Parallel Processing

Web Page: www.cs.clemson.edu/~srimani/8380_F22

Instructor: Pradip Srimani, srimani@clemson.edu

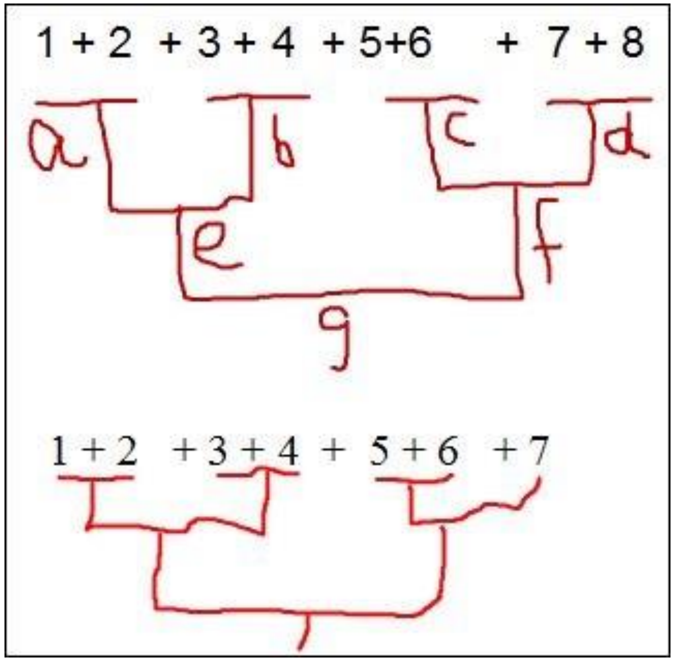
Introduction: Overview, Speed-up, Efficiency, Cost vs. Work, PRAM Model, Simple Addition, Prefix Sum

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Simple Problems

- Add n integers assuming you have multiple adders.
- Add 2 n-bit integers assuming you have multiple <u>1-bit binary full adders</u>.
- Compute x^n where n is an integer assuming you have multiple multipliers.

Add n Numbers



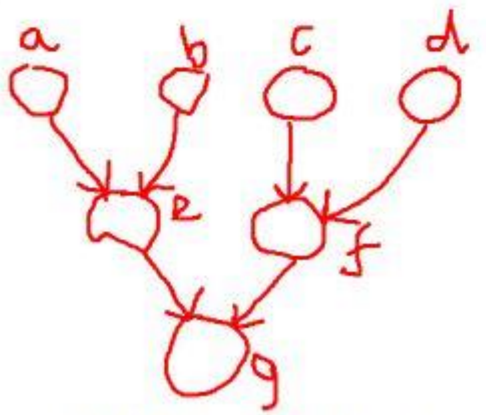
P1	1+2=3	3+7=10	18+11=28
P2	3+4=7	11+7=18	X
P3	5+6=11	X	X

Such a representation is called **Gantt Chart** (to show a specific scheduling of tasks on PEs); it's just an arbitrary example

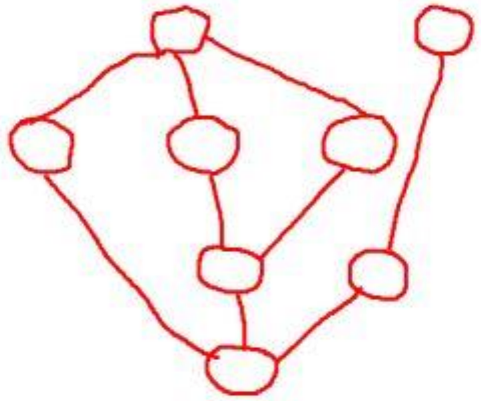
- Look at the (inverted complete) binary tree. The minimum time to add n integers is $\log_2 n$ given enough processors (PEs) – so called infinite parallelism model.
- Observe the tasks a, b, c, and d do not depend on any other subtasks to complete.
- Tasks e and f must wait (a, b) and (c, d) and so on – think in terms of a ***dependency*** *relationship between tasks*.
- Do you see 7 numbers can be added in 3-time units using 3 PEs but adding 8 integers in 3 time units you need at least 4 PEs? Write rigorous arguments why!!
- Practice with different values of n.
- Try with parenthesized arithmetic expressions.

Dependency Graph & Scheduling Informally

- While there are two types of dependency graphs in the context of parallel computing (task dependency and data dependency), we will consider the task dependency graphs for now. It's a directed graph where nodes represent tasks and directed edges represent task dependency relation, i.e., a → b means task b depends on completion of task a.
- Any valid dependency graph at least one node with zero in degree (source) and at least one node with zero out degree (sink).
- Dependency graphs, by definition, are acyclic. **Why?**
- Observe, given a task graph, the minimum feasible execution time is the length of the longest path from a source to a sink. **Why**?
- Experiment with arbitrary dependency graphs for different number of PEs; compute **Speedup** and **Efficiency**.
- Given an arbitrary graph, can you design an algorithm to test if it is a valid dependency graph?
- Do you see any similarity with topological sorting?



Addition of 8 integers.



Assume edges are top down

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

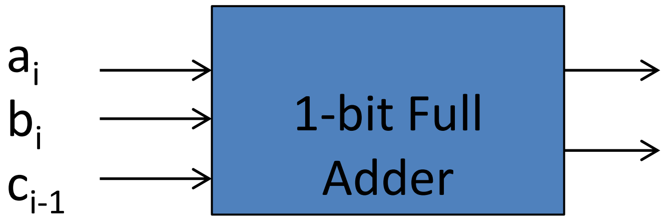
Addition of 2 n-bit binary numbers

- Consider 2 n-bit numbers a = (a_{n-1} a_{n-2} … a_0) and b = (b_{n-1} b_{n-2} … b_0). Addition (using single bit full adders) is inherently sequential because of the flow of carry bits from LSB to MSB. Assume, $a_n = b_n = 0$. Look at the example (n = 8), first.

	8	7	6	5	4	3	2	1	0
a	(0)	1	0	1	0	0	1	1	0
b	(0)	0	1	1	0	1	0	1	1
s	1	0	0	0	1	0	0	0	1

- If 1-bit full addition takes a unit time, sequential algorithm takes O(n) time. How to do it in parallel? Realize it is different from integer additions, there is no way to decompose the array – we need to think differently. What exactly is the reason the process seems inherently sequential starting from LSB to MSB?

$$c_i = \begin{cases} 0, \text{ if } i = 0 \\ a_{i-1}b_{i-1} + a_{i-1}c_{i-1} + b_{i-1}c_{i-1}, \ i > 0 \end{cases} \qquad S_i = a_i \oplus b_i \oplus c_i, \quad 0 \le i \le n$$



Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Addition of 2 n-bit binary numbers

- Consider 2 n-bit numbers a = (a_{n-1} a_{n-2} … a_0) and b = (b_{n-1} b_{n-2} … b_0). Addition (using single bit full adders) is inherently sequential because of the flow of carry bits from LSB to MSB. Assume, a_n = b_n = 0. Look at the example, first.

	8	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
a	(0)	1	0	1	0	0	1	1	0
b	(0)	0	1	1	0	1	0	1	1
s	1	0	0	0	1	0	0	0	1

- Look carefully at all the columns; do we see any property that we can exploit to parallelize the process?
- <u>We cannot do full binary bit addition at any position until we know the carry-in from the previous bit position</u>.

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

How to parallelize

- Define a pre-carry vector, $P = p_{n-1}\, p_{n-2} \ldots p_0$, such that $p_0 = 0$ and for all i, $i > 0$,

$$p_i = \begin{cases} 0, & a_{i-1} = b_{i-1} = 0 \\ 1, & a_{i-1} = b_{i-1} = 1 \\ 2, & \text{otherwise} \end{cases}$$

- (1) The pre-carry vector can be computed in unit time using n $\oplus$ operators (gates); (2) pre-carry elements are ternary but can be easily implemented by using 2 binary bits. Define a binary operator & for ternary elements

$$x \,\&\, y = \begin{cases} 0, & x = 0 \\ 1, & x = 1 \\ y, & x = 2 \end{cases}$$

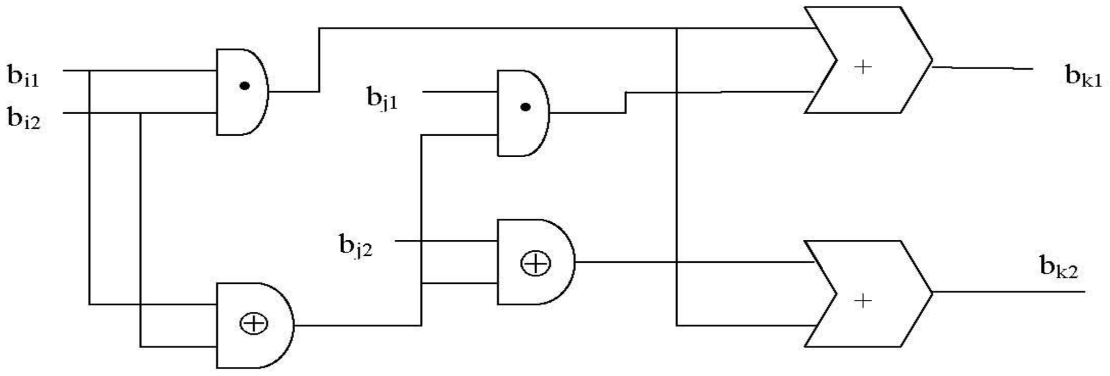
- The operator & is associative but not commutative. Observe that our desired carry bits are given by **$c_i = p_i \,\&\, (p_{i-1} \,\&\, (p_{i-2} \ldots (p_1 \,\&\, p_0) \ldots)$**
- These & operators can be executed in parallel, and the carry vector can be generated in $\lceil \log_2 n \rceil$ time units; then the sum vector can be generated in another parallel time unit.

