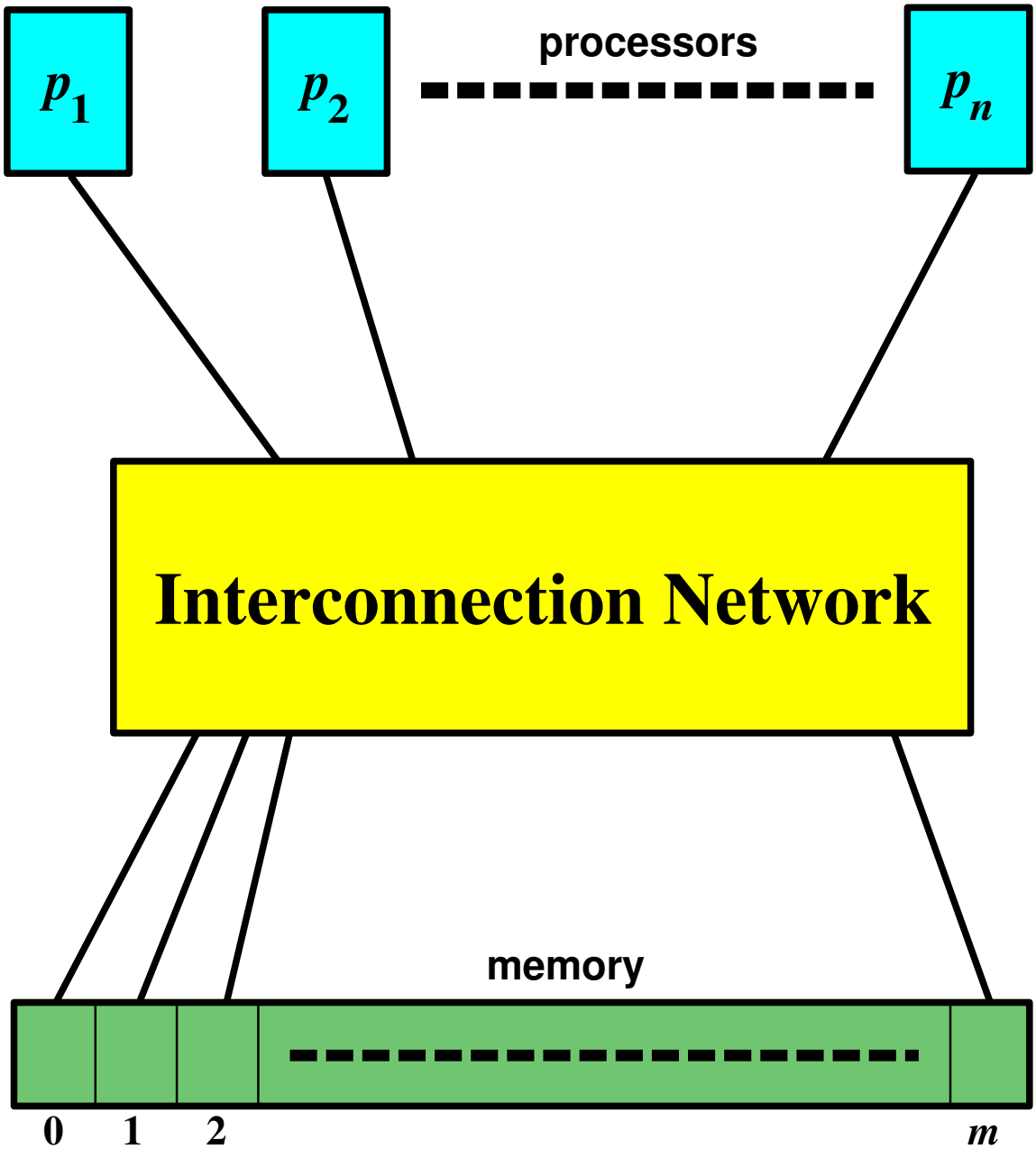
The PRAM Model

- **PRAM** = **P**arallel **R**andom **A**ccess **M**achine.
- A PRAM consists of
  1. p general-purpose processors p_1, p_2, ..., p_p.
  2. Processors are connected to a shared random access memory.
  3. Each processor has a private (or local) memory, program counter, and an unique ID (*i.e.*, i in p_i)
  4. Communications among processors must go through the shared memory
  5. Each "time" step has three phases: **(1)** the **read** phase reads information from a memory location, **(2)** the **computation** phase performs necessary computation for that step, and **(3)** the **write** phase writes information back to the shared memory.
  6. All processors are **synchronized**. They begin each step at the same time and all processors write at the same time.
  7. **PRAM is primarily an abstract programming model, rather than a machine designed to be built.**

