ENEE699/ENEE459P: Parallel Algorithms

Home page:
http://www.umiacs.umd.edu/users/vishkin/TEACHING/enee459p-f10.html

Uzi Vishkin

Electrical and Computer Engineering Dept

A. JAMES CLARK SCHOOL of ENGINEERING

UNIVERSITY OF MARYLAND 1856

4 Reasons for the approach I will present

<u>Reason 1</u> Thinking in parallel is based on first principles.

The theme:

How to Think Algorithmically in Parallel?

Merits special attention, separate from programming

Reason 2: You can handle algorithms not through programming

Recall how the standard CS curriculum looks:

- 1^{st} year of CS: programming.
- Later: design & analysis of algorithms.
- Rationale: 1st concrete programming experience, so that algorithms & analysis make sense

But, **you** can understand parallel algorithms. Then:

- learn skill of parallel programming by assignments
- 20 minute review of XMT-C. Rest from tutorial+manual

Test: are assignments on par with other approaches?

The Pain of Parallel Programming

- Parallel programming is currently too difficult:
- 40 years of parallel computing→ The world is yet to see a successful general-purpose parallel computer: Easy to program & good speedups
- To many users programming existing parallel computers is "as intimidating and time consuming as programming in assembly language" [NSF Blue-Ribbon Panel on Cyberinfrastructure]
- AMD/Intel: "Need PhD in CS to program today's multicores"
- "The Trouble with Multicore: Chipmakers are busy designing microprocessors that *most programmers can't handle*"—D. Patterson, IEEE Spectrum, July 2010.

➔ The software spiral (HW improvements ➔ SW imp ➔ HW imp) – growth engine for IT (A. Grove, Intel); Alas, now broken!

➔ Impasse SW vendors avoid investment in long-term SW development since may bet on the wrong horse.

Parallel programming education: why not same impasse?

Reason 3: Can teach common denominator

Example and Question

Today's reality 12MB on-chip caches already available. Likely to increase further ➔ to what extent should programming-for-locality be enforced by many-core vendors?

Example Matrix multiplication

- 12MB on-chip cache can fit 1000X1000 matrices
- In principle*, standard matrix-multiplication can work using high bandwidth on-chip interconnection network coupled with our (static and dynamic) prototyped prefetching mechanisms

Matrix multiplication using tiling still important for large matrices

The Open Question To what extent will we need in the future to drag mainstream programming to the low-productivity pains** experienced by parallel computing?

Not questioned some must know these techniques; may be even all of you eventually. But, should they be part of the intro?

* Reason 4 Done that: XMT. ** Previous slide

Not only us

- Parallel algorithms researchers realized decades ago that the main reason that parallel machines are difficult to program is that the bandwidth between processors/memories is so limited.
- [BMM94] suggested that: 1. Machine manufacturers see the cost benefit of lowering performance of interconnects (that were much higher in 1994), but grossly underestimate the programming difficulties and the high software development costs implied. 2. Their exclusive focus on runtime benchmarks misses critical costs, including: (i) the time to write the code, and (ii) the time to port the code to different distribution of data or to different machines that require different distribution of data.

G. Blelloch, B. Maggs, and G. Miller. The hidden cost of low bandwidth communication. In Developing a Computer Science Agenda for High-Performance Computing (Editor: U. Vishkin), pp 22–25. ACM Press, 1994.

The eXplicit MultiThreading (XMT) Easy-To-Program Parallel Computer

www.umiacs.umd.edu/users/vishkin/XMT

UMIACS
University of Maryland Institute for Advanced Computer Studies

A. JAMES CLARK SCHOOL of ENGINEERING

UNIVERSITY OF MARYLAND 1856

XMT

Algorithms

PRAM parallel algorithmic theory. “Natural selection”. **Latent**, though not widespread, **knowledgebase**

Won  the battle of ideas on parallel algorithmic thinking. No silver or bronze!

Model of choice in all theory communities. 1988-90: Big chapters in standard algorithms textbooks.

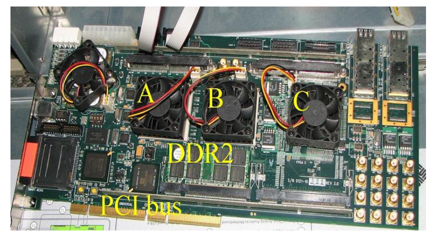
Programming & workflow

Compiler

PRAM-On-Chip HW Prototypes

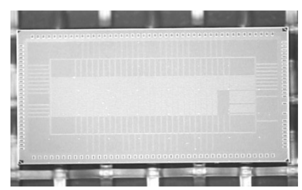
<u>64-core, 75MHz FPGA</u> of XMT

(Explicit Multi-Threaded) architecture



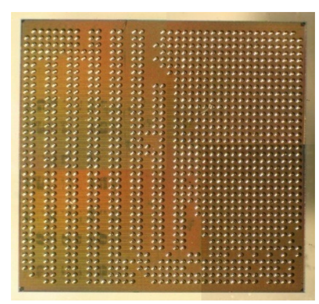
SPAA98..CF08

<u>128-core intercon. network</u>



IBM 90nm: 9mmX5mm, [HotI07];.

Asynch, [NOCS10]



<u>FPGA design→ASIC</u>

IBM 90nm: 10mmX10mm

Architecture scales to 1000+ cores on-chip

Experience with High School Students, Fall’07

1-day parallel algorithms tutorial to 12 HS students. Some (2 10th graders) managed 8 programming assignments, including 5 of the 6 in the grad course. Only help: 1 office hour/week by undergrad TA. No school credit. Part of a computer club after 8 periods/day.

One of these 10th graders: “I tried to work on parallel machines at school, but it was no fun: I had to program around their engineering. With XMT, I could focus on solving the problem that I had to solve.”

Dec’08-Jan’09: 50 HS students, by self-taught HS teacher, TJ HS, Alexandria, VA. Part of the regular curriculum.

By summer’09: 100+ K-12 students experienced XMT. ‘09 MCPS Middle school camp for children from underepresented groups. Taught at Baltimore Poly HS: 70% African Americans.

Spring’09: Course to Freshmen, UMD (strong enrollment). How will programmers have to think by the time you graduate.

➔ **SIGCSE’10 paper and CS4HS’09@CMU Keynote**

NEW: Software release

Allows to use **your own computer** for programming on an XMT environment and experimenting with it, including:

(i) Cycle-accurate simulator of the XMT machine

(ii) Compiler from XMTC to that machine

Also provided, extensive material for teaching or self-studying parallelism, including

(i) Tutorial + manual for XMTC (150 pages)

(ii) Classnotes on parallel algorithms (100 pages)

(iii) Video recording of 9/15/07 HS tutorial (300 minutes)

(iv) Video recording of Spring'09 grad course (30+ hours)

Participants

Grad students:, Aydin Balkan, PhD, George Caragea, James Edwards, David Ellison, Mike Horak, MS, Fuat Keceli, Beliz Saybasili, Alex Tzannes, Xingzhi Wen, PhD

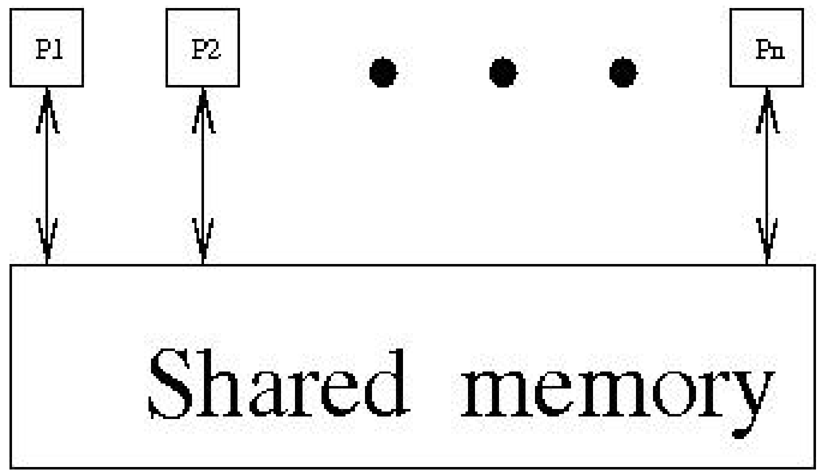
- Industry design experts (pro-bono).
- Rajeev Barua, Compiler. Co-advisor of 2 CS grad students. 2008 NSF grant.
- Gang Qu, VLSI and Power. Co-advisor.
- Steve Nowick, Columbia U., Asynch computing. Co-advisor. 2008 NSF team grant.
- Ron Tzur, Purdue U., K12 Education. Co-advisor. 2008 NSF seed funding

K12: Montgomery Blair Magnet HS, MD, Thomas Jefferson HS, VA, Baltimore (inner city) Ingenuity Project Middle School 2009 Summer Camp, Montgomery County Public Schools

- Marc Olano, UMBC, Computer graphics. Co-advisor.
- Tali Moreshet, Swarthmore College, Power. Co-advisor.
- Marty Peckerar, Microelectronics
- Igor Smolyaninov, Electro-optics
- Funding: NSF, NSA 2008 deployed XMT computer, NIH
- Reinvention of Computing for Parallelism. Selected for Maryland Research Center of Excellence (MRCE) by USM, 12/08. Not yet funded. 17 members, including UMBC, UMBI, UMSOM. Mostly applications.

Parallel Random-Access Machine/Model

PRAM:



n <u>synchronous</u> processors all having unit time access to a shared memory.
Each processor has also a local memory.
At each time unit, a processor can:

1. write into the shared memory (i.e., copy one of its local memory registers into a shared memory cell),
2. read into shared memory (i.e., copy a shared memory cell into one of its local memory registers), or
3. do some computation with respect to its local memory.

pardo programming construct

- for Pi , 1 ≤ i ≤ n pardo
- A(i) := B(i)

<u>This means</u>

The following n operations are performed concurrently: processor P1 assigns B(1) into A(1), processor P2 assigns B(2) into A(2), ….

<u>Modeling read&write conflicts to the same shared memory location</u>

Most common are:

- exclusive-read exclusive-write (EREW) PRAM: no simultaneous access by more than one processor to the same memory location for read or write purposes
- concurrent-read exclusive-write (CREW) PRAM: concurrent access for reads but not for writes
- concurrent-read concurrent-write (CRCW allows concurrent access for both reads and writes. We shall assume that in a concurrent-write model, an arbitrary processor among the processors attempting to write into a common memory location, succeeds. This is called the Arbitrary CRCW rule.

There are two alternative CRCW rules: (i) Priority CRCW: the smallest numbered, among the processors attempting to write into a common memory location, actually succeeds. (ii) Common CRCW: allows concurrent writes only when all the processors attempting to write into a common memory location are trying to write the same value.

Example of a PRAM algorithm: The summation problem

Input An array A = A(1) . . .A(n) of n numbers.

The problem is to compute A(1) + . . . + A(n).

The summation algorithm works in rounds.

Each round: add, in parallel, pairs of elements: add each odd-numbered element and its successive even-numbered element.

If n = 8, outcome of 1st round is:

A(1) + A(2), A(3) + A(4), A(5) + A(6), A(7) + A(8)

Outcome of 2nd round:

A(1) + A(2) + A(3) + A(4), A(5) + A(6) + A(7) + A(8)

and the outcome of 3rd (and last) round:

A(1) + A(2) + A(3) + A(4) + A(5) + A(6) + A(7) + A(8)

B – 2-dimensional array (whose entries are B(h,i), $0 \le h \le \log n$ and $1 \le i \le n/2^h$) used to store all intermediate steps of the computation (base of logarithm: 2).

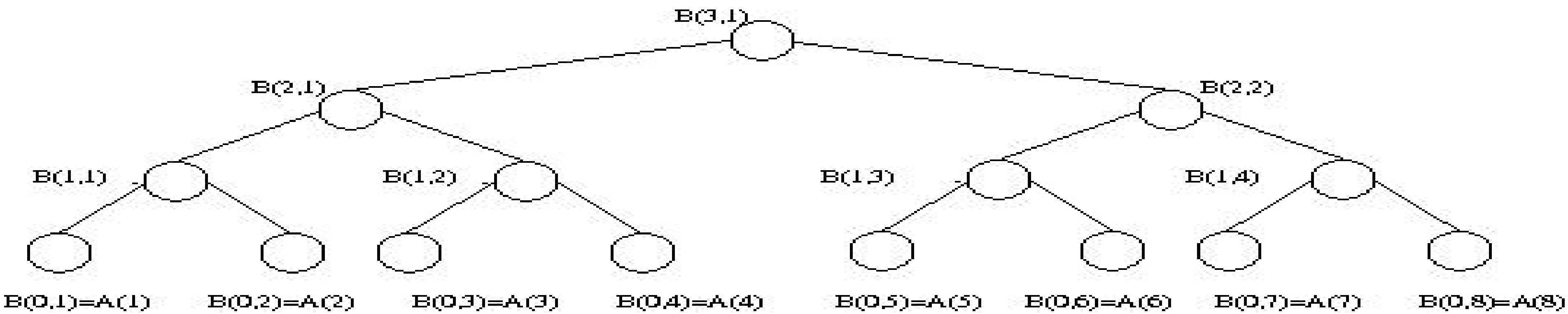
For simplicity, assume $n = 2^k$ for some integer k.

ALGORITHM 1 (Summation)

```
1. for Pi , 1 ≤ i ≤ n pardo
2.   B(0, i) := A(i)
3.   for h := 1 to log n do
4.           if i ≤ n/2^h
5.           then B(h, i) := B(h – 1, 2i – 1) + B(h – 1, 2i)
6.           else stay idle
7.   for i = 1: output B(log n, 1); for i > 1: stay idle
```

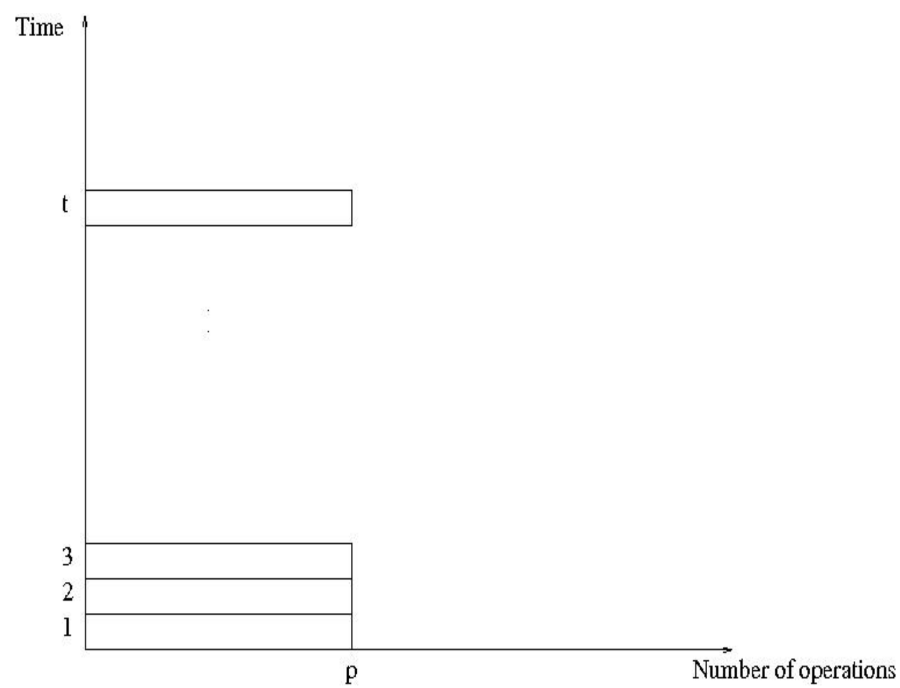
Algorithm 1 uses p = n processors.
Line 2 takes one round,
Line 3 defines a loop taking
log n rounds
Line 7 takes one round.

Summation on an n = 8 processor PRAM



Again Algorithm 1 uses p = n processors. Line 2 takes one round, line 3 defines a loop taking log n rounds, and line 7 takes one round. Since each round takes constant time, Algorithm 1 runs in O(log n) time. [When you see O ("big Oh"), think "proportional to".]