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

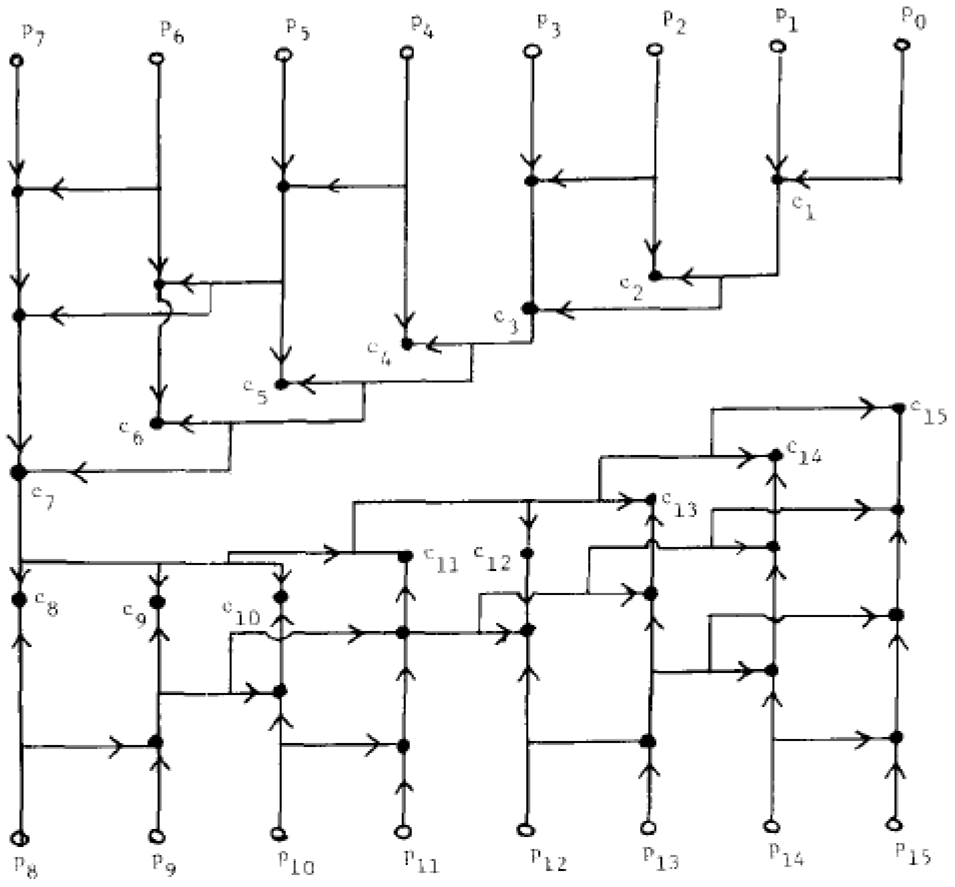
Example

	8	7	6	5	4	3	2	1	0
a	(0)	1	0	1	0	0	1	1	0
b	(0)	0	1	1	0	1	0	1	1
p		2	1	0	2	2	1	2	0
c	1	1	1	0	1	1	1	0	0
s	1	0	0	0	1	0	0	0	1

Pre-carry adder: We need two binary bits to represent a ternary bit p_i; let's assign $b_{i1} = b_{i2} = 0$ for $p_i = 0$, $b_{i1} = b_{i2} = 1$ for $p_i = 1$ and $b_{i1} = 1 \oplus b_{i2}$ for $p_i = 2$. If $p_k = p_i$ ADD& p_j, then we have $b_{k1} = b_{i1}b_{i2} + (b_{i1} \oplus b_{i2}).b_{j1}$, and $b_{k2} = b_{i1}b_{i2} + (b_{i1} \oplus b_{i2}).b_{j2}$



Pre-Carry Adder



Carry generation from precarry vector for n = 16

The number N(n) of such modules (precarry generators) required to compute the precarry vector C, for $n = 2^k$, is given by the recurrence $N(2^k) = 2N(2^{k-1}) + 2^{k-1}$; the solution is $N(2^k) = k \cdot 2^{k-1}$.

Note: The precarry logic modules can be designed as two-level AND-OR circuits just like single-bit full adders and hence the above logic operations require the same amount of time as that of a single-bit full adder. We can say that two n-bit numbers can be added in $\lceil \log_2 n \rceil + 1$ time units using this precarry technique and we call it an n-bit precarry adder. It should be noted that the precarry adder requires a maximum of $\lceil n/2 \rceil$ broadcasting communications.

Assignment

- Consider ADDA to be a function (implementing 1-bit binary full adder) that takes 3 binary bits as input and generates two binary bits – the sum bit and the carry bit as output; also assume another function ADDB (implementing a pre-carry adder) that takes two pre-carry elements as input and generate one pre-carry bit as the output. Explain all the terms, explain the approach and write a detailed pseudo code and analyze the complexity of your algorithms.

Modeling - Basics

- A sequential algorithm is evaluated by its runtime (in general, asymptotic runtime as a function of input size).
- The asymptotic runtime of a sequential program is identical on any serial platform.
- The parallel runtime of a program depends on the input size, the number of processors, and the communication parameters of the machine.
- An algorithm must therefore be analyzed in the context of the underlying platform.
- A parallel system is a combination of a parallel algorithm and an underlying platform.

Source: Joseph Ja Ja, “Introduction to Parallel Algorithms”, V. Kumar, “Introduction to Parallel Processing”

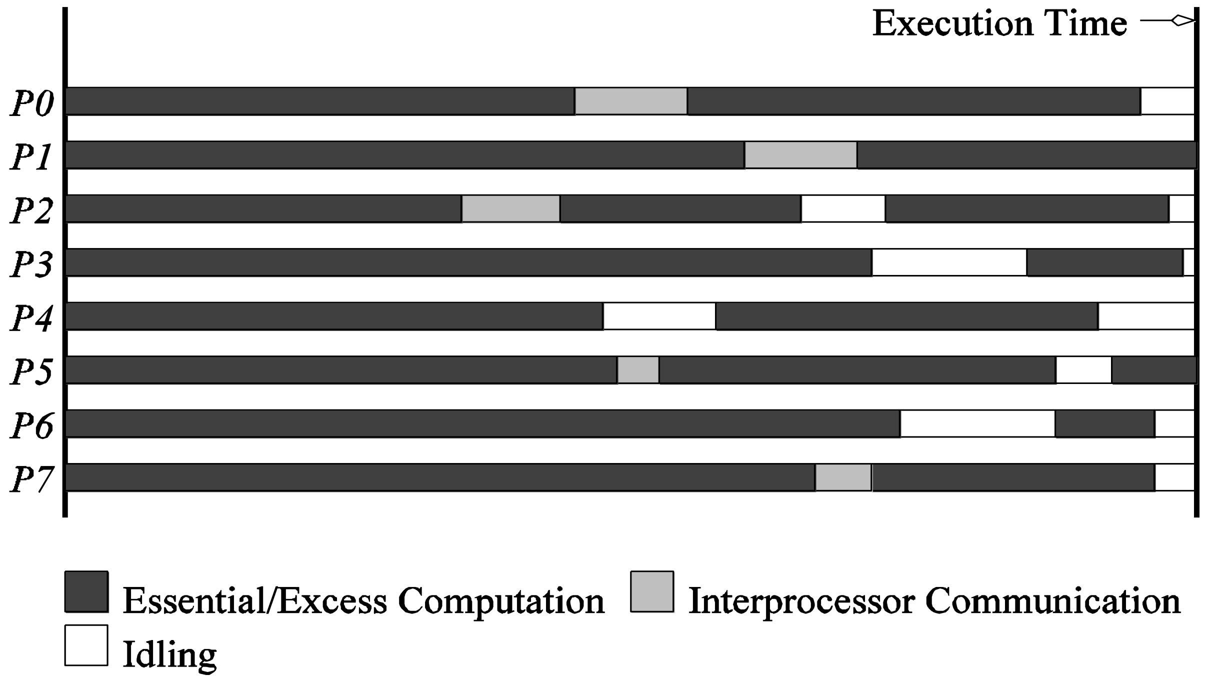
Modeling (contd.)

- Several performance measures are intuitive.
- Wall clock time – the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble. But how does this measure when the number of processors is changed, or the program is ported to another machine altogether? [Platform & Run-time environment dependence]
- How faster is the parallel version? This is related to the question – what is the baseline serial version with which we compare? Can we use a suboptimal serial program to make our parallel program look better?
- Efficiency Issues, Scalability, Cost optimality

Source: Joseph Ja Ja, “Introduction to Parallel Algorithms”, V. Kumar, “Introduction to Parallel Processing”

Sources of Overhead in Parallel Programs

- If I use two processors, shouldn't my program run twice as fast?
- No – a number of overheads, including wasted computation, communication, idling, and contention cause degradation in performance.



The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Sources of Overheads in Parallel Programs

- **Inter process interactions**: Processors working on any non-trivial parallel problem will need to talk (communicate) to each other.
- **Idling**: Processes may idle because of load imbalance, synchronization, or serial components.
- **Redundant Computation**: This is computation not performed by the serial version. This might be because the serial algorithm is difficult to parallelize, or that some computations are repeated across processors to minimize communication.

Source: Joseph Ja Ja, “Introduction to Parallel Algorithms”, V. Kumar, “Introduction to Parallel Processing”

Performance Metrics

- Serial (sequential) runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer.
- The parallel runtime is the time that elapses from the moment the first processor starts to the moment the last processor finishes execution.
- We denote the serial runtime by $\mathbf{T_1(n) = T_S(n)}$ and the parallel runtime by $\boldsymbol{T_P(n)}$, where n is the size of the input and p is the number of processors. [Note: sometimes, when it is evident from the context, size of the input is often omitted assuming all parameters are defined for same n]

Performance Metrics for Parallel Systems: Total Parallel Overhead

- Let T_{all} be the total time collectively spent by all the processing elements.
- T_S is the serial time
- Observe that $T_{all} - T_S$ is the total time spent by all processors combined in non-useful work. This is called the ***total overhead***.
- The total time collectively spent by all the processing elements

  $T_{all} = pT_P$ (***p*** is the number of processors).
- The overhead function (T_o) is given by $T_o = pT_P - T_S$
- Speed-up is $\mathbf{S_p = T_S/T_p}$, ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with ***p*** identical processing elements.

