Parallel Algorithms

Slides adapted from R. Libeskind-Hadas, I. Potapov

Parallel Algorithms:

- This week we will
  - introduce techniques for the design of efficient parallel algorithms and
  - discuss for implementing parallel algorithms.

What is Parallel Computing? (basic idea)

- Consider the problem of stacking (reshelving) a set of library books.
  - A single worker trying to stack all the books in their proper places cannot accomplish the task faster than a certain rate.
  - We can speed up this process, however, by employing more than one worker.

Solution 1

- **Assume that books are organized into shelves** and that the shelves are grouped into bays
- **One simple way** to assign the task to the workers is:
  - **To divide the books equally among them.**
  - **Each worker stacks the books one a time**
- **This division of work may not be most efficient** way to accomplish the task since
  - **The workers must walk all over the library to stack books.**

Solution 2

- **An alternative way** to divide the work is to assign a fixed and disjoint set of bays to each worker.
- As before, **each worker is assigned an equal number of books arbitrarily**.
  - If the worker finds a book that belongs to a bay assigned to him or her,
    - he or she places that book in its assignment spot
  - Otherwise,
    - He or she passes it on to the worker responsible for the bay it belongs to.
- **The second approach requires less effort from individual workers**

Problems are parallelizable to different degrees

- **For some problems**, assigning partitions to other processors might be more time-consuming than performing the processing locally.
- **Other problems may be completely serial**.
  - For example, consider the **task of digging a post hole**.
    - Although one person can dig a hole in a certain amount of time,
    - Employing more people does not reduce this time

Parallel Processing

(Several processing elements working to solve a single problem)

Primary consideration: **elapsed time**
- **NOT:** throughput, sharing resources, etc.

- Downside: **complexity**
  - system, algorithm design

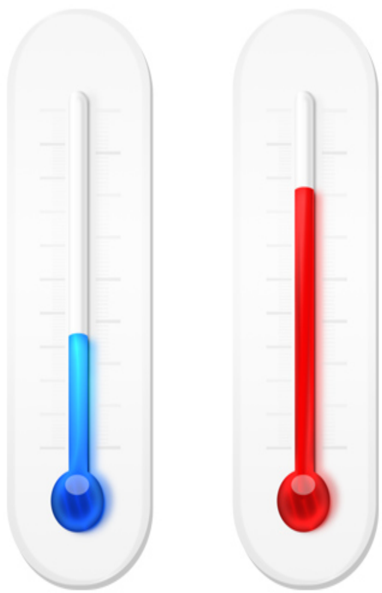
- Elapsed Time = **computation time + communication time + synchronization time**

Design of efficient algorithms

A parallel computer is of little use unless efficient parallel algorithms are available.

- The issue in designing parallel algorithms are very different from those in designing their sequential counterparts.

- A significant amount of work is being done to develop efficient parallel algorithms for a variety of parallel architectures.



Bonus time!

- Where is the average Algs student from?
- Your goal: find the class' average original US zip code.
- Rules of the game:
  - Number of students is known
  - Every person in the room can perform a single arithmetic operation per time step (+,-,x,/)
  - Use of white board is free (free reads and writes)
- Bonus points:
  - You will receive bonus participation points equivalent to the speedup over the naïve approach.

Processor Trends

- **Moore's Law**
  - performance doubles every 18 months
- **Parallelization within processors**
  - pipelining
  - multiple pipelines

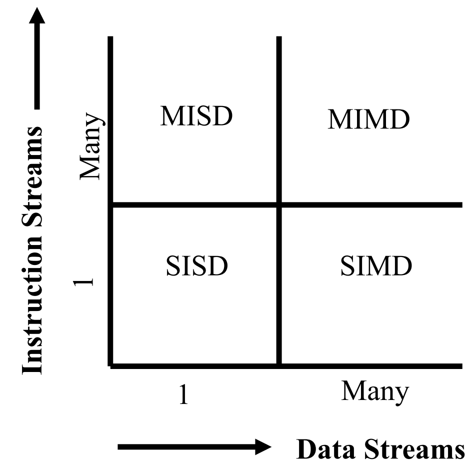
Why Parallel Computing

- **Practical:**
  - Moore's Law cannot hold forever
  - Problems must be solved fast
  - Cost-effectiveness
  - Scalability
- **Theoretical:**
  - challenging problems