So, an algorithm in the PRAM model is presented in terms of a sequence of parallel time units (or "rounds", or "pulses"); we allow p instructions to be performed at each time unit, one per processor; this means that a time unit consists of a sequence of exactly p instructions to be performed concurrently.

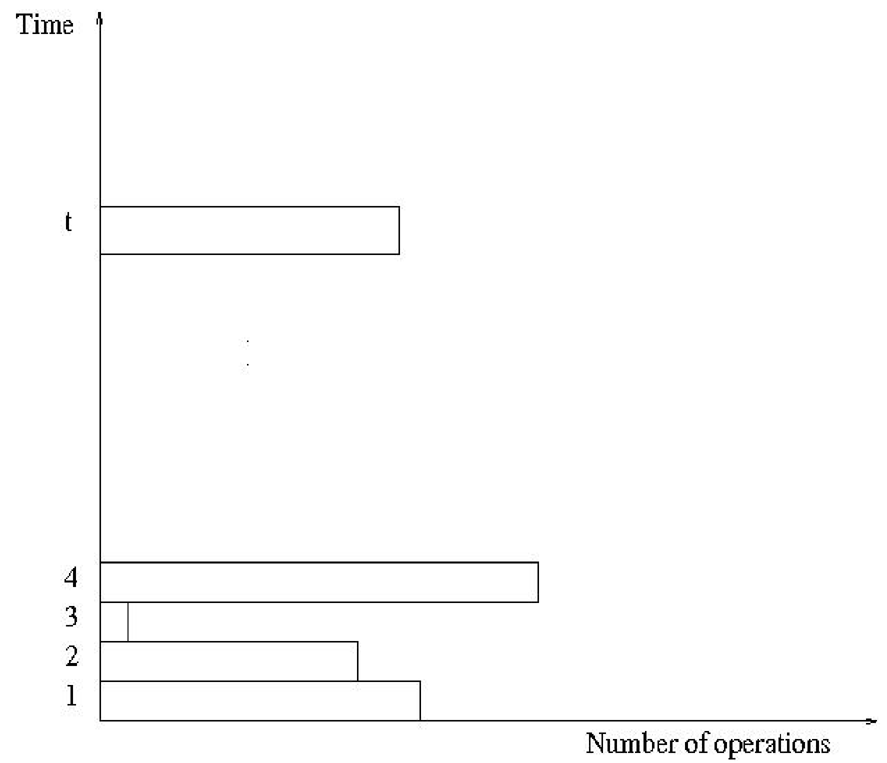


2 drawbacks to PRAM mode: (i) Does not reveal how the algorithm will run on PRAMs with different number of processors; e.g., to what extent will more processors speed the computation, or fewer processors slow it? (ii) Fully specifying the allocation of instructions to processors requires a level of detail which might be unnecessary (a compiler may be able to extract from lesser detail)

Work-Depth presentation of algorithms

Alternative model and presentation mode.

Work-Depth algorithms are also presented as a sequence of parallel time units (or "rounds", or "pulses"); however, each time unit consists of a sequence of instructions to be performed concurrently; the sequence of instructions may include any number.



WD presentation of the summation example

"Greedy-parallelism": At each point in time, the (WD) summation algorithm seeks to break the problem into as many pairwise additions as possible, or, in other words, into the largest possible number of independent tasks that can performed concurrently.

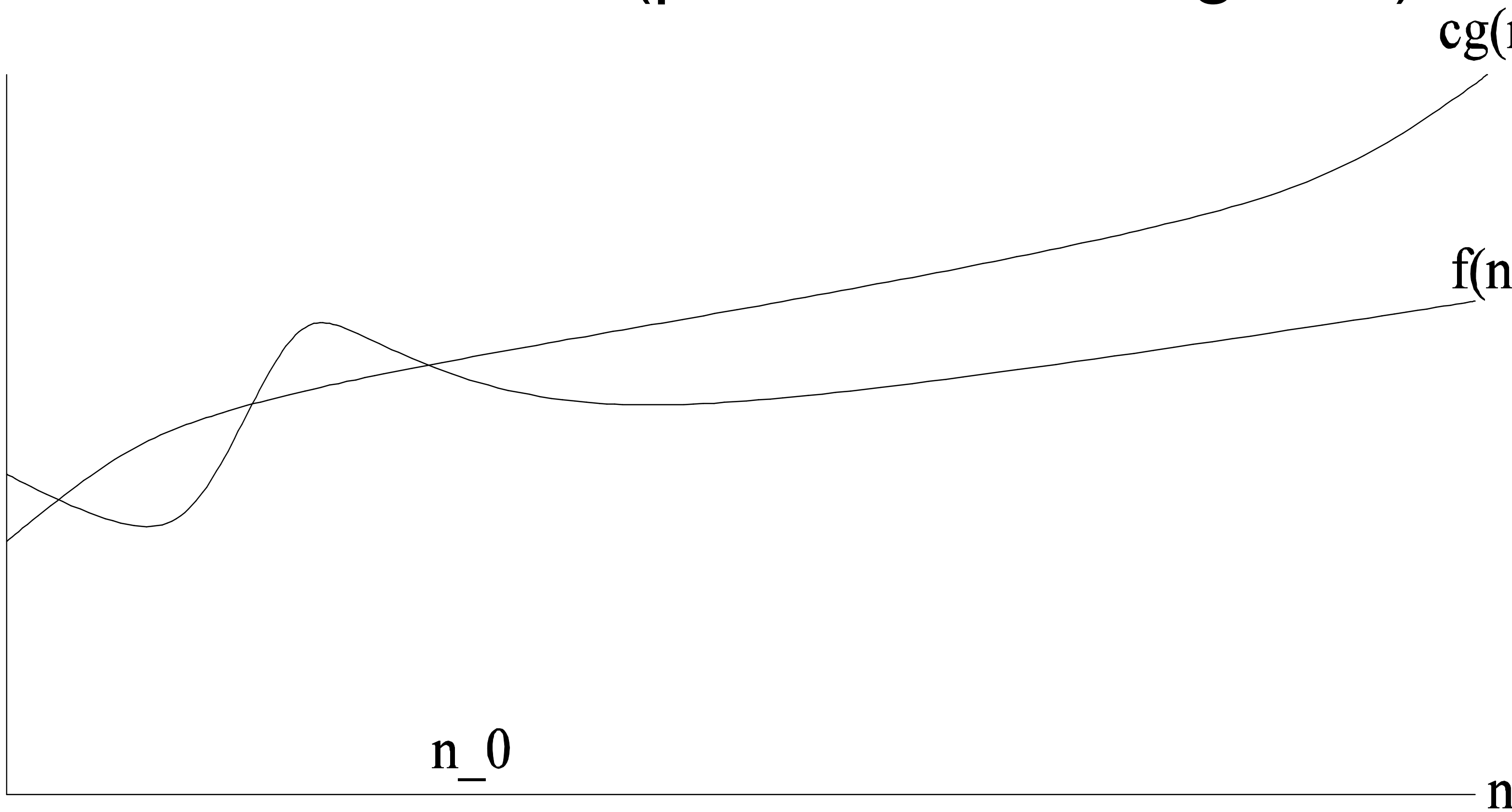
<u>ALGORITHM 2 (WD-Summation)</u>

1. for i , $1 \le i \le n$ pardo
2. B(0, i) := A(i)
3. for h := 1 to log n
4. for i , $1 \le i \le n/2^h$ pardo
5. B(h, i) := B(h – 1, 2i – 1) + B(h – 1, 2i)
6. for i = 1 pardo output B(log n, 1)

The 1st round of the algorithm (lines 1&2) has n operations. The 2nd round (lines 4&5 for h = 1) has n/2 operations. The 3rd round (lines 4&5 for h = 2) has n/4 operations. In general, the k-th round of the algorithm, $1 \le k \le \log n + 1$, has $n/2^{k-1}$ operations and round log n +2 (line 6) has one more operation (use of a pardo instruction in line 6 is somewhat artificial). The total number of operations is 2n and the time is log n + 2. We will use this information in the corollary below.

The next theorem demonstrates that the WD presentation mode does not suffer from the same drawbacks as the standard PRAM mode, and that every algorithm in the WD mode can be automatically translated into a PRAM algorithm.

O-notation (pronounced 'big-Oh')



f(n) = O(g(n))

O-notation provides an upper bound on a function within a constant factor.
f(n) = O(g(n)) if there are two positive constants n_0 and c such that for every value n >= n_0 the function f(n) lies on or below cg(n)

O-notation is the most frequently used notation in algorithm courses

The WD-presentation sufficiency Theorem

Consider an algorithm in the WD mode that takes a total of $x = x(n)$ elementary operations and $d = d(n)$ time. The algorithm can be implemented by any $p = p(n)$-processor PRAM within $O(x/p + d)$ time, using the same concurrent-write convention as in the WD presentation.

[i.e., 5 theorems: EREW, CREW, Common/Arbitrary/Priority CRCW]

<u>Proof</u>

x_i - # instructions at round i. [$x_1+x_2+..+x_d = x$]

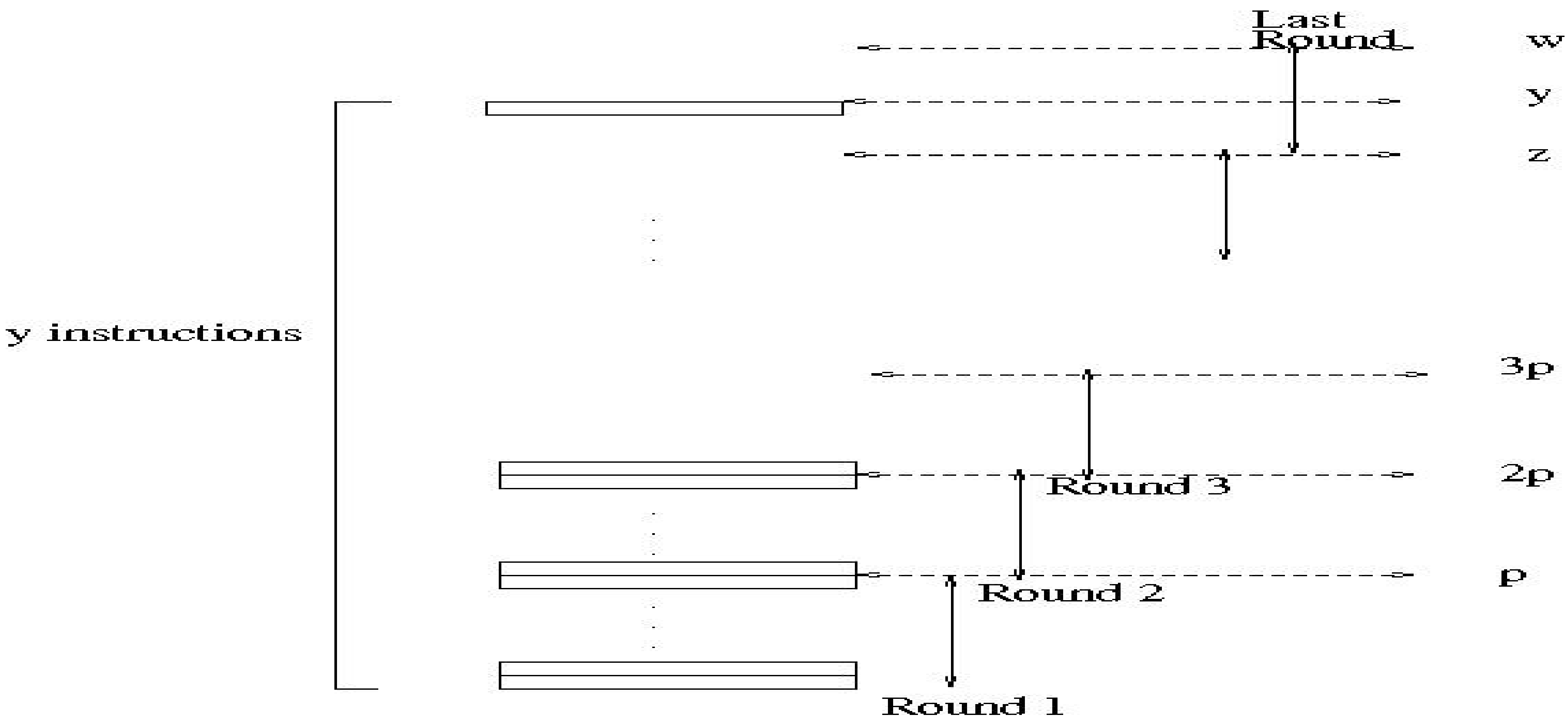
p processors can simulate x_i instructions in $\lceil x_i/p \rceil \leq x_i/p + 1$ time units. See <u>next slide</u>. Demonstration in Algorithm 2' shows why you don't want to leave this to a programmer.

Formally: first reads, then writes. Theorem follows, since

$$\lceil x_1/p \rceil + \lceil x_2/p \rceil + .. + \lceil x_d/p \rceil \leq (x_1/p + 1) + .. + (x_d/p + 1) \leq x/p + d$$

Round-robin emulation of *y* concurrent instructions

by p processors in $\lceil y/p \rceil$ rounds. In each of the first $\lceil y/p \rceil - 1$ rounds, p instructions are emulated for a total of $z = p(\lceil y/p \rceil - 1)$ instructions. In round $\lceil y/p \rceil$, the remaining $y - z$ instructions are emulated, each by a processor, while the remaining $w - y$ processor stay idle, where $w = p\lceil y/p \rceil$



Corollary for summation example

Algorithm 2 would run in $O(n/p + \log n)$ time on a p-processor PRAM.

For $p \leq n/\log n$, this implies $O(n/p)$ time. Later called both <u>*optimal speedup & linear speedup*</u>

For $p \geq n/\log n$: $O(\log n)$ time.

Since no concurrent reads or writes ➔ p-processor EREW PRAM algorithm.

ALGORITHM 2' (Summation on a p-processor PRAM)

1. for P_i , $1 \le i \le p$ pardo
2. for $j := 1$ to $\lceil n/p \rceil - 1$ do
- $B(0, i + (j-1)p) := A(i + (j-1)p)$
3. for i , $1 \le i \le n - (\lceil n/p \rceil - 1)p$
- $B(0, i + (\lceil n/p \rceil - 1)p) := A(i + (\lceil n/p \rceil - 1)p)$
- for i , $n - (\lceil n/p \rceil - 1)p \le i \le p$
- stay idle
4. for $h := 1$ to $\log n$
5. for $j := 1$ to $\lceil n/(2^h p) \rceil - 1$ do (*an instruction $j := 1$ to 0 do means: "do nothing"*)
- $B(h, i+(j-1)p) := B(h-1, 2(i+(j-1)p)-1) + B(h-1, 2(i+(j-1)p))$
6. for i , $1 \le i \le n - (\lceil n/(2^h p) \rceil - 1)p$
- $B(h, i + (\lceil n/(2^h p) \rceil - 1)p) := B(h-1, 2(i + (\lceil n/(2^h p) \rceil - 1)p) - 1) +$
- $B(h-1, 2(i + (\lceil n/(2^h p) \rceil - 1)p))$
- for i , $n - (\lceil n/(2^h p) \rceil - 1)p \le i \le p$
- stay idle
7. for $i = 1$ output $B(\log n, 1)$; for $i > 1$ stay idle

Nothing more than plugging in the above proof.

<u>Main point of this slide</u>: compare to Algorithm 2 and decide, which one you like better

But is WD mode as easy as it gets? Hold on…Key question for this presentation

Measuring the performance of parallel algorithms

A problem. Input size: *n.* A parallel algorithm in WD mode. Worst case time: *T(n);* work*: W(n).*

4 alternative ways to measure performance:

1. W(n) operations and T(n) time.
2. P(n) = W(n)/T(n) processors and T(n) time (on a PRAM).
3. W(n)/p time using any number of p ≤ W(n)/T(n) processors (on a PRAM).
4. W(n)/p + T(n) time using any number of p processors (on a PRAM).

