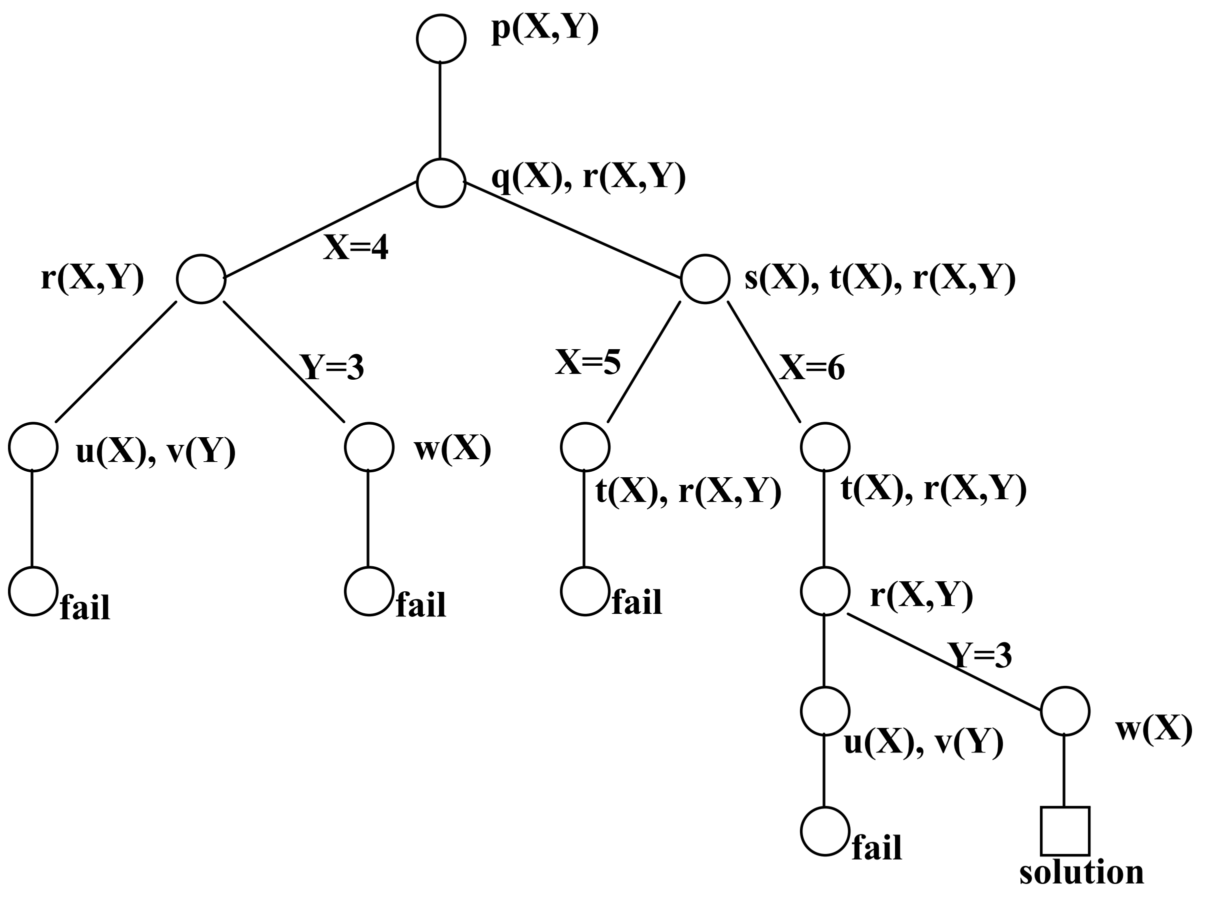
### Query

```
?-p(X, Y).
```

### Answer

```
X = 6,  Y = 3
```

# Solution Tree (Depth First)



## Prolog Sort of Knows Numbers

**Triangular Numbers**

```
tri (0, 0).
tri (N, R) :-
    N1 = N - 1,
    tri (N1, R1),
    R = N + R1.
```

**Query**

```
?-tri (4, Ans).
```

**Answer**

```
Ans = 10
```

**Query**

```
?-tri (Row, 10).
```

**Answer**

Maybe

# CLP-R Really Knows Real Numbers

**Triangular Numbers**

```
tri (0, 0).
tri (N, N + R) :-
    N ≥ 1,
    tri (N-1, R).
```

**Query**

```
?-tri (4, Ans).
```

**Answer**

```
Ans = 10
```

**Query**

```
?-tri (Row, 10).
```

**Answer**

```
Row = 4
```

## Logic Programming and Parallelism

**In logic programming there are 3 clear opportunities for parallelism. Or-parallel pursues multiple choices. This requires separate traces with separate data spaces for each option so backtracking can be done and so the variable bindings of the choices are kept separate. And-parallel makes a choice and then follows one or more of the terms in this clause in parallel. Only one data space is required since each path shares the same bindings. The problem here is variable locking or data flow (generator-consumer) analysis to avoid variable locking. Unification-parallel matches heads of clauses in parallel. This is orthogonal to the other two types of parallelism. It is, in fact, possible to combine all three in a given system.**

## And-Parallelism

In the clause

$H :- B_1 , B_2 , \dots , B_n.$

The subgoals $B_1 , B_2 , \dots , B_n$ must be simultaneously satisfied. An obvious form of parallelism is to do all n subgoals in parallel. There are some associated problems

1) What if two or more of the subgoals reference the same variable? This can create a problem if both try to write it at the same time. They might both assume success if the variable starts as unbound (free to receive a value.)

2) What happens if a conflict is found? Won't backtracking be very complicated?

## Sharing a Variable

Example:

```
    F(X,Y) :– G(X), H(X,Y).
    G(0).
    H(1,1).
    H(0,0).
```

Clearly

```
    ?– F(X,Y)
```

should be answered X=0,Y=0.

But, how do we protect X if we are matching G(X) to G(0) at the same time as we are matching H(X,Y) to H(1,1)? The conflict may go unnoticed!

We could add a lock to X’s access. But that would make access very time consuming.

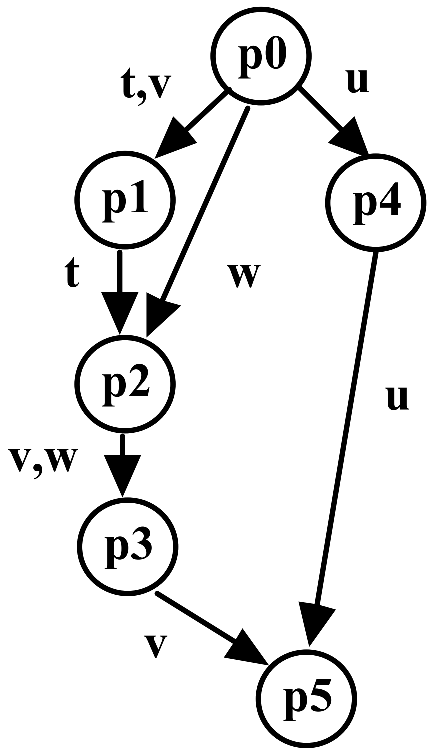
We could apply data flow analysis – either static or dynamic.

We could use a generator / consumer approach.

## Data Flow Analysis

**Consider the following abstract clause**

```
p0(t, u, v, w) :–
    p1(t,v),
    p2(t,w),
    p3(v,w),
    p4(u),
    p5(u,v).
```



**Once p0 is unified to some goal, we must satisfy the subgoals p1, p2, p3 , p4 and p5. We could run p1 and p4 in parallel. Once p1 is done we could start a process for p2. p3 has to await completion of p2, and p5 must await completion of both p3 and p4. This approach would retain the LR semantics whenever a choice is possible.**

## Generator / Consumer

**A problem with the data flow approach is that it doesn't work very effectively if we can't distinguish clauses that assign from clauses that use bindings. The generator consumer approaches are similar to data flow, except that they release a subgoal for processing as soon as its turn comes or its variables have bindings assigned by others. So for the previous example, we would release all subgoals if p0's unification bound all variables. Or less dramatically, we would release p2 with p1 and p4, if p0 provided a binding for t.**

## Backtracking

Once an error is found in one of the subgoals of a goal B, we must backtrack in an intelligent manner. This is a difficult topic that I will not really cover, but I'll point out a few problems.

Bindings must be undone.

We cannot backtrack above some choice point that could have succeeded.

It is foolish to try an option that is guaranteed to fail.

Backtracking is considerably harder if we add in or-parallelism.

## Guarded Clauses

**A guarded clause is one that starts with a set of right-hand conditions that must be satisfied for the clause to apply. Guards are like selects in Ada and can be used effectively to control parallelism.**

**A clause of the form**

**H :– $G_1$ , $G_2$ , … , $G_m$ | $B_1$ , $B_2$ , … , $B_n$.**

**means that the m guards, $G_1$ , $G_2$ , … , $G_m$, must be satisfied in order to satisfy H through the n subgoals, $B_1$ , $B_2$ , … , $B_n$.**

## Strand

Strand focuses on what is called committed choice. That is, it commits to one of the <u>or</u> choices and parallelizes on the <u>ands</u>. This avoids keeping multiple traces and having to backtrack, but does not avoid the problems of data flow. Since Strand is a write-once (single assignment) language, a variable whose value is bound cannot be overwritten. In fact, an attempt to change the value is an error that leads to backtrack in normal Prolog, but denotes an error in a committed choice language.