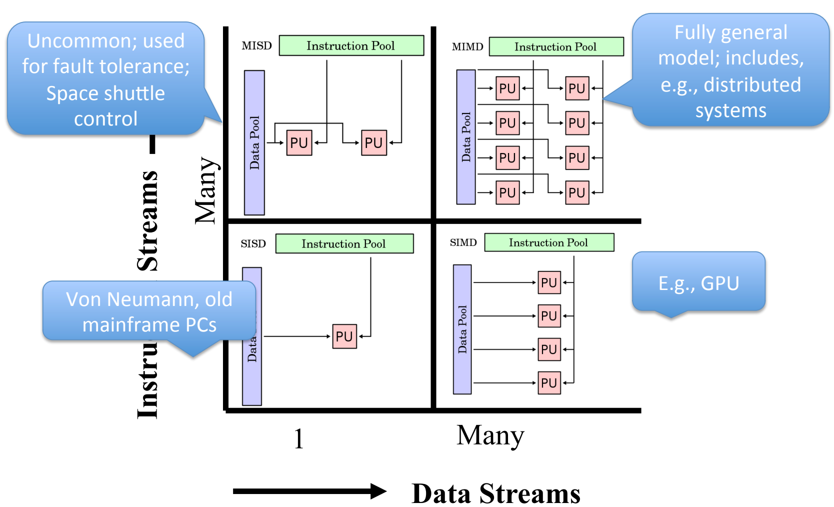
Fundamental Issues

- Is the problem amenable to parallelization?
- How to decompose the problem to exploit parallelism?
- What machine architecture should be used?
- What parallel resources are available?
- What kind of speedup is desired?

Flynn's Taxonomy



Flynn's Taxonomy



Parallel Architectures

- **Multiple processing elements**
- **Memory:**
  - shared
  - distributed
  - hybrid
- **Control:**
  - centralized
  - distributed

Parallel vs Distributed Computing

- **Parallel:**
  - several processing elements concurrently solving a single same problem
- **Distributed:**
  - processing elements do not share memory or system clock

Metrics

A measure of relative performance between a multiprocessor system and a single processor system is the *speed-up* $S(p)$, defined as follows:

$$S(p) = \frac{\text{Execution time using a single processor system}}{\text{Execution time using a multiprocessor with } p \text{ processors}}$$

$$S(p) = \frac{T_1}{T_p} \qquad Efficiency = \frac{S_p}{p}$$

$$Work = p \times T_p$$

Efficient and optimal parallel algorithms

- **A parallel algorithm is efficient** iff
  - it is fast (e.g. polynomial time) and
  - the product of the parallel time and number of processors is close to the time of at the best know sequential algorithm

$$T_{sequential} \approx T_{parallel} \cdot N_{processors}$$

- **A parallel algorithms** is optimal iff this product is of the same order as the best known sequential time

Metrics

- **Parallel algorithm is cost-optimal:**
  Work = sequential time
  $W_p = T_1$
  $E_p = 100\%$

- **Critical when down-scaling:**
  parallel implementation may become slower than sequential
  $T_1 = n^3$
  $T_p = n^{2.5}$ when $p = n^2$
  $W_p = n^{4.5}$

Amdahl's Law

- ***f* = fraction of the problem that's inherently sequential**

  (1 – *f*) = fraction that's parallel

- **Parallel time T_p:**

$$T_p = f + (1 - f)/p$$

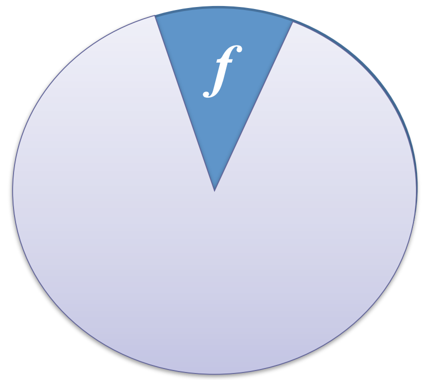
- **Speedup with *p* processors:**

$$S_p = \frac{1}{f + \frac{1-f}{p}}$$

What kind of speed-up may be achieved?