The PRAM Model



- **The EREW Model**
  1. EREW: Exclusive-Read Exclusive-Write.
  2. Only one processor can have access to the same location at the same time.
- **The CREW Model**
  1. CREW: Concurrent-Read Exclusive-Write.
  2. Only one processor can write to the same location at the same time. However, more than one processors can read from the same location at the same time.
- **CRCW Model**
  1. CRCW: Concurrent-Read Concurrent-Write.
  2. Allows reading from and writing to the same location at the same time.
- **Resolving Write-Conflicts for CRCW:**
  1. **COMMON:** only allows all processors to write the **same** information to the same location at the same time. Otherwise, the algorithm halts.
  2. **ARBITRARY:** allows an arbitrary value to be written.
  3. **PRIORITY:** The processor with lowest (or highest) index always succeeds.

- **Input/Output Convention:**
  1. Let $\mathcal{P}$ be a PRAM and its memory be numbered as M_0, M_1, ...
  2. The input to $\mathcal{P}$ is a string $x \in \{0,1\}^n$ (*i.e.*, a length n string of 0's and 1's).
  3. The value n is placed in M_0, and the i-th symbol of x is placed in M_i.
  4. The output of $\mathcal{P}$ follows the same rule. Since the output is $y \in \{0,1\}^m$ (*i.e.*, a length m string of 0's and 1's), m is placed in M_0 and the i-th symbol of y is placed in M_i.
- **PRAM $\mathcal{P}$ computes in <u>parallel time</u> $t(n)$ and <u>processor</u> $p(n)$ if for every input $x \in \{0,1\}^n$, $\mathcal{P}$ halts within at most $t(n)$ time steps, activates at most $p(n)$ processors, and generates some output $y \in \{0,1\}^*$.**
- **PRAM $\mathcal{P}$ computes in <u>sequential time</u> $t(n)$ if it computes in parallel time $t(n)$ with <u>one</u> processor.**

Circuit Families

- **A *family of circuit* is an <u>infinite</u> sequence $\mathcal{C} = \{C_0, C_1, C_2, \dots\}$ of Boolean circuits, where C_i has i input variables.**
- **Why a "family"?**
  In our previous simulation of Turing machines with Boolean circuits, we need to know the length of the input, and, as a result, the simulating Boolean circuit is built with respect to the length of a particular input string. Therefore, a family of Boolean circuits can handle input strings of all possible length.
- **A family $\mathcal{C} = \{C_0, C_1, C_2, \dots\}$ of Boolean circuits is *uniform* if there is a log-space bound Turing machine $\mathcal{M}$ which generates circuit C_n on input 1^n.**
- **Why "uniform"?** Non-uniform families can compute undecidable languages! Here, "uniform" also means that the component Boolean circuits are "created" by the same Turning machine $\mathcal{M}$.

Functions Computed by Circuit Families

- **Suppose $\mathcal{C} = \{C_0, C_1, C_2, \ldots\}$ is a Boolean circuit family. Each circuit C_i of this family computes a function $f_i : \{0,1\}^i \mapsto \{0,1\}^{m(i)}$, where $m(i)$ is the number of output bits that is a function of the number of input bits i. The *function computed by family* $\mathcal{C}$, denoted as $f_{\mathcal{C}}$, is the function**

$$f_{\mathcal{C}} : \{0,1\}^* \mapsto \{0,1\}^*$$

defined by $f_{\mathcal{C}}(x) = f_{|x|}(x)$.

1. Intuitively, circuit C_i takes an input string of length i and computes an output string of $m(i)$.
2. The length of the output string depends on the length of the input string.
3. An input string x is supplied to function $f_{|x|}$ that accepts input strings of length $|x|$. Then, $f_{|x|}$ computes an output, which is the output of the family.