General forms of clauses in Strand are

$H \;:\!-\; G_1 ,\; G_2 ,\; \ldots\; ,\; G_m \mid B_1 ,\; B_2 ,\; \ldots\; ,\; B_n$. where $m,n \geq 0$

Recall that lower case names are constants (or predicates or functions) and upper case names are variables.

max(X,Y,Z) :- X>Y | Z := X.

max(X,Y,Z) :- X=<Y | Z := Y.

In standard Prolog, we would write as

max(X,Y,X) :- X>Y.

max(X,Y,Y) :- Y>=X.

## Examples in Strand

```
power(X,N,R) :- N==0 | R := 1.       % := is simple assign
power(X,N,R) :- N == 1 | R := X.
power(X,N,R) :- N>1 |
                    N1 is N-1,       % is means calculation
                    power(X,N1,R1),
                    R is X * R1.
Consumer-Producer in Strand:
main( ) :-                          % initial process pool
    producer(100, Buffer),
    consumer (Buffer).              % buffer is shared by tasks
producer(Count, Buffer) :- Count > 0 |        % guard
    get_input(Input),                   % keyboard
    Buffer := [Input | Buffer1]         % push
    Count1 is Count-1,                  % single assignment
    producer(Count1,Buffer1).           % new producer
producer(Count, Buffer) :- Count == 0 |       % guard
    Buffer := [ ]                       % no new data
consumer([Head | Tail]) :-
    Output := Head,                     % display
    consumer(Tail).                     % new consumer
consumer([ ]) :-                        % empty buffer
    Output := “Finished”.               % display
```

## Summation in Strand

```
main() :-                       % initial process pool
     generator(5,Stream),       % gen 1 … 5
     sum (Stream,Sum).          % capture the sum

generator(N,S) :- N > 0 |       % more to go
     S := [N|S1],               % add N to list
     N1 is N-1,                 % one fewer to do
     generator(N1,S1)           % new generator

generator(0,S) :-               % implicit guard
     S := [ ]                   % no new data

sum(L,Sum) :-
     sum1(L,0,Sum).             % intial sum is 0

sum1([X | Rest], A, Sum) :-
     A1 is A + X,               % add X
     sum1(Rest, A1, Sum).       % complete the sum

sum1([ ], A, Sum) :-
     Sum := A.                  % all done
```

## Philosophical Programmers

There are four programmers and only two keyboards and two monitors. Each programmer has a keyboard on one side and a monitor on the other. Programmers ponder until inspired. Once inspired, a programmer tries to acquire a keyboard and a monitor. Having completed a problem, the programmer returns to pondering.

```
main :–
   prog(ponder, P1), prog(ponder, P2),
   prog(ponder, P3), prog(ponder, P4),
   merger([merge(P1), merge(P2), merge(P3), merge(P4)],S),
   monitor(S).

monitor(In) :– initial(C),monitor(In,0,Q,Q,C).

initial(S) :–  S := [set,set].
```

In Strand, a Merger is like a blackboard or message center. It guarantees that the order in which messages are received is the order in which they are output.

## More Philosophical Programmers

```
monitor([req(R)|In],N,F,B,[]) :–
   B := [R|B1], N1 is N+1,
   monitor(In,N1,F,B1,[]).
monitor([req(R1)|In],N,F,B,[R|C]) :–
   R1 := R,
   monitor(In,N,F,B,C).
monitor([rel(R)|In],N,[R1|F],B,C) :–
   R1 := R, N1 is N-1,
   monitor(In,N1,F,B,C).
monitor([rel(R)|In],0,F,B,C) :–
   monitor(In,0,F,B,[R|C]).
monitor([],0,_,_,_).

prog(ponder,S) :– prog(inspired,S).
prog(inspired,S) :–
   S := [req(R)|S1],
   prog1(inspired,S1,R).
prog(program,S) :–
   S := [rel(set)|S1],
   prog(ponder,S1).
prog1(inspired,S,set) :– prog(program,S).
```

## Binary Tree Building in Strand

```
main(BinTree) :–
    generator(5, Stream),
    streamInsert(Stream, [ ], Bintree).
generator(Count, Buffer) :– Count > 0 |
    get_input(Data),    % Bind kb input (a number) to Data
    Buffer := [Data | Buffer1],
    Count1 is Count – 1,
    generator(Count1, Buffer1).
generator(Count, Buffer) :– Count == 0 |
    Buffer := [ ].

streamInsert([ ], A, A).
streamInsert([X | Rest], A, B) :–
    insert(X, A, C),
    streamInsert(Rest, C, B).

insert(X, [ ], [X]).                % [X] is same as [X | Rest] where Rest == [ ]
insert(X, [X | Rest], [X | Rest]).
insert(X, [Y | Rest], [Y | [X] ]) :– Rest == [ ], X < Y |
    true.                           % just need to satisfy the guards
insert(X, [Y | Rest], [Y | [ [ ] | [X]] ]) :– Rest == [ ], X>Y|
    true.                           % just need to satisfy the guards
insert(X, [Y | [L | R] ], [Y | [L1 | R] ]) :– X < Y | insert(X, L, L1).
insert(X, [Y | [L | R] ], [Y | [L | R1] ]) :– X > Y | insert(X, R, R1).
```

## Monitors via Semaphores

**Shared variables**

```
sem e= 1;           // one per monitor
int nc = 0;         // one per cond
queue q;            // one per cond
sem private [N];    // one entry per process

entry:        P(e);

wait(cv):     cv.nc++; cv.q.insert(myId); V(e); P(private[myId]); P(e);

signal(cv):   if (cv.nc > 0) {
                  cv.nc--;
                  V(private[cv.q.remove()]);
              }

exit:         V(e);
```

## Channels

```
chan ch ( signature )
      chan input ( char )  // used for character messages
      chan disk_access ( int cylinder, int block, int count, char* buf )
      chan result[n] (int)

send ch (args)
receive ch (args)       // blocking
empty (ch);             // predicate
```

## Line Assembly – not assembly line

```
chan input(char), output(char[MaxLine]);

process charToLine {
    char line[MaxLine]; int i = 0;
    while (true) {
        receive input (line[i]);
        while (line[i]!=CR && i<MaxLine) receive input (line[++i]);
    }
    line[i] = EOL;
    send output (line);
    i = 0;
}
```

## Sorting Network

```
chan in1(int), in2(int), out(int);

process merge {
    int v1, v2;
    receive in1(v1);
    receive in2(v2);
    while (v1 != EOS && v2 != EOS) {
        if (v1 <= v2) { send out(v1); receive in1(v1); }
        else { send out(v2); receive in2(v2); }
    }
    if (v1==EOS)
        while (v2 != EOS) { send out(v2); receive in2(v2); }
    else
        while (v1 != EOS) { send out(v1); receive in1(v1); }
    send out(EOS);
}
```

## Client/Server – one service

```
chan request(int clientID, types of input values);
chan reply[n](result type);

process server {
    int clientID;
    while (true) {
        receive request(clientID, inputs);
        // carry out operation
        send reply[clientID](results);
    }
}
process client [j=0 to n-1] {
    send request(j, args);
    receive reply[j](results);
}
```

## Client/Server – several services

```
chan request(int clientID, op, types of input values);
chan reply[n](result type);

process server {
    int clientID; op_kind kind;
    while (true) {
        receive request(clientID, kind, inputs);
        // cases to carry out various kinds of ops
        send reply[clientID](results);
    }
}
process client [j=0 to n-1] {
    send request(j, op, args);
    receive reply[j](results);
}
```

## Self-Scheduling Disk Driver

```
chan request(int clientID, int cyl, other arg types);
chan reply[n](result type);
process diskDriver {
    queue left, right; int clientID, cyl, headpos=1, nsaved=0;
    while (true) {
        while (!empty(request) || nsaved==0) {
            receive request(clientID, cyl, …);
            if (cyl<=headpos) left.insert(clientId, cyl, …);
            else right.insert(clientId, cyl, …);
            nsaved++;
        }
        if (left.size() == 0) get request from right
        if (right.size() == 0) get request from left
        else get better of two
        change head pos; nsaved--;
        send reply[clientId](result);
    }
}
```

## CSP – Communicating Sequential Processes

**Simple Communication**

```
    B ! e
        send expression e to process B
    A ? x
        accept an expression copied into variable x from process A
```

**Each process blocks until a match occurs (rendezvous)**

**More complex version of this is**

```
    Destination ! port (e1, e2, … , e4);
    Source ? port (x1, x2, … , xn);
```

**do B1 $\rightarrow$ S1 [ ] B2 $\rightarrow$ S2 [ ] … [ ] Bk $\rightarrow$ Sk od**

**if B1 $\rightarrow$ S1 [ ] B2 $\rightarrow$ S2 [ ] … [ ] Bk $\rightarrow$ Sk fi**

**are guarded commands that lead to choice if more than one guard is true and failure if all are false. Think of [ ] as "or." More complex guarded communications will be discussed later.**

## GCD in CSP

```
process GCD {
    int id, x, y;
    do true →
        Client[*] ? args(id, x, y);
        do
            x>y → x = x – y;
        [ ]
            x<y → y = y – x;
        od
        Client[id] ! result(x);
    od
}
```