Exercise 1: The above four ways for measuring performance of a parallel algorithms form six pairs. Prove that the pairs are all 'asymptotically equivalent'.

Note: 1➔3 is the WD-presentation sufficiency theorem

Goals for Designers of Parallel Algorithms

Suppose 2 parallel algorithms for same problem:

1. $W_1(n)$ operations in $T_1(n)$ time. 2. $W_2(n)$ operations, $T_2(n)$ time.

General guideline: algorithm 1 more efficient than algorithm 2 if $W_1(n) = o(W_2(n))$, regardless of $T_1(n)$ and $T_2(n)$; if $W_1(n)$ and $W_2(n)$ grow asymptotically the same, then algorithm 1 is considered more efficient if $T_1(n) = o(T_2(n))$.

Good reasons for avoiding strict formal definition—only guidelines

<u>Example</u> $W_1(n)=O(n), T_1(n)=O(n); W_2(n)=O(n \log n), T_2(n)=O(\log n)$ Which algorithm is more efficient?

Algorithm 1: less work. Algorithm 2: much faster.

In this case, both algorithms are probably interesting. Imagine two users, each interested in different input sizes and in different target machines (different # processors). For one user Algorithm 1 faster. For second user Algorithm 2 faster.

Known unresolved issues with asymptotic worst-case analysis.

Nicknaming speedups

Suppose T(n) best possible worst case time upper bound on serial algorithm for an input of length n for some problem. (T(n) is serial time complexity for problem.)

Let W(n) and $T_{par}(n)$ be work and time bounds of a parallel algorithm for same problem.

The parallel algorithm is *work-optimal*, if W(n) grows asymptotically the same as T(n). A work-optimal parallel algorithm is *work-time-optimal* if its running time $T_{par}(n)$ cannot be improved by another work-optimal algorithm.

What if serial complexity of a problem is unknown?

Still an accomplishment if T(n) is best known and W(n) matches it. Called *linear speedup*. Note: can change if serial improves.

Recall main reasons for existence of parallel computing:

- Can perform better than serial
- (it is just a matter of time till) Serial cannot improve anymore

Default assumption regarding shared memory access resolution

Since all conventions represent virtual models of real machines: strongest model whose implementation cost is "still not very high", would be practical.

Simulations results + UMD PRAM-On-Chip architecture

➔ Arbitrary CRCW

NC Theory

Good serial algorithms: poly time.

Good parallel algorithm: poly-log time, poly processors.

Was much more dominant than what's covered here in early 1980s. Fundamental insights. Limited practicality.

In choosing abstractions: fine line between helpful and "defying gravity"

Technique: Balanced Binary Trees;

Problem: Prefix-Sums

Input: Array A[1..n] of elements. Associative binary operation, denoted $*$, defined on the set: $a * (b * c) = (a * b) * c$.

($*$ pronounced "star"; often "sum": addition, a common example.)

The n prefix-sums of array A are:

$A(1)$

$A(1) * A(2)$

..

$A(1) * A(2) * .. * A(i)$

..

$A(1) * A(2) * \quad .. \quad * A(n)$

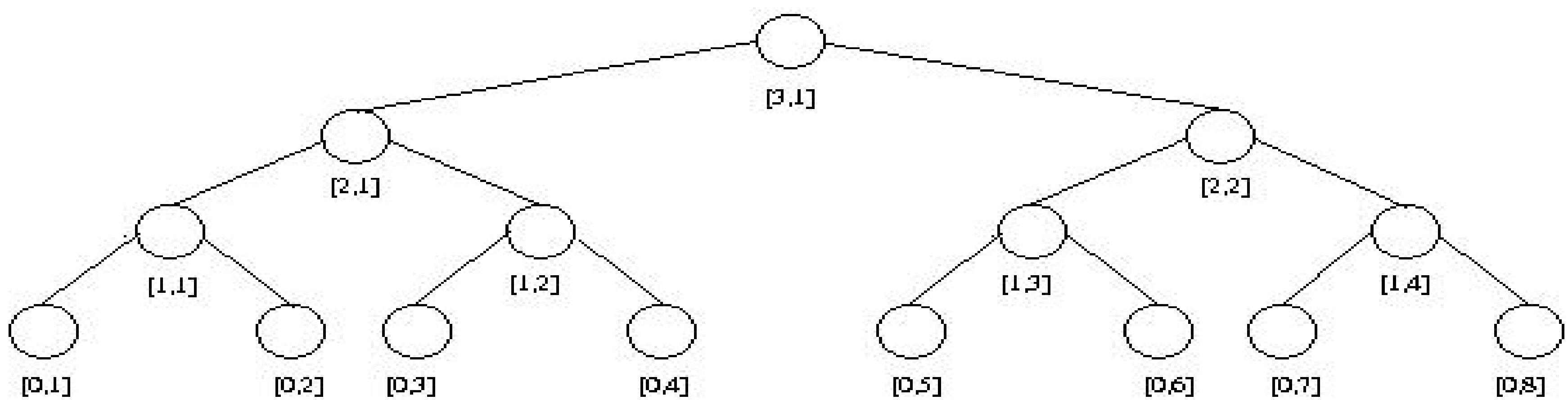
Prefix-sums is perhaps the most heavily used routine in parallel algorithms.

ALGORITHM 1 (Prefix-sums)

1. for i , $1 \le i \le n$ pardo
- $B(0, i) := A(i)$
2. for h := 1 to log n
3. for i , $1 \le i \le n/2^h$ pardo
- $B(h, i) := B(h - 1, 2i - 1) * B(h - 1, 2i)$
4. for h := log n to 0
5. for i even, $1 \le i \le n/2^h$ pardo
- $C(h, i) := C(h + 1, i/2)$
6. for i = 1 pardo
- $C(h, 1) := B(h, 1)$
7. for i odd, $3 \le i \le n/2^h$ pardo
- $C(h, i) := C(h + 1, (i - 1)/2) * B(h, i)$
8. for i , $1 \le i \le n$ pardo
- Output $C(0, i)$

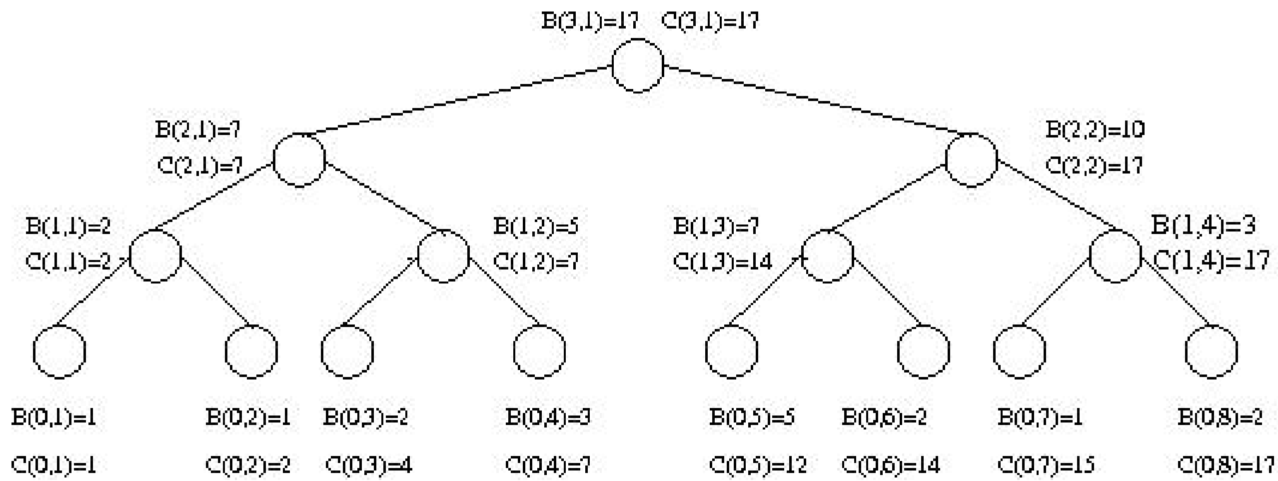
Summation (as before)

C(h,i) – prefix-sum of rightmost leaf of [h,i]



Prefix-sums algorithm

Example



Complexity Charge operations to nodes. Tree has 2n-1 nodes.
No node is charged with more than O(1) operations.
➔W(n) = O(n). Also T(n) = O(log n)
Theorem: The prefix-sums algorithm runs in O(n) work and O(log n) time.

Application - the Compaction Problem

The Prefix-sums routine is heavily used in parallel algorithms. A trivial application follows:

Input Array A = A[1. . N] of elements, and binary array B = B[1 . . n].

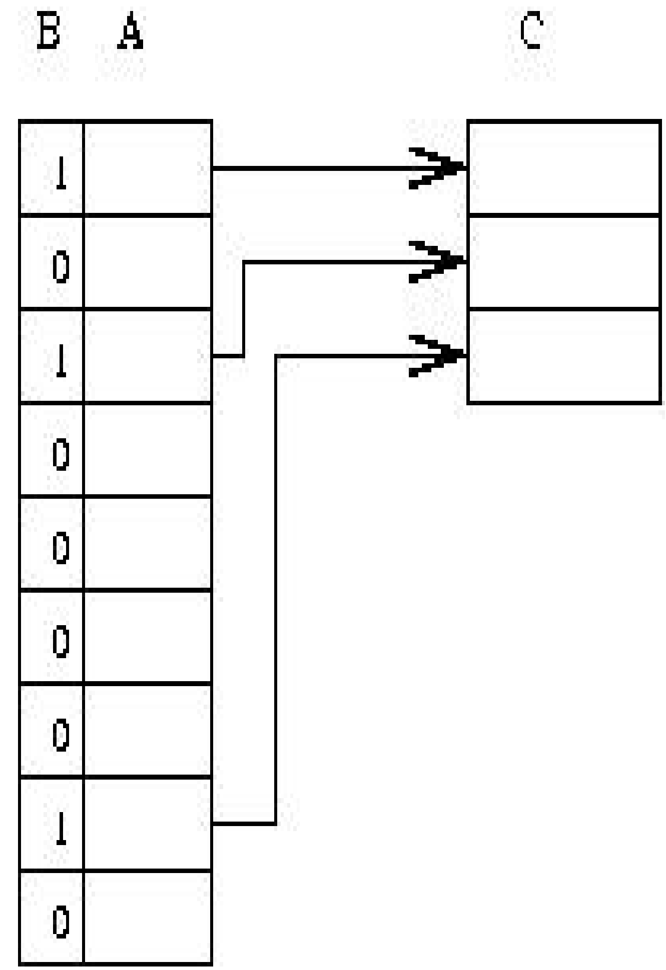
Map each value i, $1 \le i \le n$, where B(i) = 1, to the sequence (1, 2, . . . , s); s is the (a priori unknown) numbers of ones in B. Copy the elements of A accordingly.

The solution is order preserving. But, quite a few applications of compaction do not require that.

For computing the mapping, simply find prefix sums with respect to array B.

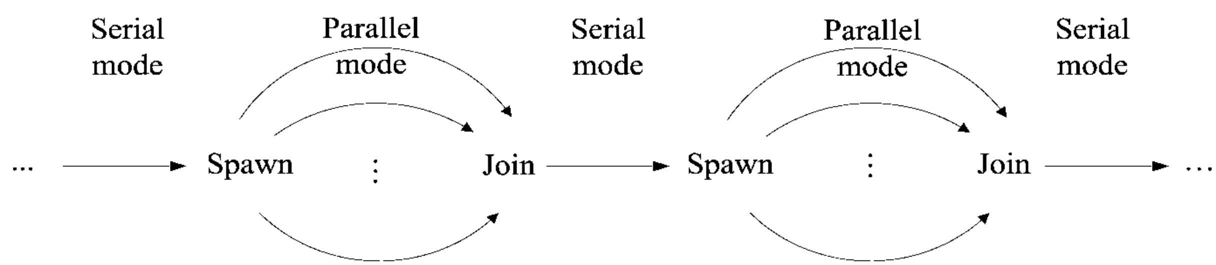
Consider an entry B(i) = 1. If the prefix sum of i is j then map A(i) into C(j).

Theorem The compaction algorithm runs in O(n) work and O(log n) time.



Snapshot: XMT High-level language

XMTC: Single-program multiple-data (SPMD) extension of standard C.
Includes Spawn and PS - a multi-operand instruction.
Short (not OS) threads.



Cartoon Spawn creates threads; a thread progresses at its own speed and expires at its Join.
Synchronization: only at the Joins.
So, virtual threads avoid busy-waits by expiring.
New: Independence of order semantics (IOS).

XMT High-level language (cont'd)

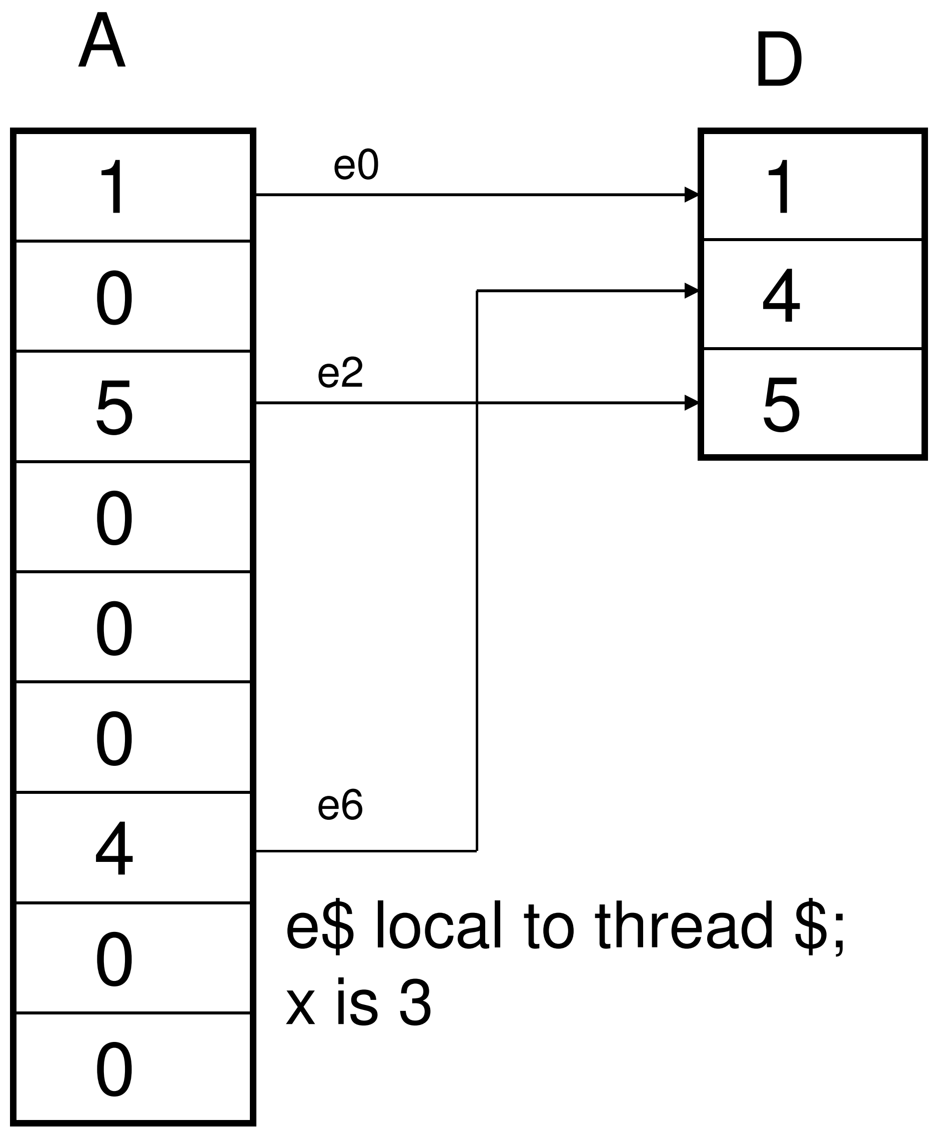
The array compaction problem

Input: A[1..n]. Map in some order all A(i) not equal 0 to array D.