Two Important Theorems

- **The functions $f : \{0,1\}^* \mapsto \{0,1\}^*$ computed by uniform Boolean circuit families of polynomial-size and poly-logarithmic depth are the same as those computed by the CREW PRAMs with a polynomial number of processors and poly-logarithmic time.**
- **If a P-complete language can be computed in poly-logarithmic time with polynomial many processors on a CREW PRAM, then so all languages in P, and all languages in P are fully parallelizable.**
- Note that "poly-logarithmic" means $O(\log_2^k |x|)$ for some constant k.

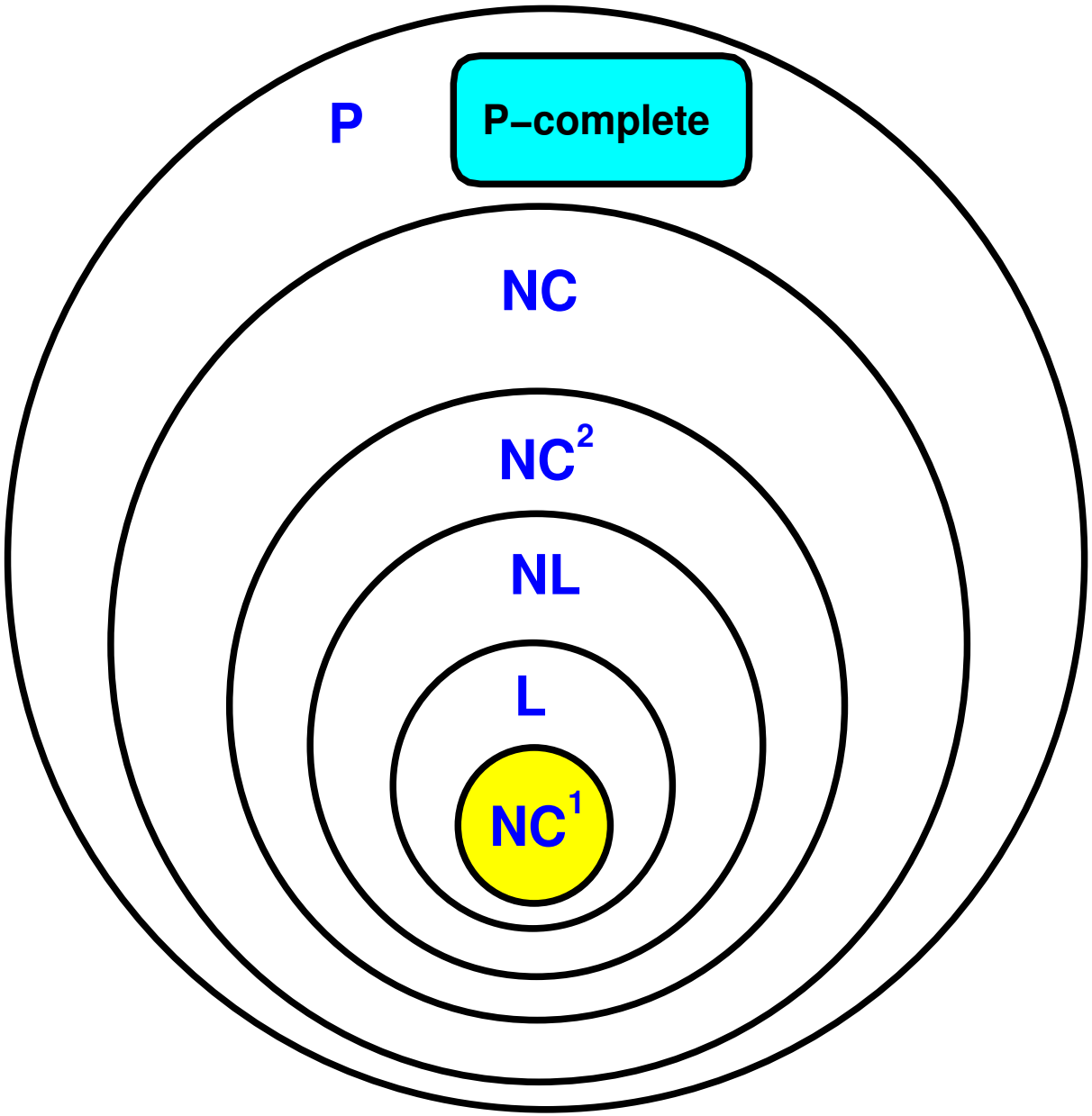
Efficiently Parallelizable Languages

- **The class $\mathbf{NC}^k$ contains all languages L accepted by a uniform family of Boolean circuits of polynomial size in $|x|$ and depth $O(\log_2^k |x|)$, where x is an input string.**
- **The class NC is the union of all $\mathbf{NC}^k$, $k \geq 1$:**