**Client[i] does GCD ! args(i,v1,v2); GCD ? result(r);**

## Guarded Communication

**B; C $\rightarrow$ S1**

**B is an optional Boolean expression; C is a communication primitive**

**B and C together are the guard.**

**A guard succeeds if B is true and C causes no delay.**

**A guard blocks if B is true but C is not ready.**

**A guard fails if B is false.**

**A do or if will choose non-deterministically when multiple choices succeed.**

**A do or if will block if none of its choice succeed and at least one blocks.**

**A do terminates if all its guarded choices fail.**

**An if fails if all its guarded choices fail.**

## Allocator in CSP

```
process allocator {
    int avail = MaxUnits; set units = initial values;
    int index. unitId;
    do avail>0;
        Client[*] ? acquire(index) →
            avail --;
            remove(units, unitId);
            Client[index] ! reply(unitId);
        [ ]
        Client[*] ? release(index) →
            avail++;
            insert(units, unitId);
    od
}
```

## Program Flow Analysis

**Basic type is Scalar Analysis**

- **Concentrates on simple variable names**
- **Indexed array ref. A[I] is treated as a reference to all of object A**
- **This basic coverage ignores aliasing (multiple names for same object)**

**Basic Block**

- **One in, one out sequence of code**

**Local Analysis – done on single basic blocks**

**Intraprocedural Analysis – done within procedures**

**Interprocedural Analysis – done across procedures**

**Control Flow**

- **intra creates flow graph with procedure entry as initial node**
- **inter creates a call graph with main body as initial node**

**Data Flow**

- **determines accessibility of definitions and uses to each other**
- **UD chaining – given a variable use, what definitions reach this use**
- **DU chaining – given a variable definition, what uses are made of it**

## Data Flow Notations

**Program P consists of procedures, one of which is denoted p.**

**We assume one entry / one exit procedures.**
**A flowgraph G = (N, E, s) refers to a directed graph (N, E) and an initial node s in N, where there is a path from s to every node of G. Nodes can be statements or basic blocks. Commonly, they are the latter.**

```
Program SquareRoot;
var    L, N, K, M : integer; C : boolean;
begin
        (* start of block B1 *)    read(L); N := 0; K := 0; M := 1;        (* end of block B1 *)
        loop
                (* start of block B2 *)
                K := K + M; C := K > L;
                if C then break;    (* end of block B2 *)
                (* start of block B3 *)    N := N + 1; M := M + 2  (* end of block B3 *)
        end loop;
        (* start of block B4 *)    write(N)      (* end of block B4 *)
end.  (* SquareRoot *)
```

# Extracting Loops

**Let G = (N,E,s)**

**(1) a node s’ $\in$ N is the entry point for a loop in G iff there is an n’ $\in$ N such that (n’,s’) $\in$ E and s’ $\le$ n’. (n’ branches back)**

**(2) Let s’ be an entry point of a loop. The max loop with entry s’ is G’ = (N’,E’,s’), where N’ = {n” | $\exists$ a path from n” to s’ which contains only nodes “dominated” by s’}. s’ dominates n” if s’ is on every path from s (start node) to n”. E’=E $\cap$ (N’$\times$N’)**

**To do data flow analysis we wish to obey dominances, doing loop entries before their bodies, if conditions before their choices, etc.**

## Depth First Numbering

**A depth first traversal can be used to number nodes so that**

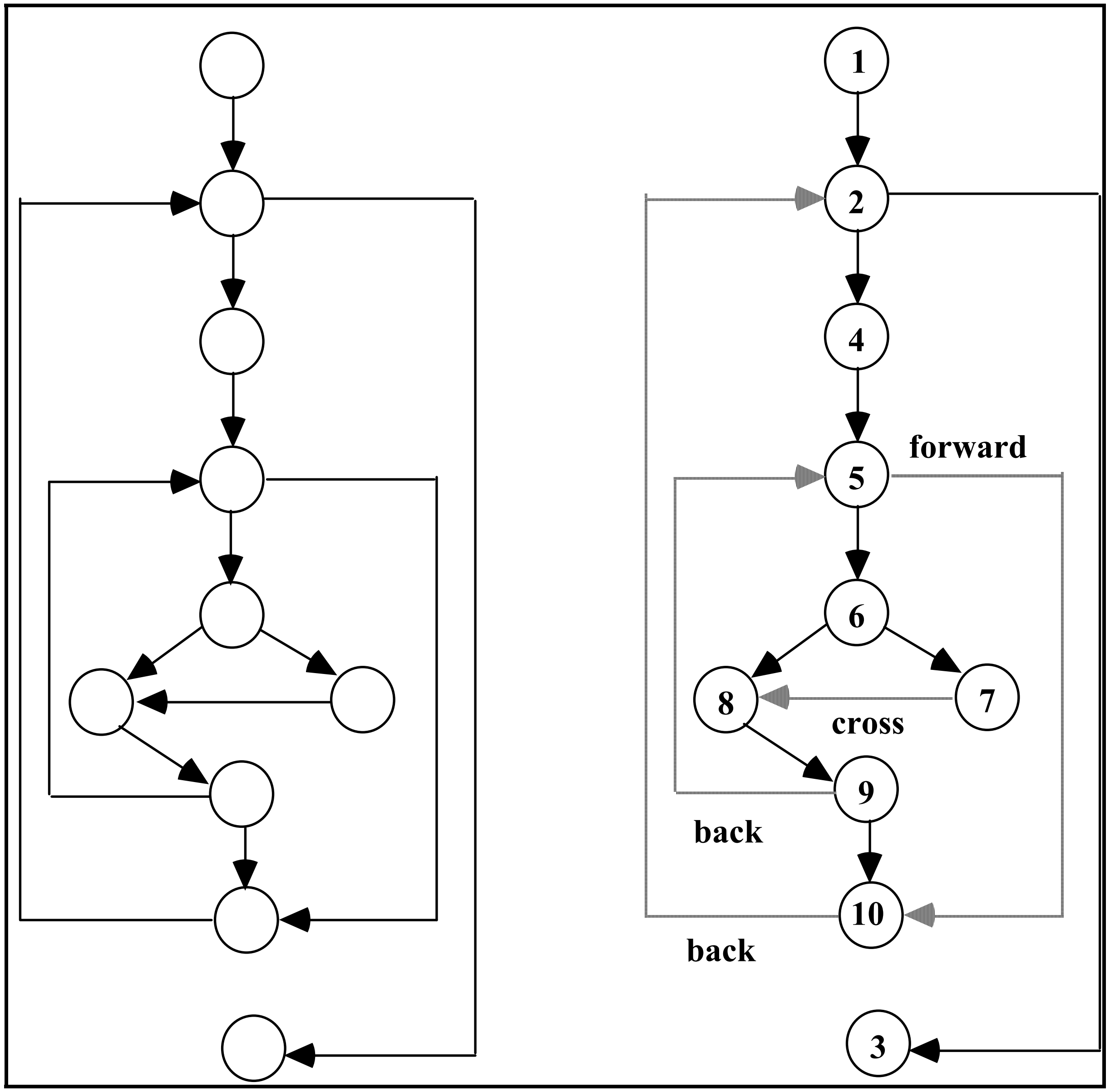
**n’ < n (n’ dominates n) implies #(n’) < #(n).**

**This is a total ordering that obeys all the restrictions of the partial ordering ≤.**

```
DFT( G : flowgraph )            (* G = (N,E,s) *)
    E’ = { };
    i := | N |;
    for every n in N do mark[n] := false;
    search( s )

Search( n : node )
    mark[n] := true;
    while unmarked_successors[n] ≠ { } do begin
        n’ := select( unmarked_successor[n] );
        E’ := E’ + { (n,n’) };
        Search( n’ )
    end; (* while *)
    rPostOrder[n] := i;
    i := i – 1
```

**This produces one of the natural orders. Visiting nodes based on these numbers speeds up data flow analysis.**
**Note n ≤ n’ implies rPostOrder[n] ≤ rPostOrder[n’].**
**Arcs are forward (unvisited node); back (visited but not numbered); cross (numbered).**
**Back arcs denote loops.**

# Categorizing Arcs in DFS Tree



## More Notation

**S_DEFS = { s | s is a statement that defines variables }**

**S_USES = { s | s is a statement that uses variables }**

**DEF[s] = { v | s is a definition of variable v }**

**USE[s] = { v | s is a use of variable v }**

**DEF[n] = { v | $\exists$ an outward exposed defn of v in n }**

**USE[n] = { v | $\exists$ an outward exposed use of v in n }**

**PRE[n] = VAR – DEF[n] /* preserved defs */**

**S_DEF[n] = { s | s is an outward exposed defn in n }**

**S_USE[n] = { s | s is an outward exposed use in n }**

**S_PRE[n] = { s’ | s’ $\in$ S_DEFS and, for all**
**s $\in$ S_DEF[n], DEF[s'] $\neq$ DEF[s] } // PRE stands for preserves**

**Reaching Definitions**

**RD[n] = { s | s $\in$ S_DEFS and s reaches n }**

**UD[n, v] = { s’ | s’ $\in$ RD[n] and v $\in$ DEF[s'] }**

**DU[n’, v] = {s | s $\in$ S_USE[n] for some n $\in$ N,**
**v $\in$ USE[s] and s’ $\in$ UD[n, v] }**

## Types of Data Flow

**Notation: For any node n, pred[n] is the set of all immediate predecessors of n and succ[n] is the set of all immediate successors.**

**ReachIn[n] = { s | p $\in$ pred[n] and s $\in$ ReachOut[p] }**

**ReachOut[n] = (ReachIn[n] $\cap$ S_PRE[n]) $\cup$ S_DEF[n]**

**In some papers this is (In[n] - Kill[n]) + Gen[n]**

**In any case, we have a recurrence relation and hence seek a fixed point. We want the least fixed point.**

**MAY – determine if a property may be possible. This is attacked by assuming no elements satisfy, then union in all those that might have the property. By starting with the empty set, we get the Least Upper Bound (LUB). This is conservative.**