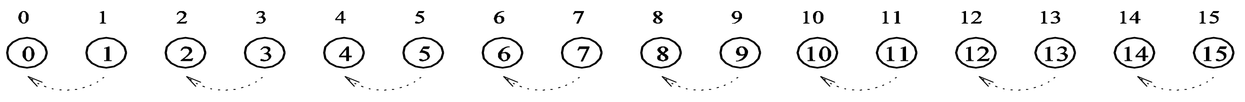
Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Definitions

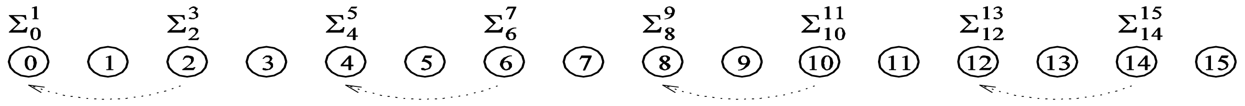
- In the sequential world, performance is determined by instruction rate (CPU cycle time and memory access time) of the –processor and the execution requirement of the software.
- In the parallel world, things are a bit more complicated – we need to consider both hardware and software – the number of processors (hardware) and the structure of the software.
- Let *p* = number of processors (# PEs), *n* = size of the problem, $T_p(n)$ = execution time when p processors are available to work on a problem of size n [if *n* is omitted, we assume we are talking about the problem of same size); also, that includes all overhead as well.
- Assume [to make life simple]: (1) all processors are identical; (2) Ignore all overheads (communication, I/O, resource constraints, context switching, etc.) **Speed-up** = $S_p = T_1/T_p$, **Efficiency** = $E_p = S_p/p$ [normalized speed-up] ; T_i is the parallel execution time with i processors.
- Speed-up is linear if efficiency $E_p = \alpha$, some constant ≤ 1 and is ideal if $\alpha = 1$. Linear speed-up is not possible for an algorithm for all possible problem sizes, because of (1) communication overhead (2) resource contentions (3) structure of software. **Is super-linear ($\alpha > 1$) speed-up possible?**

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

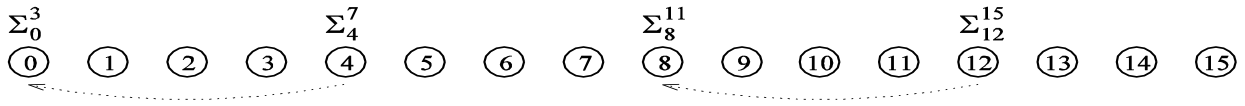
Example: Add n numbers by n PEs in $\log_2 n$ time



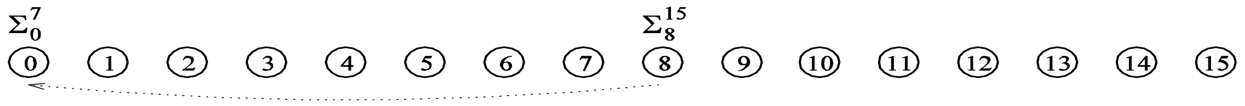
(a) Initial data distribution and the first communication step


Σ_0^1 Σ_2^3 Σ_4^5 Σ_6^7 Σ_8^9 Σ_{10}^{11} Σ_{12}^{13} Σ_{14}^{15}

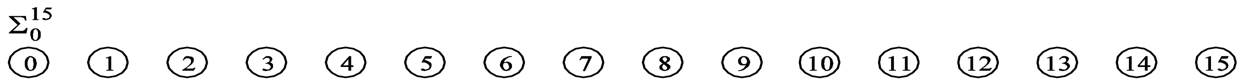
(b) Second communication step


Σ_0^3 Σ_4^7 Σ_8^{11} Σ_{12}^{15}

(c) Third communication step



(d) Fourth communication step



(e) Accumulation of the sum at processing element 0 after the final communicatio

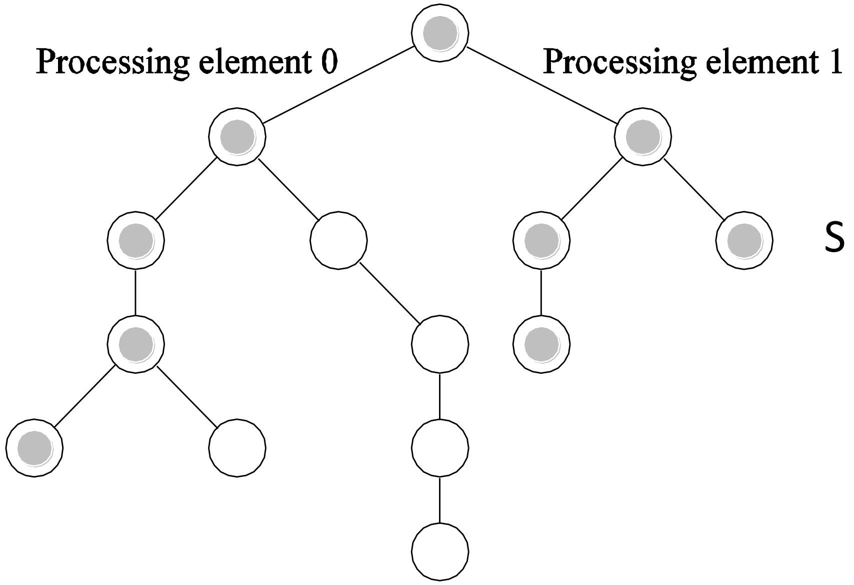
Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Performance Metrics: Example (continued)

- If an addition takes constant time, say, t_c and communication of a single word takes time $t_s + t_w$, we have the parallel time $T_n = \Theta(\log n)$
- We know that $T_S = \Theta(n)$
- Speedup S_n is given by $S = \Theta(n / \log n)$
- Efficiency E_n is given by $E_n = S_n/n = 1/\log n$

Super-linear Speed-up

One reason for super-linearity is that the parallel version does less work than corresponding serial algorithm.



Searching an unstructured tree for a node with a given label, ‘S’, on two processing elements using depth-first traversal. The two-processor version with processor 0 searching the left sub tree and processor 1 searching the right sub tree expands only the shaded nodes before the solution is found. The corresponding serial formulation expands the entire tree. The serial algorithm does more work than the parallel algorithm.

Super-linear Speedup

- Super-linear speedup is attributed to caching effects. When a problem is split onto multiple processors, the sub-problems are smaller. With smaller problem size, cache hit rate is most likely higher, and the result, even after considering the communications time, is still better than the time on a single processor with more cache misses. The number of process switches and the cache flushing that occurs with each process switch are reduced; thus, fewer instructions (less switches) are executed faster (fewer cache faults).
- There are two possible cache effects. (1) Assume a processor has 64KB of unified cache. If data set + code fits into 64KB, a good improvement than when it doesn't. (2) the limited TLB size: 64 entries for 4KB pages, i.e., 256KB mapped simultaneously at most. If a single PE's code + data fits into the TLB*, there is a marked difference; the differences are exaggerated if code makes repeated sweeps over these data regions.
- Resource-based super-linearity: The higher aggregate cache/memory bandwidth can result in better cache-hit ratios, and therefore super-linearity. Example: A processor with 64KB of cache yields an 80% hit ratio. If two processors are used, since the problem size/processor is smaller, the hit ratio goes up to 90%. Of the remaining 10% access, 8% come from local memory and 2% from remote memory. If DRAM access time is 100 ns, cache access time is 2 ns, and remote memory access time is 400ns, this corresponds to a speedup of 2.43!

*"A translation lookaside buffer (TLB) is a cache that memory management hardware uses to improve virtual address translation speed. The majority of desktop, laptop, and server processors includes one or more TLBs in the memory management hardware, and it is nearly always present in any hardware that utilizes paged or segmented virtual memory."

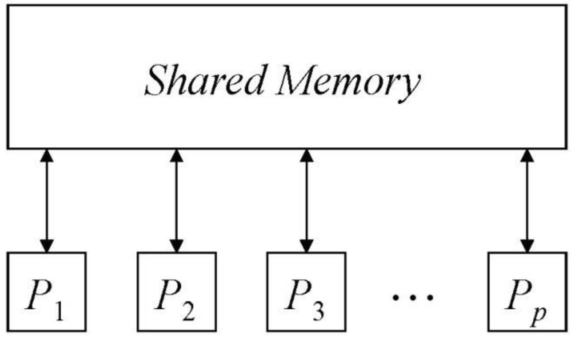
Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Two Basic Models

- Shared Memory
  - Each PE has a unique *id, 1 …p*
  - Synchronous (**PRAM**) – either SIMD or MIMD – size of data exchanged between local and global memory represents communication costs – A = B+C where all are shared variables implicitly means reading from shared memory to local memory and writing from local memory to shared memory – EREW, CREW, CRCW (common, priority, arbitrary) – asynchronous shared memory model will need explicit synchronization primitives.
- Network Model
  - A network is a graph (V, E) – a node is a PE, and an arc is a bidirectional communication channel – a PE has its local memory but no shared memory – info exchange is done via 2 primitives: *send (X, i) and receive (Y, i)* – message passing model – network topology (linear array, mesh, hypercube) and routing (store & forward, wormhole) are important

PRAM model: synchronous shared memory model

- Well developed body of techniques to handle different classes of computational problems
- Focus on structural properties of the problem (removes algorithmic details of synchronization & communication issues)
- Does not depend on network topology – many network algorithms can be directly derived from PRAM algorithms.
- Provides explicit understanding of the operations to be performed at each time unit and explicit allocations of processors to computational jobs at each time unit.



The synchronous PRAM model has a similarity with data-parallel execution on a SIMD machine: (1) All processors execute the same program; (2) All processors execute the same PRAM step instruction stream in "lock-step"; (3) Effect of operation depends on local data; (4) Instructions can be selectively disabled (for if-then-else flow).

PRAM Programming References:
- A PRAM oriented programming system, CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 9(3), 163–179 (MARCH 1997)
- Practical PRAM Programming, by Jörg Keller, Christoph W. Kessler, Jesper L. Träff [Check the Project homepages from the web site.]

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Classification of PRAM Model

- A PRAM step ("clock cycle") consists of three phases
  1. *Read*: each processor may read a value from shared memory
  2. *Compute*: each processor may perform operations on local data
  3. *Write*: each processor may write a value to shared memory
- Model is refined for concurrent read/write capability
  - Exclusive Read Exclusive Write (EREW)
  - Concurrent Read Exclusive Write (CREW)
  - Concurrent Read Concurrent Write (CRCW)
- CRCW PRAM: what to do with concurrent writes?
  - Common CRCW: all processors must write the same value
  - Arbitrary CRCW: one of the processors succeeds in writing
  - Priority CRCW: processor with highest priority succeeds in writing

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Relevance of the PRAM Model

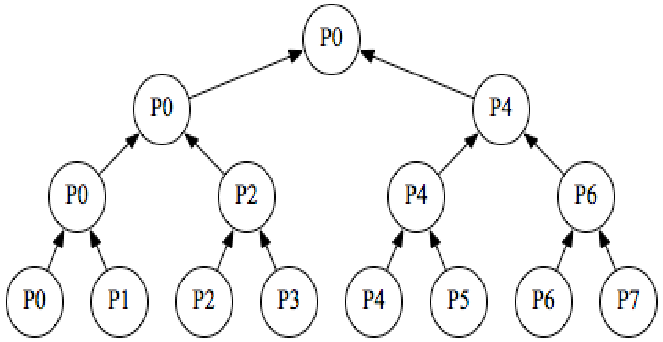
- The central question in complexity analysis for sequential algorithms is the P = NP question. The corresponding question in the complexity analysis of parallel algorithms is the P = NC question. NC is the class of all problems which, with a polynomial number of PEs, can be solved in poly-logarithmic time. An algorithm of size n is poly-logarithmic if it can be solved in $O(\log(n)^c)$ time with $O(n^k)$ PEs, where c and k are constants.
- Problems in NC can be solved efficiently on a parallel computer. "NC" is an abbreviation of "Nick (Pippenger)'s Class."
- A common criticism of the PRAM model is the unrealistic assumption of an immediately addressable unbounded shared parallel memory. This criticism, like the criticism of $O(n^{17})$ "polynomial" time algorithms, often comes from a misunderstanding of the role of theory:
  - Theory is not everything: theoretical results are not to be taken as is and implemented by engineers.
  - Theory is also not nothing: the fact that an algorithm cannot in general be implemented as is does not mean it is meaningless.
- When a $O(n^{17})$ algorithm is designed for a problem, it does not lead to a practical way to solve that problem, while it proves something inherent about the problem, namely that it is in P, and thus that its hardness does not grow exponentially with the input size. Hopefully, in the process of proving this result key insights may be developed that can later be used in practical algorithms for this or other problems, or for other theoretical results. Similarly, the design of PRAM algorithms for a problem proves something inherent to the problem (namely that it is parallelizable) and can in turn lead to new ideas.