$$\mathbf{NC} = \bigcup_{k \geq 1} \mathbf{NC}^k$$

- Steven Cook coined the name **NC** for "Nick's class" because Nick Pippenger was the first person to recognize its importance.
- **Cook-Pippenger Thesis: A language is *highly parallel* if there exists a PRAM that accepts it in poly-logarithmic time and polynomial number of processors.**
- Thus, all languages in **NC** are highly parallelizable.
- **For $k \geq 2$, $\mathbf{NC}^1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}^2 \subseteq \mathbf{NC}^k \subseteq \mathbf{NC} \subseteq \mathbf{P}$**
- **Another big question: $\mathbf{P} \stackrel{?}{=} \mathbf{NC}$**

Important Hierarchy between P and NC



Speedup and Efficiency

- **Running Time of a Parallel Algorithm:** $T(n, p)$, where n is the input size and p is the number of processors used.
- **Speedup Ratio:**

$$S(p) = \frac{T(n, 1)}{T(n, p)}$$

  This measures how fast a PRAM algorithms is, compared with a sequential one.
- **Perfect Speedup:** The best case is, of course, $S(p) = p$. That is, using p processors is p times faster than using 1 processor.
- **Efficiency:** $E(n, p)$

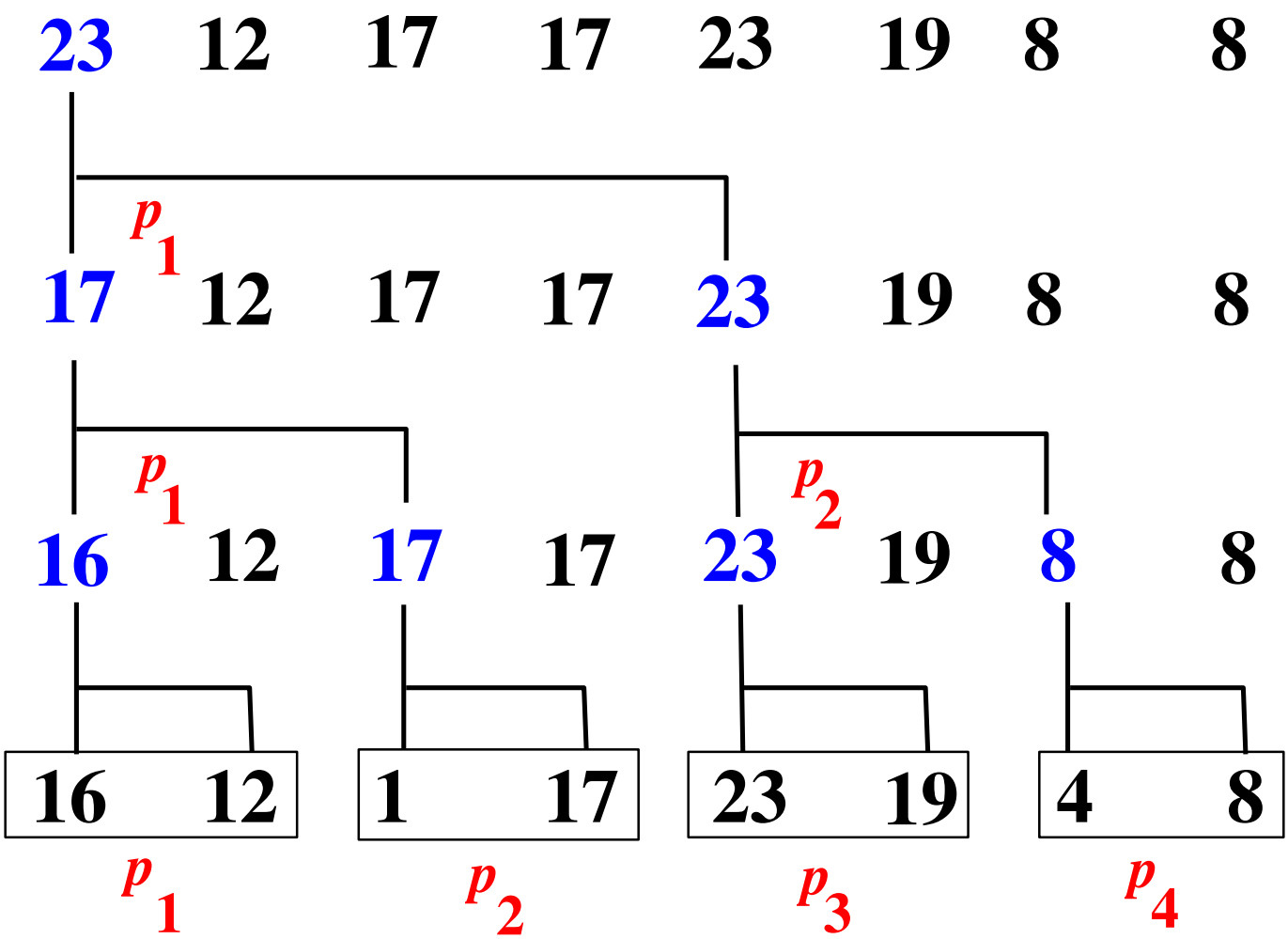
$$E(n, p) = \frac{S(p)}{p} = \frac{T(n, 1)}{pT(n, p)}$$