**MUST – determine if a property must be true. This is attacked by assuming all elements satisfy, then intersecting all those that must have the property. By starting with the everything, we get Greatest Lower Bound (GLB). This is conservative.**

**FORWARD FLOW – information flows from the root towards leaves of the control flow graph.**

**BACKWARD FLOW – information goes from the leaves towards the root of the control flow graph.**

**Reaching Definitions is MAY / FORWARD FLOW**

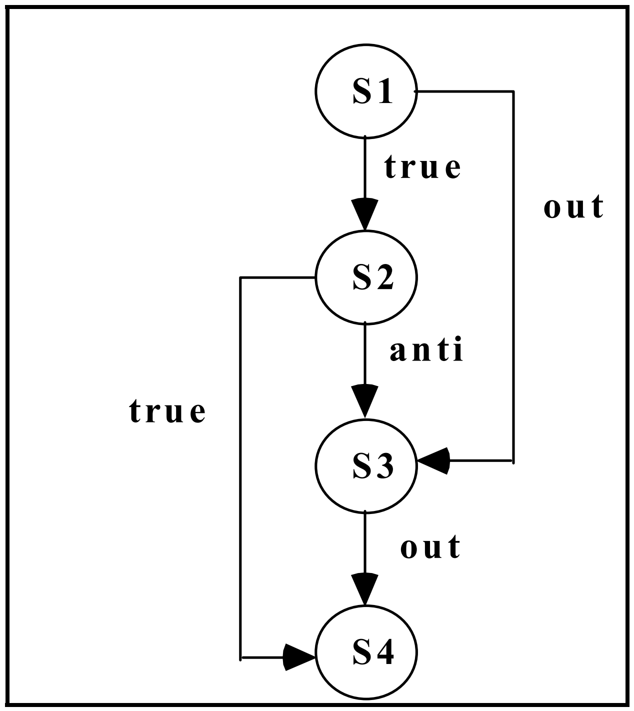
## Reaching Definitions Algorithm

```
For i := 1 to NBlocks do begin
    ReachOut[i] := S_DEF[i];
    ReachIn[i] := { }
end;
change := true;
while change do begin
    change := false;
    for i := 1 to NBlocks do begin
        newIn := { s | p ∈ pred[n] & s ∈ ReachOut[p] };
        if ReachIn[i] ≠ newIn then begin
            ReachIn[i] := newIn;
            oldOut := ReachOut[i];
            ReachOut[i] :=
                (ReachIn[i] ∩ S_PRE[i]) ∪ S_DEF[i];
            if oldOut ≠ ReachOut[i] then change := true
        end
    end
end
```

## Scalar Data Dependence

```
S1:       A := 1.0;
S2:       B := A + 3.1415;
S3:       A := .333 * (C – D);
…         …
S4:       A := (B * 3.8) / 2.718;
```

**S2 is true dependent on S1**
**S3 is anti-dependent on S2**
**S4 is output dependent on S3**



**Can use scalar data flow analysis to determine these dependencies.**

## Vector Data Dependence

```
        for i := 1 to 100 do begin
S:              A[2*i] := B[i] + 1;
S':             D[i] := A[2*i + 1]
        end
```

**If treat A, B and D as scalars then S' is true dependent on S and S is anti-dependent on S'. But it can't be so since S references only even numbered elements of A and S' references only off numbered elements of A. Thus, we can do the iterations independently. But how do we recognize this? The basis is Diophantine analysis – provided indices are linear in the *<u>for</u>* variable. In above, we can ask if there is an integral solution to**

**$1 \le X, Y \le 100$ such that $2X = 2Y + 1$**

**The answer is no, hence the indices cannot overlap. Even if we had for i:=1 to N, we can determine this.**

```
        for i := 2 to 10 do begin
S:              A[i] := B[i] + 1;
S':             D[i] := A[i – 1]
        end
```

**The relation is $X = Y - 1$, for $2 \le X, Y \le 10$. Can solve for all $2 \le X \le 9$, so there is true dependence.**

## Testing Data Dependence

**There are exact and inexact (but faster) tests for the existence of solutions to linear Diophantine equations. There is no test for polynomials of degree ≥ 4, and in fact exact solutions for lower degree polynomials are very hard.**

**One simple test is the GCD (Greatest Common Divisor) test. It is easiest seen by example.**

```
        for i := 1 to N do
                for j := 2 to M do begin
S:                      A[2*i + 2*j] := …;
…                       …
S':                     … := A[4*i – 6*j + 3]
                end
```

**These are independent if there is no solution to $2A + 2B = 4C - 6D + 3$**

**Can rewrite as $2A + 2B - 4C + 6D = 3$**

**But evenness of left says no solution is possible. This is recognized by gcd(left) = 2, gcd(right) = 3, but 2 is not a divisor of 3.**

**The technique is conservative, especially since it ignores regions. So, it says the following are possibly dependent**

```
        for i := 0 to 10 do
                for j := 0 to 10 do begin
S:                      A[2*i + j] := …;
…                       …
S':                     … := A[–i + 2*j – 21]
                end
```

**which translates to $2A + B + C - 2D = -21$. gcd(left)=1; gcd(right) = 21. But the restriction that $0 \le A, B, C, D \le 10$ can be used to deny a solution since the left side can be no smaller than -20.**

## Examples of Vectorizing

```
        for i := 1 to N do
S:              A[i + 1] := A[i] * B[i]     (* True Dependence *)
=====================================================
        for i := 1 to 100 do begin
S:              D[i] := A[i – 1] * D[i];    (* S depends on S’ *)
S’:             A[i] := B[i] + C[i]
        end
```

<u>**Reorder S and S’**</u>

```
        for i := 1 to 100 do begin
S’:             A[i] := B[i] + C[i]
S:              D[i] := A[i – 1] * D[i];    (* S depends on S’ *)
        end
```

<u>**Loop Distribution**</u>

```
        for i := 1 to 100 do
S’:             A[i] := B[i] + C[i]
        for i := 1 to 100 do
S:              D[i] := A[i – 1] * D[i];    (* S depends on S’ *)
```

<u>**Change to Vector Operations**</u>

```
S’:             A[1:100] := B[1:100] + C[1:100]
S:              D[1:100] := A[0:99] * D[1:100];
=====================================================
        for i := 1 to N do
                for j := 1 to N do
                        C[i, j] := C[i – 1, j] – D[i – 1, j + 1]
```

**Dependence is on outer loop only, so vectorize as**

```
        for i := 1 to N do
                C[i, 1:N] := C[i – 1, 1:N] – D[i – 1, 2:N+1]
```

## Program Transformations Used to Parallelize Code

Privatization -- Give each process a copy of a variable

Scalar Expansion -- Replace a scalar by an array

Loop Distribution -- Split one loop into two separate ones

Loop Fusion -- Combine two loops into one

Loop Interchange -- Interchange inner and outer loops

Loop Unrolling -- Replace loop body and do fewer iterations

Strip Mining -- Divide iterations of one loop into two nested loops

Unroll and jam -- Combine interchange, strip mining and unrolling

Loop Skewing -- Alter loop bounds to expose wavefront parallelism

Loop Blocking (Tiling) -- Divide iteration space into rectangular blocks

## Processes Scheduling Problem

A Process Scheduling Problem can be described by

m processors $P_1$, $P_2$, …, $P_m$,

processor timing functions $S_1$, $S_2$, …, $S_m$, each describing how the corresponding processor responds to an execution profile,

additional resources $R_1$, $R_2$, …, $R_k$, e.g., memory and other serially reusable items,

a transmission cost matrix $C_{ij}$ $(1 \le i , j \le m)$, based on processor data sharing,

tasks to be executed $T_1$, $T_2$, …, $T_n$,

task execution profiles $A_1$, $A_2$, …, $A_n$,

a partial order defined on the tasks
such that $T_i < T_j$ means that $T_i$ must complete before $T_j$ can start execution,

a communication matrix $D_{ij}$ $(1 \le i , j \le n)$ where $D_{ij}$ can be non-zero only if $T_i < T_j$,

weights $W_1$, $W_2$, …, $W_n$ interpreted as the cost of deferring execution of a task.

## Scheduling of Processes and NP-Completeness

The intent of a scheduling algorithm is to minimize the sum of the weighted completion times of all tasks, while obeying the constraints of the task system. Weights can be made unusually large to impose actual deadlines.

The general scheduling problem is quite complex, but even simpler instances, where the processors are uniform, there are no additional resources, there is no data transmission, the execution profile is just processor time and the weights are uniform, are very hard.

In fact, if we just specify the time to complete each task and we have no partial ordering, then finding an optimal schedule on two processors is an NP-complete problem. (The notion of NP Complete is on other overheads.)

## 2-Processor Scheduling

The problem of optimally scheduling n tasks $T_1$, $T_2$, …, $T_n$ onto 2 processors with an empty partial order < is the same as that of dividing a set of positive whole numbers into two subsets, such that the numbers are as close to evenly divided. So, for example, given the numbers

3, 2, 4, 1

we could try a “greedy” approach as follows:
put 3 in set 1
put 2 in set 2
put 4 in set 2 (total is now 6)
put 1 in set 1 (total is now 4)

This is not the best solution. A better option is to put 3 and 2 in one set and 4 and 1 in the other. Such a solution would have been attained if we did a greedy solution on a sorted version of the original numbers. In general, however, sorting doesn’t always work.