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Sum (Reduction) on a PRAM

- Input: an array A of size $n = 2^k$ stored in the shared memory of a PRAM with n processors; initialized local variables are n and the processor number is i.
- Output: sum of the entries of A stored in the shared location S; the array A is not destroyed.
- Code for processor P_i

```
begin
1. read (A(i), a)
2. write (a, B(i))
3. for h = 1 to log n do
   if (i ≤ n/2^h) then B(i) = B(2i) + B(2i-1);
   if i = 1 then S = B(1)
end
```



Note:

1. During steps 1 and 2 a copy B of A is created in the shared memory. Step 3 is based on a balanced tree whose leaves are the entries of A. Note that we use n processors for an array A of size n – not all processors are busy all the time; only processor P_1 is busy during execution of the algorithm. <u>All operations by all processors are synchronous</u>. Execution time = $O(\log_2 n)$, Work done = O(n).
2. Can easily be done in place.

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Matrix Multiplication on PRAM

- **Input**: Two n by n matrices A and B stored in shared memory, $n = 2^k$. Initialized local variables are n and a triple of indices (i,j,k) to identify the processor. We assume we have n^3 PEs, denoted by $P_{i,j,k}$ where $1 \le i, j, k \le n$.
- **Output**: A n by n matrix C = A*B stored in shared memory
- Code for the processor Pi,j,k

$$C_{ij} = \sum_{k=1}^{n} A_{ik} * B_{kj}$$

```
begin
1. D(i,j,k) = A(i,k)*B(k,j)
2. for h = 1 to log n do {if k ≤ n/2^h then D(i, j,k) = D(i, j,2k-1)+D(i,j,2k) }
3. if k=1 then C(i,j) = D(i,j,1)
end
```

Note:

1. This is an example of CREW algorithm – processors $P_{i,1,k}$, $P_{i,2,k}$, … $P_{i,n,k}$ all require A(i,k) and B(k,j) from the shared memory. The algorithm takes O(log n) parallel steps if we use n^3 processors – work done is $O(n^3 \log n)$ – not all processors are busy all the time. Do a Gantt chart to see how the efficiency is.
2. We can convert this algorithm to do Boolean multiplication as well. We replace * (integer multiply) by $*_b$ (Boolean multiply, i.e., Boolean AND) and + (integer add) by $+_b$ (Boolean add, i.e., Boolean OR).

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Performance Metrics

- Let Q be the problem for which we have a PRAM algorithm that runs in T(n) time using P(n) processors for an instance of size n. The *time-processor product* C(n) = T(n)*P(n) represents **cost** of the parallel algorithm – the obvious sequential algorithm will run in O(C(n)) time (a single processor simulating P(n) processors in O(P(n)) time).
- Generalize to p ≤ P(n) processors as follows: for each of the parallel steps we use p processors to simulate the original P(n) processors in O(P(n)/p) sub steps in the obvious way – the simulation takes O(T(n)P(n)/p) time – total parallel time with p processors is O(C(n)/p + T(n)) [Explain]. Note that P(n) represents, in a sense, the maximal parallelism inherent in the problem and the algorithm.
- For the matrix multiplication problem $\Rightarrow$ n^3 processors and O(log n) time and O(n^3log n) cost and O(n^3log n/p + log n) parallel time with an arbitrary p number of processors

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Work-Time Framework

- The refinement is done in two steps:
  - Upper level (WT presentation) $\Rightarrow$ describe the algorithm in terms of sequence of time units, where each time unit may include any number of concurrent operations – **work** performed by a parallel algorithm is *total number of operations* used. We use a statement like "for L $\le$ i $\le$ U pardo statement" – the *statement* (which can be a sequence of statements) following the pardo depends on the index i – the statements corresponding to all values of i between L and U are executed concurrently.
  - Lower Level (WT Scheduling principle): Let $W_i(n)$ be the number of operations performed in time unit i, where $1 \le i \le T(n)$. Simulate each set of $W_i(n)$ operations in $\le \left\lceil \frac{W_i(n)}{p} \right\rceil$ parallel steps. If the simulation is successful, the p-processor PRAM algorithm takes

$$\le \sum_i \left\lceil \frac{W_i(n)}{p} \right\rceil \le \sum_i \left(\left\lfloor \frac{W_i(n)}{p} \right\rfloor + 1 \right) \le \left\lfloor \frac{W(n)}{p} \right\rfloor + T(n) \text{ parallel steps}$$

The Sum $S = \sum_{1 \le i \le n} A(i)$ [Higher Level]

```
begin
1. For $1 \le i \le n$ pardo
        $B(i) = A(i)$
2. For $h = 1$ to $\log n$ do
        for $1 \le i \le n/2^h$ pardo
                $B(i) = B(2i-1) + B(2i)$
3. $S = B(1)$
end
```

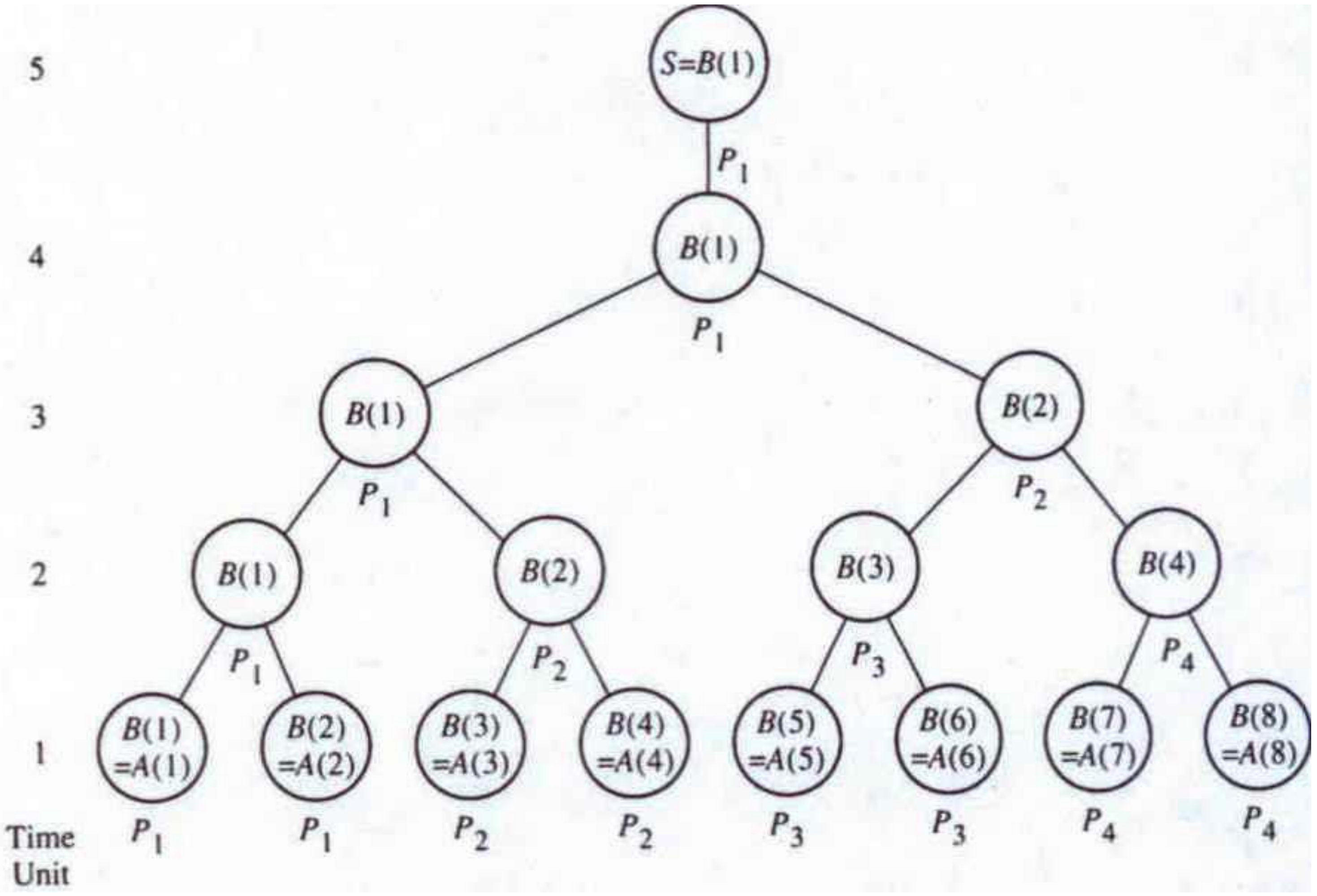
Note: No mention of how many processors or how operations are allocated to processors – stated only in terms of time units – each time unit may include arbitrary number of concurrent operations. We have log n + 2 time units – so parallel run time is O(log n). Observe that n operations are done in the first-time unit (step 1), j-th time unit (iteration h = j-1 of step 2) includes $n/2^{j-1}$ operations $\Rightarrow$ work performed $W(n)$ = total number of operations = $n + \{n/2^0 + n/2^1 + \ldots n/2^{\log n}\} + 1 = O(n)$. <u>The main advantage is we don't have to deal with processors, we can concentrate on inherent parallelism.</u>

Sum algorithm [lower level] for P_s, $1 \le s \le p = 2^q \le n$

```
begin
1. for j = 1 to m (= n/p) do
      B(m(s-1) + j) = A(m(s-1) + j);
2. for h = 1 to log n do
      2.1  if (k-h-q ≥ 0) then {for j = $2^{k-h-q}$(s-1) to $2^{k-h-q}$s do B(j) = B(2j-1)+B(2j)}
      2.2  else {if s ≤ $2^{k-h}$ then B(s) = B(2s-1) +B(2s) }
3. if s=1 then S = B(1)
end
```

Note: Step 1 takes O(n/p) time (each processor does n/p operations) – the h-th operation of step 2 takes $O(n/2^h p)$ time, since a processor has to do at least $\lfloor n/2^h p \rfloor$ operations – step 3 takes O(1) time $\Rightarrow$ adding up, total run time $T_p(n)$ is given by O(n/p +log n) as predicted by the WT scheduling principle.