- Part *f* is computed by a single processor
- Part (1-*f*) is computed by *p* processors, *p>1*

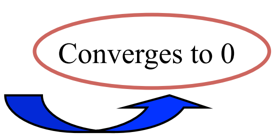
Basic observation: Increasing p we cannot speed-up part f.



Amdahl's Law

- **Upper bound on speedup (*p* = ∞)**

$$S_p = \frac{1}{f + \frac{1-f}{p}} \qquad S_\infty = \frac{1}{f}$$

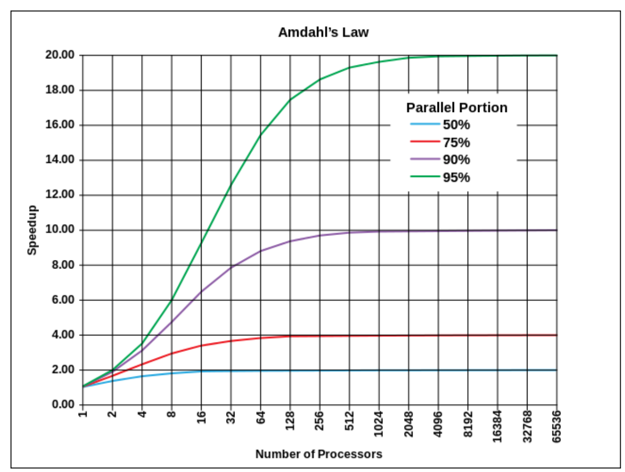


- **Example:**

  ***f* = 2%**

  ***S* = 1 / 0.02 = 50**

Amdahl's Law

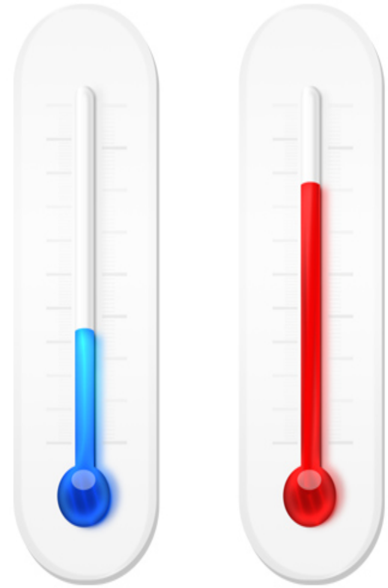


The main open question

- The basic parallel complexity class is **NC**.
- **NC** is a class of problems computable in poly-logarithmic time ($\log^c n$, for a constant c) using a polynomial number of processors.
- **P** is a class of problems computable sequentially in a polynomial time

The main open question in parallel computations is

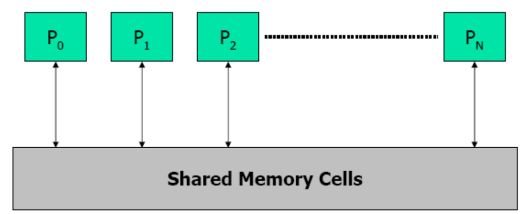
NC = P ?



Parallel or Distributed?

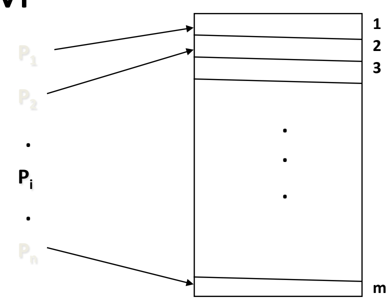
- ATM Machines
- Map Reduce
- Distributed Database
- Two servers sharing the workload of routing mail
- Internet
- GPU-based algorithms
- Supercomputer
- Cellular Network

PRAM model



PRAM

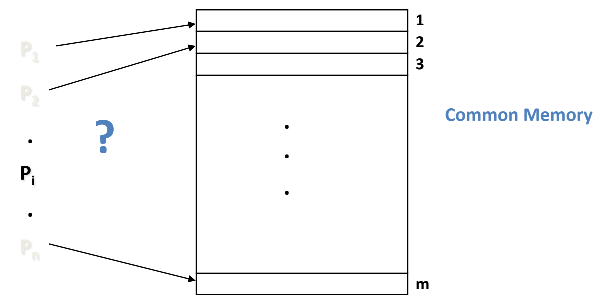
- PRAM - Parallel Random Access Machine
- Shared-memory multiprocessor
- unlimited number of processors, each
  - has unlimited local memory
  - knows its ID
  - able to access the shared memory in constant time
  - unlimited shared memory



A very reasonable question: Why do we need a PRAM model?