## 2-Processor Scheduling

Try the unsorted list
7, 7, 6, 6, 5, 4, 4, 5, 4

Greedy (Always in one that is least used)
7, 6, 5, 5 = 23
7, 6, 4, 4, 4 = 25

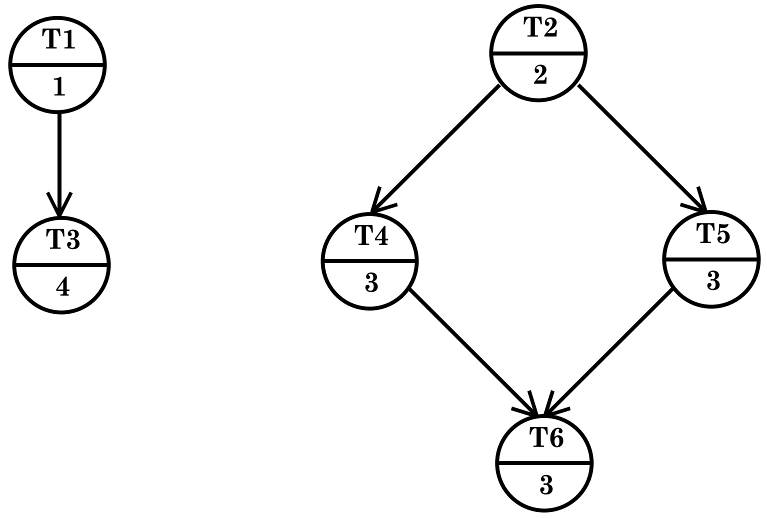
Optimal
7, 6, 6, 5 = 24
7, 4, 4, 4, 5 = 24

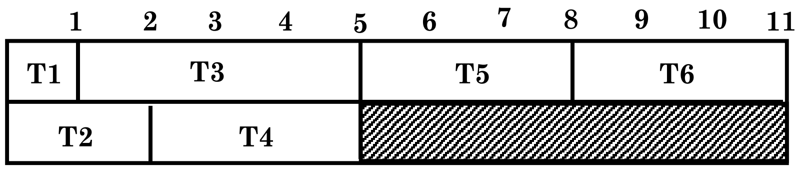
Sort it
7, 7, 6, 6, 5, 5, 4, 4, 4

7, 6, 5, 4, 4 = 26
7, 6, 5, 4 = 22

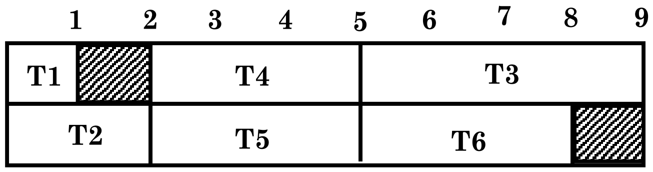
Even worse than greedy unsorted

# 2-Processor with Partial Ordering

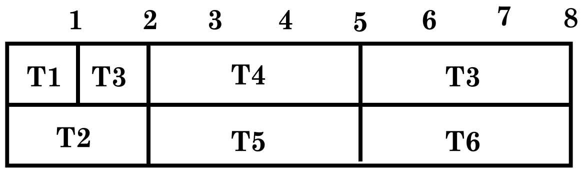




List Schedule with L = {T1,T2,T3,T4,T5,T6}

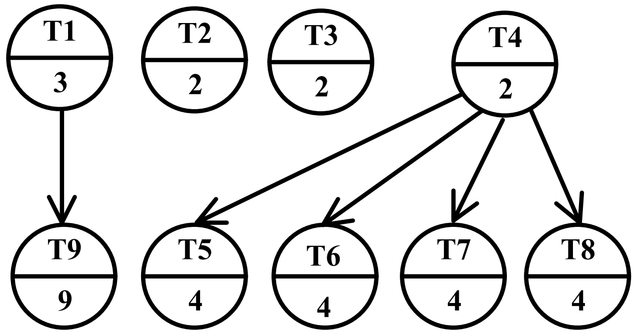


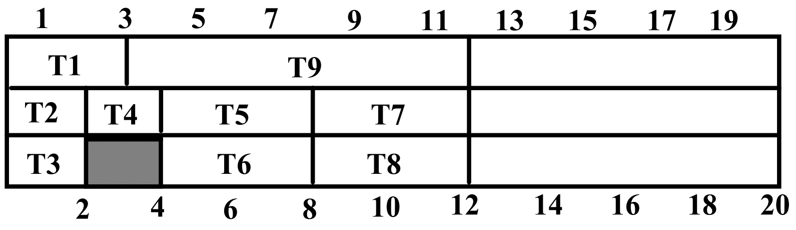
Non-Preemptive, Delays Allowed



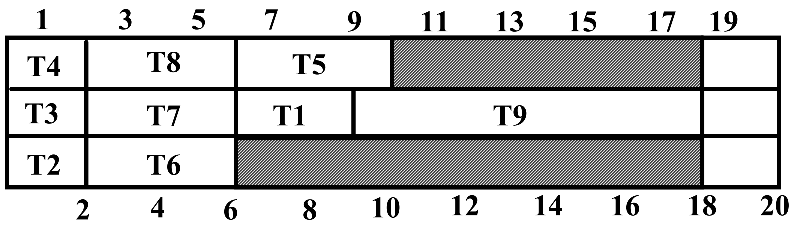
Preemptive

## Anomalies Everywhere

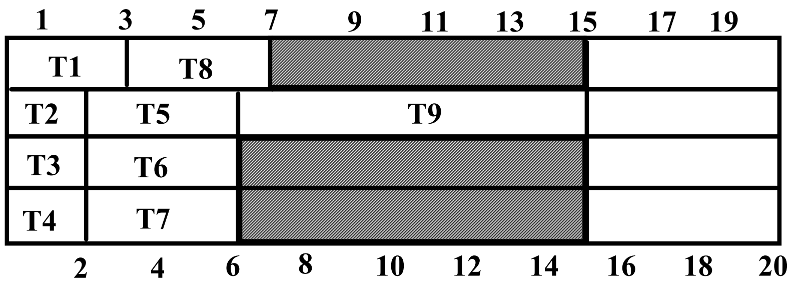




List Schedule with L = {T1,T2,T3,T4,T5,T6,T7,T8,T9}

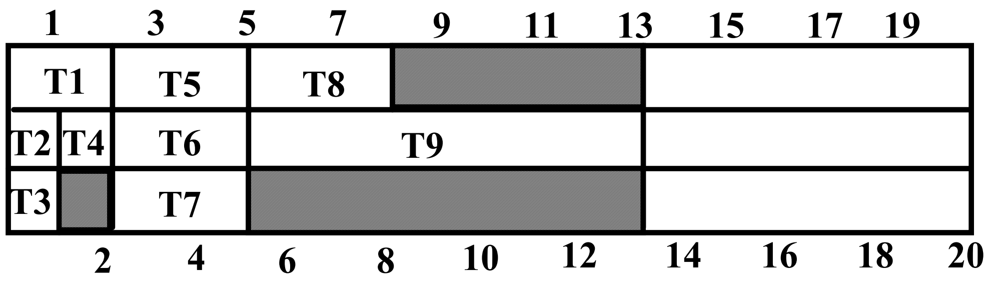


List Schedule with L = {T9,T8,T7,T6,T5,T4,T3,T2,T1}

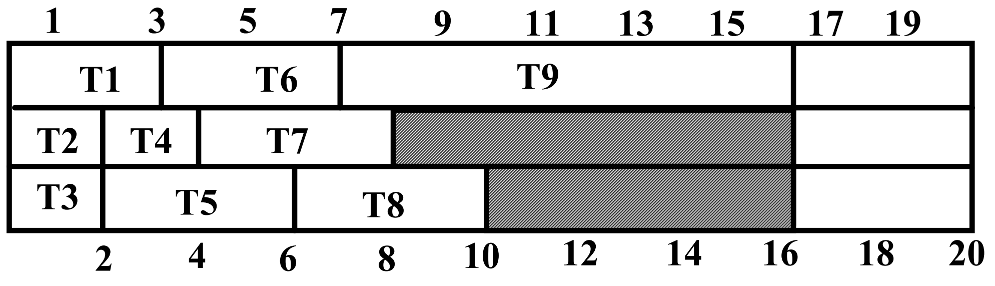


Use Original List with 4 Processors

## More Anomalies



**Original List Schedule but with All Times Reduced by 1**



**Original List Schedule but with T5 and T6 Independent**

## NP Problems

There is a large class of problems for which no fast algorithms have been devised, but for which no proof has ever been presented that confirms the inherent intractability of these problems.

Of particular interest is a class of problems that can be solved in polynomial time, provided we can always make the correct decision whenever the algorithm has a choice between courses of action.

For example, consider the simple 2-processor scheduling problem, restructured as a decision problem.  We could just guess which processor to assign to each task.  Then it would be a simple matter to check to see if our guesses were correct.  This algorithm would clearly be polynomial, but it would only work if our guesses were correct on the first try.  If, in contrast, we had to try another guess and then another guess, we would be no better off than running a try all combinations algorithm.

Such problem are said to be in NP, the class of problems solvable in polynomial time by a non-deterministic algorithm.

## NP Problems and Parallelism

The class NP includes all easy problems, since, if we can solve a problem in polynomial time without guessing, we can clearly solve it in polynomial time with guesses.

The class NP can also be categorized as consisting of problems that can be solved in polynomial time on a machine that has an unbounded number of processors (the ultimate parallelism). This should be evident, since we could alter the non-determinism so that it starts a separate machine for each guess. We might have an exponential number of processors running in parallel, but no one of them will take more than polynomial time to gets its task done. We then say "yes" to the original question if any of the processors says "yes".