Example Execution



Processor allocation for computing sum of 8 elements with 4 processors – note that no new idea is involved in doing the lower-level details – mechanical work, often time cumbersome, but necessary for implementation.

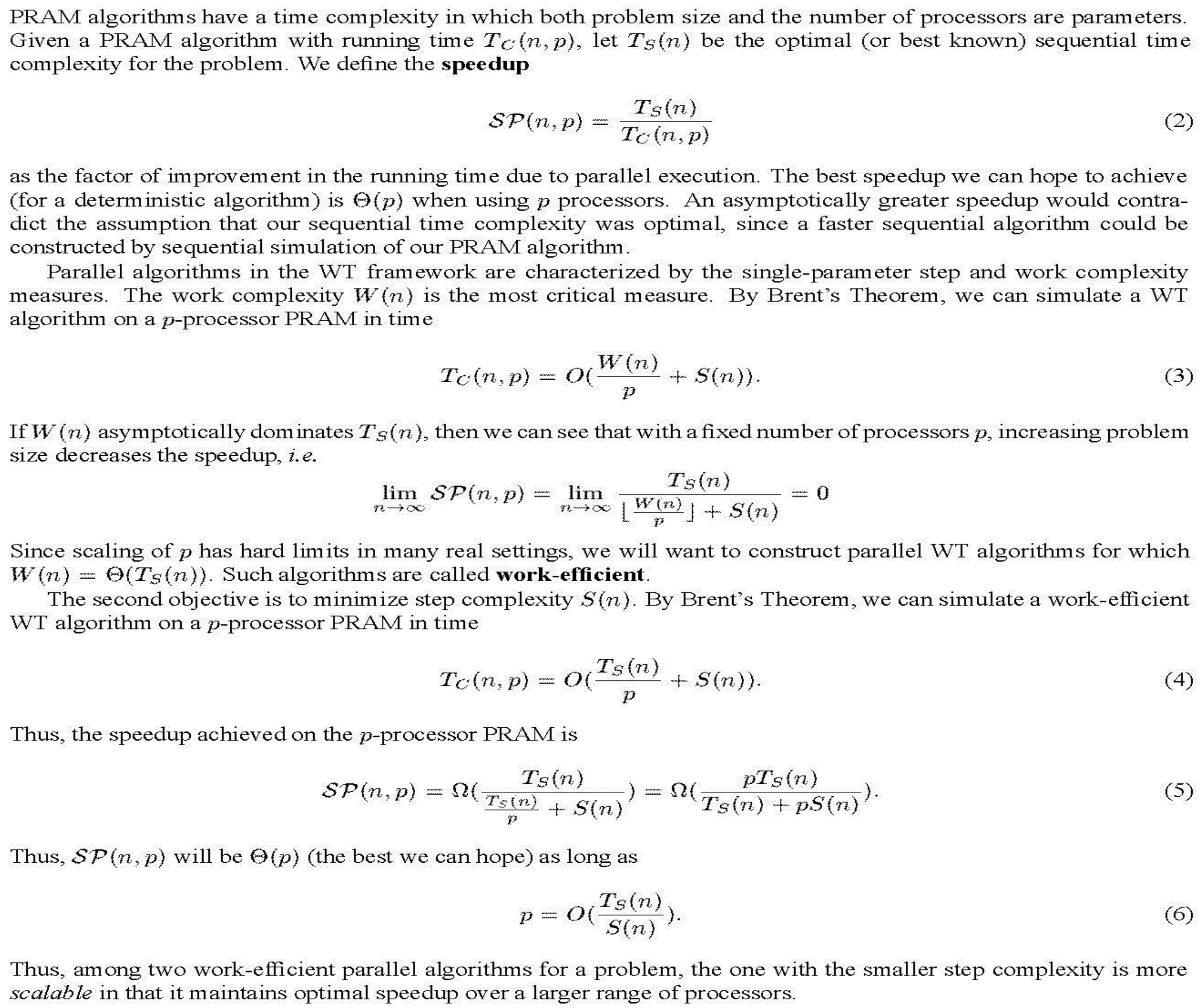
Source: Joseph Ja Ja, “Introduction to Parallel Algorithms”, V. Kumar, “Introduction to Parallel Processing”

Work Versus Cost

- Given a parallel algorithm running in time $T(n)$ and using a total of $W(n)$ operations [WT presentation level], it can be simulated on a p-processor PRAM in $T_p(n) = O(W(n)/p + T(n))$ time by WT scheduling principle. The corresponding cost is $C_p(n) = T_p(n)*p = O(W(n) + T(n)*p)$
- $W(n)$ [work] and $C_p(n)$ are equal only when $p = O(W(n)/T_p(n))$ asymptotically (we always have $C_p(n) \geq W(n)$, for any p) – otherwise, $W(n)$ measures total number of operations by the algorithm (nothing to do with the number of processors employed – unbounded parallelism) and $C_p(n)$ measures the cost of the algorithm relative to the number p of the processors available.
- Consider the sum example. $W(n) = O(n)$, $T(n) = O(\log n)$ and $C_p(n) = O(n + p \log n)$. Both are of equal order when p is $O(n/\log n)$ (optimal number of processors!) – when p is larger than this, not all processors are busy all the time.
- A parallel algorithm is called work-optimal iff $W(n) = O(T^*(n))$ where $T^*(n)$ is the time needed by provably the best sequential algorithm, regardless of the parallel running time of the algorithm (no redundant computations are introduced to increase parallelism).

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

An Alternate Modeling to make p and n explicit⭐



PRAM algorithms have a time complexity in which both problem size and the number of processors are parameters. Given a PRAM algorithm with running time $T_C(n,p)$, let $T_S(n)$ be the optimal (or best known) sequential time complexity for the problem. We define the **speedup**

$$\mathcal{SP}(n,p) = \frac{T_S(n)}{T_C(n,p)} \tag{2}$$

as the factor of improvement in the running time due to parallel execution. The best speedup we can hope to achieve (for a deterministic algorithm) is $\Theta(p)$ when using p processors. An asymptotically greater speedup would contradict the assumption that our sequential time complexity was optimal, since a faster sequential algorithm could be constructed by sequential simulation of our PRAM algorithm.

Parallel algorithms in the WT framework are characterized by the single-parameter step and work complexity measures. The work complexity $W(n)$ is the most critical measure. By Brent's Theorem, we can simulate a WT algorithm on a p-processor PRAM in time

$$T_C(n,p) = O(\frac{W(n)}{p} + S(n)). \tag{3}$$

If $W(n)$ asymptotically dominates $T_S(n)$, then we can see that with a fixed number of processors p, increasing problem size decreases the speedup, *i.e.*

$$\lim_{n\to\infty} \mathcal{SP}(n,p) = \lim_{n\to\infty} \frac{T_S(n)}{\lfloor \frac{W(n)}{p} \rfloor + S(n)} = 0$$

Since scaling of p has hard limits in many real settings, we will want to construct parallel WT algorithms for which $W(n) = \Theta(T_S(n))$. Such algorithms are called **work-efficient**.

The second objective is to minimize step complexity $S(n)$. By Brent's Theorem, we can simulate a work-efficient WT algorithm on a p-processor PRAM in time

$$T_C(n,p) = O(\frac{T_S(n)}{p} + S(n)). \tag{4}$$

Thus, the speedup achieved on the p-processor PRAM is

$$\mathcal{SP}(n,p) = \Omega(\frac{T_S(n)}{\frac{T_S(n)}{p} + S(n)}) = \Omega(\frac{pT_S(n)}{T_S(n) + pS(n)}). \tag{5}$$

Thus, $\mathcal{SP}(n,p)$ will be $\Theta(p)$ (the best we can hope) as long as

$$p = O(\frac{T_S(n)}{S(n)}). \tag{6}$$

Thus, among two work-efficient parallel algorithms for a problem, the one with the smaller step complexity is more *scalable* in that it maintains optimal speedup over a larger range of processors.

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Simultaneous Max & Min Computation Efficiently

- Pick 2 elements(a, b), compare them. (say a > b)
- Update min by comparing (min, b); Update max by comparing (max, a)
- Then move to next 2 elements, do the same and move on. This way you would do 3 comparisons for 2 elements, amounting to 3N/2 total comparisons for N elements.
- Writing program is relatively easy. Try recursion first.
- Do you see the embedded binary tree? Can you parallelize your code like we did the sum? Note: computing sum, maximum or minimum is almost identical. Practice. Think of recursion.

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Using Balanced Trees

Prefix Sum

Given an array A[1 .. n], the prefix sum array PS[1.. n] is

$$PS_i = \sum_{j=1}^{i} A_i$$

Example: A[5,6,7,8] $\rightarrow$ PS[5,11,18,26]

What is Prefix Sum?

Given an array A[1 .. n], the prefix sum array PS[1.. n] is

Example: A[5,6,7,8] → PS[5,11,18,26]

$$PS_i = \sum_{j=1}^{i} A_i$$

Sequential Program → O(n) sequential execution time and O(n) work – both linear in size n of the array

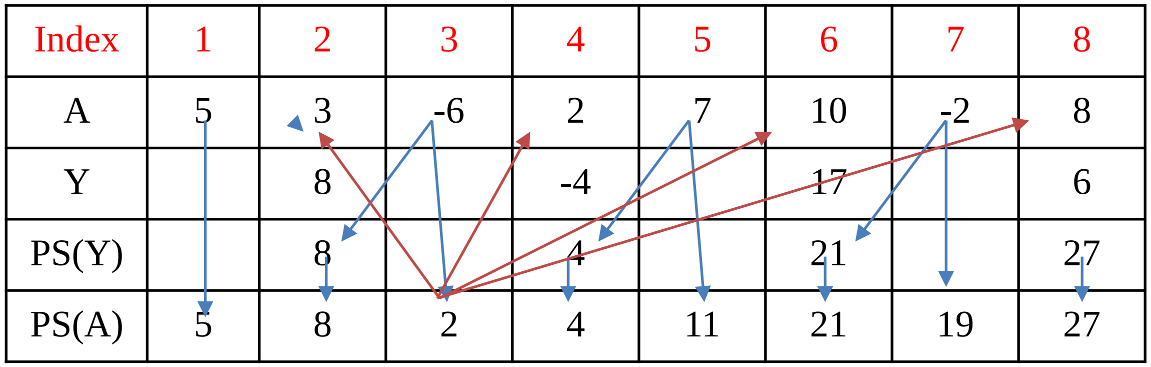
```
Procedure PS (A, n, S)
    $S_1 = A_1$;
    if n = 1 then exit;
    else { for j = 2 to n do $S_j = S_{j-1} + A_j$ }
```

Note: Can easily be done in-place. It looks inherently sequential; is it?