Essence of an XMT-C program

```
int x = 0; /*formally: psBaseReg x=0*/
spawn(0, n-1) /* Spawn n threads; $
    ranges 0 to n – 1 */
{ int e = 1;
  if (A[$] not-equal 0)
    { ps(e,x);
      D[e] = A[$] }
}
n = x;
```

Notes: (i) PS is defined next (think F&A). See results for e0,e2, e6 and x. (ii) Join instructions are implicit.



e$ local to thread $;
x is 3

XMT Assembly Language

Standard assembly language, plus 3 new instructions: Spawn, Join, and PS.

The PS multi-operand instruction

New kind of instruction: Prefix-sum (PS).

Individual PS, PS Ri Rj, has an inseparable (“atomic”) outcome:

(i) Store Ri + Rj in Ri, and

(ii) store original value of Ri in Rj.

Several successive PS instructions define a multiple-PS instruction. E.g., the sequence of k instructions:

PS R1 R2; PS R1 R3; ...; PS R1 R(k + 1)

performs the prefix-sum of base R1 elements R2,R3, ...,R(k + 1) to get:

R2 = R1; R3 = R1 + R2; ...; R(k + 1) = R1 + ... + Rk; R1 = R1 + ... + R(k + 1).

Idea: (i) Several ind. PS’s can be combined into one multi-operand instruction.

(ii) Executed by a new multi-operand PS functional unit.

Mapping PRAM Algorithms onto XMT

(1st visit of this slide)

(1) PRAM parallelism maps into a thread structure

(2) Assembly language threads are not-too-short (to increase locality of reference)

(3) the threads satisfy IOS

<u>How</u> (summary):

I. Use work-depth methodology [SV-82] for "thinking in parallel". The rest is skill.

II. Go through PRAM or not.

For performance-tuning, in order to later teach the compiler. (To be suppressed as it is ideally done by compiler):

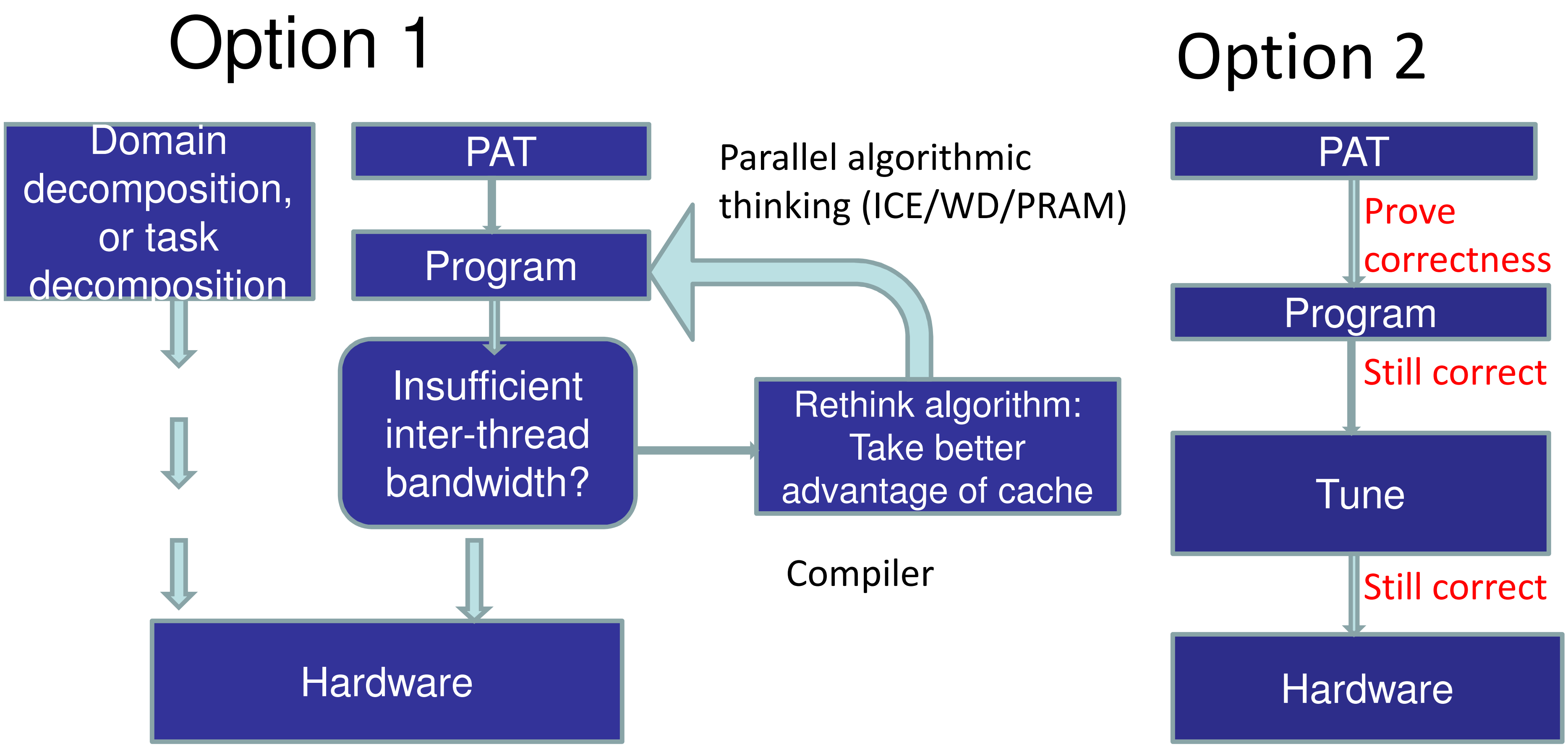
Produce XMTC program accounting also for:

(1) Length of sequence of round trips to memory,

(2) QRQW.

Issue: nesting of spawns.

Workflow from parallel algorithms to programming versus trial-and-error



Is Option 1 good enough for the parallel programmer's model?
Options 1B and 2 start with a PRAM algorithm, but not option 1A.
Options 1A and 2 represent workflow, but not option 1B.

Not possible in the 1990s.
Possible now: XMT@UMD
Why settle for less?

<u>Exercise 2</u> Let A be a memory address in the shared memory of a PRAM. Suppose all p processors of the PRAM need to “know” the value stored in A. Give a fast EREW algorithm for broadcasting A to all p processors. How much time will this take?

<u>Exercise 3</u> Input: An array A of n elements drawn from some totally ordered set. The <u>minimum problem</u> is to find the smallest element in array A.
(1) Give an EREW PRAM algorithm that runs in O(n) work and O(log n) time.
(2) Suppose we are given only $p \leq n/\log n$ processors numbered from 1 to p. For the algorithm of (1) above, describe the algorithm to be executed by processor i, $1 \leq i \leq p$.
The prefix-min problem has the same input as for the minimum problem and we need to find for each i, $1 \leq i \leq n$, the smallest element among A(1),A(2), . . . ,A(i).
(3) Give an EREW PRAM algorithm that runs in O(n) work and O(log n) time for the prefix-min problem.

<u>Exercise 4</u> The <u>nearest-one problem</u> is defined as follows. Input: An array A of size n of bits; namely, the value of each entry of A is either 0 or 1. The nearest-one problem is to find for each i, $1 \leq i \leq n$, the largest index $j \leq i$, such that A(j) = 1.
(1) Give an EREW PRAM algorithm that runs in O(n) work and O(log n) time.
The input for the <u>segmented prefix-sums problem</u> includes the same binary array A as above, and in addition an array B of size n of numbers. The segmented prefix-sums problem is to find for each i, $1 \leq i \leq n$, the sum $B(j) + B(j + 1) + . . . + B(i)$, where j is the nearest-one for i (if i has no nearest-one we define its nearest-one to be 1).
(2) Give an EREWPRAM algorithm for the problem that runs in O(n) work and O(log n) time.

Recursive Presentation of the Prefix-Sums Algorithm

Recursive presentations are useful for describing both serial and parallel algorithms. Sometimes they shed new light on a technique being used.

PREFIX-SUMS($x_1, x_2, \ldots, x_m; u_1, u_2, \ldots, u_m$)

1. if $m = 1$ then $u_1 := x_1$; exit
2. for i, $1 \le i \le m/2$ pardo
- $y_i := x_{2i-1} * x_{2i}$
3. PREFIX-SUMS($y_1, y_2, \ldots, y_{m/2}; v_1, v_2, \ldots, v_{m/2}$)
4. for i even, $1 \le i \le m$ pardo
- $u_i := v_{i/2}$
5. for $i = 1$ pardo
- $u_1 := x_1$
6. for i odd, $3 \le i \le m$ pardo
- $ui := v_{(i-1)/2} * x_i$

To start, call: PREFIX-SUMS(A(1),A(2), . . . ,A(n);C(0, 1),C(0, 2), . . . ,C(0, n)).

Complexity Recursive presentation can give concise and elegant complexity analysis. Excluding the recursive call in instruction 3, routine PREFIX-SUMS, requires: $\le \alpha$ time, and $\le \beta m$ operations for some positive constants α and β. The recursive call is for a problem of size $m/2$. Therefore,

$T(n) \le T(n/2) + \alpha$

$W(n) \le W(n/2) + \beta n$

Their solutions are $T(n) = O(\log n)$, and $W(n) = O(n)$.

Exercise 5: Multiplying two n × n matrices A and B results in another n × n matrix C, whose elements $c_{i,j}$ satisfy $c_{i,j} = a_{i,1}b_{1,j} + .. + a_{i,k}b_{k,j} + .. + a_{i,n}b_{n,j}$.

(1) Given two such matrices A and B, show how to compute matrix C in O(log n) time using n^3 processors.

(2) Suppose we are given only $p \le n^3$ processors, which are numbered from 1 to p. Describe the algorithm of item (1) above to be executed by processor i, $1 \le i \le p$.

(3) In case your algorithm for item (1) above required more than $O(n^3)$ work, show how to improve its work complexity to get matrix C in $O(n^3)$ work and O(log n) time.

(4) Suppose we are given only $p \le n^3/\log n$ processors numbered from 1 to p. Describe the algorithm for item (3) above to be executed by processor i, $1 \le i \le p$.

Merge-Sort

Input: Two arrays A[1. . n], B[1. . m]; elements from a totally ordered domain S. Each array is monotonically non-decreasing.

Merging: map each of these elements into a monotonically non-decreasing array C[1..n+m]

<u>The partitioning paradigm</u>

n: input size for a problem. Design a 2-stage parallel algorithm:

1. Partition the input into a large number, say p, of independent small jobs AND size of the largest small job is roughly n/p.
2. Actual work - do the small jobs concurrently, using a separate (possibly serial) algorithm for each.

<u>Ranking Problem</u>

Input: Same as for merging.

For every 1<=i<= n, RANK(i,B), and 1<=j<=m, RANK(j,A)

Example: A=[1,3,5,7,9],B[2,4,6,8]. RANK(3,B)=2;RANK(1,A)=1

Merging algorithm (cnt'd)

<u>Observe</u> Merging & Ranking: really same problem.

<u>Show</u> M➔R in W=O(n),T=O(1) (say n=m):

C(k)=A(i) ➔ RANK(i,B)=k-i

<u>Show</u> R➔M in W=O(n),T=O(1):

RANK(i,B)=j➔C(i+j)=A(i)

<u>"Surplus-log" parallel algorithm for the Ranking</u>

for 1 ≤ i ≤ n pardo

- Compute RANK(i,B) using standard binary search
- Compute RANK(i,A) using binary search

Complexity: W=(O(n log n), T=O(log n)

Serial 'zipper' algorithm for ranking

SERIAL – RANK(A[1 . .];B[1. .])

i := 0 and j := 0; add two auxiliary elements A(n+1) and B(n+1), each larger than both A(n) and B(n)

while i ≤ n or j ≤ n do

- if A(i + 1) < B(j + 1)
- then RANK(i+1,B) := j; i := i + 1
- else RANK(j+1),A) := i; j := j + 1

In words Starting from A(1) and B(1), in each round:

1. compare an element from A with an element of B
2. determine the rank of the smaller among them

Complexity: O(n) time (and O(n) work...)

Linear work parallel merging

Partitioning for 1 ≤ i ≤ n/p pardo [p <= n/log and p | n]

- b(i):=RANK(p(i-1) + 1),B) using binary search
- a(i):=RANK(p(i-1) + 1),A) using binary search

Actual work

Observe Ranking task can be broken into 2p independent “slices”.

Example of a slice

Start at A(p(i-1) +1) and B(b(i)).

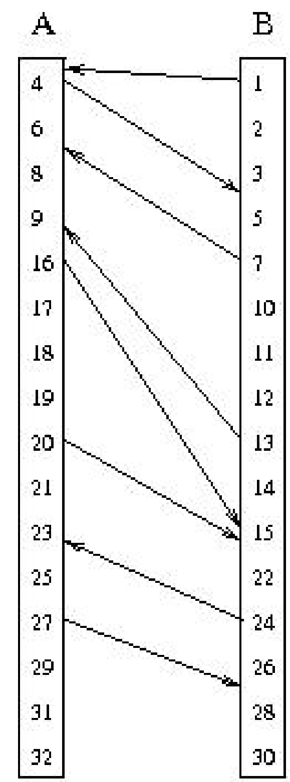
Using serial ranking advance till:

Termination condition

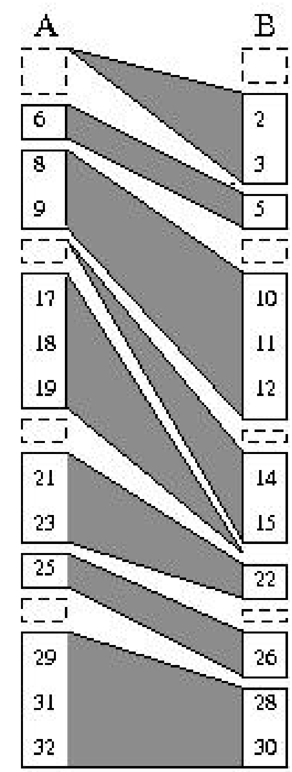
Either A(pi+1) or some B(jp+1) loses

Parallel algorithm

2p concurrent threads



Step 1
partitioning



Step 2
actual work

Linear work parallel merging (cont'd)

Observation 2p slices. None larger than 2n/p.

(not too bad since average is 2n/2p=n/p)

Complexity Partitioning takes O(p log n) work and O(log n) time, or O(n) work and O(log n) time. Actual work employs 2p serial algorithms, each takes O(n/p) time. Total work is O(n) and time is O(log n), for p=n/log n.

Exercise 6: Consider the merging problem as above. Consider a variant of the above merging algorithm where instead of fixing x (p above) to be n/ log n, x could be any positive integer between 1 and n.
Describe the resulting merging algorithm and analyze its time and work complexity as a function of both x and n.

Exercise 7: Consider the merging problem as above, and assume that the values of the input elements are not pair wise distinct. Adapt the merging algorithm for this problem, so that it will take the same work and the same running time.

Exercise 8: Consider the merging problem as above, and assume that the values of n and m are not equal. Adapt the merging algorithm for this problem. What are the new work and time complexities?

Exercise 9: Consider the merging algorithm as above. Suppose that the algorithm needs to be programmed using the smallest number of Spawn commands in an XMT-C single-program multiple-data (SPMD) program. What is the smallest number of Spawn commands possible? Justify your answer.
(Note: This exercise should be given only after XMT-C programming has been introduced.)

Technique: Divide and Conquer
Problem: Sort (by-merge)

Input: Array A[1 .. n], drawn from a totally ordered domain.

Sorting: reorder (permute) the elements of A into array B, such that $B(1) \leq B(2) \leq . . . \leq B(n)$.

Sort-by-merge: classic serial algorithm. This known algorithm translates directly into a reasonably efficient parallel algorithm.