## NP-Complete Problems

The class NP has some members that are the hardest ones in this class. These problems are called NP-Complete, and are such that, if any of them submits to a fast algorithm, then all the NP problems will have been shown to be easy. Similarly, if any can be shown to be intractable then all NP-complete problems will have been shown to be intractable. The 2-processor scheduling and the bin packing problems are instances of NP-complete problems.

One of the big problems of modern computer science is the question
"Is P = NP?"
Here P stands for the class of problems that can be solved in polynomial time by a conventional, deterministic algorithm running on a machine with a bounded number of processors. The solution to this question lies in our being able to determine the complexity of any NP-complete problem. If we can demonstrate a conventional polynomial algorithm for any NP complete problem, then all such problems are in P, and hence are tractable. If any NP-complete problem can be shown inherently exponential, then $P \neq NP$.

## Heuristics and NP-Completeness

While it is not known whether or not P = NP?, it is clear that we need to “solve” problems that are NP-complete since many practical scheduling and networking problems are in this class.  For this reason we often choose to find good “heuristics” which are fast and provide acceptable, though not perfect, answers.  The First Fit and Best Fit algorithms we previously discussed are examples of such acceptable, imperfect solutions to bin packing.

## Critical Path or Level Strategy – UET

A UET is a Unit Execution Tree. Our Tree is funny. It has a single leaf by standard graph definitions.

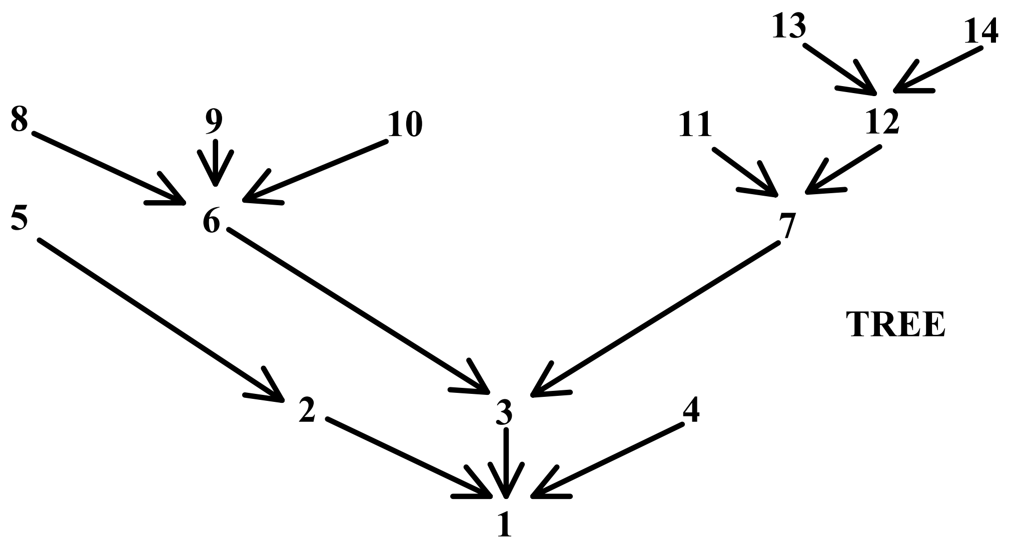
1. Assign L(T) = 1, for the leaf task T
2. Let labels 1, …, k-1 be assigned. If T is a task with lowest numbered immediate successor then define L(T) = k (non-deterministic)

This is an order n labeling algorithm that can easily be implemented using a breadth first search.

Note: This can be used for a forest as well as a tree. Just add a new leaf. Connect all the old leafs to be immediate successors of the new one. Use the above to get priorities, starting at 0, rather than 1. Then delete the new node completely.

Note: This whole thing can also be used for anti-trees. Make a schedule, then read it backwards. You cannot just reverse priorities.

## Applying Level Strategy to UET



| | | | | | |
|---|---|---|---|---|---|
| 14 | 12 | 8 | 6 | 3 | 1 |
| 13 | 10 | 7 | 4 | ▨ | ▨ |
| 11 | 9 | 5 | 2 | ▨ | ▨ |

M=3

**Theorem:** Level Strategy is optimal for unit execution, m arbitrary, forest precedence

## Level Strategy – DAG with Unit Time

1. Assign L(T) = 1, for an arb. leaf task T
2. Let labels 1, …, k-1 be assigned. For each task T such that

   {L(T’) is defined for all T’ in Successor(T)}

   Let N(T) be decreasing sequence of set members in

   {S(T’) | T’ is in S(T)}

   Choose T* with least N(T*).
   Define L(T*) = K.

This is an order $n^2$ labeling algorithm. Scheduling with it involves n union / find style operations. Such operations have been shown to be implementable in nearly constant time using an “amortization” algorithm.

**Theorem**: Level Strategy is optimal for unit execution, m=2, dag precedence.

# Sample Scheduling Question#1

Consider the simple scheduling problem where we have a set of independent tasks running on a fixed number of processors, and we wish to minimize the time at which the last task completes.

How would a list (first fit, no preemption) strategy schedule tasks with the following IDs and execution times onto four processors? Answer by showing a Gantt chart for the resulting schedule (write the task ID into each time/processor slot used.)

**(T1,1)** **(T2,1)** **(T3,3)** **(T4,3)** **(T5,2)** **(T6,2)** **(T7,4)**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

Now show what would happen if the times were sorted non-decreasing.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

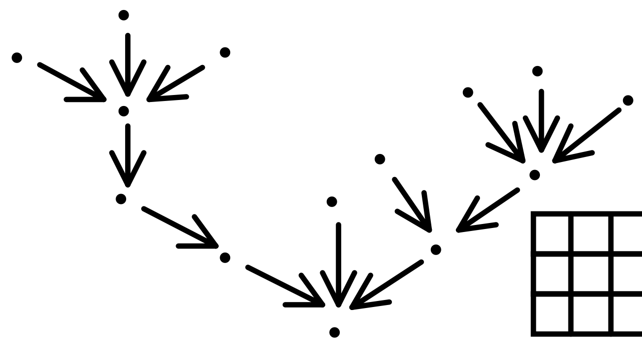
Now show what would happen if the times were sorted non-increasing.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

This problem is, in general, NP-complete. Given this fact, what is the complexity of the best known optimal scheduling algorithm? Is this the theoretical lower bound on the problem's inherent complexity? If not, why not? If so, how do we know this?

# Sample Scheduling Question#2

Some scheduling problems can be efficiently solved using a level (critical path) algorithm. The first step of such an algorithm is the assignment of priorities (lowest is 1) to each task and the creation of a list schedule based on these priorities. Unit execution time tasks with a forest (or anti-forest) task graph are amenable to a level algorithm. Given the following such system, assign priorities to the right of each task as represented by a dot (•), then show the resultant 3-processor schedule.

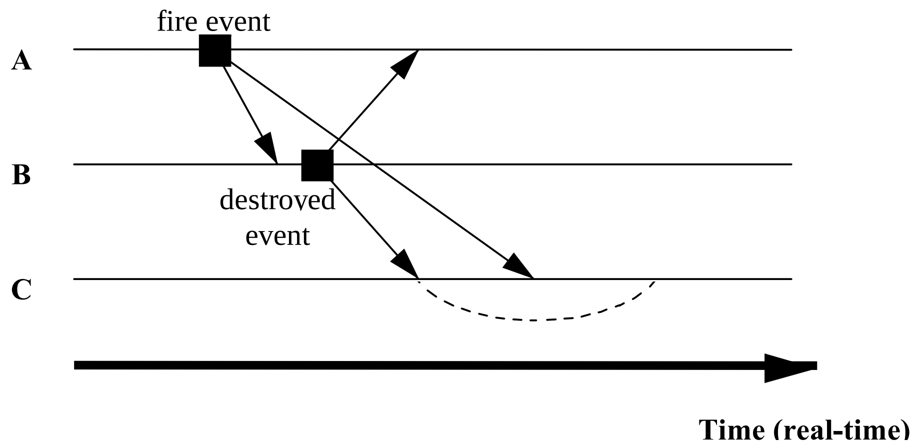


| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |

## HLA Time Management

**Timing Problems**

**Critical when there's an observer/interactor**



**Message Ordering**

**Receive Order**

**Priority Order**

**Time Stamp Order**

**Causal Order**

**CATOCS (**causally and totally ordered communications support**)**

## IBM TSpaces and our Bid.com Example -- BidItem

```
package bid;

import java.io.Serializable;

/**
 * Title:       TSpaces bid.com
 * Description:  A prototypical bid system implemented using IBM TSpaces
* @author Charles E. Hughes
*/

public class BidItem implements Serializable {

  String item;
  String price;

  public BidItem(String item, String price) {
    this.item = item;
    this.price = price;
  }
}
```

## BidTuple # 1

```
package bid;

import com.ibm.tspaces.*;
import java.io.Serializable;

public class BidTuple extends SubclassableTuple implements Serializable {

 /**
 ** Default constructor will build a template that would match
 ** all bid tuples in the space.
 */
 public BidTuple() throws TupleSpaceException {
  super(new Field(String.class),new Field(String.class));
 }

 /**
 ** Constructor with only key specified  will build a template
 ** for retrieving the data
 */
 public BidTuple(String key)  throws TupleSpaceException {
  super(key,new Field(String.class));
 }
```

