The PRAM Model for Parallel Computation

References

1. Selim Akl, Parallel Computation: Models and Methods, Prentice Hall, 1997, Updated online version available through website.
2. Selim Akl, The Design of Efficient Parallel Algorithms, Chapter 2 in “Handbook on Parallel and Distributed Processing” edited by J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram, Springer Verlag, 2000.
3. Selim Akl, Design & Analysis of Parallel Algorithms, Prentice Hall, 1989.
4. Henri Casanova, Arnaud Legrand, and Yves Robert, Parallel Algorithms, CRC Press, 2009.
5. Cormen, Leisterson, and Rivest, Introduction to Algorithms, 1st edition (i.e., older), 1990, McGraw Hill and MIT Press, Chapter 30 on parallel algorithms.
6. Phillip Gibbons, Asynchronous PRAM Algorithms, Ch 22 in Synthesis of Parallel Algorithms, edited by John Reif, Morgan Kaufmann Publishers, 1993.
7. Joseph JaJa, An Introduction to Parallel Algorithms, Addison Wesley, 1992.
8. Michael Quinn, Parallel Computing: Theory and Practice, McGraw Hill, 1994
9. Michael Quinn, Designing Efficient Algorithms for Parallel Computers, McGraw Hill, 1987.

Outline

- Computational Models
- Definition and Properties of the PRAM Model
- Parallel Prefix Computation
- The Array Packing Problem
- Cole’s Merge Sort for PRAM
- PRAM Convex Hull algorithm using divide & conquer
- Issues regarding implementation of PRAM model

Concept of "Model"

- An abstract description of a real world entity
- Attempts to capture the essential features while suppressing the less important details.
- Important to have a model that is both precise and as simple as possible to support theoretical studies of the entity modeled.
- If experiments or theoretical studies show the model does not capture some important aspects of the physical entity, then the model should be refined.
- Some people will not accept most abstract model of reality, but instead insist on reality.
    - Sometimes reject a model as invalid if it does not capture every tiny detail of the physical entity.

Parallel Models of Computation

- Describes a class of parallel computers
- Allows algorithms to be written for a general model rather than for a specific computer.
- Allows the advantages and disadvantages of various models to be studied and compared.
- Important, since the life-time of specific computers is quite short (e.g., 10 years).

Controversy over Parallel Models

- Some professionals (often engineers) will not accept a parallel model if
  - It does not capture every detail of reality
  - It cannot currently be built
- Engineers often insist that a model must be valid for any number of processors
  - Parallel computers with more processors than the number of atoms in the observable universe are unlikely to be built in the foreseeable future.
    - If they are ever built, the model for them is likely to be vastly different from current models today.
  - Even models that allow a billion or more processors are likely to be very different from those supporting at most a few million processors.

The PRAM Model

- PRAM is an acronym for

  Parallel Random Access Machine
- The earliest and best-known model for parallel computing.
- A natural extension of the RAM sequential model
- More algorithms designed for PRAM than any other model.

The RAM Sequential Model

- RAM is an acronym for Random Access Machine
- RAM consists of
  - A *memory* with M locations.
    - Size of M can be as large as needed.
  - A *processor* operating under the control of a sequential program which can
    - load data from memory
    - store date into memory
    - execute arithmetic & logical computations on data.
  - A *memory access unit (MAU)* that creates a path from the processor to an arbitrary memory location.

RAM Sequential Algorithm Steps

- A *READ* phase in which the processor reads datum from a memory location and copies it into a register.
- A *COMPUTE* phase in which a processor performs a basic operation on data from one or two of its registers.
- A *WRITE* phase in which the processor copies the contents of an internal register into a memory location.

PRAM Model Discussion

- Let $P_1, P_2, \ldots, P_n$ be identical processors
- Each processor is a RAM processor with a private local memory.
- The processors communicate using *m* shared (or global) memory locations, $U_1, U_2, \ldots, U_m$.
  - Allowing both local & global memory is typical in model study.
- Each P_i can read or write to each of the *m* shared memory locations.
- All processors operate synchronously (i.e. using same clock), but can execute a different sequence of instructions.
  - Some authors inaccurately restrict PRAM to simultaneously executing the same sequence of instructions (i.e., SIMD fashion)
- Each processor has a unique index called, the processor ID, which can be referenced by the processor's program.
  - Often an unstated assumption for a parallel model

PRAM Computation Step

- Each PRAM step consists of three phases, executed in the following order:
  - A *read phase* in which each processor may read a value from shared memory
  - A *compute* phase in which each processor may perform basic arithmetic/logical operations on their local data.
  - A *write phase* where each processor may write a value to shared memory.
- Note that this prevents reads and writes from being simultaneous.
- Above requires a PRAM step to be sufficiently long to allow processors to do different arithmetic/logic operations simultaneously.

SIMD Style Execution for PRAM

- Most algorithms for PRAM are of the single instruction stream multiple data (SIMD) type.
  - All PEs execute the same instruction on their own datum
  - Corresponds to each processor executing the same program synchronously.
  - PRAM does not have a concept similar to SIMDs of all active processors accessing the '*same local memory location'* at each step.

SIMD Style Execution for PRAM (cont)

- PRAM model was historically viewed by some as a shared memory SIMD.
  - Called a SM SIMD computer in [Akl 89].
  - Called a SIMD-SM by early textbook [Quinn 87].
  - PRAM executions required to be SIMD [Quinn 94]
  - PRAM executions required to be SIMD in [Akl 2000]

The Unrestricted PRAM Model

- The unrestricted definition of PRAM allows the processors to execute different instruction streams as long as the execution is synchronous.
  - Different instructions can be executed within the unit time allocated for a step
  - See JaJa, pg 13
- In the Akl Textbook, processors are allowed to operate in a “totally asychronous fashion”.
  - See page 39
  - Assumption may have been intended to agree with above, since no charge for synchronization or communications is included.

Asynchronous PRAM Models

- While there are several asynchronous models, a typical asynchronous model is described in [Gibbons 1993].
- The asychronous PRAM models do not constrain processors to operate in lock step.
  - Processors are allowed to run synchronously and then charged for any needed synchronization.
- A non-unit charge for processor communication.
  - Take longer than local operations
  - Difficult to determine a “fair charge” when message-passing is not handled in synchronous-type manner.
- Instruction types in Gibbon’s model
  - Global Read, Local operations, Global Write, Synchronization
- Asynchronous PRAM models are useful tools in study of actual cost of asynchronous computing
- The word ‘PRAM’ usually means ‘synchronous PRAM’

Some Strengths of PRAM Model

JaJa has identified several strengths designing parallel algorithms for the PRAM model.

- PRAM model removes algorithmic details concerning synchronization and communication, allowing designers to focus on obtaining maximum parallelism
- A PRAM algorithm includes an explicit understanding of the operations to be performed at each time unit and an explicit allocation of processors to jobs at each time unit.
- PRAM design paradigms have turned out to be robust and have been mapped efficiently onto many other parallel models and even network models.

PRAM Strengths (cont)

- PRAM strengths - Casanova et. al. book.
  - With the wide variety of parallel architectures, defining a precise yet general model for parallel computers seems hopeless.
    - Most daunting is modeling of data communications costs within a parallel computer.
  - A reasonable way to accomplish this is to only charge unit cost for each data move.
    - They view this as ignoring computational cost.
  - Allows minimal computational complexity of algorithms for a problem to be determined.
  - Allows a precise classification of problems, based on their computational complexity.

PRAM Memory Access Methods

- *Exclusive Read (ER):* Two or more processors can not simultaneously read the same memory location.
- *Concurrent Read (CR):* Any number of processors can read the same memory location simultaneously.
- *Exclusive Write (EW):* Two or more processors can not write to the same memory location simultaneously.
- *Concurrent Write (CW):* Any number of processors can write to the same memory location simultaneously.

Variants for Concurrent Write

- *Priority CW*: The processor with the highest priority writes its value into a memory location.
- *Common CW:* Processors writing to a common memory location succeed only if they write the same value.
- *Arbitrary CW:* When more than one value is written to the same location, any one of these values (e.g., one with lowest processor ID) is stored in memory.
- *Random CW:* One of the processors is randomly selected write its value into memory.

Concurrent Write (cont)

- *Combining CW:* The values of all the processors trying to write to a memory location are combined into a single value and stored into the memory location.
  - Some possible functions for combining numerical values are SUM, PRODUCT, MAXIMUM, MINIMUM.
  - Some possible functions for combining boolean values are AND, INCLUSIVE-OR, EXCLUSIVE-OR, etc.

ER & EW Generalizations

- Casanova et.al. mention that sometimes ER and EW are generalized to allow a bounded number of read/write accesses.
- With EW, the types of concurrent writes must also be specified, as in CW case.

Additional PRAM comments

1. PRAM encourages a focus on minimizing computation and communication steps.
   - Means & cost of implementing the communications on real machines ignored
2. PRAM is often considered as unbuildable & impractical due to difficulty of supporting parallel PRAM memory access requirements in constant time.
3. However, Selim Akl shows a complex but efficient MAU for all PRAM models (EREW, CRCW, etc) that can be supported in hardware in O(lg n) time for n PEs and O(n) memory locations. (See [2. Ch.2].
4. Akl also shows that the sequential RAM model also requires O(lg m) hardware memory access time for m memory locations.
   - Some strongly criticize PRAM communication cost assumptions but accept without question the cost in RAM memory cost assumptions.