- to make it easy to reason about algorithms
- to achieve complexity bounds
- to analyze the maximum parallelism

PRAM MODEL



PRAM **n RAM processors connected to a common memory of m cells**

ASSUMPTION: at each time unit each P_i can read a memory cell, make an internal computation and write another memory cell.

CONSEQUENCE: any pair of processor P_i P_j can communicate in constant time!

P_i writes the message in cell x at time t
P_i reads the message in cell x at time t+1

Summary of assumptions for PRAM

PRAM

- Inputs/Outputs are placed in the shared memory (designated address)
- Memory cell stores an arbitrarily large integer
- Each instruction takes unit time
- Instructions are synchronized across the processors

PRAM Instruction Set

- accumulator architecture
  - memory cell R_0 accumulates results
- multiply/divide instructions take only constant operands
  - prevents generating exponentially large numbers in polynomial time

PRAM Complexity Measures

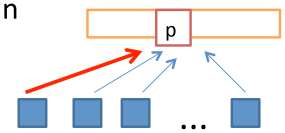
- for each individual processor
  - ***time***: number of instructions executed
  - ***space***: number of memory cells accessed
- PRAM machine
  - ***time***: time taken by the longest running processor
  - ***hardware***: maximum number of active processors

Two Technical Issues for PRAM

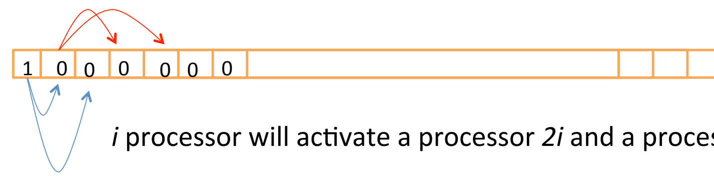
- How processors are activated
- How shared memory is accessed

Processor Activation

- P_0 places the number of processors (p) in the designated shared-memory cell
  - each active P_i, where $i < p$, starts executing
  - $O(1)$ time to activate
  - all processors halt when P_0 halts



- Active processors explicitly activate additional processors via FORK instructions
  - tree-like activation
  - $O(\log p)$ time to activate



i processor will activate a processor *2i* and a processor *2i+1*

PRAM

- Too many interconnections gives problems with synchronization
- **However it is the best conceptual model** for designing efficient parallel algorithms
  - due to simplicity and possibility of simulating efficiently PRAM algorithms on more realistic parallel architectures

Basic parallel statement

for all x in X do in parallel
instruction (x)

→

For each x PRAM will assign a processor which will execute *instruction(x)*

Shared-Memory Access

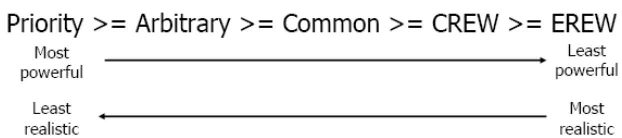
Concurrent (C) means, many processors can do the operation simultaneously in the same memory

Exclusive (E) not concurent

- EREW (Exclusive Read Exclusive Write)
- CREW (Concurrent Read Exclusive Write)
  - Many processors can read simultaneously the same location, but only one can attempt to write to a given location
- ERCW (Exclusive Read Concurrent Write)
- CRCW (Concurrent Read Concurrent Write)
  - **Many processors can write**/read **at**/from **the same memory location**

Concurrent Write (CW)

- What value gets written finally?
- Priority CW – processors have priority based on which write value is decided
- Common CW – multiple processors can simultaneously write only if values are the same
- Arbitrary/Random CW – any one of the values are randomly chosen



Example CRCW-PRAM

- Initially
  - table **A** contains values 0 and 1
  - **output** contains value 0

$$\text{for each } 1 \le i \le 5 \text{ do in parallel}$$
$$\text{if } A[i] = 1 \text{ then output=1;}$$