Recursive description (assume $n = 2^l$ for some integer $l \geq 0$):

MERGE – SORT(A[1 .. n];B[1 .. n])

if n = 1

then return B(1) := A(1)

else call, in parallel,

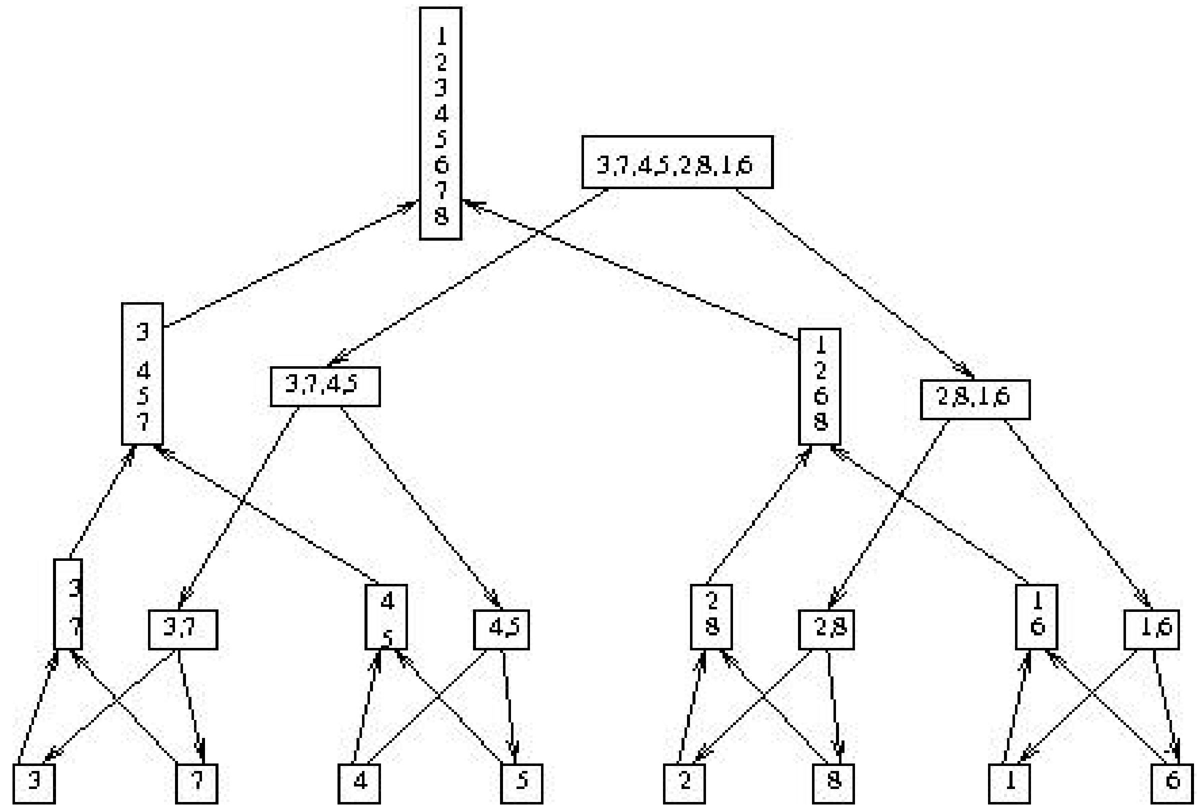
- MERGE – SORT(A[1 .. n/2];C[1 .. n/2]) and

- MERGE – SORT(A[n/2 +1 .. n);C[n/2 + 1 .. n])

Merge C[1 .. n/2] and C[n/2 +1) .. N] into B[1 .. N]

Merge-Sort

Example:



<u>Complexity</u> The linear work merging algorithm runs in O(log n) time.
Hence, time and work for merge-sort satisfy:
$T(n) \leq T(n/2) + \alpha \log n$; $W(n) \leq 2W(n/2) + \beta n$ where $\alpha, \beta > 0$ are constants.
Solutions: $T(n) = O(\log^2 n)$ and $W(n) = O(n \log n)$.

Merge-sort algorithm is a “balanced binary tree” algorithm. See above figure and try to give a non-recursive description of merge-sort.

PLAN

1. Present 2 general techniques:
- Accelerating cascades
- Informal Work-Depth—what "thinking in parallel" means in this presentation
2. Illustrate using 2 approaches for the selection problem: deterministic (clearer?) and randomized (more practical)
3. Program (if you wish) the latter

Problem: Selection

Input: Array A[1..n] from a totally ordered domain; integer k, $1 \leq k \leq n$. A(j) is k-th smallest in A if $\leq k-1$ elements are smaller and $\leq n-k$ elements are larger.

Selection problem: find a k-th smallest element.

Example. A=[9,7,2,3,8,5,7,4,2,3,5,6]; n=12;k=4. Either A(4) or A(10) (=3) is 4-th smallest. For k=5, A(8)=4 is the only 5-th smallest element.

Instances of selection problem: (i) for k=1, the minimum element, (ii) for k=n, the maximum (iii) for $k = \lceil n/2 \rceil$, the median.

Accelerating Cascades - Example

Get a fast O(n)-work selection algorithm from 2 “pure” selection algorithms:

(1) Algorithm 1 has O(log n) iterations. Each reduces a size m instance of selection in O(log m) time and O(m) work to an instance whose size is ≤ 3m/4. Why is the complexity of Algorithm 1 $O(\log^2 n)$ time and O(n) work?

(2) Algorithm 2 runs in O(log n) time and O(n log n) work.

Pros: Algorithm 1: only O(n) work. Algorithm 2: less time.

Accelerating cascades technique way for deriving a single algorithm that is both: fast and needs O(n) work.

Main idea start with Algorithm 1, but not run it to completion. Instead, switch to Algorithm 2, as follows:

Step 1 Use Algorithm 1 to reduce selection from n to ≤ n/ log n. Note: O(log log n) rounds are enough, since for $(3/4)^r n \le n/\log n$, we need $(4/3)^r \ge \log n$, implying $r = \log_{4/3}\log n$.

Step 2 Apply Algorithm 2.

Complexity Step 1 takes O(log n log log n) time. The number of operations is n+(3/4)n+.. which is O(n). Step 2 takes additional O(log n) time and O(n) work. In total: O(log n log log n) time, and O(n) work.

Accelerating cascades is a practical technique.

Algorithm 2 is actually a sorting algorithm.

Accelerating Cascades

Consider the following situation: for problem of size n, there are two parallel algorithms.

Algorithm A: $W_1(n)$ and $T_1(n)$. Algorithm B: $W_2(n)$ and $T_2(n)$ time. Suppose: Algorithm A is more efficient ($W_1(n) < W_2(n)$), while Algorithm B is faster ($T_2(n) < T_1(n)$). Assume also: Algorithm A is a "reducing algorithm": Given a problem of size n, Algorithm A operates in phases. Output of each successive phase is a smaller instance of the problem. The accelerating cascades technique composes a new algorithm as follows:

Start by applying Algorithm A. Once the output size of a phase of this algorithm is below some threshold, finish by switching to Algorithm B.

Algorithm 1, and IWD Example

Note: not just a selection algorithm. Interest is broader, as the informal work-depth (IWD) presentation technique is illustrated. In line with the IWD presentation technique, some missing details for the current high-level description of Algorithm 1 are filled in later.

Input Array A[1..n]; integer k, $1 \leq k \leq n$.

Algorithm 1 works in “reducing” ITERATIONS:

Input: Array B[1..m]; $1 \leq k_0 \leq m$. Find k_0-th element in B.

Main idea behind a reducing iteration is: find an element α of B which is guaranteed to be not too small (≤ m/4 elements of B are smaller), and not too large (≤ m/4 elements of B are larger). Exact ranking of α in B enables to conclude that at least m/4 elements of B do not contain the k_0-th smallest element. Therefore, they can be discarded. The other alternative: the k_0-th smallest element (which is also the k-th smallest element with respect to the original input) has been found.

ALGORITHM 1 - High-level description (Assume: log m and m/ log m are integers.)

1. for i, $1 \le i \le n$ pardo
   B(i) := A(i)
2. k_0 := k; m := n
3. while m > 1 do

3.1. Partition B into m/log m blocks, each of size log m as follows. Denote the blocks $B_1,..,B_{m/\log m}$, where B_1=B[1..logm],..,$B_{m/\log m}$=B[m+1−log m..m].

3.2. for block Bi, $1 \le i \le m/\log m$ pardo
   compute the median α_i of Bi, using a linear time serial selection algorithm

3.3. Apply a sorting algorithm to find α the median of medians $(\alpha_1, \ldots, \alpha_{m/\log m})$.

3.4. Compute s_1, s_2 and s_3. s_1: # elements in B smaller than α, s_2: # elements equal to α, and s_3: # elements larger than α.

3.5. There are three possibilities:

3.5.1 (i) $k_0 \le s_1$: the new subset B (the input for the next iteration) consists of the elements in B, which are smaller than α ($m := s_1$; k_0 remains the same)

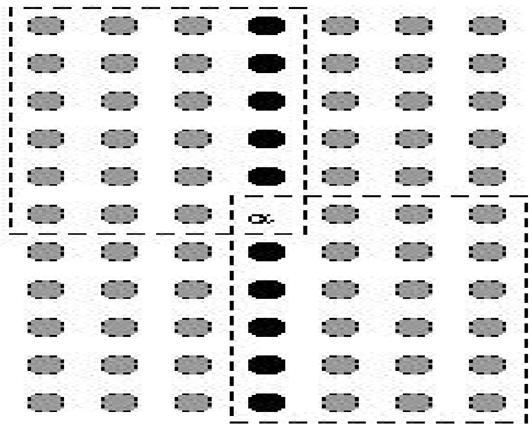
3.5.2 (ii) $s_1 < k_0 \le s_1 + s_2$: α is the k_0-th smallest element in B; algorithm terminates

3.5.3 (iii) $k_0 > s_1 + s_2$: the new subset B consists of the elements in B, which are larger than α ($m := s_3$; $k_0 := k_0 - (s_1 + s_2)$)

4. (we can reach this instruction only with m = 1 and k_0 = 1)
   B(1) is the k_0-th element in B.

Reducing Lemma At least m/4 elements of B are smaller than α, and at least m/4 are larger.

Proof



Corollary 1 Following an iteration of Algorithm 1 the value of m decreases so that the new value of m is at most (3/4)m.

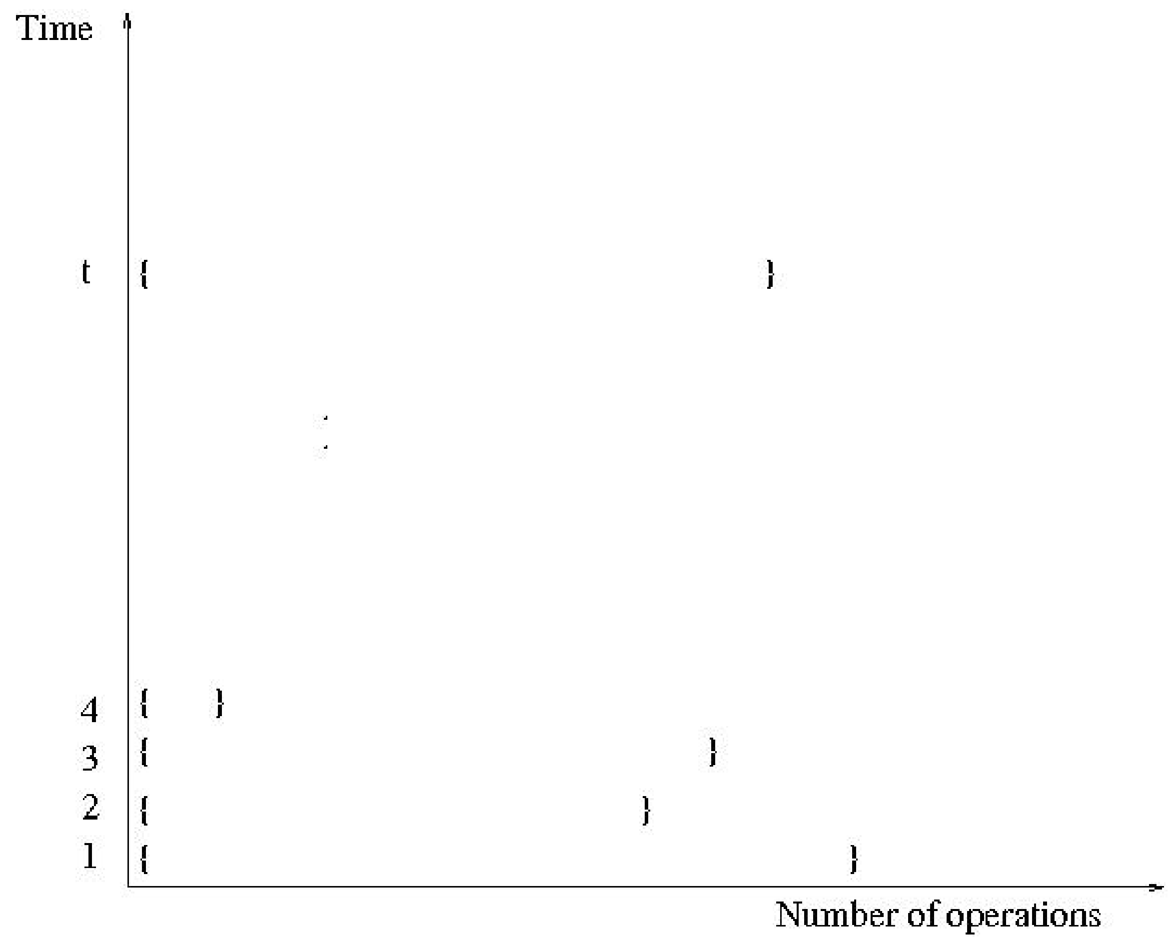
Informal Work-Depth (IWD) description

Similar to Work-Depth, the algorithm is presented in terms of a sequence of parallel time units (or "rounds"); however, at each time unit there is a set containing a number of instructions to be performed concurrently

Descriptions of the set of concurrent instructions can come in many flavors. Even implicit, where the number of instruction is not obvious.

Example Algorithm 1 above: The input (and output) for each reducing iteration is given as a set. We were also not specific on how to compute s_1, s_2 and s_3.

The main methodical issue addressed here is how to train CS&E professionals "to think in parallel". Here is the informal answer: train yourself to provide IWD description of parallel algorithms. The rest is detail (although important) that can be acquired as a skill (also a matter of training).



The Selection Algorithm (wrap-up)

To derive the lower level description of Algorithm 1, simply apply the prefix-sums algorithm several times.

Theorem 5.1 Algorithm 1 solves the selection problem in $O(\log^2 n)$ time and $O(n)$ work. The main selection algorithm, composed of algorithms 1 and 2, runs in $O(n)$ work and $O(\log n \log\log n)$ time.

Exercise 10 Consider the following sorting algorithm. Find the median element and then continue by sorting separately the elements larger than the median and the ones smaller than the median. Explain why this is indeed a sorting algorithm. What will be the time and work complexities of such algorithm?

Recap: (i) Accelerating cascades framework was presented and illustrated by selection algorithm. (ii) A top-down methodology for describing parallel algorithms was presented. Its upper level, called Informal Work-Depth (IWD), is proposed as the essence of thinking in parallel.

Randomized Selection

Parallel version of serial randomized selection from CLRS, Ch. 9.2

<u>Input</u> Array A[p...r]

```
RANDOMIZED_PARTITION(A,p,r)
1.    i := RANDOM (p,r)
/*Rearrange A[p...r]: elements <=
      A(i) followed by those > A(i)*/
2.    exchange A(r) ←→A(i)
3.    return PARTITION(A,p,r)
```

```
PARTITION(A,p,r)
1.    x := A(r)
2.    i := p-1
3.    for j := p to r-1
4.        if A(j) <= x
5.        then i := i+1
6.                exchange A(i) ←→A(j)
7.    exchange A(i+1) ←→A(r)
8.    Return i+1
```

<u>Input</u> Array A[p...r], i. Find i-th smallest

```
RANDOMIZED_SELECT(A,p,r,i)
1. if p=r
2. Then return A(p)
3. q := RANDOMIZED_PARTITION(A,p,r)
4. k := q-p+1
5. if i=k
6. then return A(q)
7. else if i < k
8.        then return
           RANDOMIZED_SELECT(A,p,q-1,i)
9.        else return
           RANDOMIZED_SELECT(A,q+1,r,i-k)
```

Basis for proposed programming project

Integer Sorting

<u>Input</u> Array A[1..n], integers from range [0..r–1]; n and r are positive integers.