Parallel Prefix Computation

- EREW PRAM Model is assumed for this discussion
- A *binary operation* on a set S is a function

$$\oplus : S \times S \rightarrow S.$$

- Traditionally, the element $\oplus(s_1, s_2)$ is denoted as

$$s_1 \oplus s_2.$$

- The binary operations considered for prefix computations will be assumed to be
  - *associative*: $(s_1 \oplus s_2) \oplus s_3 = s_1 \oplus (s_2 \oplus s_3)$
- Examples
  - Numbers: addition, multiplication, max, min.
  - Strings: concatenation for strings
  - Logical Operations: **and**, **or**, **xor**
- Note: $\oplus$ is not required to be commutative.

Prefix Operations

- Let s_0, s_1, ... , s_{n-1} be elements in S.
- The computation of p_0, p_1, ... ,p_{n-1} defined below is called *prefix computation*:

$$p_0 = s_0$$

$$p_1 = s_0 \oplus s_1$$

.

.

.

$$p_{n-1} = s_0 \oplus s_1 \oplus \ldots \oplus s_{n-1}$$

Prefix Computation Comments

- *Suffix* computation is similar, but proceeds from right to left.
- A binary operation is assumed to take constant time, unless stated otherwise.
- The number of steps to compute p_{n-1} has a lower bound of $\Omega(n)$ since n-1 operations are required.
- Next visual diagram of algorithm for n=8 from Akl's textbook. (See Fig. 4.1 on pg 153)
  - This algorithm is used in PRAM prefix algorithm
  - The same algorithm is used by Akl for the hypercube (Ch 2) and a sorting combinational circuit (Ch 3).

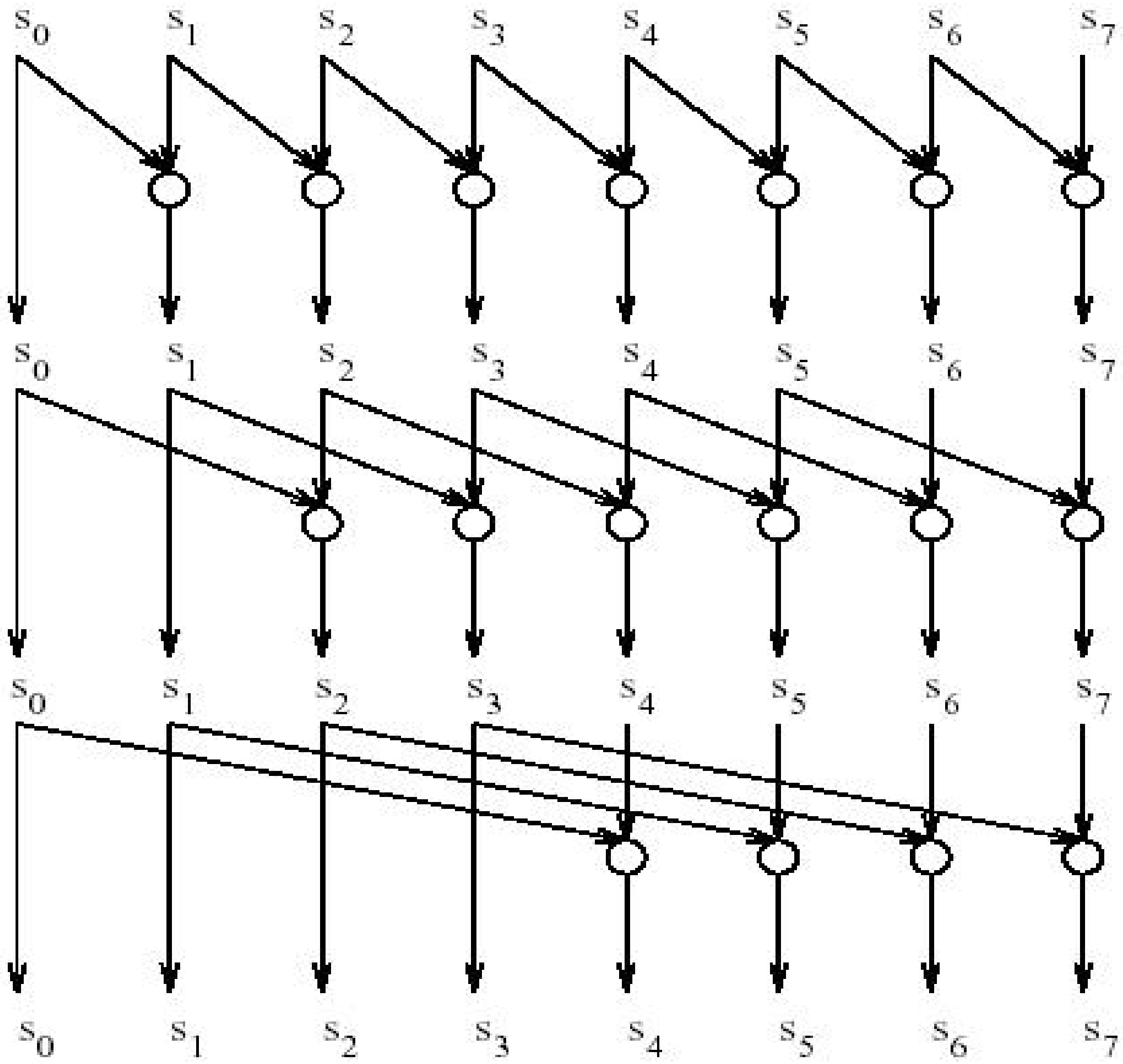


Figure 4.1: Prefix computation on the PRAM.

EREW PRAM Prefix Algorithm

- Assume PRAM has n processors, P_0, P_1 , ... , P_{n-1}, and n is a power of 2.
- Initially, P_i stores x_i in shared memory location s_i for i = 0,1, ... , n-1.
- Algorithm Steps:

for j = 0 to (lg n) -1, do

for i = 2^j to n-1 in parallel do

h = i - 2^j

$s_i = s_h \oplus s_i$

endfor

endfor

Prefix Algorithm Analysis

- Running time is $t(n) = O(\lg n)$
- Cost is $c(n) = p(n) \times t(n) = O(n \lg n)$
- Note not cost optimal, as RAM takes $O(n)$

Example for Cost Optimal Prefix

- Sequence – 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
- Use $n / \lceil \lg n \rceil$ PEs with lg(n) items each
- 0,1,2,3 4,5,6,7 8,9,10,11 12,13,14,15
- <u>STEP 1</u>: Each PE performs sequential prefix sum
- 0,1,3,6 4,9,15,22 8,17,27,38 12,25,39,54
- <u>STEP 2</u>: Perform parallel prefix sum on last nr. in PEs
- 0,1,3,6 4,9,15,28 8,17,27,66 12,25,39,120
- Now prefix value is correct for last number in each PE
- <u>STEP 3</u>: Add last number of each sequence to incorrect sums in next sequence (in parallel)
- 0,1,3,6 10,15,21,28 36,45,55,66 78,91,105,120

A Cost-Optimal EREW PRAM Prefix Algorithm

- In order to make the prefix algorithm optimal, we must reduce the cost by a factor of lg n.
- We reduce the nr of processors by a factor of lg n (and check later to confirm the running time doesn't change).
- Let $k = \lceil \lg n \rceil$ and $m = \lceil n/k \rceil$
- The input sequence $X = (x_0, x_1, ..., x_{n-1})$ is partitioned into m subsequences $Y_0, Y_1, ..., Y_{m-1}$ with k items in each subsequence.
  – While Y_{m-1} may have fewer than k items, without loss of generality (WLOG) we may assume that it has k items here.
- Then all sequences have the form,

$$Y_i = (x_{i*k}, x_{i*k+1}, \ldots, x_{i*k+k-1})$$

PRAM Prefix Computation (X, $\oplus$,S)

- **Step 1**: For $0 \leq i < m$, each processor P_i computes the prefix computation of the sequence $Y_i = (x_{i*k}, x_{i*k+1}, \ldots, x_{i*k+k-1})$ using the RAM prefix algorithm (using $\oplus$) and stores prefix results as sequence $s_{i*k}, s_{i*k+1}, \ldots, s_{i*k+k-1}$.
- **Step 2:** All m PEs execute the preceding PRAM prefix algorithm on the sequence $(s_{k-1}, s_{2k-1}, \ldots, s_{n-1})$
  - Initially P_i holds s_{i*k-1}
  - Afterwards P_i places the prefix sum $s_{k-1} \oplus \ldots \oplus s_{ik-1}$ in s_{ik-1}
- **Step 3:** Finally, all P_i for $1 \leq i \leq m-1$ adjust their partial value sums for all but the final term in their partial sum subsequence by performing the computation

$$s_{ik+j} \leftarrow s_{ik+j} \oplus s_{ik-1}$$

for $0 \leq j \leq k-2$.