  It is the quotient of dividing the speedup ratio by the number of processors used. $E(n, p)$ is usually less than 1 since processor time may be wasted. The best case is $E(n, p) = 1$.

Finding Maximum (EREW Model)

- **<u>Problem:</u> Given an array of n integers, find the maximum.**
- This problem can be solved using $\lfloor n/2 \rfloor$ processors in $\lceil \log_2 n \rceil$ steps. Thus, $T(n, \lfloor n/2 \rfloor) = \lceil \log_2 n \rceil$.
- A straightforward sequential algorithm can solve this problem in $n-1$ comparisons, $T(n, 1) = n - 1$. Thus, the efficiency measure is $E(n, n/2) \approx 1/\log_2 n$
- **What if we do not have $n/2$ processors?**
  1. Subdivide the input into $n/\log_2 n$ groups, with about $\log_2 n$ elements per group.
  2. Each group is assigned with a processor to find the maximum of that group. It takes $n/\log_2 n$ steps sequentially.
  3. Then, use the EREW algorithm for finding the maximum of these $n/\log_2 n$ "maximums." It takes less than $\log_2 n$ steps.
  4. The total number of steps is $T(n, \lceil n/\log_2 n \rceil) \approx 2\log_2 n$ steps. So, we have managed to keep the time complexity within $O(\log_2 n)$ steps!