- The program computes the "Boolean OR" of A[1], A[2], A[3], A[4], A[5]

Example CREW-PRAM

- Assume initially table **A** contains [0,0,0,0,0,1] and we have the parallel program

$$\text{for each } 1 \le i \le 5 \text{ do in parallel}$$
$$A[i]; = A[i] + A[i+1]$$

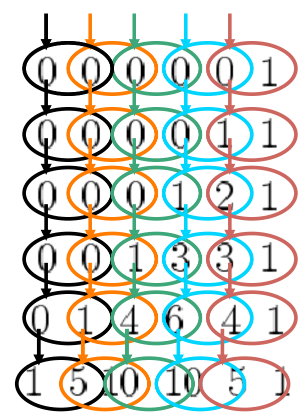
Worksheet: What is the output of this program for t = 1,2,…,6?

Pascal triangle

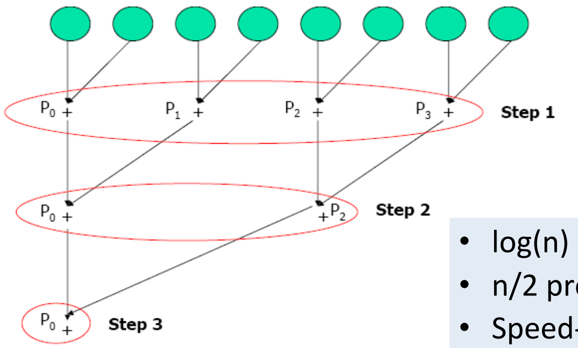
$$\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \ldots \binom{n}{n}$$
$$\text{for } n = 0, 1, 2, 3, 4, 5, 6.$$

PRAM CREW

$$\text{for each } 1 \le i \le 5 \text{ do in parallel}$$
$$A[i]; = A[i] + A[i+1]$$



Parallel Addition



- log(n) steps=time needed
- n/2 processors needed
- Speed-up = n/log(n)
- Efficiency = 1/log(n)
- Applicable for other operations too
  +, *, <, >, == etc.

Membership problem

- p processors PRAM with n numbers (p ≤ *n)*
- Does x exist within the n numbers?
- P0 contains x and finally P0 has to know

Algorithm

step1: Inform everyone what x is

step2: Every processor checks [n/p] numbers and sets a flag

step3: Check if any of the flags are set to 1

WORKSHEET	EREW	CREW	CRCW
Step 1:			
Step 2:			
Step 3:			

Membership problem

- p processors PRAM with n numbers (p ≤ *n)*
- Does x exist within the n numbers?
- P0 contains x and finally P0 has to know

Algorithm

step1: Inform everyone what x is

step2: Every processor checks [n/p] numbers and sets a flag

step3: Check if any of the flags are set to 1

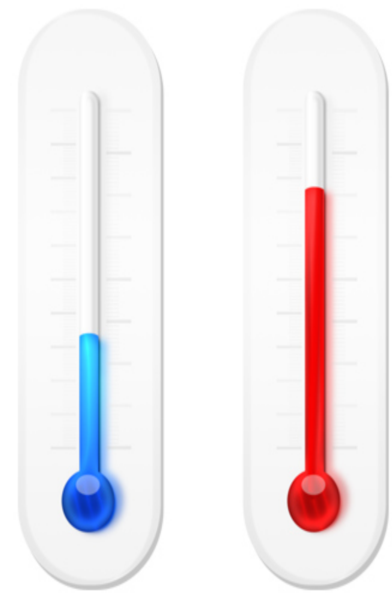
EREW	CREW	CRCW (common)
log(p)	1	1
n/p	n/p	n/p
log(p)	log(p)	1

THE PRAM IS A THEORETICAL (UNFEASIBLE) MODEL

- **The interconnection network between processors and memory would require a very large amount of area .**
- **The message-routing on the interconnection network would require time proportional to network size (i.e. the assumption of a constant access time to the memory is not realistic).**

Why is PRAM useful?