Algorithm Analysis

- **Analysis:**
  - Step 1 takes $O(k) = O(\lg n)$ time.
  - Step 2 takes $O(\lg m) = O(\lg n/k)$
    $= O(\lg n - \lg k) = O(\lg n - \lg \lg n)$
    $= O(\lg n)$
  - Step 3 takes $O(k) = O(\lg n)$ time
  - The running time for this algorithm is $O(\lg n)$.
  - The cost is $O((\lg n) \times n/(\lg n)) = O(n)$
  - Cost optimal, as the sequential time is $O(n)$
- The combined pseudocode version of this algorithm is given on pg 155 of the Akl textbook

The Array Packing Problem

- Assume that we have
  - an array of *n* elements, X = $\{x_1, x_2, \ldots, x_n\}$
  - Some array elements are *marked* (or *distinguished*).
- The requirements of this problem are to
  - pack the marked elements in the front part of the array.
  - place the remaining elements in the back of the array.
- While not a requirement, it is also desirable to
  - maintain the original order between the marked elements
  - maintain the original order between the unmarked elements

A Sequential Array Packing Algorithm

- Essentially "burn the candle at both ends".
- Use two pointers q (initially 1) and r (initially n).
- Pointer q advances to the right until it hits an unmarked element.
- Next, r advances to the left until it hits a marked element.
- The elements at position q and r are switched and the process continues.
- This process terminates when $q \geq r$.
- This requires $O(n)$ time, which is optimal. (why?)

Note: This algorithm does not maintain original order between elements

EREW PRAM Array Packing Algorithm

1. Set s_i in P_i to 1 if x_i is marked and set $s_i = 0$ otherwise.
2. Perform a prefix sum on $S =(s_1, s_2 ,..., s_n)$ to obtain destination $d_i = s_i$ for each marked x_i .
3. All PEs set $m = s_n$, the total nr of marked elements.
4. P_i sets s_i to 0 if x_i is marked and otherwise sets $s_i = 1$.
5. Perform a prefix sum on S and set $d_i = s_i + m$ for each unmarked x_i .
6. Each P_i copies array element x_i into address d_i in X.

Array Packing Algorithm Analysis

- Assume n/lg(n) processors are used above.
- Optimal prefix sums requires O(lg n) time.
- The *__EREW broadcast__* of s_n needed in Step 3 takes O(lg n) time using either
  1. a binary tree in memory (See Akl text, Example 1.4.)
  2. or a prefix sum on sequence $b_1,\ldots,b_n$ with

     $b_1 = a_n$ and $b_i = 0$ for $1 < i \leq n$)
- All and other steps require constant time.
- Runs in O(lg n) time, which is cost optimal. (why?)
- Maintains original order in unmarked group as well

Notes:

- Algorithm illustrates usefulness of Prefix Sums

There many applications for Array Packing algorithm.

__Problem__: Show how a PE can broadcast a value to all other PEs in EREW in O(lg n) time using a binary tree in memory.

List Ranking Algorithm (Using Pointer Jumping)

- Problem: Given a linked list, find the location of each node in the list.
- Next algorithm uses the pointer jumping technique
- Ref: Pg 6-7 Casanova, et.al. & Pg 236-241 Akl text. In Akl's text, you should read prefix sum on pg 236-8 first.
- Assume we have a linked list L of n objects distributed in PRAM's memory
- Assume that each P_i is in charge of a node i
- Goal: Determine the distance d[i] of each object in linked list to the end, where d is defined as follows:

$$d[i] = \begin{cases} 0 & \text{if } next[i] = nil \\ d[next\ [i]] + 1 & \text{if } next[i] \neq nil \end{cases}$$

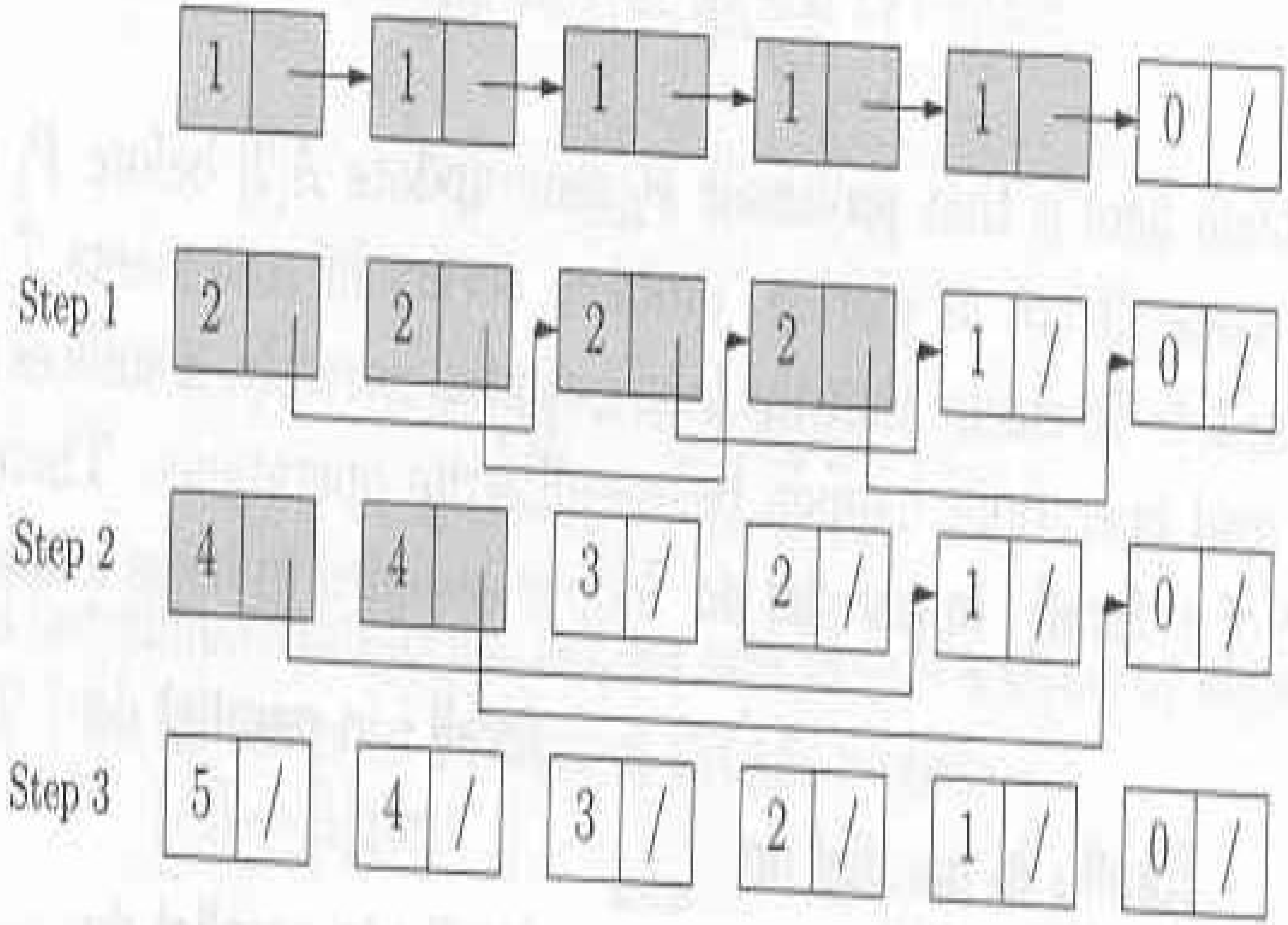
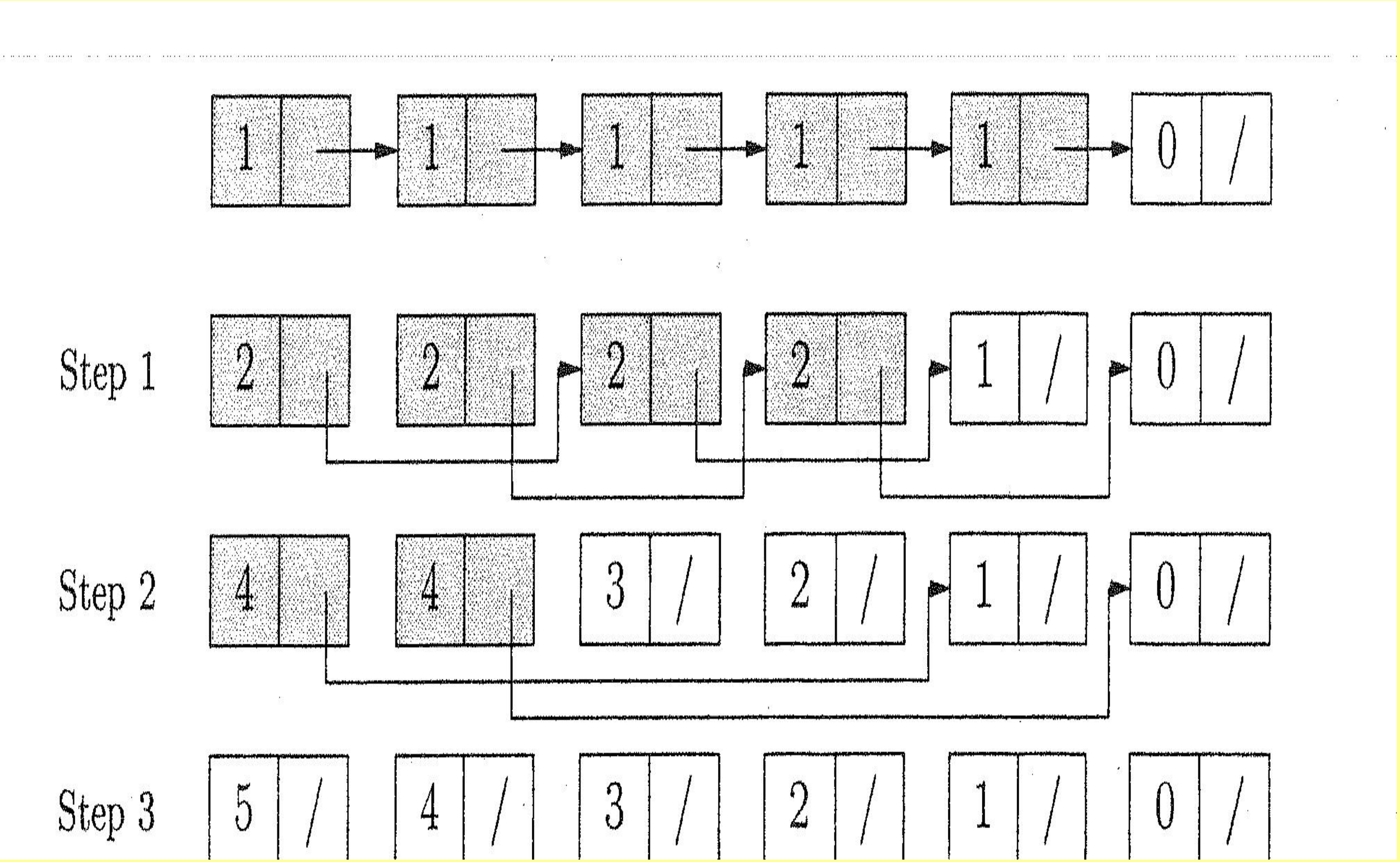