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

How to parallelize

- Example: Consider an array of size n = 8 (assume n = 2^k, for some integer k) and do $A_{2i-1} + A_{2i}$ for i=1 to n/2 to generate array Y and generate the prefix sum of the smaller array of half the size (we will soon see how).



Index	1	2	3	4	5	6	7	8
A	5	3	-6	2	7	10	-2	8
Y		8		-4		17		6
PS(Y)		8		4		21		27
PS(A)	5	8	2	4	11	21	19	27

These elements are already in their place

- Observe that once we have PS(Y), we can generate PS(A) in one parallel step if we have n/2 processors using the additions as shown by arrows. Why?
- We generate PS(Y) by using recursion; note that the recursion stops as soon as we arrive an array of size 2.
- *Is there any error somewhere in the table??*

Parallel Prefix in PRAM

```
Procedure P_S (A, n, S);
    Step 1: if $n = 1$ then {$S_1 = A_1$; exit}
    Step 2: for $i = 1$ to $n/2$ pardo {$Y_i = A_{2i-1} + A_{2i}$}
    Step 3 : Call P_S (Y, $n/2$, Z)
    Step 4: for $i = 1$ to $n$ pardo
        {$i = 1 \Rightarrow S_1 = A_1$
         $i = \text{even} \Rightarrow S_i = Z_{i/2}$
         $i = \text{odd} \Rightarrow S_i = Z_{(i-1)/2} + A_i$ }
```

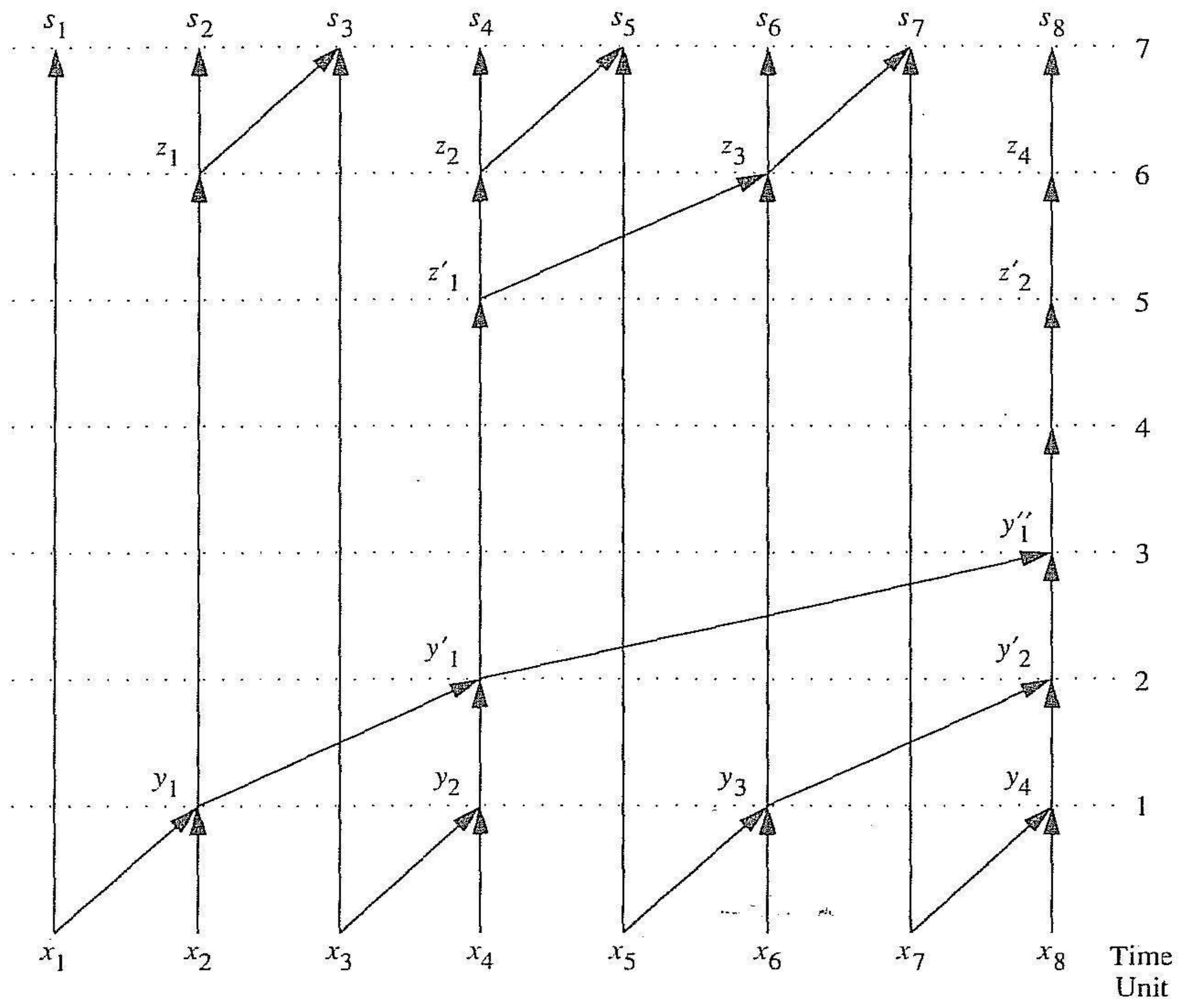
Unit parallel time

Note: The program is recursive; $O(\log_2 n)$ parallel execution time (the subarray size decreases by a factor of 2 at each recursive call) with unbounded parallelism (number of PEs needed is n/2. Remember to use the subscripts correctly.

Do the Gantt chart with n/2 processors; compute speedup, cost, efficiency; repeat for $p < n/2$. Draw your conclusions; what did you find?

Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

How it works on an array of 8 elements



Source: Joseph Ja Ja, "Introduction to Parallel Algorithms", V. Kumar, "Introduction to Parallel Processing"

Complexity

The running time T(n) and the work W(n) required by the algorithm satisfy the following recurrences:

$$T(n) = T(n/2) + a$$

$$W(n) = W(n/2) + bn$$

where a and b are constants. The solutions of these recurrences are

$$T(n) = O(\log n) \text{ and } W(n) = O(n).$$

Note: The parallel run time is logarithmic in size of the array and the total work is still linear – *work optimal*.

Non-Recursive Prefix_Sum on PRAM

We first consider a simpler version using a 2-dimensional array that kind of removes the recursion from the algorithm in a straightforward way. Let A[1..n] denote the array of given n elements (integers). Let B(h,j) be the sets of auxiliary variables where $0 \leq h \leq \log_2 n$ and $1 \leq j \leq n/2^h$. The array B is used to record the information in the binary tree nodes during a forward traversal, whereas the array C is used during the backward traversal of the tree.

Input: An array A of size $n = 2^h$, where h is a nonnegative integer
Output: A matrix C such that C(0,j) is the jth prefix sum for $1 \leq j \leq n$

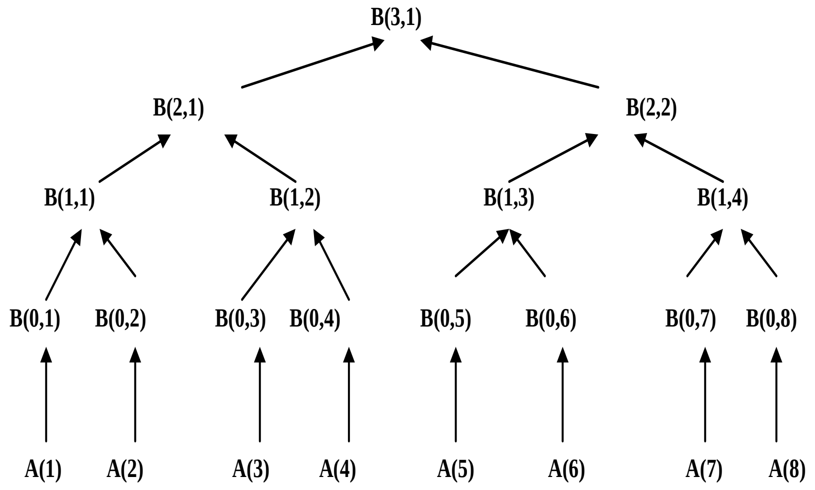
Pseudo Code

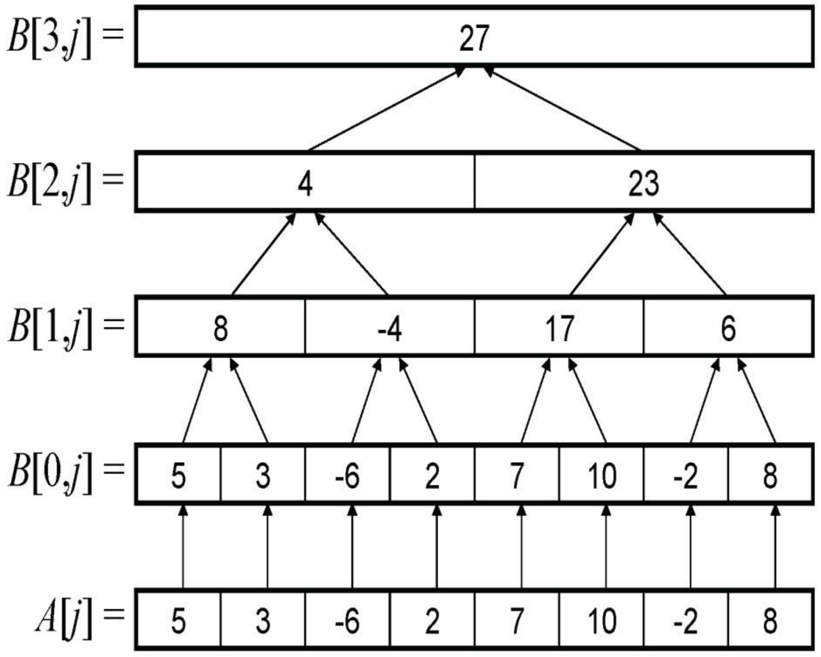
```
begin
for 1 ≤ j ≤ n pardo
            set B(0,j) = A(j)
for h =1 to log n do
          for 1 ≤ j ≤ n/2h  pardo
              set B(h,j) = B(h-1,2j-1) * B(h-1,2j)
for h = log n to 0 do
           for 1 ≤ j ≤ n/2h  pardo
               { j even      : set C(h,j) = C(h+1, j/2)
                   j =1        : set C(h,1) = B(h,1)
             j odd             : set C(h,j) = C(h+1, (j-1)/2) * B(h,j)  }
end
```