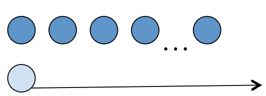
- Algorithm's designers can forget the communication problems and focus their attention on the parallel computation only.
- There exist algorithms simulating any PRAM algorithm on bounded degree networks (e.g., each step can be simulated with a slow-down of log2n/log logn on tree-mesh structure).
- Any problem that can be solved for a p processor PRAM in t steps can be solved in a p' processor PRAM in t' =O(tp/p') steps
- Instead of design ad hoc algorithms for bounded degree networks, design more general algorithms for the PRAM model and simulate them on a feasible network.



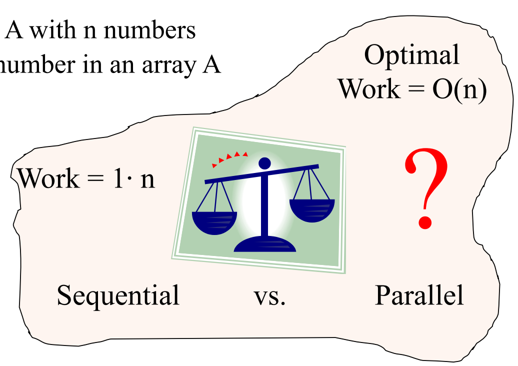
Min of n numbers

- Input: Given an array A with n numbers
- Output: the minimal number in an array A

Sequential algorithm



At least n comparisons should be performed!!!



Work = (num. of processors) · (time)

Worksheet:

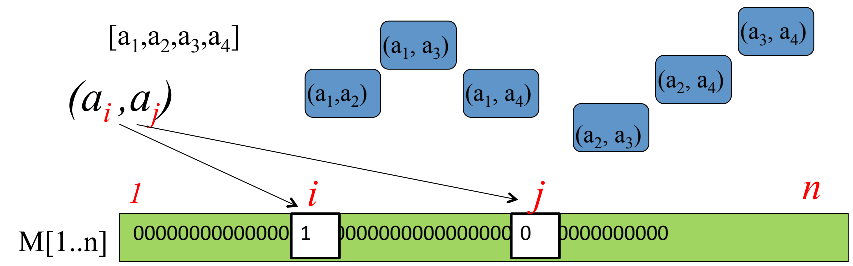
- Write a parallel algorithm for finding the min of *n* numbers in *constant* time.
- How many processors do you need?

Mission: Impossible …
computing in a constant time

- Archimedes: *Give me a lever long enough and a place to stand and I will move the earth*
- NOWDAYS….
  Give me a parallel machine with enough processors and I will find the smallest number in any giant set in a constant time!

Parallel solution 1
Min of n numbers

- Comparisons between numbers can be done independently
- The second part is to find the result using concurrent write mode
- For n numbers ----> we have ~ n^2 pairs



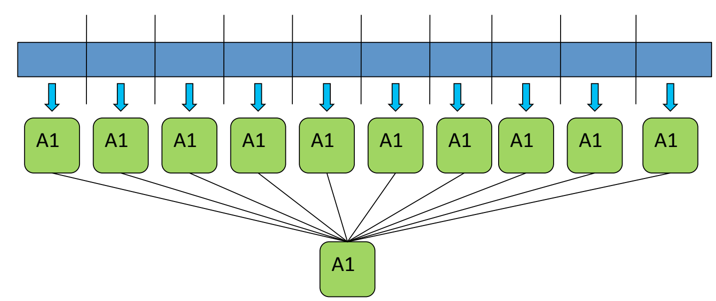
If $a_i > a_j$ *then* a_i *cannot be the minimal number*

The following program computes MIN of n numbers stored in the array C[1..n] in O(1) time with n^2 processors.

Algorithm A1

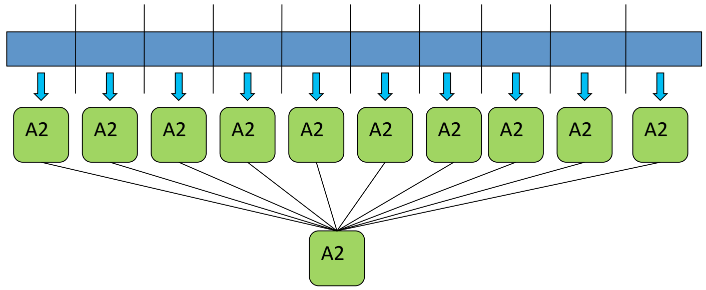
for each $1 \le i \le n$ *do in parallel*
    $M[i]:=0$
for each $1 \le i,j \le n$ *do in parallel*
    if $i \ne j$ $C[i] \le C[j]$ *then* $M[j]:=1$
for each $1 \le i \le n$ *do in parallel*
    if $M[i]=0$ *then output:=i*