FIGURE 1.2: Typical execution of the list ranking algorithm. Gray cells indicate active values, i.e., values that are in the process of being computed.

Backup of Previous Diagram



```
RANK_COMPUTATION(L)
  forall i in parallel do                                   { Initialization }
    if next[i] = Nil then d[i] ← 0 else d[i] ← 1
  while there exists a node i such that next[i] ≠ Nil do    { Main loop }
    forall i in parallel do
      if next[i] ≠ Nil then
        d[i] ← d[i] + d[next[i]]
        next[i] ← next[next[i]]
```

ALGORITHM 1.1: List ranking algorithm.

Potential Problems?

- Consider following steps:
  7. d[i] = d[i] + d[next[i+1]]
  8. next[i] = next[next[i]]
- Casanova, et.al, pose below problem in Step7
  - P_i reads d[i+1]and uses this value to update d[i]
  - P_{i-1} must read d[i] to update d[i-1]
  - Computation fails if P_i change the value of d[i] before P_{i-1} can read it.
- This problem should not occur, as all PEs in PRAM should execute algorithm synchronously.
  - The same problem is avoided in Step 8 for the same reason

Potential Problems? (cont.)

- Does Step 7 (&Step 8) require CR PRAM?

$$d[i] = d[i] + d[next[i]]$$

  - Let j = next[i]
  - Casanova et.al. suggests that P_i and P_j may try to read d[j] concurrently, requiring a CR PRAM model
  - Again, if PEs are stepping through the computations synchronously, EREW PRAM is sufficient here
- In Step 4, PRAM must determine whether there is a node i with next[i] ≠ nil. A CWCR solution is:
  - In Step 4a, set *done* to false
  - In Step 4b, all PE write boolean value of "next[i] = nil" using CW-common write.
- A EREW solution for Step 7 is given next

Rank-Computation using EREW

- <u>Theorem</u>: The Rank-Computation algorithm only requires EREW PRAM
  - Replace Step 4 with
    - For step = 1 to $\lceil \log n \rceil$ do,
- Akl raises the question of what to do if an unknown number of processors Pi, each of which is in charge of node i (see pg 236).
  - In this case, it would be necessary to go back to the CRCW solution suggested earlier.

PRAM Model Separation

- We next consider the following two questions
  - Is CRCW strictly more powerful than CREW
  - Is CREW strictly more powerful that EREW
- We can solve each of above questions by finding a problem that the leftmost PRAM can solve faster than the rightmost PRAM

CRCW Maximum Array Value Algorithm

CRCW Compute_Maximum (A,n)

- **Algorithm requires** $O(n^2)$ PEs, $P_{i,j}$.
  1. forall $i \in \{0, 1, \ldots, n-1\}$ in parallel do
     - $P_{i,0}$ sets m[i] = True
  2. forall $i, j \in \{0, 1, \ldots, n-1\}^2$, $i \neq j$, in parallel do
     - [if A[i] < A[j] then $P_{i,j}$ sets m[i] = False
  3. forall $i \in \{0, 1, \ldots, n-1\}$ in parallel do
     - If m[i] = True, then $P_{i,0}$ sets *max* = A[i]
  4. Return *max*
- Note that on n PEs do EW in steps 1 and 3
- The write in Step 2 can be a "common CW"
- Cost is $O(1) \times O(n^2)$ which is $O(n^2)$

CRCW More Powerful Than CREW

- The previous algorithm establishes that CRCW can calculate the maximum of an array in O(1) time
- Using CREW, only two values can be merged into a single value by one PE in a single step.
  - Therefore the number of values that need to be merged can be halved at each step.
  - So the fastest possible time for CREW is $\Omega(\log n)$

CREW More Powerful Than EREW

- Determine if a given element e belongs to a set $\{e_1, e_2, \ldots, e_n\}$ of n distinct elements
- CREW can solve this in O(1) using n PEs
  - One PE initializes a variable *result* to false
  - All PEs compare e to one e_i.
  - If any PE finds a match, it writes "true" to *result.*
- On EREW, it takes $\Omega(\log n)$ steps to broadcast the value of e to all PEs.
  - The number of PEs with the value of e can be doubled at each step.

Simulating CRCW with EREW

<u>**Theorem**</u>: An EREW PRAM with p PEs can simulate a **common** CRCW PRAM with p PEs in O(log p) steps using O(p) extra memory.

- See Pg 14 of Casanova, et. al.
- The only additional capabilities of CRCW that EREW PRAM has to simulate are CR and CW.
- Consider a CW first, and initially assume all PE participate.
- EREW PRAM simulates this CW by creating a $p \times 2$ array A with length p

Simulating Common CRCW with EREW

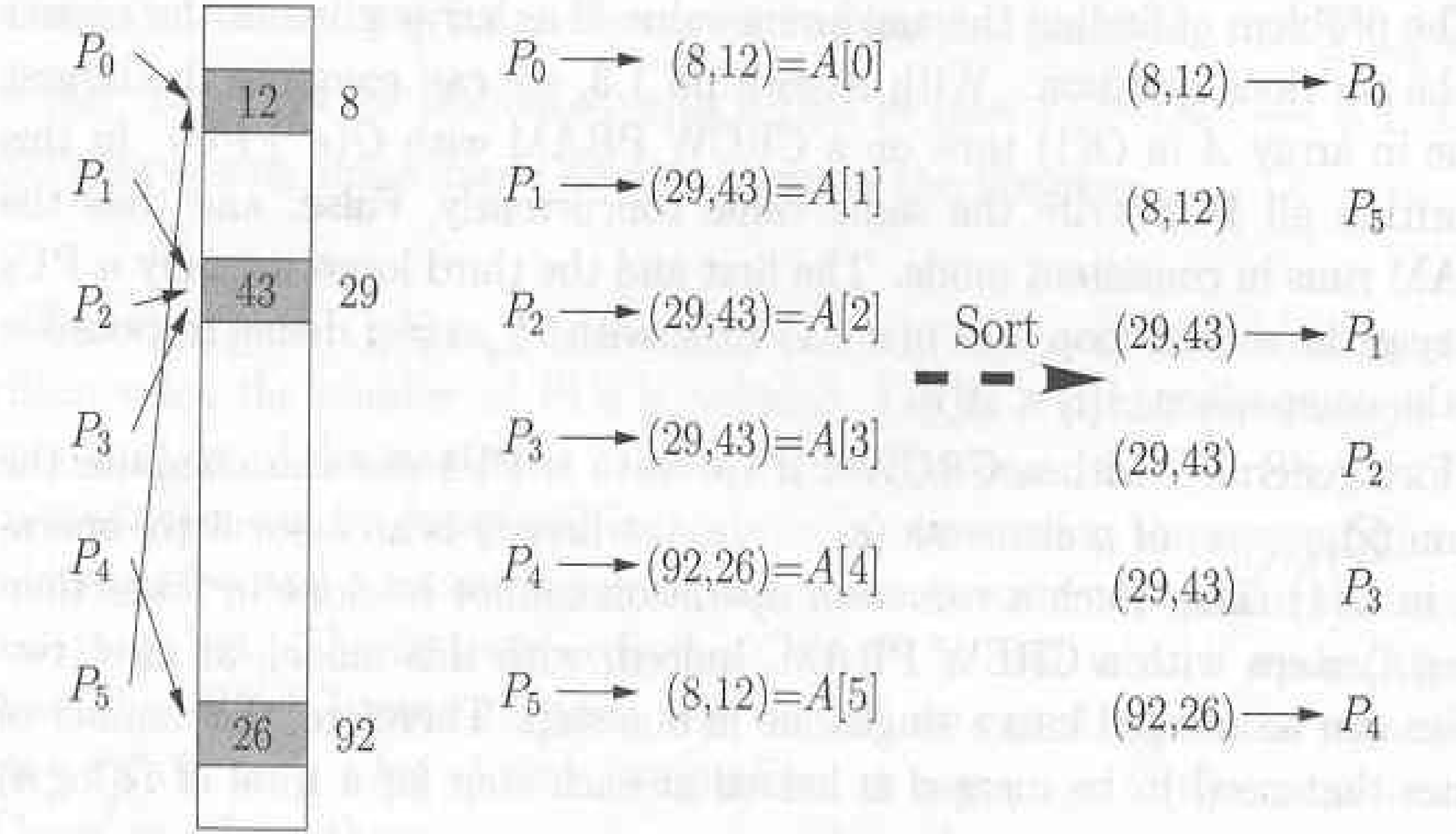
- When a CW write is simulated, PRAM EREW PE $_j$ writes
  - The memory cell address wishes to write to in A(j,0)
  - The value it wishes into memory in A(j,1).
  - If any PE j does not participate in CW, it will write -1 to A(j,0).
- Next, sort A by its first column. This brings all of the CW to same location together.
- If memory location in A(0,1) is not -1, then PE 0 writes the data value in A(0,1) to memory location value stored in A(0,1).