## BidTuple # 2

```
/**
 ** Constructor with both key and data specified will create
 ** a Tuple that can be written to BidSpaces
 */
 public BidTuple(String key, String data) throws TupleSpaceException {
   super(key,data);
 }

 /**
 ** This is an example of defining a method within the SublassableTuple
 ** that will hide some of the ugly Tuple and Field code.
 ** This method will return the data Field from the tuple.
 */
 public String getData()  throws TupleSpaceException {
   // extract the contents of the 2nd field.
   return (String)this.getField(1).getValue();
 }

 public boolean matches( SuperTuple t ) {
   // This message will show up in the output from the TSServer.
   Debug.out("SubclassMyTuple.matches() called by server!!!!");
   return super.matches(t);
 }  // end matches()
}
```

## RegisterTuple # 1

```
package bid;

import com.ibm.tspaces.*;
import java.io.Serializable;

public class RegisterTuple extends SubclassableTuple implements Serializable {

 // Default constructor will build a template that would match registration tuples in space.
  public RegisterTuple() throws TupleSpaceException {
    super(new Field(String.class), new Field(String.class));
  }
// Constructor with only key specified will build a template for retrieving the data
  public RegisterTuple(String key)  throws TupleSpaceException {
    super(key,new Field(String.class));
  }

 // Constructor with both key and data for a Tuple that can be written to BidSpaces
  public RegisterTuple(String key, String data) throws TupleSpaceException {
    super(key,data);
  }
```

## RegisterTuple # 2

```
 /**
** Access fields without the ugly Tuple and Field code.
*/
public String getData()  throws TupleSpaceException {
  // extract the contents of the 2nd field.
  return (String)this.getField(1).getValue();
}

public String getName()  throws TupleSpaceException {
  // extract the contents of the 2nd field.
  return (String)this.getField(0).getValue();
}

/**
 * Tuple matching will now be traced
 */
public boolean matches( SuperTuple t ) {
  // This message will show up in the output from the TSServer.
  Debug.out("SubclassMyTuple.matches() called by server!!!!");
  return super.matches(t);
}  // end matches()
```

## TSpacesBidDotCom # 2

```
package bid;
import javax.swing.UIManager;
import java.awt.*;
import com.ibm.tspaces.*;

public class TSpacesBidDotCom {
  boolean packFrame = false;

  /**Construct the application*/
  public TSpacesBidDotCom() {
    BidFrame frame = new BidFrame();
    // Pretty up the window …
    Debug.setDebugOn(true);
    BidAgent agent = new BidAgent();
    frame.setAgent(agent);   agent.setFrame(frame);   agent.sayHello();
  }
  /**Main method*/
  public static void main(String[] args) {
    try {  UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());  }
    catch(Exception e) {e.printStackTrace();}
    new TSpacesBidDotCom();
  }
}
```

## BidAgent # 1

```
package bid;
import java.util.Vector;
import com.ibm.tspaces.*;
import java.io.Serializable;

public class BidAgent implements Serializable, Callback {
  TupleSpace space;
  int myId;
  BidFrame frame;

  BidAgent() {
    try {
      space = new TupleSpace("TSpaceBidDotCom");
      Transaction trans = new Transaction();  trans.addTupleSpace(space);
      trans.beginTrans();
        Tuple id = space.take("ID", new Field(Integer.class));
        if (id == null) {myId = 1;  space.write("Offer", new Vector()); }
        else   myId = ((Integer)id.getField(1).getValue()).intValue();
        space.write("ID", new Integer(myId+1));
      trans.commitTrans();
      Tuple hello = new Tuple("Hello", new Integer(myId), new Field(Integer.class));
      space.eventRegister(TupleSpace.WRITE, hello, this, true);
      Tuple offer = new Tuple("Offer", new Field(Vector.class));
      space.eventRegister(TupleSpace.WRITE, offer, this, true);
    } catch (Exception e) {System.out.println(e); System.exit(1);}
  }
```

## BidAgent # 2

```
 void sayHello() {
  try {for (int i = 1; i < myId; i++) space.write("Hello", new Integer(i), new Integer(myId)); } catch …
 }
 void offer(String item, String price) {
  try {
    Tuple offer = space.take("Offer", new Field(Vector.class));
    Vector offerings = (Vector)(offer.getField(1).getValue());  offerings.add(item);  offerings.add(price);
     space.write("Offer", offerings);
  } catch (Exception e) {System.out.println("Offer " + e); System.exit(1);}
 }
 public boolean call(String eventname, String tsName, int seqNum,  SuperTuple tuple, boolean exception) {
  try { if (!exception) {
      if (((String)tuple.getField(0).getValue()).equals("Hello")) {
        int friend = ((Integer)tuple.getField(2).getValue()).intValue();
        System.out.println("Hello to #"+myId+" from #"+friend); }
      else if (((String)tuple.getField(0).getValue()).equals("Offer"))
        frame.setOfferings((Vector)tuple.getField(1).getValue());      }
  } catch (Exception e) {System.out.println("Call " + e); System.exit(1);}
  return false;
 }
 void setFrame(BidFrame frame) {
  this.frame = frame;
  try {
    Tuple tuple = space.read("Offer", new Field(Vector.class));
    frame.setOfferings((Vector)tuple.getField(1).getValue());
  } catch (Exception e) {System.out.println("Offer " + e); System.exit(1);}
 }
}
```

## BidFrame # 1

```
package bid;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
import com.ibm.tspaces.*;

public class BidFrame extends JFrame {
  BidAgent agent;

  JPanel contentPane;
  JLabel bidListLabel = new JLabel();
  JButton offerButton = new JButton();
  JButton acceptButton = new JButton();
  JToggleButton bidButton = new JToggleButton();
  JTextField itemField = new JTextField();
  JLabel itemLabel = new JLabel();
  JTextField priceField = new JTextField();
  JLabel priceLabel = new JLabel();
  JComboBox offeringsList = new JComboBox();
```

## BidFrame # 2

```
/**Construct the frame*/
public BidFrame() {
  enableEvents(AWTEvent.WINDOW_EVENT_MASK);
  try {
    jbInit();
  }
  catch(Exception e) {
    e.printStackTrace();
  }
}
/**Component initialization*/
private void jbInit() throws Exception  {
  // Initialize GUI stuff …
  offerButton.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
      offerButton_actionPerformed(e);
    }
  });
}
/**Overridden so we can exit when window is closed*/
protected void processWindowEvent(WindowEvent e) {
  super.processWindowEvent(e);
  if (e.getID() == WindowEvent.WINDOW_CLOSING) {
    System.exit(0);
  }
}
```

## BidFrame # 3

```
 void offerButton_actionPerformed(ActionEvent e) {
  agent.offer(itemField.getText(), priceField.getText());
 }

 void setAgent(BidAgent agent) {
  this.agent = agent;
 }

 void setOfferings(Vector offerings) {
  Iterator iter = offerings.iterator();
  offeringsList.removeAllItems();
  while (iter.hasNext()) {
   String item = (String)iter.next();
   String price = (String)iter.next();
   offeringsList.addItem(item + " at " + price);
   offeringsList.revalidate();
  }
 }

}
```

## Simulating Pure Tuple Space in an Applet

```
public class Tuple extends Object {
   static private int tupleCount = 0;
   private String key;
   private String who;
   private int id;
   private Object value;

   public Tuple(String key, String who, Object value) {
      this.key = key; this.who = who;  this.value = value;  id = ++tupleCount;
   }
   public void setFields(Tuple tuple) {
      key = tuple.key;  who = tuple.who;  id = tuple.id; value = tuple.value;
   }
   public void appendHelper(String helper) {
      who = who + "+" + helper;
   }
   public String key() { return key; }

   public String who() { return who; }

   public String id() { return String.valueOf(id); }

   public Object value() { return value; }
 }
```

## TupleSpace Applet # 1

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class TupleSpace extends Applet {
   private MultiHashtable tuples = new MultiHashtable();
   private List display = new List(10);

   // adds tuple to tuples. tuple.key() is used as key
   public void out(Tuple tuple) {
      tuples.put(tuple.key(),tuple);
      displayKeys();
   }

   // uses tuple.key() to get and delete matching tuple;
   // passed tuple is replaced by one extracted from tuple space;
   // passed who and value are irrelevant
   // blocks until successful
   public void in(Tuple tuple) {
      Tuple t;
      while ((t = (Tuple) tuples.remove(tuple.key())) == null) {
        try {Thread.sleep(100);} catch (InterruptedException e){}
     }
      tuple.setFields(t);
      displayKeys();
   }
```

## TupleSpace Applet # 2

```
// uses tuple.key() to get and delete matching tuple;
// passed tuple is replaced by one extracted from tuple space;
// passed who and value are irrelevant
// true if success, false otherwise
public boolean inp(Tuple tuple) {
   Tuple t = (Tuple) tuples.remove(tuple.key());
   if (t != null) {
      tuple.setFields(t);
   }
   displayKeys();
   return t != null;
}

// uses tuple.key() to get matching tuple;
// passed tuple is replaced by one read from tuple space;
// passed who and value are irrelevant
// blocks until successful
public void rd(Tuple tuple) {
   Tuple t;
   while ((t = (Tuple) tuples.get(tuple.key())) == null) {
     try {Thread.sleep(100);}
     catch (InterruptedException e){}
  }
   tuple.setFields(t);
}
```

# TupleSpace Applet # 3

```
// uses tuple.key() to get matching tuple;
// passed tuple is replaced by one read from tuple space;
// passed who and value are irrelevant
// true if success, false otherwise
public boolean rdp(Tuple tuple) {
   Tuple t = (Tuple) tuples.get(tuple.key());
   if (t != null) {
      tuple.setFields(t);
   }
   return t != null;
}