Source: Joseph Ja Ja, “Introduction to Parallel Algorithms”, V. Kumar, “Introduction to Parallel Processing”

Data Flow (Forward)

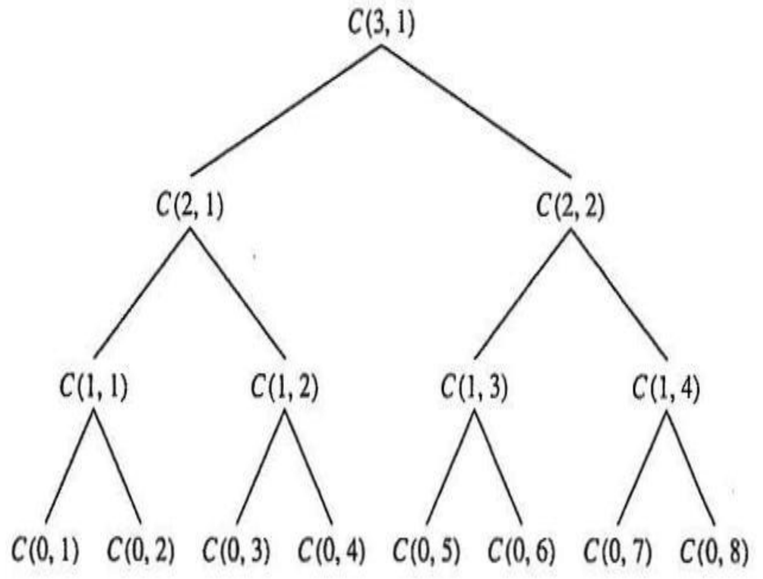
Run time: The parallel run time is O($\log_2 n$) – there are two sequential for loops each of which runs exactly $\log_2 n$ times.



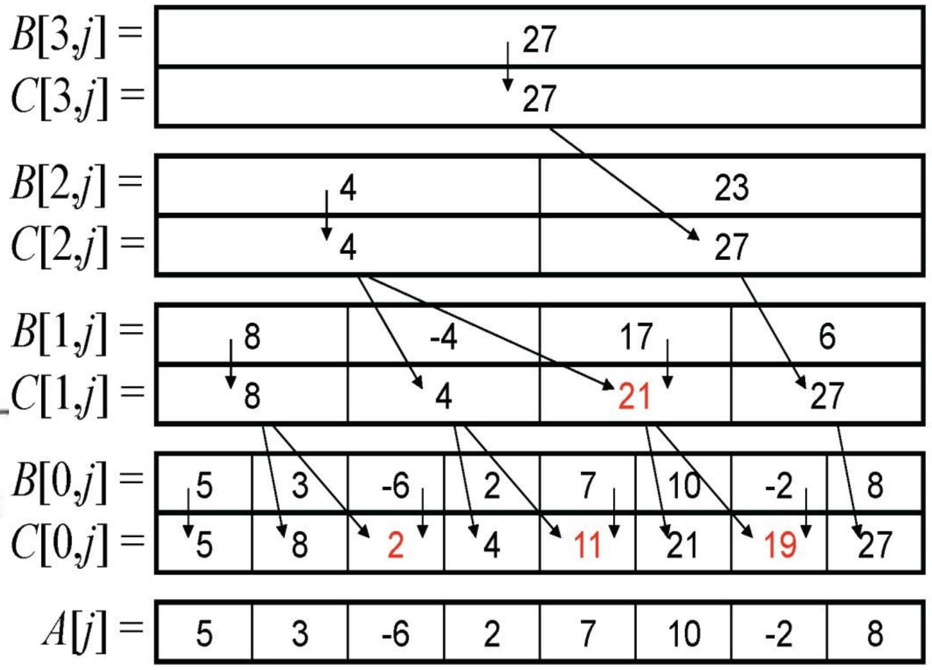


Note: This is the forward traversal (of the logical binary tree) – this is bottom-up. The backward traversal for C-array is the same except it is top-down

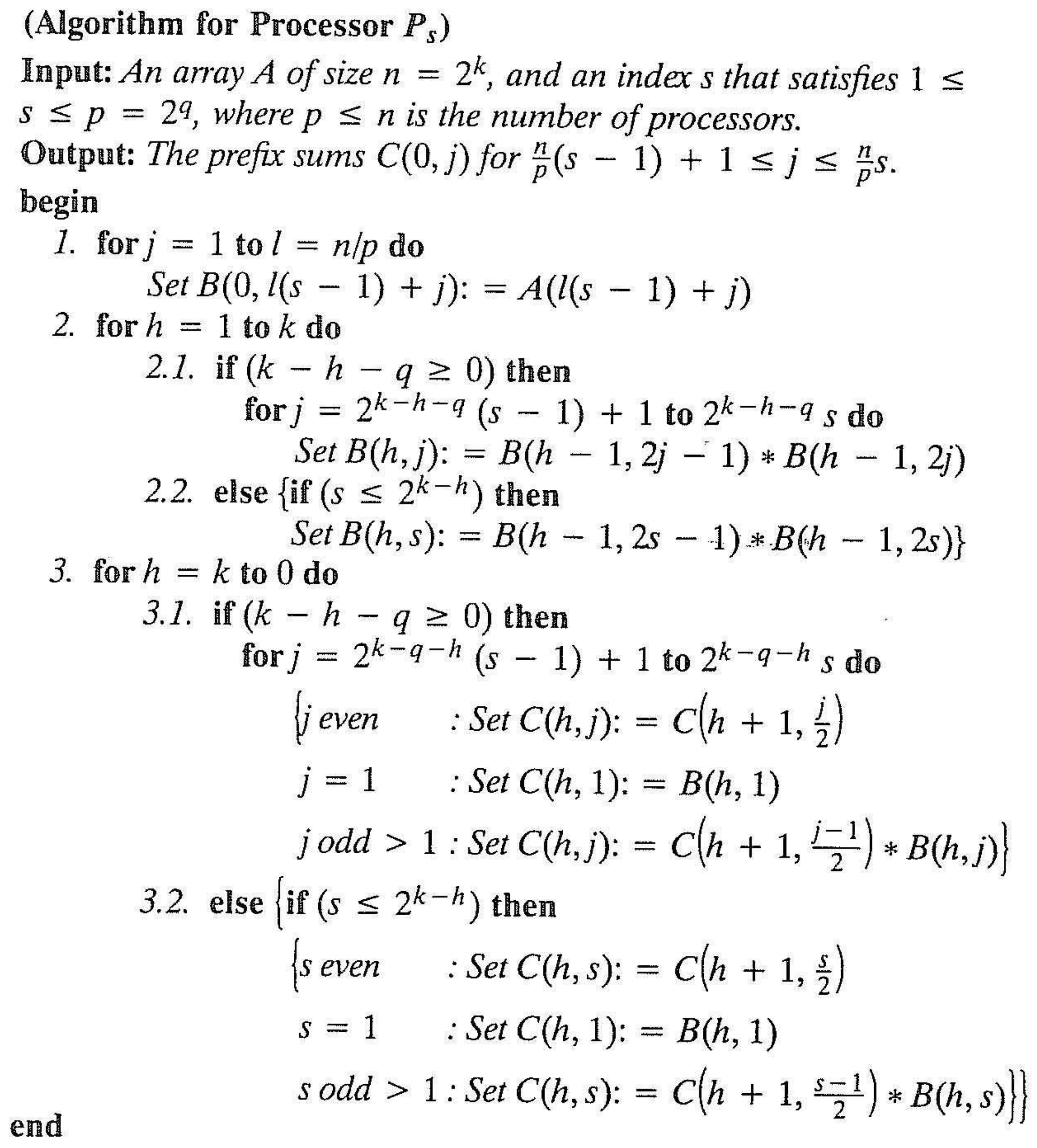
Data Flow (backward)



The elements of the array C as generated by the top-down (backward) traversal of the binary tree corresponding to the nonrecursive prefix-sums algorithm.



B[3,j] = 27
C[3,j] = 27
B[2,j] = 4 | 23
C[2,j] = 4 | 27
B[1,j] = 8 | -4 | 17 | 6
C[1,j] = 8 | 4 | 21 | 27
B[0,j] = 5 | 3 | -6 | 2 | 7 | 10 | -2 | 8
C[0,j] = 5 | 8 | 2 | 4 | 11 | 21 | 19 | 27
A[j] = 5 | 3 | -6 | 2 | 7 | 10 | -2 | 8



```
(Algorithm for Processor $P_s$)
Input: An array $A$ of size $n = 2^k$, and an index $s$ that satisfies $1 \le s \le p = 2^q$, where $p \le n$ is the number of processors.
Output: The prefix sums $C(0, j)$ for $\frac{n}{p}(s-1)+1 \le j \le \frac{n}{p}s$.
begin
  1. for $j = 1$ to $l = n/p$ do
         Set $B(0, l(s-1)+j) := A(l(s-1)+j)$
  2. for $h = 1$ to $k$ do
         2.1. if $(k-h-q \ge 0)$ then
                  for $j = 2^{k-h-q}(s-1)+1$ to $2^{k-h-q}s$ do
                      Set $B(h, j) := B(h-1, 2j-1) * B(h-1, 2j)$
         2.2. else {if $(s \le 2^{k-h})$ then
                      Set $B(h, s) := B(h-1, 2s-1) * B(h-1, 2s)$}
  3. for $h = k$ to $0$ do
         3.1. if $(k-h-q \ge 0)$ then
                  for $j = 2^{k-q-h}(s-1)+1$ to $2^{k-q-h}s$ do
                      {$j$ even      : Set $C(h, j) := C\left(h+1, \frac{j}{2}\right)$
                       $j = 1$       : Set $C(h, 1) := B(h, 1)$
                       $j$ odd $> 1$ : Set $C(h, j) := C\left(h+1, \frac{j-1}{2}\right) * B(h, j)$}
         3.2. else {if $(s \le 2^{k-h})$ then
                      {$s$ even      : Set $C(h, s) := C\left(h+1, \frac{s}{2}\right)$
                       $s = 1$       : Set $C(h, 1) := B(h, 1)$
                       $s$ odd $> 1$ : Set $C(h, s) := C\left(h+1, \frac{s-1}{2}\right) * B(h, s)$}}
end
```

Source: Joseph Ja Ja, “Introduction to Parallel Algorithms”, V. Kumar, “Introduction to Parallel Processing”

Analysis & Processor Allocation

- If n = 2^k, there are 2k+2 parallel steps; if $W_{i,m}$ = work done in ith step during mth iteration,

$$W_{1,1} = n, W_{2,m} = n/2^m, W_{3,m} = 2^m, \quad 0 \le m \le k$$

- So, total work W(n) is

$$W_{1,1} + \sum_{m=1}^{k} W_{2,m} + \sum_{m=0}^{k} W_{3,m} = n + n(1 - \frac{1}{n}) + 2n - 1 = O(n)$$

- Now, if we have only p=2q PEs, divide the work evenly among all PEs in each parallel step; see how it works.
- Question: Can we do a non recursive solution without using additional arrays, i.e., in-place?