PRAM Simulations (cont)

- All PEs j for j>0 read memory address in A(j,0) and A(j-1,0)
  - If memory location in A(j,0) is -1, PE j does not write.
  - Also, if the two memory addresses are the same, PE j does not write to memory.
  - Otherwise, PE j writes data value in A(j,1) to memory location in A(j,0).
- Cole's algorithm that EREW can sort n items in log(n) time is needed to complete this proof. It is discussed next in Casanova et.al. for CREW.

Problem:

- This proof is invalid for CRCW versions stronger than common CRCW, such as combining.



(a) CRCW write access

(b) EREW simulation

FIGURE 1.5: Simulation of concurrent writes with exclusive writes.

Cole’s Merge Sort for PRAM

- Cole’s Merge Sort runs on EREW PRAM in O(lg n) using O(n) processors, so it is cost optimal.
  - The Cole sort is significantly more efficient than most other PRAM sorts.
- Akl calls this sort “PRAM SORT” in book & chptr (pg 54)
  - A high level presentation of EREW version is given in Ch. 4 of Akl’s online text and also in his book chapter
- A complete presentation for **CREW** PRAM is in JaJa.
  - JaJa states that the algorithm he presents can be modified to run on EREW, but that the details are non-trivial.
- Currently, this sort is the best-known PRAM sort & is usually the one cited when a cost-optimal PRAM sort using O(n) PEs is needed.

References for Cole's EREW Sort

Two references are listed below.

- Richard Cole, Parallel Merge Sort, SIAM Journal on Computing, Vol. 17, 1988, pp. 770-785.
- Richard Cole, Parallel Merge Sort, Book-chapter in “Synthesis of Parallel Algorithms”, Edited by John Reif, Morgan Kaufmann, 1993, pg.453-496

Comments on Sorting

- A CREW PRAM algorithm that runs in
  O((lg n) lg lg n) time
  and uses O(n) processors which is much simpler is given in JaJa's book (pg 158-160).
  - This algorithm is shown to be work optimal.
- Also, JaJa gives an O(lg n) time randomized sort for CREW PRAM on pages 465-473.
  - With high probability, this algorithm terminates in O(lg n) time and requires O(n lg n) operations
    - i.e., with high-probability, this algorithm is work-optimal.
- Sorting is often called the "queen of the algorithms":
    - A speedup in the best-known sort for a parallel model usually results in a similar speedup other algorithms that use sorting.

Cole's CREW Sort

- Given in 1986 by Cole [43 in Casanova]
- Also, sort given for EREW in same paper, but is even more difficult.
- The general idea of algorithm technique follows:
  - Based on classical merge sort, represented as a binary tree.
  - All merging steps at a given level of the tree must be done in parallel
  - At each level, two sequences each of arbitrary size must be merged in O(1) time.
    - Partial information from previous merges is used to merge in constant time, using a very clever technique.
  - Since there are log n levels, this yields a log n running time.

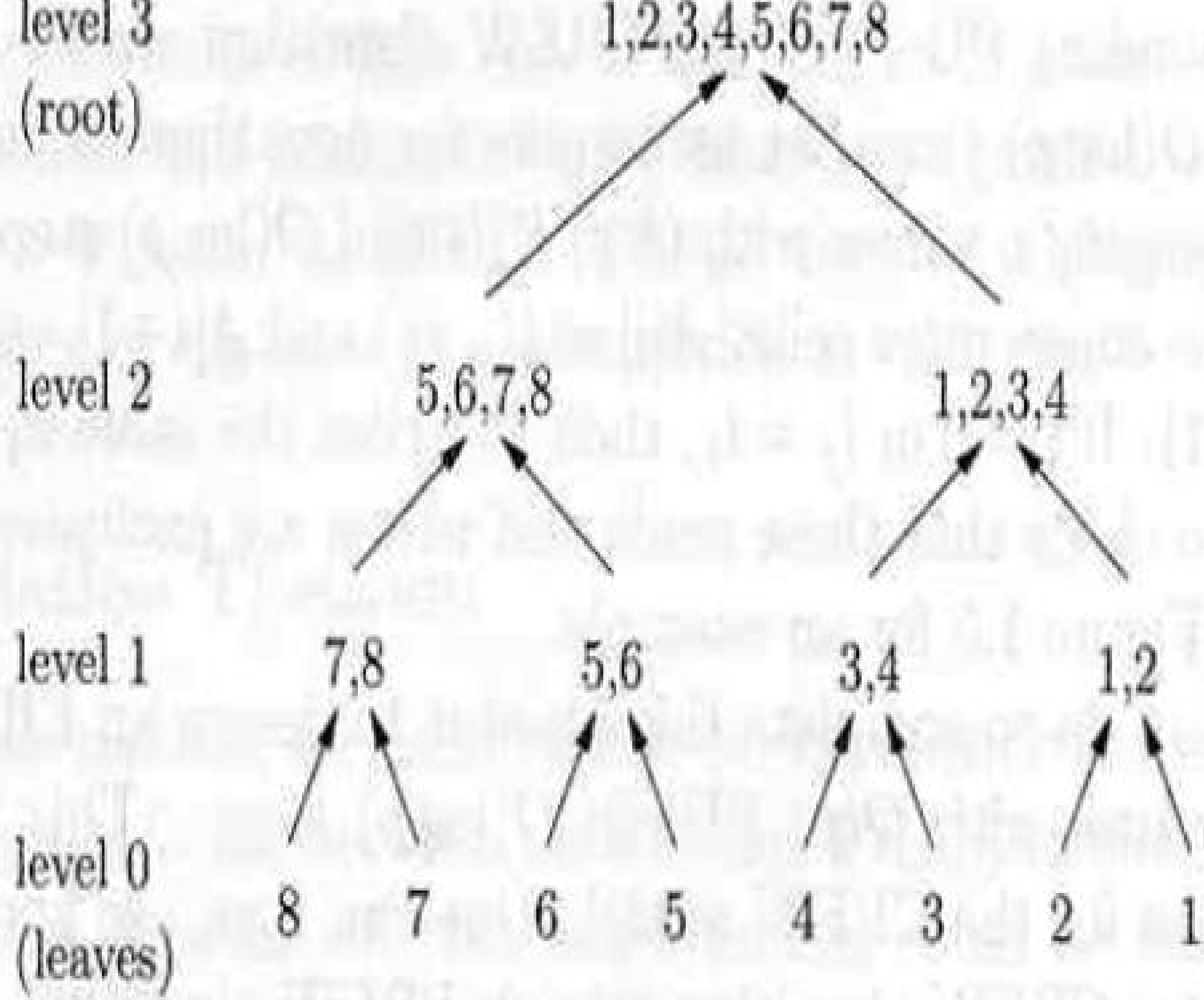


FIGURE 1.6: Example binary tree for Cole's parallel merge sort.

Cole's EREW Sort (cont)

- Defn: A sequence L is called a *good sampler* (GS) of sequence J if, for any k≥1, there are at most 2k+1 elements of J between k+1 consecutive elements of {-∞} ∪ L ∪{+ ∞}
  - Intuitively, elements of L are almost uniformly distributed among elements of J.