// randomly selects one of the TupleSpaceClient applets;
// invokes its eval service with tuple as arg
public void eval(Tuple tuple) {
   Vector collection = new Vector();
     Enumeration e = getAppletContext().getApplets();
     while (e.hasMoreElements()) {
     Applet applet = (Applet)e.nextElement();
     if (applet instanceof TupleSpaceClient) {
        collection.addElement(applet);
     }
  }
  if (!(collection.isEmpty())) {
     int select = (int) (collection.size()*Math.random());
     ((TupleSpaceClient) collection.elementAt(select)).eval(tuple);
  }
}
```

## TupleSpace Applet # 4

```
 public void displayKeys() {
   display.removeAll();
   Enumeration e = tuples.keys();
     while (e.hasMoreElements()) {
       String key = (String) e.nextElement();
       display.add(key + "(" + String.valueOf(tuples.size(key)) + ")");
  }
 }

 public void init() {
     Label label = new Label("Tuple Space Server",Label.CENTER);
   add(label);
   add(display);
 }

 public void paint(Graphics g) {
   g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
 }

 public String getAppletInfo() {
   return "TupleSpace by Charles E. Hughes";
 }

}
```

## TupleSpaceClient # 1

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Enumeration;

public class TupleSpaceClient extends Applet {
   protected TextField key = new TextField("Enter Key", 24);
   protected TextField sender = new TextField("Sender Name", 24);
   protected TextField tupleId = new TextField("Tuple Number", 24);
   protected String object = "OBJECT";
   protected String appletName;
   protected TupleSpace tupleSpace = null;

   protected void delay(int maxSeconds) {
            try {Thread.sleep((int)(maxSeconds*1000 * Math.random()));}
            catch (InterruptedException e){}
        }

   protected void getServer() {
      if (tupleSpace == null) {
        Enumeration e = getAppletContext().getApplets();
        while (e.hasMoreElements()) {
            Applet applet = (Applet)e.nextElement();
            if (applet instanceof TupleSpace) tupleSpace = (TupleSpace) applet;
        }
      }
   }
```

## TupleSpaceClient # 2

```
protected void clearResultFields() {
   sender.setText("");
   tupleId.setText("");
}

class OUTButtonHandler implements ActionListener {
   public void actionPerformed(ActionEvent event) {
      getServer();
      delay(5);
      tupleSpace.out(new Tuple(key.getText(),appletName,object));
      clearResultFields();
   }
}

class INButtonHandler implements ActionListener {
   public void actionPerformed(ActionEvent event) {
      getServer();
      Tuple tuple = new Tuple(key.getText(),appletName,object);
      clearResultFields();
      delay(5);
      tupleSpace.in(tuple);
      sender.setText(tuple.who());
      tupleId.setText(tuple.id());
   }
}
```

## TupleSpaceClient # 3

```
class INPButtonHandler implements ActionListener {
   public void actionPerformed(ActionEvent event) {
      getServer();
      Tuple tuple = new Tuple(key.getText(),appletName,object);
      clearResultFields();
      delay(5);
      if (tupleSpace.inp(tuple)) {
         sender.setText(tuple.who());
         tupleId.setText(tuple.id());
      } else {
         sender.setText("NO TUPLE");
      }
   }
}

class RDButtonHandler implements ActionListener {
   public void actionPerformed(ActionEvent event) {
      getServer();
      Tuple tuple = new Tuple(key.getText(),appletName,object);
      clearResultFields();
      delay(5);
      tupleSpace.rd(tuple);
      sender.setText(tuple.who());
      tupleId.setText(tuple.id());
   }
}
```

## TupleSpaceClient # 4

```
class RDPButtonHandler implements ActionListener {
   public void actionPerformed(ActionEvent event) {
      getServer();
      Tuple tuple = new Tuple(key.getText(),appletName,object);
      clearResultFields();
      delay(5);
      if (tupleSpace.rdp(tuple)) {
         sender.setText(tuple.who());
         tupleId.setText(tuple.id());
      } else {
         sender.setText("NO TUPLE");
      }
   }
}

class EVALButtonHandler implements ActionListener {
   public void actionPerformed(ActionEvent event) {
      getServer();
      clearResultFields();
      delay(5);
      tupleSpace.eval(new Tuple(key.getText(),appletName,object));
   }
}
```

## TupleSpaceClient # 5

```
class EVALTuple extends Thread {
   private Tuple tuple;

   public EVALTuple(Tuple tuple) {
      this.tuple = tuple;
   }

   public void run() {
      delay(10);
      getServer();
      tuple.appendHelper(appletName);
      tupleSpace.out(tuple);
   }
}

public void eval(Tuple tuple) {
   new EVALTuple(tuple).start();
}
```

## TupleSpaceClient # 6

```
  public void init() {
     appletName = getParameter("NAME");
     Label label = new Label(appletName, Label.CENTER);
     add(label);  add(key);  add(sender);  add(tupleId);
     Button outButton = new Button("OUT"); add(outButton);
     outButton.addActionListener(new OUTButtonHandler());
     Button inButton = new Button("IN"); add(inButton);
     inButton.addActionListener(new INButtonHandler());
     Button inpButton = new Button("INP"); add(inpButton);
     inpButton.addActionListener(new INPButtonHandler());
     Button rdButton = new Button("RD"); add(rdButton);
     rdButton.addActionListener(new RDButtonHandler());
     Button rdpButton = new Button("RDP"); add(rdpButton);
     rdpButton.addActionListener(new RDPButtonHandler());
     Button evalButton = new Button("EVAL"); add(evalButton);
     evalButton.addActionListener(new EVALButtonHandler());
  }

  public void paint(Graphics g) {
     g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
  }

  public String getAppletInfo() {
     return "TupleSpaceClient by Charles E. Hughes";
  }

}
```

## MultiHashtable # 1

```
import java.lang.*;
import java.util.*;
public class MultiHashtable extends Hashtable {

   public synchronized Object get(Object key) {
      Object obj = super.get(key);
      if (obj == null) return null;
      if (!(obj instanceof Vector)) return null;
      return ((Vector) obj).elementAt((int)(((Vector) obj).size()*Math.random()));
   }

   public synchronized Object remove(Object key) {
      Object obj = super.get(key);
      if (obj == null) return null;
      if (!(obj instanceof Vector)) return null;
      int choice = (int)(((Vector) obj).size()*Math.random());
      Object result = ((Vector) obj).elementAt(choice);
      ((Vector) obj).removeElementAt(choice);
      if (((Vector) obj).size() == 0)  super.remove(key);
      return result;
   }
```

## MultiHashtable # 2

```
   public synchronized Object put(Object key,Object value) {
      Object obj = super.get(key);
      if (obj == null) {
         obj = new Vector();
      }
      if (!(obj instanceof Vector)) return null;
      Object result = ((Vector) obj).clone();
      ((Vector) obj).addElement(value);
      super.put(key,obj);
      return result;
   }

   public synchronized int size(Object key) {
      Object obj = super.get(key);
      if (obj == null) return 0;
      if (!(obj instanceof Vector)) return 0;
      return ((Vector) obj).size();
   }
}
```

## Centralized versus symmetric versus ring reduction

**Reduction is so common that it deserves another visit.**

**The issues that arise here are all related to how much parallel communication we can sustain. That is highly dependent on the architecture with which we are working. Specifically, it depends on the interconnection network, a topic we studied earlier in the term.**

**The next two overheads summarize results concerning the effects of such interconnections on various communication tasks. Vipin Kumar's  book is an excellent source of detailed discussions of this topic.**

## Store and Forward Broadcasting

| Operation | Ring | 2d Mesh | Hypercube |
|---|---|---|---|
| One-to-all | $(t_s + t_w m) * \lceil p/2 \rceil$ | $2(t_s + t_w m) * \lceil \sqrt{p}/2 \rceil$ | $(t_s+t_w m) \lg p$ |
| All-to-all | $(t_s+t_w m) * (p-1)$ | $2t_s(\sqrt{p}-1) + t_w m (p-1)$ | $t_s \lg p + t_w m (p-1)$ |
| One-to-all personalized | $(t_s+t_w m) * (p-1)$ | $2t_s(\sqrt{p}-1) + t_w m (p-1)$ | $t_s \lg p + t_w m (p-1)$ |
| All-to-all personalized | $(t_s+t_w m p/2) * (p-1)$ | $(2t_s+t_w m p) * (\sqrt{p}-1)$ | $(t_s+t_w m p/2) * \lg p$ |
| Circular q-shift | $(t_s+t_w m) * \lfloor p/2 \rfloor$ | $(t_s+t_w m) * (2\lfloor \sqrt{p}/2 \rfloor+1)$ | $(t_s+t_w m) * (2 \lg p - 1)$ |

## Cut Through Broadcasting

| Operation | Ring | 2d Mesh | Hypercube |
|---|---|---|---|
| One-to-all | $(t_s+t_w m) \lg p + t_h(p-1)$ | $(t_s+t_w m) \lg p + 2t_h(\sqrt{p}-1)$ | |
| All-to-all personalized | | | $(t_s+t_w m)(p-1) + (t_h/2)\, p \lg p$ |
| Circular q-shift | | | $t_s+t_w m + t_h(\lg p - \gamma(q))$ |

$\gamma(q)$ is the number of times 2 divides q. $\lg p - \gamma(q)$ is longest path for a circular q-shift

# THE END