Sorting: rank from smallest to largest.

Assume n is divisible by r. Typical value for r might be $n^{1/2}$; other values possible.

Two comments about the parallel integer sorting algorithm:

(i) Its performance depends on the value of r, and unlike other parallel algorithms we have seen, its running time may not be bounded by $O(\log^k n)$ for any constant k ("poly-logarithmic"). It is a remarkable coincidence that the literature includes only very few work-efficient non ploy-log parallel algorithms.

(ii) It already lent itself to efficient implementation on a few parallel machines in the early 1990s. (Remark later.)

The algorithm work as follows:

1. Partition A into n/r subarrays: B_1=A[1..r]..$B_{n/r}$=A[n−r+1..n]. Using serial bucket sort (see Exercise 12 below), sort each subarray separately (and in parallel for all subarrays). Also compute: (1) number(v,s) - the number of elements whose value is v in subarray B_s, for 0≤v≤ r−1, and 1≤s≤n/r; and (2) serial(i) - the number of elements A(j) such that A(j)=A(i) and precede element i in its subarray B_s (i.e., serial(i) counts only j < i, where $\lceil j/r \rceil = \lceil i/r \rceil$ = s), for 1 ≤ i ≤ n.

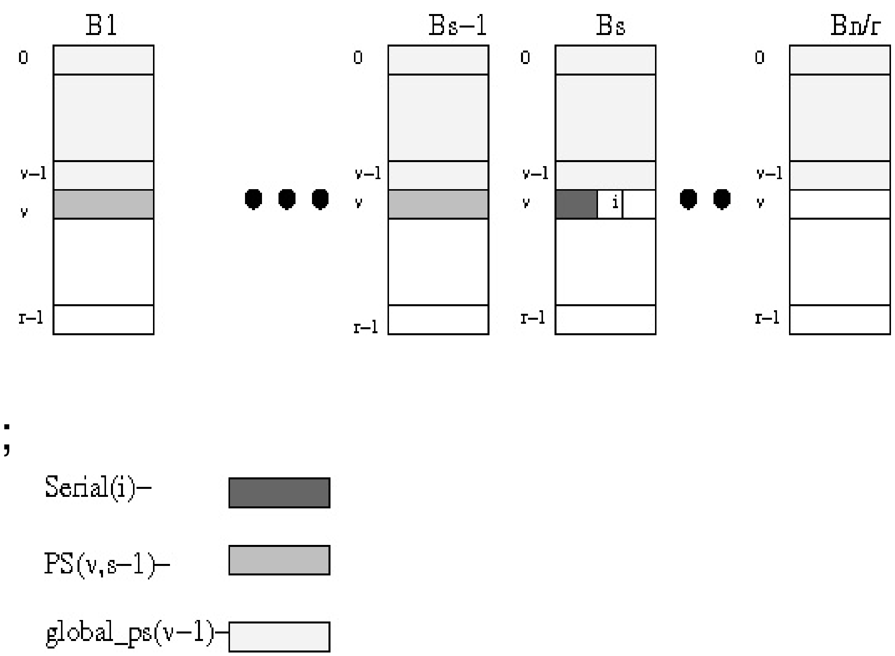
Example B_1=(2,3,2,2) (r=4). Then, number(2,1) = 3, and serial(3)=1.

2. Separately (and in parallel) for each value 0 ≤ v ≤ r−1 compute the prefix-sums of number(v,1), number(v,2) .. number(v,n/r) into ps(v,1), ps(v,2) .. ps(v,n/r), and their sum (the number of elements whose value is v) into cardinality(v).

3. Compute the prefix sums of cardinality(0), cardinality(1) .. cardinality(r−1) into global−ps(0), global−ps(1) .. global−ps(r−1).

4. In parallel for every element i, 1≤i≤n [Let v = A(i) and B_s the subarray of element i (s = $\lceil i/r \rceil$]: The rank of element i is 1+serial(i)+ps(v,s−1)+global−ps(v−1) [where ps(0,s)=0 and global−ps(0)=0]

Exercise 11: Describe the integer sorting algorithm in a "parallel program", similar to the pseudo-code that we usually give.



Complexity

1: T=O(r), W=O(r) per subarray; total: T=O(r), W=O(n).

2: r computations; each T=O(log(n/r)),W=O(n/r); total T=O(log n), W=O(n) work.

3: T=O(log r), W=O(r).

4: T=O(1), W=O(n) work.

Total: T=O(r + log n), W=O(n).

Theorem 6.1: (1) The integer sorting algorithm runs in $O(r+\log n)$ time and $O(n)$ work. (2) The integer sorting algorithm can be applied to run in time $O(k(r^{1/k}+\log n))$ and $O(kn)$ work for any positive integer k.

Showed (1). For (2): radix sort using the basic integer sort (BIS) algorithm:

A sorting algorithm is stable if for every pair of two equal input elements $A(i) = A(j)$ where $1 \le i < j \le n$, it ranks element i lower than element j.

Observe: BIS is stable.

Only outline the case $k = 2$.

<u>2-step algorithm for an integer sort problem with $r=n$ in $T=O(\sqrt{n})$ $W=O(n)$</u>

Note: the big Oh notation suppresses the factor $k=2$.

Assume that $\sqrt{n}$ is an integer.

<u>Step 1</u> Apply BIS to keys $A(1) \pmod{\sqrt{n}}$, $A(2) \pmod{\sqrt{n}}$ .. $A(n) \pmod{\sqrt{n}}$. If the computed rank of an element i is j then set $B(j) := A(i)$.

<u>Step 2</u> Apply again BIS this time to key $\lfloor B(1)/\sqrt{n} \rfloor$, $\lfloor B(2)/\sqrt{n} \rfloor$ .. $\lfloor B(n)/\sqrt{n} \rfloor$.

<u>Example</u> 1. Suppose UMD has 35,000 students with social security number as IDs. Sort by IDs. The value of k will be 4 since $\sqrt{1B} \le 35{,}000$ and 4 steps are used.

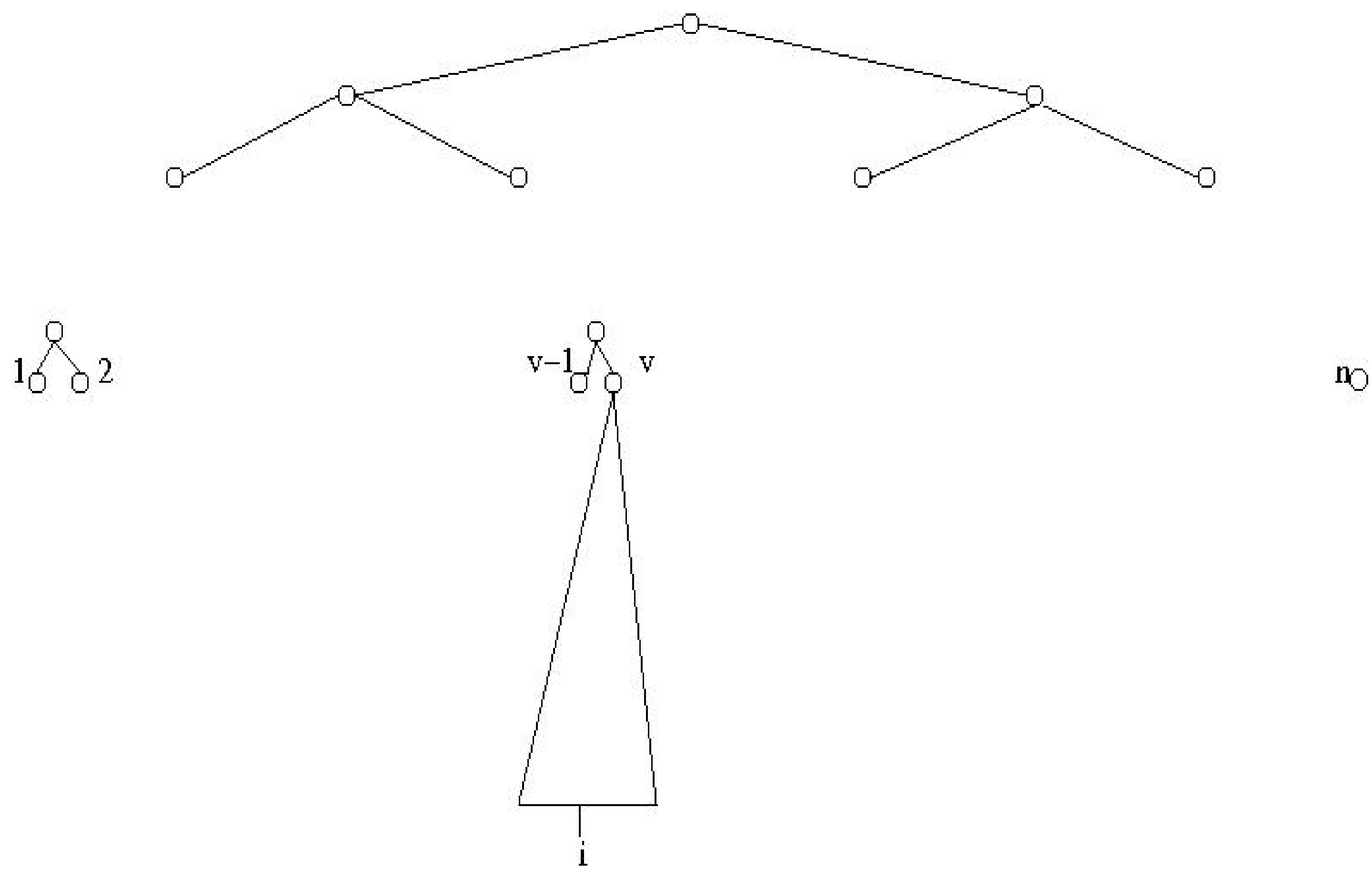
2. Let A=10,12,9,2,3,11,10,12,4,5,9,4,3,7,15,1 with $n=16$ and $r=16$. Keys for Step 1 are values modulo 4: 2,0,1,2,3,3,2,0,0,1,1,0,3,3,3,1. Sorting & assignment to array B: 12,12,4,4,9,5,9,1,10,2,10,3,11,3,15. Keys for Step 2 are $\lfloor v/4 \rfloor$, where v is the value of an element of B (i.e., $\lfloor 9/4 \rfloor=2$). The keys are 3,3,1,1,2,1,2,0,2,0,2,0,2,0,3. The result relative to the original values of A is 1,2,3,3,4,5,7,9,9,10,10,11,12,12,15.

Remarks 1. This simple integer sorting algorithm has led to efficient implementation on parallel machines such as some Cray machines and the Connection Machine (CM-2). [BLM+91] and [ZB91] report giving competitive performance on the machines that they examined. Given a parallel computer architecture where the local memories of different (physical) processors are distant from one another, the algorithm enables partitioning of the input into these local memories without any inter-processor communication. In steps 2 and 3, communication is used for applying the prefix-sums routine. Over the years, several machines had special constructs that enable very fast implementation of such a routine.
2. Since the theory community looked favorably at the time only on poly-log time algorithm, this practical sorting algorithm was originally presented in [CV-86] for a routine for sorting integers in the range 1 to log n as was needed for another algorithm.

Exercise 12: (Redundant if you remember the serial bucket-sort algorithm).
The serial bucket-sort (called also bin-sort) algorithm works as follows. Input: An array $A = A(1), \ldots, A(n)$ of integers from the range $[0, \ldots, n-1]$. For each value v, $0 \le v \le n-1$, the algorithm forms a linked list of all elements $A(i) = v$, $0 \le i \le n-1$. Initially, all lists are empty. Then, at step i, $0 \le i \le n-1$, element A(i) is inserted to the linked list of value v, where $v = A(i)$. Finally, the linked list are traversed from value 0 to value $n-1$, and all the input elements are ranked. (1) Describe this serial bucket-sort algorithm in pseudo-code using a “structured programming style”. Make sure that the version you describe provides stable sorting. (2) Show that the time complexity is O(n).

The orthogonal-tree algorithm

Integer sorting problem Range of integers: [1 .. n]. In a nutshell: the algorithm is a big prefix-sum computation with respect to the data structure below. For each integer value v, $1 \leq v \leq n$, it has an n-leaf balanced binary tree.



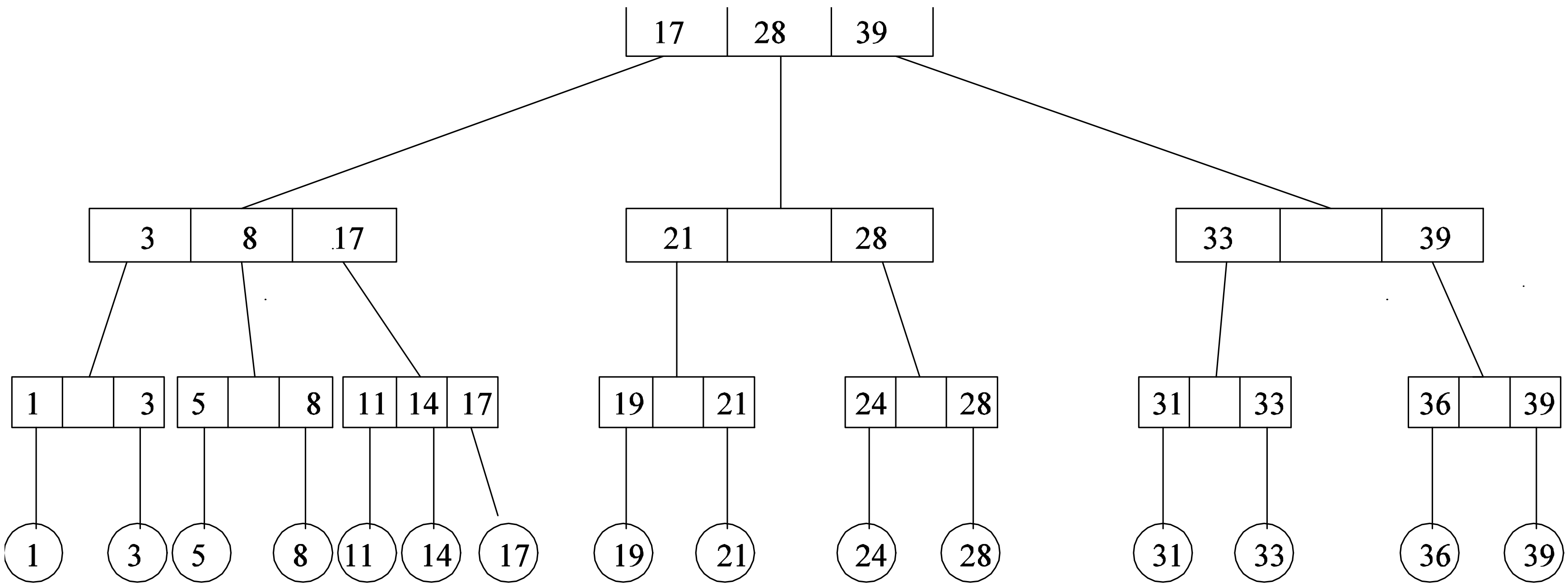
1 (i) In parallel, assign processor i, $1 \le i \le n$ to each input element A(i). Focus on one element A(i). Suppose A(i) = v.
(ii) Advance in log n rounds from leaf i in tree v to its root. In the process, compute the number of elements whose value is v. When 2 processors "meet" at an internal node of the tree one of them proceeds up the tree; the 2nd "sleep-waits" at that node.
The plurality of value v is now available at leaf v of the top (single) binary tree that will guide steps 2 and 3 below..
2 Using a similar log n-round process, processors continue to add up these pluralities; in case 2 processors meet, one proceeds and the other is left to sleep-wait.
The total number of all pluralities (namely n) is now at the root of the upper tree. Step 3 computes the prefix-sums of the pluralities of the values into leaves of the top tree.
3 A log n-round "playback" of Step 2 from the root of the top tree its leaves follows. [Exercise: figure out how to obtain prefix-sums of the pluralities of values at leaves of the top tree.] Only interesting case: internal node where a processor was left sleep-waiting in Step 2. Idea: wake this processor up, send the waking processor and the just awaken one with prefix-sum values in the direction of its original lower tree.
The objective of Step 4 is to compute the prefix-sums of the pluralities of the values at every leaf of the lower trees that holds an input element-- the leaves active in Step 1(i).
4 A log n-round "playback" of Step 1, starting in parallel at the roots of the lower trees. Each of the processors ends at the original leaf in which it started Step 1. [Exercise: Same as Step 3]. Waking processors and computing prefix-sums: Step 3.