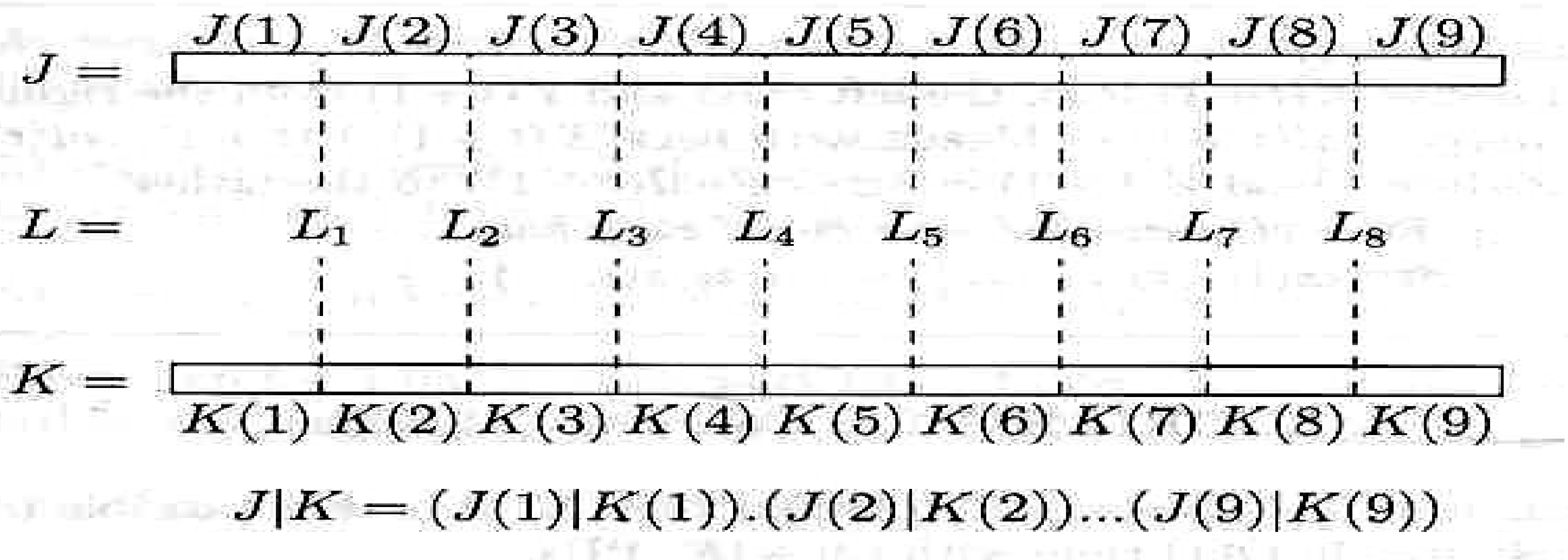


FIGURE 1.7: Merging J and K with the help of L.

Key is to use sorting tree of Fig 1.6 in a pipelined fashion. A good sampler sequence is built at each level for next level.

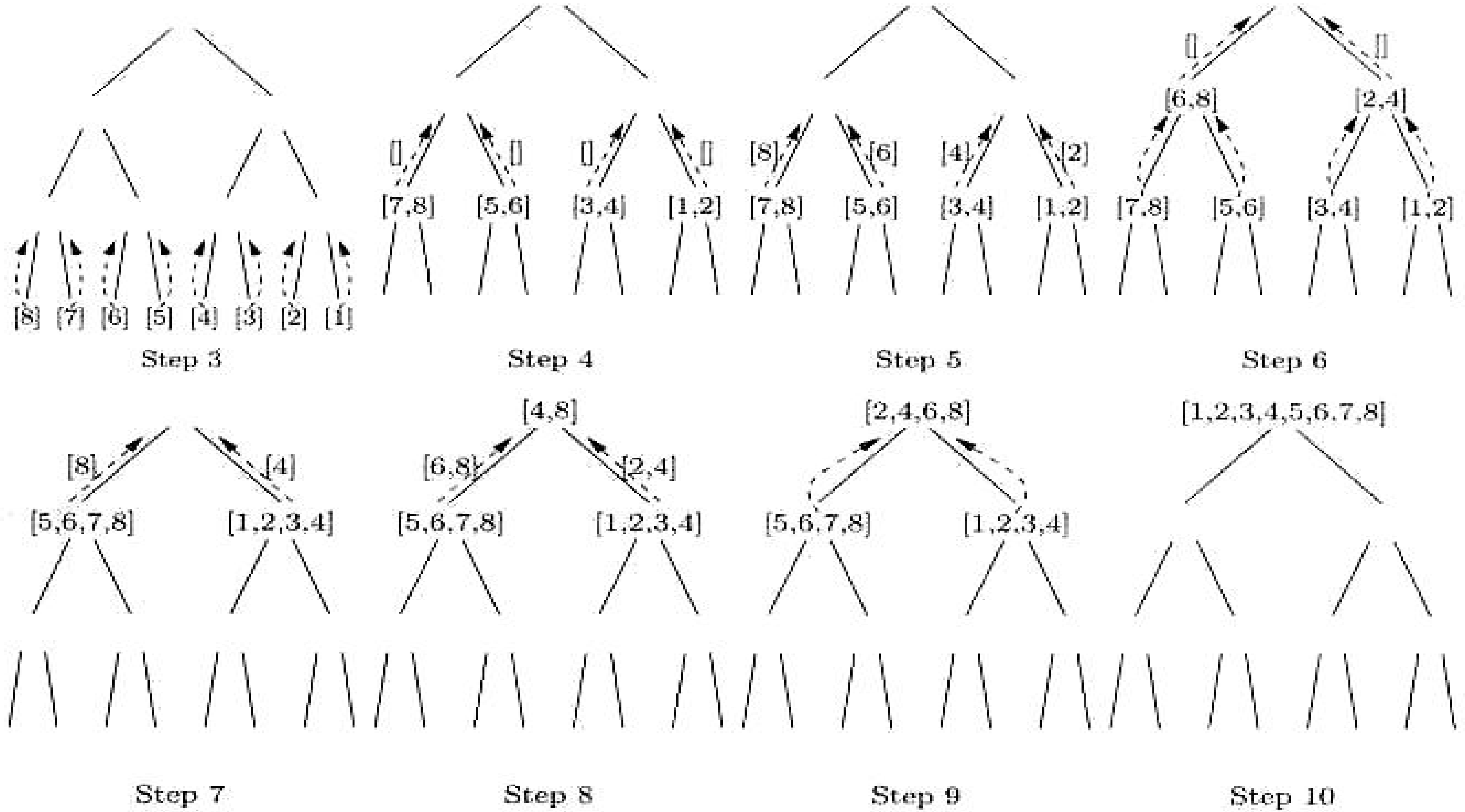


FIGURE 1.8: Sorting an array of size 8 with Cole's parallel merge sort.

Divide & Conquer PRAM Algorithms
(Reference: Akl, Chapter 5)

- Three Fundamental Operations
  - *Divide* is the partitioning process
  - *Conquer* is the process of solving the base problem (without further division)
  - *Combine* is the process of combining the solutions to the subproblems
- Merge Sort Example
  - *Divide* repeatedly partitions the sequence into halves.
  - *Conquer* sorts the base set of one element
  - *Combine* does most of the work. It repeatedly merges two sorted halves
- Quicksort Example
  - The *divide* stage does most of the work.

An Optimal CRCW PRAM Convex Hull Algorithm

- Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of points in the Euclidean plane (i.e., E^2-space).
- The convex hull of Q is denoted by $CH(Q)$ and is the smallest convex polygon containing Q.
  - It is specified by listing convex hull corner points (which are from Q) in order (e.g., clockwise order).
- Usual Computational Geometry Assumptions:
  - No three points lie on the same straight line.
  - No two points have the same x or y coordinate.
  - There are at least 4 points, as $CH(Q) = Q$ for $n \leq 3$.

PRAM CONVEX HULL(n,Q, $CH(Q)$)

1. Sort the points of Q by x-coordinate.
2. Partition Q into $k = \sqrt{n}$ subsets $Q_1, Q_2, . . . , Q_k$ of k points each such that a vertical line can separate Q_i from Q_j
   – Also, if $i < j$, then Q_i is left of Q_j.
3. For i = 1 to k , compute the convex hulls of Q_i in parallel, as follows:
   – if $|Q_i| \leq 3$, then CH(Qi) = Q_i
   – else (using k=$\sqrt{n}$ PEs) call PRAM CONVEX HULL(k, Qi, CH(Q_i))
4. Merge the convex hulls in $\{CH(Q1), CH(Q2), . . . , CH(Q_k)\}$ into a convex hull for Q.