From n^2 processors to $n^{1+1/2}$



Step 1: Partition into disjoint blocks of size $\sqrt{n}$
Step 2: Apply A1 to each block $n\sqrt{n}$
Step 3: Apply A1 to the results from the step 2 $\sqrt{n}$

From $n^{1+1/2}$ processors to $n^{1+1/4}$



Step 1: Partition into disjoint blocks of size $\sqrt{n}$

Step 2: Apply A2 to each block

Step 3: Apply A2 to the results from the step 2

$n^2 \to n^{1+1/2} \to n^{1+1/4} \to n^{1+1/8} \to n^{1+1/16} \to \ldots \to n^{1+1/k} \sim n^1$

- Assume that we have an algorithm A_k working in O(1) time with $n^{1+\varepsilon_k}$ processors

Algorithm A_{k+1}

1. Let $\alpha=1/2$
2. Partition the input array C of size n into disjoint blocks of size n^α each
3. Apply in parallel algorithm A_k to each of these blocks
4. Apply algorithm A_k to the array C' consisting of n/n^α minima in the blocks.

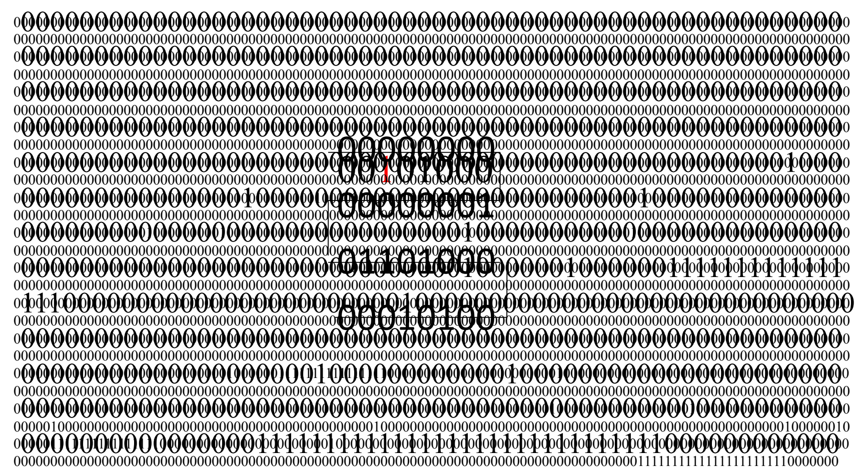
Complexity

- We can compute minimum of n numbers using CRCW PRAM model in O(log log n) with n processors by applying a strategy of partitioning the input

$$\text{Work} = n \cdot \log\log n$$

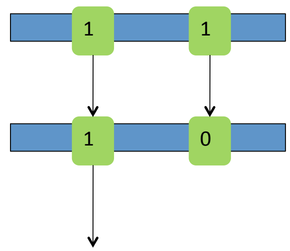
Mission: Impossible (Part 2)

Computing a position of the first one in the sequence of 0's and 1's in a constant time.



Problem 2.
Computing a position of the first one in the sequence of 0' s and 1' s.

Algorithm A
(2 parallel steps and n^2 processors)
for each $1 \le i < j \le n$ *do in parallel*
if $C[i] = 1$ *and* $C[j] = 1$ *then* $C[j] := 0$
for each $1 \le i \le n$ *do in parallel*
if $C[i] = 1$ *then FIRST-ONE-POSITION:=i*



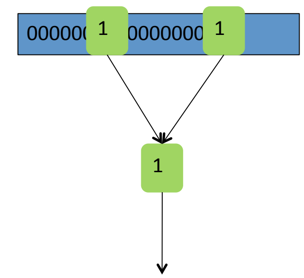
FIRST-ONE-POSITION(C)=4
for the input array
C=[0,0,0,1,0,0,0,1,1,1,0,0,0,1]