Exercise 13: (i) Show how to complete the above description into a sorting algorithm that runs in T=O(log n), W=O(n log n) and $O(n^2)$ space. (ii) Explain why your algorithm indeed achieves this complexity result.

2-3 (or B-) Trees

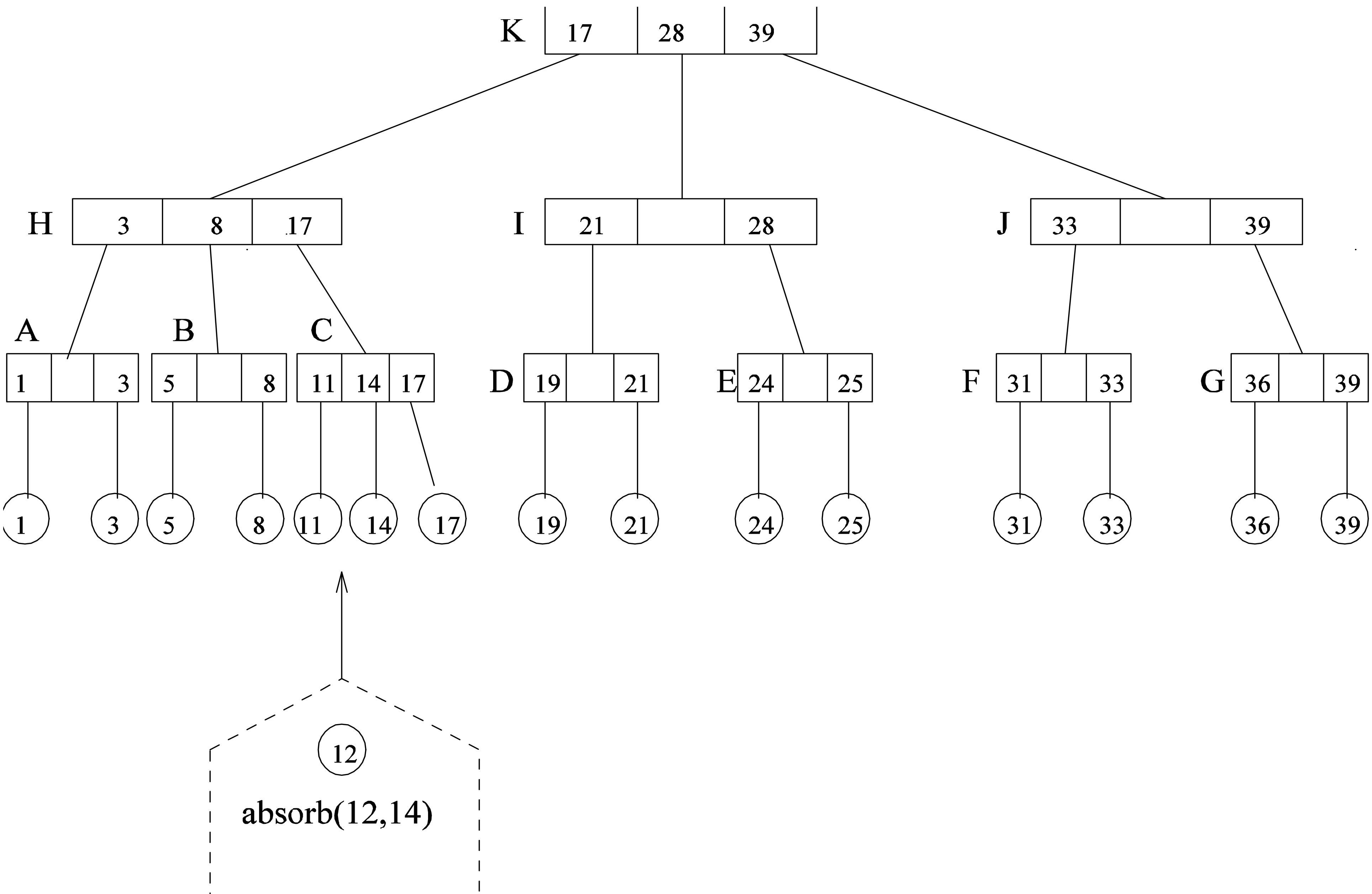
Get material from pages 42-51 in
http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.pdf

2-3 tree

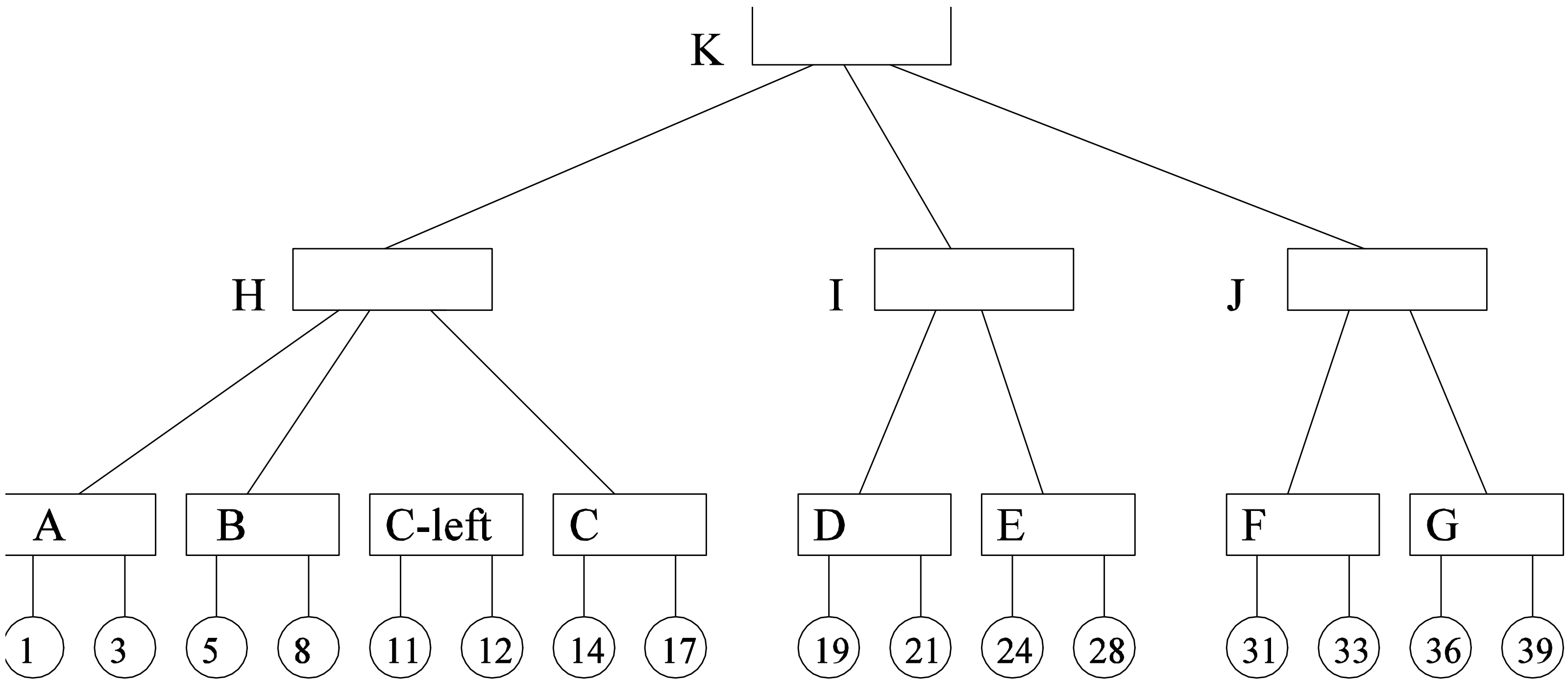


Search takes O(log n) time

Begin with insert(12)
Translate to absorb(12,14)

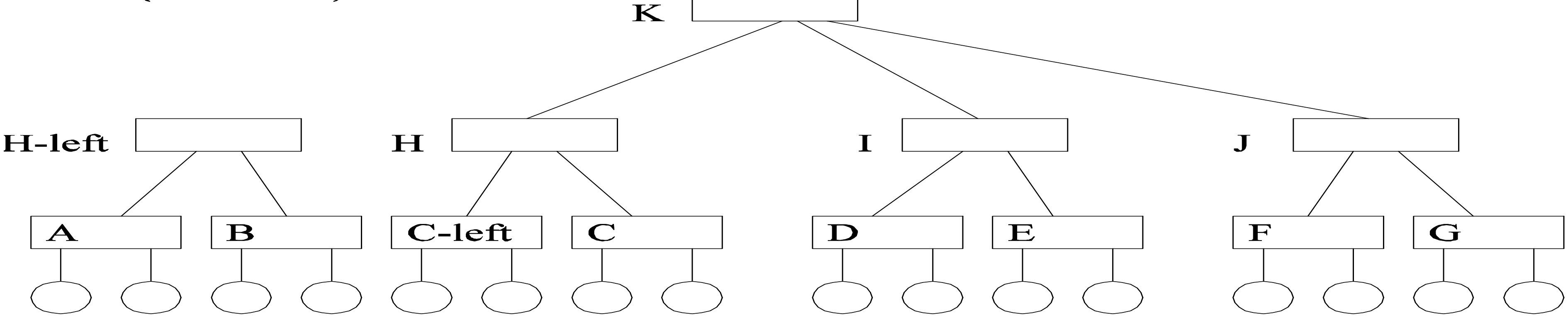


absorb(C-LEFT,C)

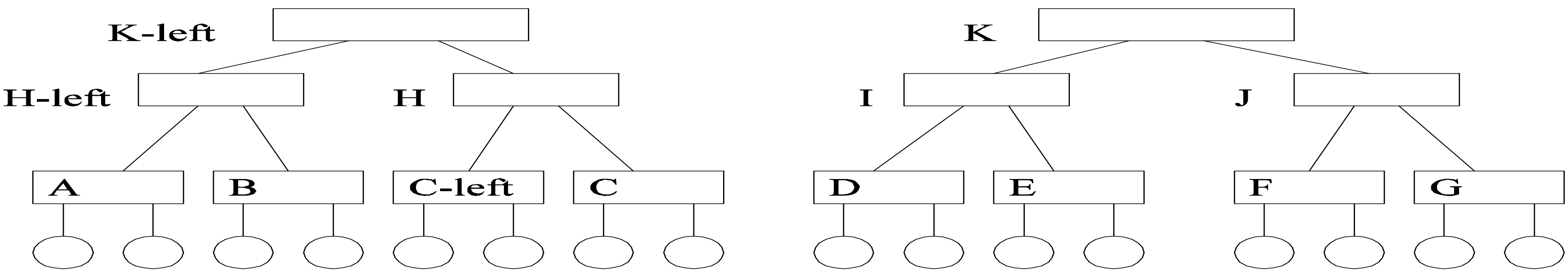


Complete insert(12)

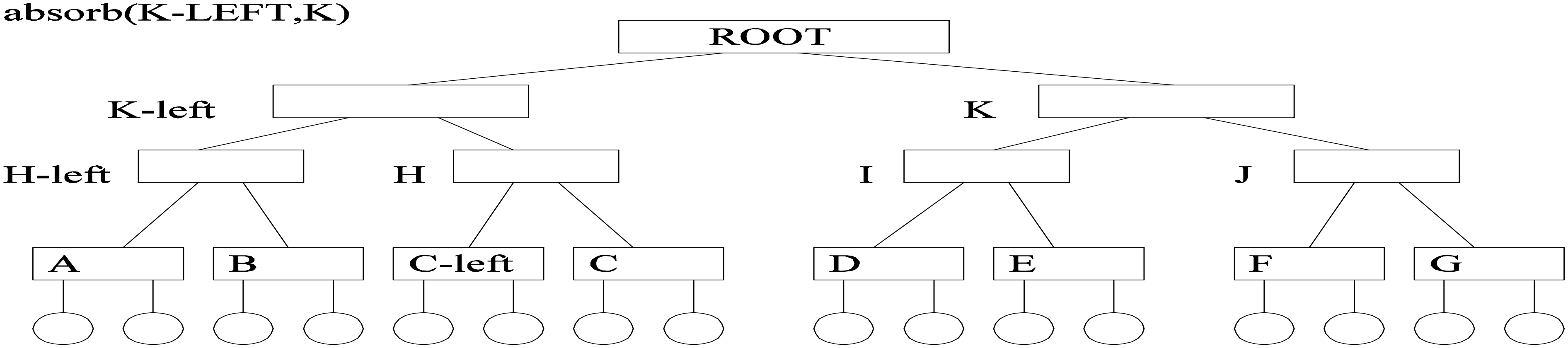
absorb(C-   LEFT)



absorb(H-LEFT,H)



absorb(K-LEFT,K)



Complexity

- Insert takes O(log n) time

Can we insert k elements in parallel?

- Yes, but if not too many need to be inserted at the same place (between 2 leaves).
- Idea let us restrict our attention to a ‘restricted parallel problem’: only one element can be inserted between 2 leaves (one largest element, or one smallest can be handled separately)
- Observation as we climb in synchronous steps the situation is not getting worse: ***no more than one new node for every old node***

Complexity

- Restricted problem with k elements to insert: k processors, O(log n) time

What if the problem is not restricted?

- Any k elements need to be inserted

<u>Step 1</u> Sort them [O(log k) time, O(k log k) work]

<u>Step 2</u> How can the restricted problem help?

Idea?

Unrestricted insert

- Insert middle element
- Recur

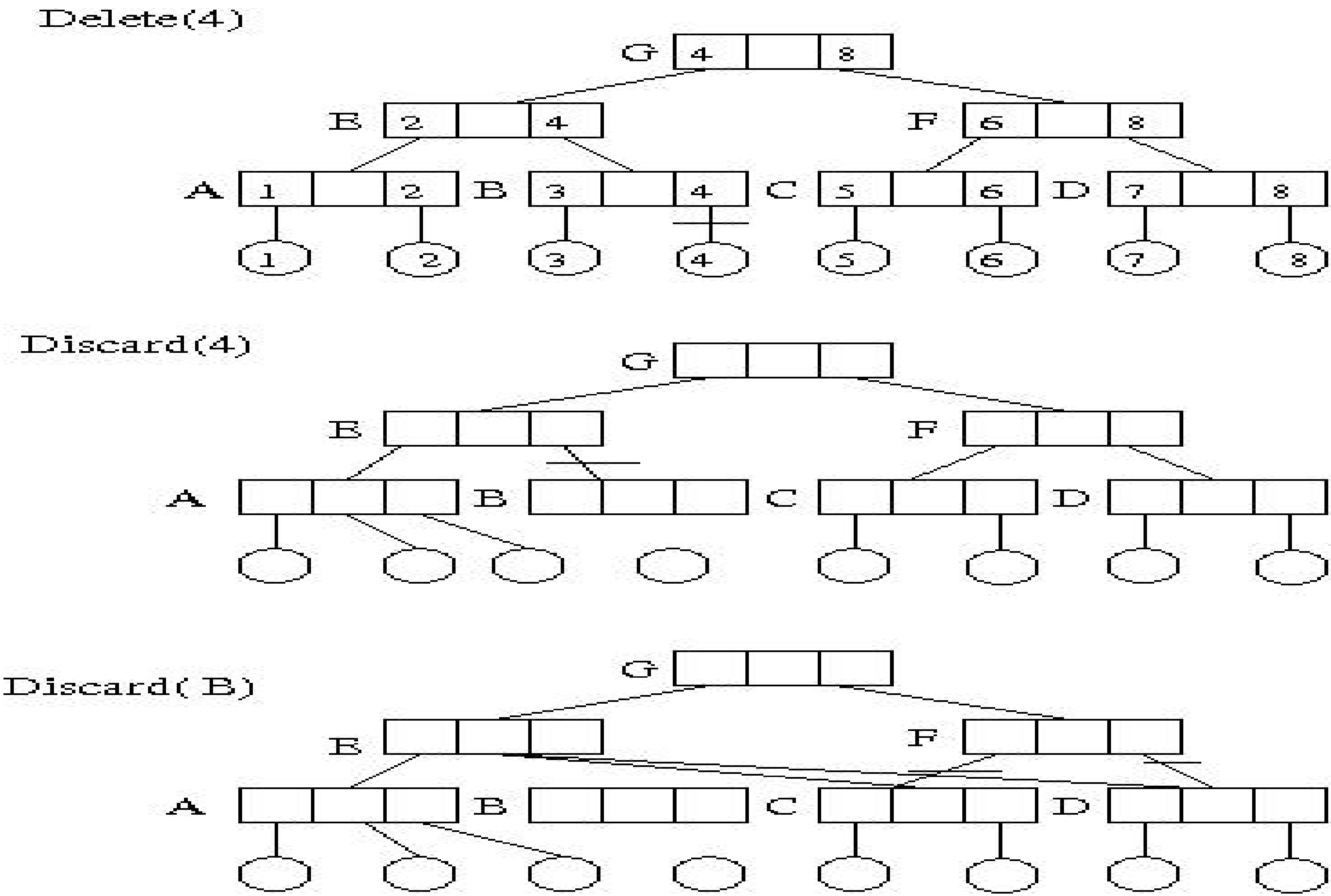
Complexity $O(\log n \log k)$ time, $O(k \log n)$ work

Can this be improved?