Merging $\sqrt{n}$ Convex Hulls

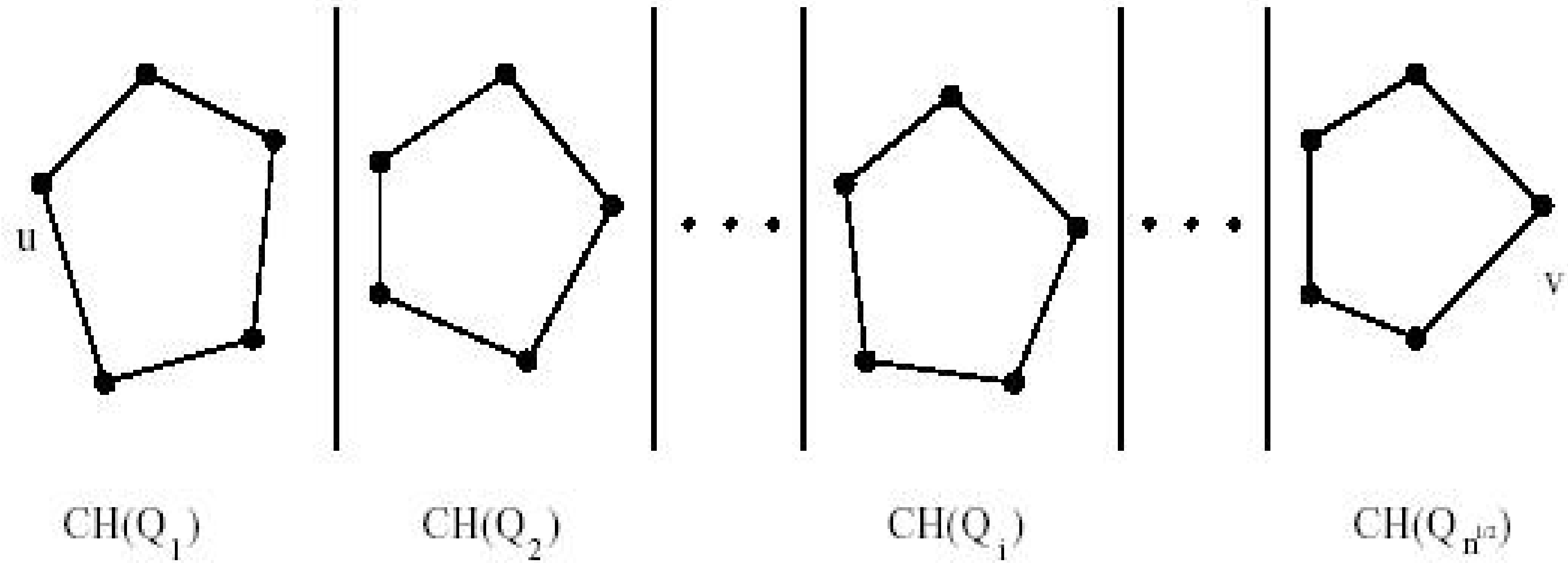


Figure 5.11: Convex polygons to be merged in the computation of $CH(Q)$.

Details for Last Step of Algorithm

- The last step is somewhat tedious.
- The upper hull is found first. Then, the lower hull is found next using the same method.
  - Only finding the upper hull is described here
  - Upper & lower convex hull points merged into ordered set
- Each $CH(Q_i)$ has $\sqrt{n}$ PEs assigned to it.
- The PEs assigned to $CH(Q_i)$ (in parallel) compute the upper tangent from $CH(Q_i)$ to another $CH(Q_j)$.
  - A total of n-1 tangents are computed for each $CH(Q_i)$
  - Details for computing the upper tangents will be discussed separately

The Upper and Lower Hull

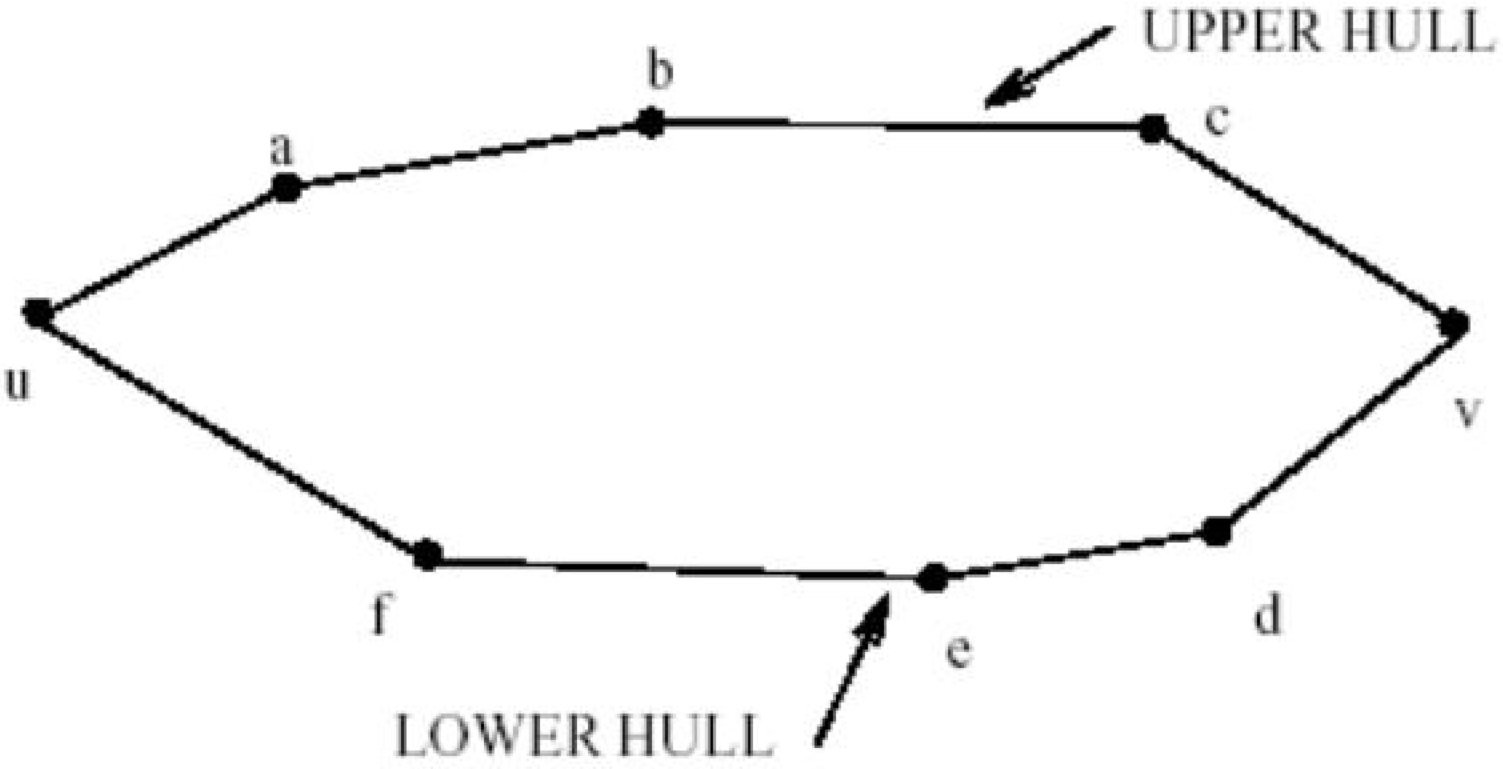


Figure 5.12: The upper and lower hulls of $CH(Q)$.

Last Step of Algorithm (cont)

- Among the tangent lines to $CH(Q_i)$ and polygons to the left of $CH(Q_i)$, let L_i be the one with the smallest slope.
  - Use a MIN CW to a shared memory location
- Among the tangent lines to $CH(Q_i)$ and polygons to the right, let R_i be the one with the largest slope.
  - Use a MAX CW to a shared memory location
- If the angle between L_i and R_i is less than 180 degrees, no point of $CH(Q_i)$ is in $CH(Q)$.
  - See Figure 5.13 on next slide (from Akl's Online text)
- Otherwise, all points in $CH(Q)$ between where L_i touches $CH(Q_i)$ and where R_i touches $CH(Q_i)$ are in $CH(Q)$.
- Array Packing is used to combine all convex hull points of $CH(Q)$ after they are identified.