Get Rid of the 2-D Array

```
Procedure Prefix_PRAM (in: A[1..n], out: Prefix[1..n])
    for $1 \leq i \leq n$ do in parallel
        Prefix[i] = A[i-1] + A[i]
    end in parallel
    k:= 2;
    while k < n do
        for $k+1 \leq i \leq n$ do in parallel
            Prefix[i] := Prefix[i-k] + Prefix[i]
        end in parallel
        k := k + k
    end while
end Procedure
```

Analysis

- PEs are numbered from 1 to n. In the first parallel step, each processor i computes the sum of A[i] and A[i-1]. In the following steps, the processors use values of previous prefix sums from other processors to compute their prefix sums. The time complexity of the complete algorithm is O (log n).
- Note that for each step, less and less processors take part in computation. Analyzing the algorithm shows that this number increases two times each iteration. In the very first step (the initial for loop) each processor does a single addition, so there is linear work done in this step. Then, the while loop is performed log n times where each time the number of processors doing the computation decreases. In the first iteration, processors 2 through n do work; in the second iteration processors 4 through n do the work and so on. So, total work done in the entire while loop is

$$= (n-2) + (n-4) + (n-8) + \dots\dots + (n-n/2)$$

$$= n \log n - (2+4+8+16 + \dots + n/2)$$

$$= n \log n \quad - \sum_{i=1}^{\log n/2} 2^i = O\left(n \log n - \frac{2^{\log(n/2)-1} - 1}{2-1} + 1\right) = O(n \log n)$$

Pointer Jumping

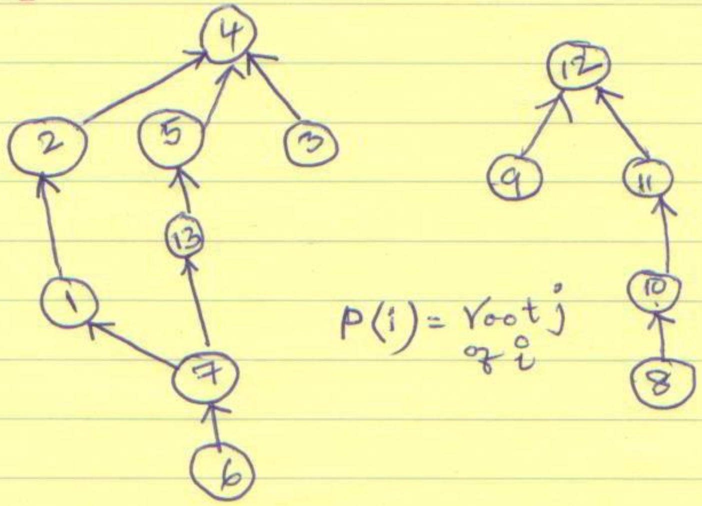
Finding the Roots of a Forest

Rooted Forest Example

Any error?

Rooted forest Example

$$\begin{array}{cccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 12 & 13 & \\ P = [& 2 & 4 & 4 & 4 & 4 & 7 & 13 & 10 & 12 & 11 & 12 & 5 &] \end{array}$$



$P(i)$ = root of i

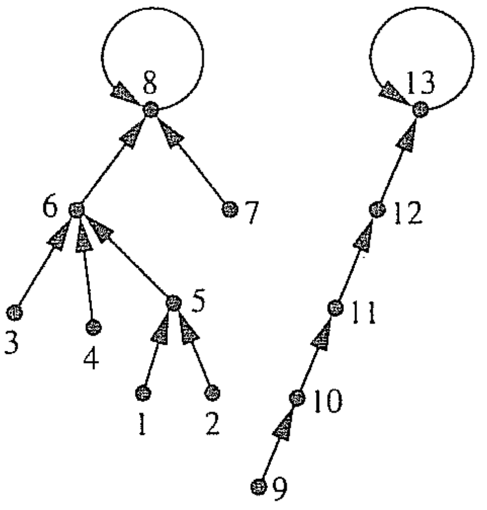
We need to get S[ ] array to denote root of each node.

$S = [4\ 4\ 4\ 4\ 4\ 4\ 4\ 12\ 12\ 12\ 12\ 12\ 4]$

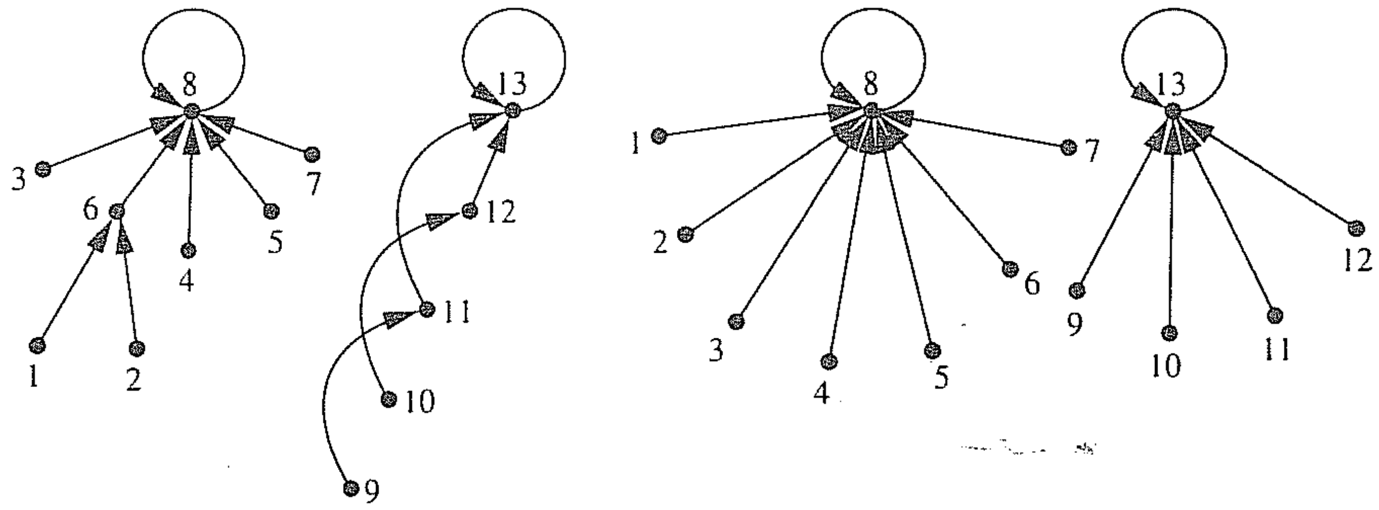
Roots of a Forest

- A *rooted-directed tree* T is a directed graph with a special node r such that (1) every v $\in$V - {r} has out degree 1, and the out degree of r is 0, and (2) for every v $\in$ V - {r}, there exists a directed path from v to r. The special node r is called the root of T. It follows that a rooted directed tree is a directed graph whose undirected version is a rooted tree such that each arc of T is directed from a node to that node's parent.
- The forest F is specified by an array P of length n such that P(i) = j iff (i,j) is an arc in F; that is, j is the parent of i in a tree of F. For simplicity, if i is a root, we set P(i) = i. The problem is to determine the root S(j) of the tree containing the node j, for each j between 1 and n.
- Sequential Algorithm – (1) Identify roots; (2) Reverse the links in each tree; (3) do a depth-first or breadth first search starting from the root of each tree. Total run time is O(n) and total work is O(n) where n is the number of nodes in the forest; write the complete code yourself.
- The ***pointer jumping*** technique allows the fast processing of data stored in the form of a set (forest) F of rooted-directed trees.

Pointer Jumping Example



(a)



Source: Joseph Ja Ja, “Introduction to Parallel Algorithms”, V. Kumar, “Introduction to Parallel Processing”

Roots of a Forest

Input: Predecessor Array P[1..n], Output: Array S[1..n]

```
for $1 \leq i \leq n$ pardo
    Set S(i) : = P(i)
    while (S(i)) ≠ S(S(i)) do
        Set S(i) : = S(S(i))
par end
```

Analysis: If h is the height of the tree with n nodes, the number of parallel steps [parallel run time] is $O(\log_2 h)$ – since h = O(n) in the worst case, worst case parallel run time is $O(\log_2 n)$. Total work done is clearly $O(n\log_2 h)$ or O(n log n) in the worst case. This clearly not work optimal since a linear work sequential algorithm exists. The algorithm requires concurrent-read capability because different nodes could have the same *S value* – it is a CREW PRAM algorithm.

Let us consider a slight variation of the problem.

Parallel Prefix

Let us assume next that each node *i in the forest F contains a weight W(i). The* pointer jumping technique can be also used to compute, for each node *i*, *the* sum of the weights stored in the nodes on the path from the node *i to the root* of i's root. [Note: This computation amounts to generating the prefix sums of several sequences of elements such that each sequence is given by the order of the nodes as they appear on each path]

```
for $1 \le i \le n$ **pardo**
    **Set** $S(i) := P(i)$ & **if** $(P(i) = P(P(i)) \wedge (i \neq 1)$ **then** $W(i) = W(i) + W(P(i))$
    **while** $(S(i) \neq S(S(i))$ **do**
        Set $W(i) := W(i) + W(S(i)$
        Set $S(i) := S(S(i))$
```

Parallel Run time is O($\log_2 n$) and Work done is O(n log n) – not work optimal.

Note: An important special case is when each tree is just a path represented by a **linked list**. The problem of computing prefix sums on linked lists is called **parallel prefix**. Why is it different from our previous prefix sum on a given array?

Test

- Write Something; Pradip Srimani MWrite Something; Pradip Srimani Monai, Rono
- onai, Rono
- Ha Ha Really; Pradip Srimani Monai, Rono
- Write Something; Pradip Srimani Monai, Rono
- ; Pradip Srimani Monai, Rono
- Write Something; Pradip Srimani Monai, Rono

Write Something; Pradip Srimani Monai, Rono
- ~~Write Something~~; Pradip Srimani Monai, Rono
- Write Something; Pradip Srimani Monai, Rono
- pradip