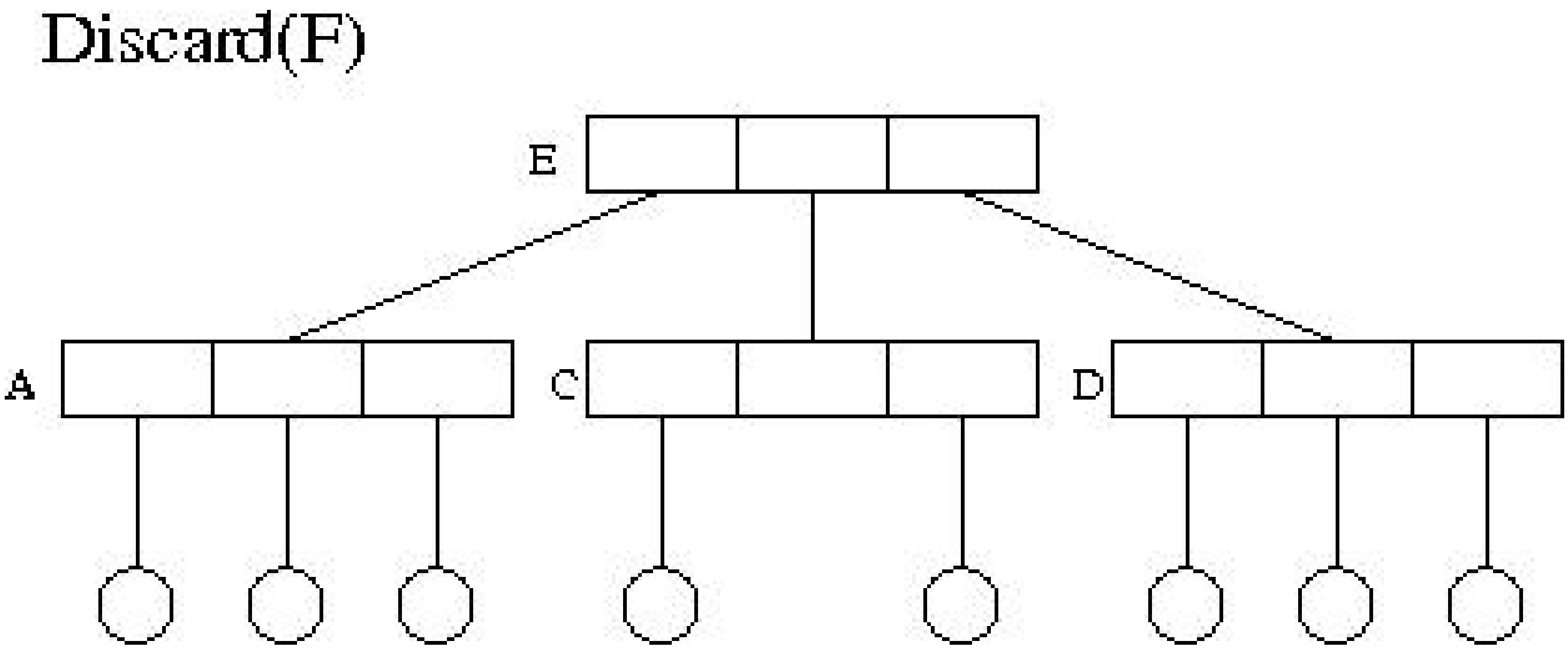
Idea?

Complexity $O(\log n + \log k) = O(\log n)$ time, $O(k \log n)$ work

Delete(4): first discard(4), second discard(B)



Finally discard(F)



Complexity

- Delete takes O(log n) time

Can we delete k elements in parallel?

Yes, but if not too many need to be deleted near one another (e.g., 2 adjacent leaves).

- <u>Idea</u> let us restrict our attention to a 'restricted parallel problem': if a leaf is deleted its predecessor leaf is not.
- <u>Observation</u> as we climb in synchronous steps the situation is not getting worse: ***no more than one deleted node for every remaining node***

Complexity

- Restricted problem with k elements to delete: k processors, O(log n) time

What if the problem is not restricted?

- Any k elements need to be deleted

Step 1 Remove all nonleaves and sort the leaves [O(log k) time, O(k log k) work]

Step 2 How can the restricted problem help?

Idea?

Unrestricted delete

- Delete every second element
- Recur

<u>Complexity</u> O(log n log k) time, O(k log n) work

Can this be improved?

Idea?

<u>Complexity</u> O(log n + log k) = O(log n) time, O(k log n) work

Mapping PRAM Algorithms onto XMT

(revisit of this slide)

(1) PRAM parallelism maps into a thread structure

(2) Assembly language threads are not-too-short (to increase locality of reference)

(3) the threads satisfy IOS

<u>How</u> (summary):

I. Use work-depth methodology [SV-82] for "thinking in parallel". The rest is skill.

II. Go through PRAM or not.

III. Produce XMTC program accounting also for:

(1) Length of sequence of round trips to memory,

(2) QRQW.

Issue: nesting of spawns.

<u>Compiler roadmap</u>:

➔ Produce performance-tuned examples➔ "teach the compiler"➔ Programmer: produce simple XMTC programs

My reading of the state of the art

Ongoing transition of mainstream computing to parallelism.

Traditional CS R&D communities good at keeping intra-community coherence.

But, problems with platforms inter-community coherence

2 Examples noted in course

1. Parallel HW not engineered for ease-of-programming and to connect to the main parallel algorithmic theory of CS

Stay tuned Can HW be out of synch with others? Will the reality check of too few programmers make difference?

2. Parallel Algorithms of yore – just theory. XMT: vertical integration/coherence require a lot of work, but are worth it

Back-up slides

But coming up with a whole theory of parallel algorithms is a complex mental problem

How to address that?

1. Address first the easiest problem(s) you don't know to solve.

   Provided a surprising structure, as illustrated next.

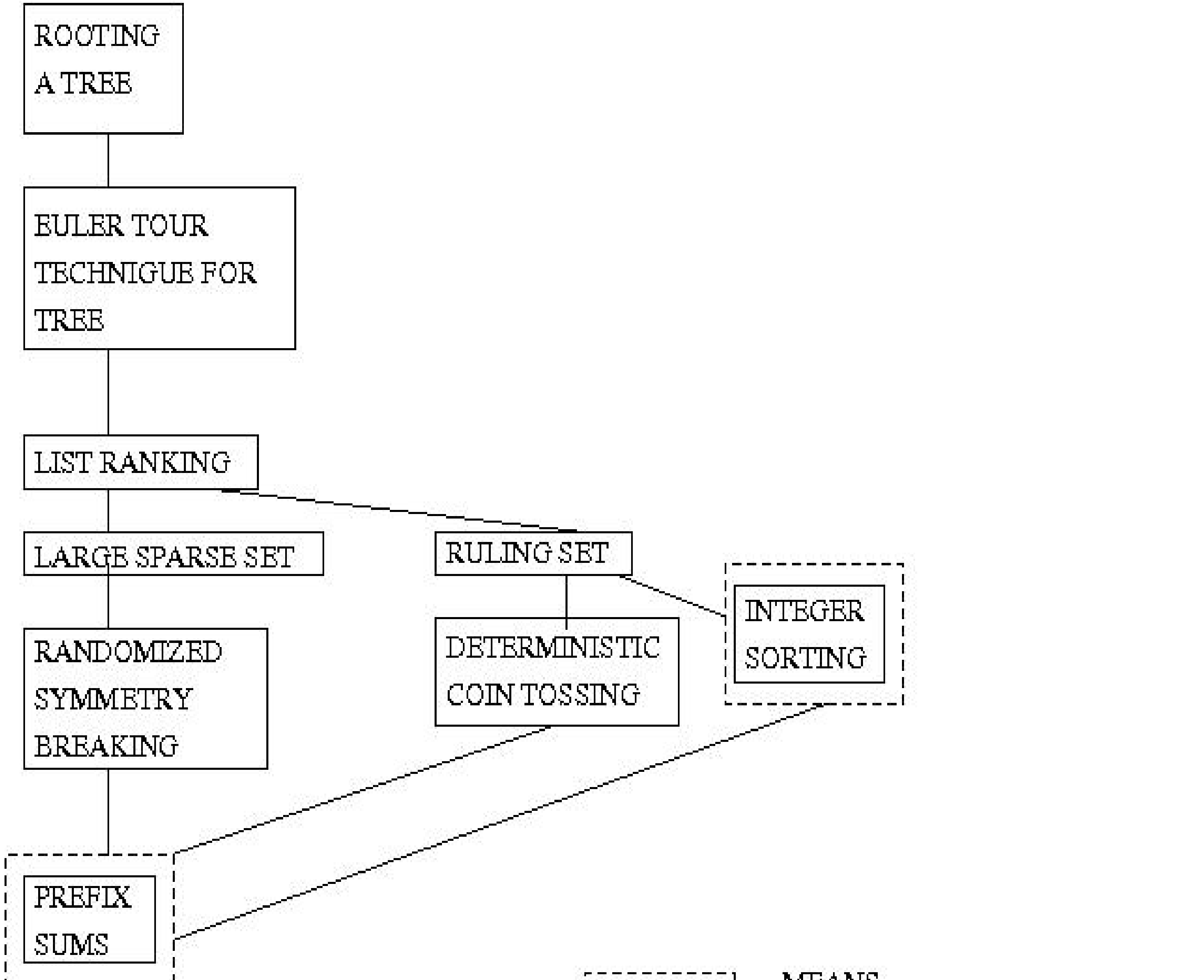
2. Do what computer scientists do best: develop/identify/fit the correct level of abstraction to each problem

   Has been a key point of this presentation

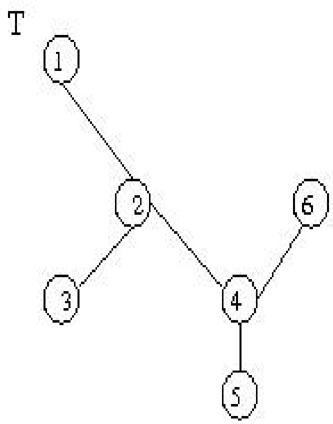
List Ranking Cluster: Euler tours; pointer jumping; randomized and deterministic symmetry breaking

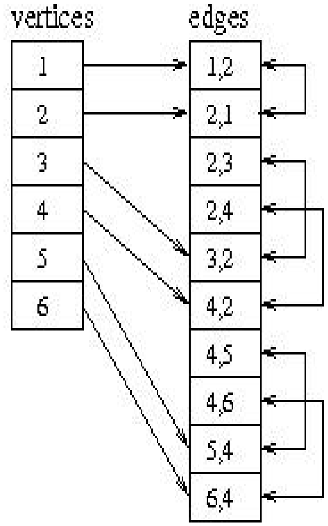
Tree rooting: a “toy problem” that will motivate the presentation.
Input T(V,E), and some specified vertex r in V . V – vertices. E – undirected edges, contains unordered pairs of vertices.
Tree rooting problem For each edge, select a direction, so that the resulting directed graph T’(V,E’) is a (directed) rooted tree whose root is vertex r; e.g., if (u, v) is in E and vertex v is closer to the root r than vertex u then u → v is in E.

Euler tour technique: constant-time optimal-work reduction of tree rooting, and other tree problems, to the list ranking problem.
This section can be viewed as an extensive top-down description of an algorithm for any of these tree problems, since the list ranking algorithms that follow are also described in a top-down manner. Top-down structures of problems and techniques from the involved to the elementary have become a “trade mark” of the theory of parallel algorithms, as reviewed in [Vis91]. Such fine structures highlight the elegance of this theory and are modest, yet noteworthy, of fine structures that exist in some classical fields of Mathematics. However, they are rather unique for Combinatorics-related theories. Figure to illustrate this structure:


ROOTING A TREE
EULER TOUR TECHINIGUE FOR TREE
LIST RANKING
LARGE SPARSE SET
RULING SET
INTEGER SORTING
RANDOMIZED SYMMETRY BREAKING
DETERMINISTIC COIN TOSSING
PREFIX SUMS
MEANS–
NOT COVERED IN THIS SECTIO

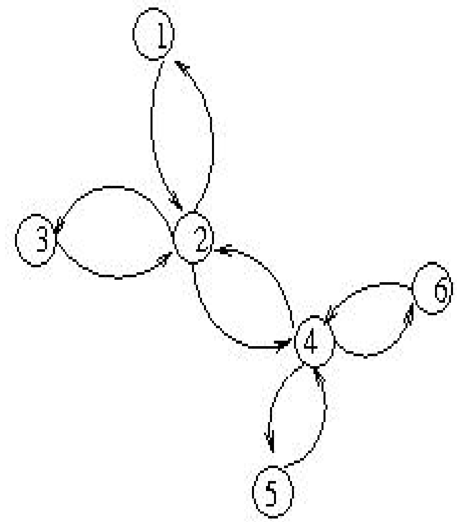
Tree T and its input representation



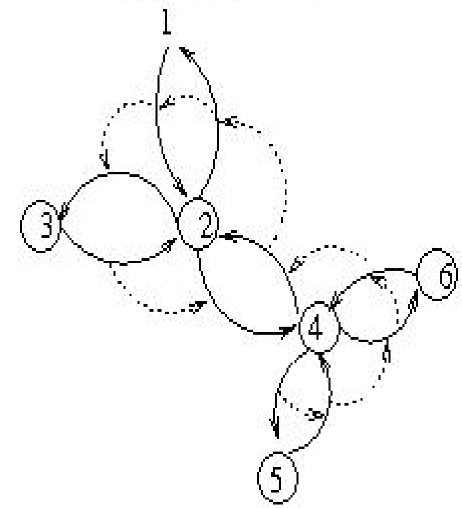


The Euler-tour technique

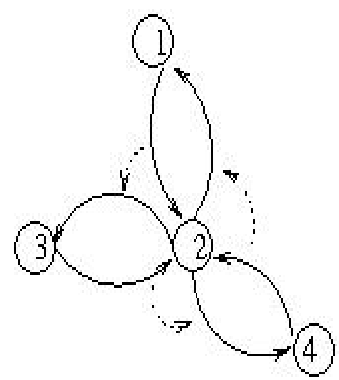
Step 1



Steps 2&3, r=4



Step 2 for vertex 2



Step 4