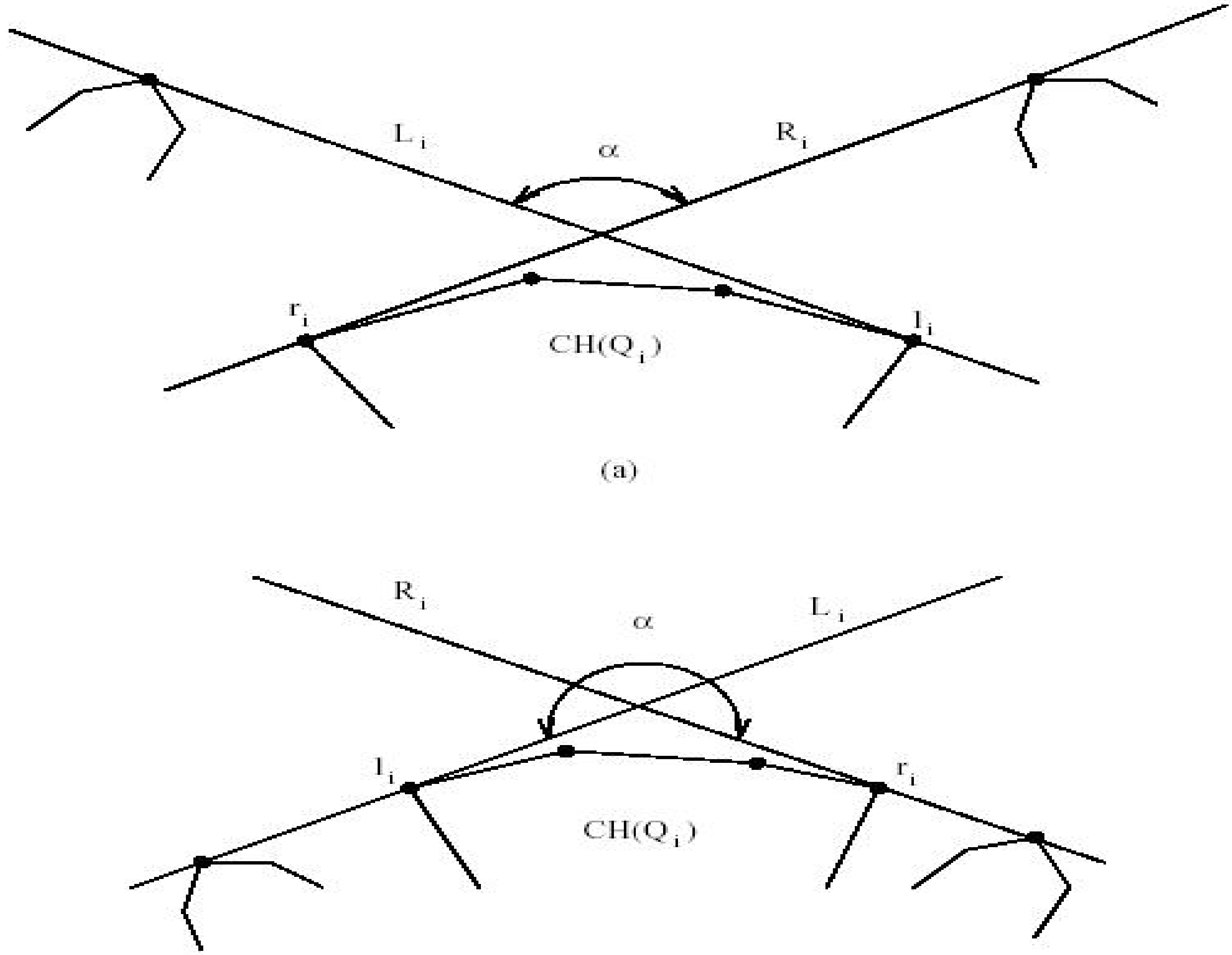


Figure 5.13: Identifying points on the upper hull of $CH(Q)$: (a) α smaller than 180 degrees; (b) α larger than 180 degrees.

Algorithm for Upper Tangents

- Requires finding a straight line segment tangent to $CH(Q_i)$ and $CH(Q_j)$, as given by line $\overline{sw}$ using a binary search technique
  - See Fig 5.14(a) on next slide
- Let s be the mid-point of the ordered sequence of corner points in $CH(Q_i)$.
- Similarly, let w be the mid-point of the ordered sequence of convex hull points in $CH(Q_i)$.
- Two cases arise:
  - $\overline{sw}$ is the upper tangent of $CH(Q_i)$ and we are done.
  - Otherwise, on average one-half of the remaining corner points of $CH(Q_i)$ and/or $CH(Q_j)$ can be removed from consideration.
- Preceding process is now repeated with the mid-points of two remaining sequences.

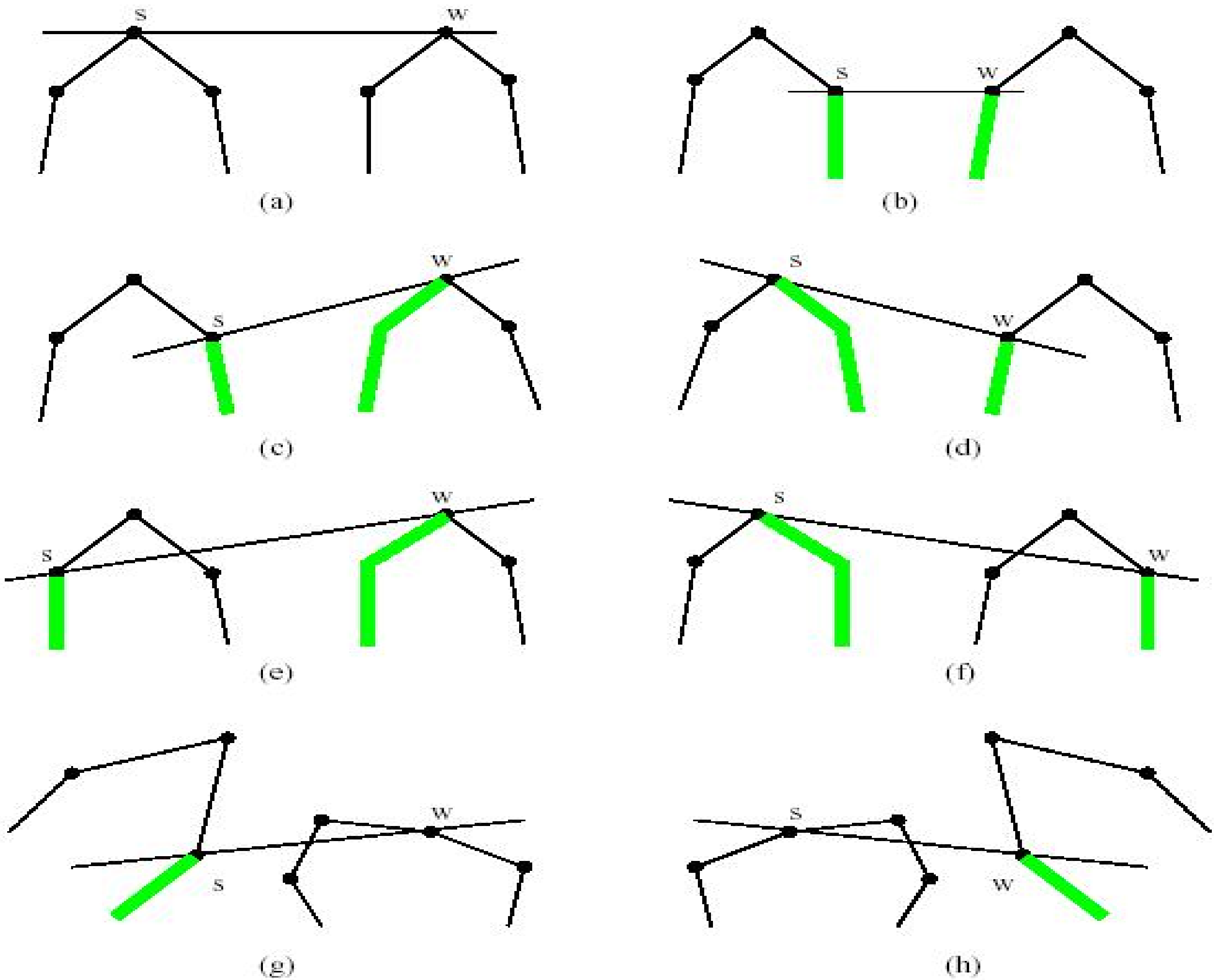


Figure 5.14: Computing the upper tangent: (a)–(h) Eight simple cases.

PRAM Convex Hull Complexity Analysis

- Step 1: The sort takes $O(lg\ n)$ time.
- Step 2: Partition of Q into subsets takes $O(1)$ time.
  - Here, Q_i consist of points q_k where $k = (i-1)\sqrt{n} + r$ for $1 \leq i \leq \sqrt{n}$
- Step 3: The recursive calculations of $CH(Q_i)$ for $1 \leq i \leq \sqrt{n}$ in parallel takes $t(\sqrt{n})$ time (using $\sqrt{n}$ PEs for each Q_i).
- Step 4: The big steps here require $O(\lg n)$ and are
  - Finding the upper tangent from $CH(Q_i)$ to $CH(Q_j)$ for each i, j pair takes $O(\lg \sqrt{n}) = O(\lg n)$
  - Array packing used to form the ordered sequence of upper convex hull points for Q.
- Above steps find the upper convex hull. The lower convex hull is found similarly.
  - Upper & lower hulls can be merged in O(1) time to be an (counter)/clockwise ordered set of hull points.

Complexity Analysis (Cont)

- Cost for Step 3: Solving the recurrence relation

$$t(n) = t(\sqrt{n}) + \beta \lg n$$

  yields

$$t(n) = O(\lg n)$$

- Running time for PRAM Convex Hull is $O(\lg n)$ since this is maximum cost for each step.
- Then the cost for PRAM Convex Hull is

$$C(n) = O(n \lg n).$$

Optimality of PRAM Convex Hull

Theorem: A lower bound for the number of sequential steps required to find the convex hull of a set of planar points is $\Omega(n \lg n)$

- Let $X = \{x_1, x_2, \ldots, x_n\}$ be any sequence of real numbers.
- Consider the set of planar points

  $$Q = \{ (x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2) .$$
- All points of Q lie on the curve $y = x^2$, so all points of Q are in $CH(Q)$.
- Apply *any* convex hull algorithm to Q.

Optimality of PRAM Convex Hull (cont)

- The convex hull produced is ‘sorted’ by the first coordinate, assuming the following rotation.
  - A sequence may require an around-the-end rotation of items to get the least x-coordinate to occur first.
  - Identifying smallest term and rotating A takes only linear (or O(n)) time.
- The process of sorting has a lower bound of *n* lg *n* basic steps.
- All of the above steps used to sort this sequence with the exception of finding the convex hull require only linear time.
- Consequently, a worst case lower bound for computing the convex hull is $\Omega(n \lg n)$ steps.