After the first parallel step C will contain a single element 1

Reducing number of processors

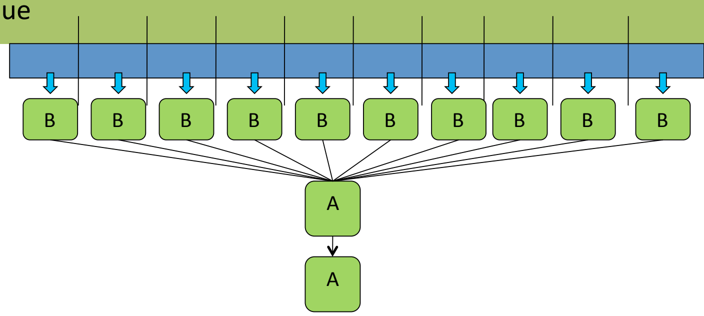
Algorithm B –
it reports if there is any one in the table.

There-is-one:=0
for each $1 \le i \le n$ *do in parallel*
if $C[i] = 1$ *then There-is-one:=1*



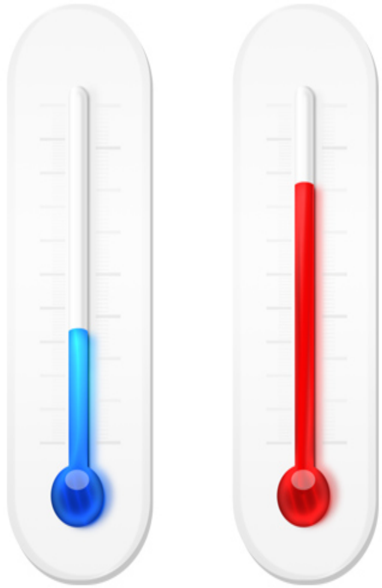
Now we can merge two algorithms A and B

1. Partition table C into segments of size $\sqrt{n}$
2. In each segment apply the algorithm B
3. Find position of the first one in these sequence by applying algorithm A
4. Apply algorithm A to this single segment and compute the final value



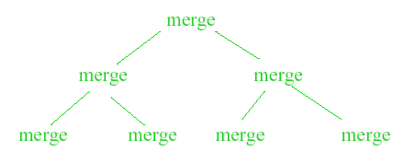
Complexity

- We apply an algorithm A twice and each time to the array of length $\sqrt{n}$ which need only $(\sqrt{n})^2 = n$ processors
- The time is O(1) and number of processors is n.



Optimal sorting in log(n) steps
Cole's algorithm

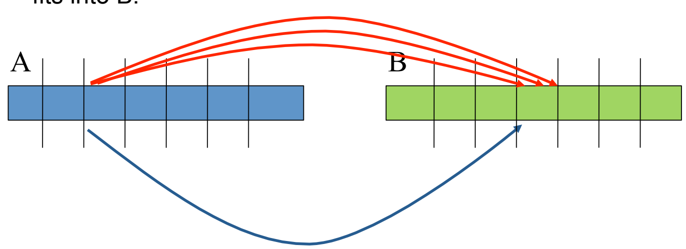
- Suppose we know how to merge two increasing sequences in *log(log(n))* steps
- Then we can climb up the merging tree and spend only *log(log(n))* per level, thus getting a parallel sorting technique in *log(n) log(log(n))*

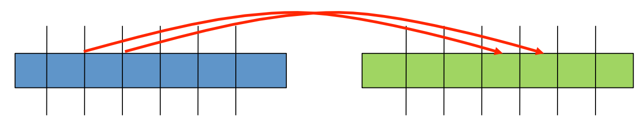


- Merges at the same level are performing in parallel

How to merge in log(log(n)) time with n processors

- Let A and B are to sorted sequences of size n
- Divide A,B into $\sqrt{n}$ blocks of length $\sqrt{n}$
- Compare first elements of each block in A with first elements of each block in B
- Then compare first elements of each block in A with each element in a "suitable" block of B
- At this point we know where all first elements of each block in A fits into B.





- Thus the problem has been reduced to a set of disjoint problems each of which involves merging of block of $\sqrt{n}$ elements of A with some consecutive piece of B.
- Recursively we solve these problems
- The parallel time t(n) satisfies to

$t(n) \leq 2 + t(\sqrt{n})$ implying $t(n)=O(\log(\log(n)))$