Finding Maximum
EREW Model



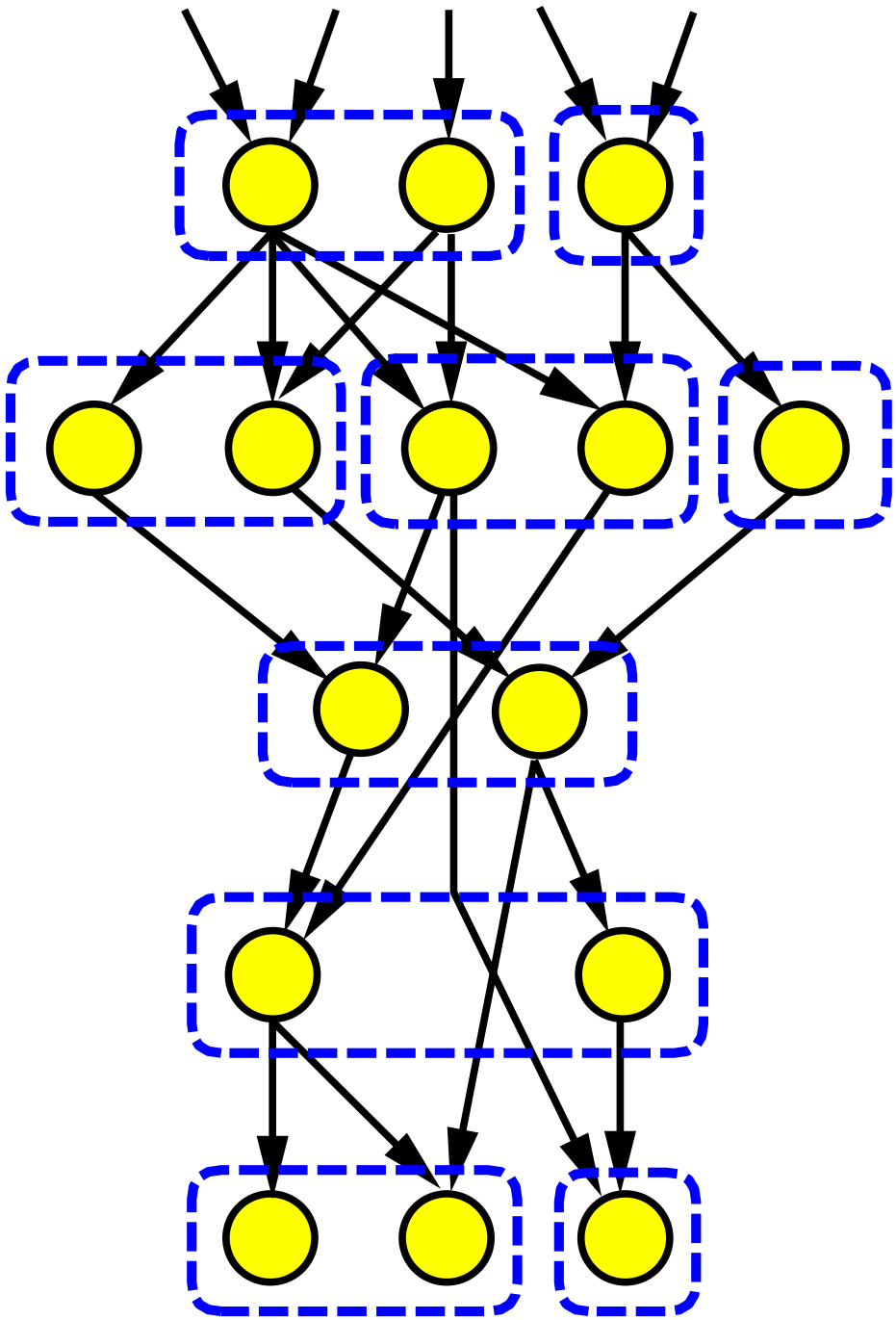
Brent's Lemma

- **A parallel algorithm is *static* if the assignment of processors to actions is predefined.**
- **Brent's Lemma: If there exists an EREW static algorithm with $T(n,p) = O(t)$, such that the total number of steps (over all processors) is s, then there exists an EREW static algorithm with $T(n, s/t) = O(t)$.**
- **Proof:**
  1. Let a_i be the *total* number of computations by *all* processors in step i. Then, $s = \Sigma_{i=1}^{t} a_i$.
  2. Suppose we want to execute the same algorithm with s/t processors.
  3. If $a_i \leq \frac{s}{t}$, there are enough processors for step i.
  4. Otherwise, the ith step is replaced with $\lceil a_i/(s/t) \rceil$ steps in which the available s/t processors emulate the steps taken by the p processors in the original.
  5. The total number of steps becomes:

$$\sum_{i=1}^{t} \left\lceil \frac{a_i}{(s/t)} \right\rceil \leq \sum_{i=1}^{t} \left\lceil \frac{a_i \cdot t}{s} + 1 \right\rceil t + \frac{t}{s} \sum_{i=1}^{t} a_i = 2t$$

Brent's Lemma

step	*work*
1	$a_1 = 3$
2	$a_2 = 5$
3	$a_3 = 2$
4	$a_4 = 2$
5	$a_5 = 3$



total work = 15 = 3 + 5 + 2 + 2 + 3
processors = 2

Finding Maximum (CRCW Model)

- Let the input be **distinct** values $a_1, a_2, \ldots, a_n$. Let the work array be $v_1, v_2, \ldots, v_n$.
- v_i's are initialized to 1's.
- We shall use $n(n-1)/2$ processors. Processor P_{ij} is for data pair a_i and a_j.
- It takes *two* steps to find the maximum:
  1. Processor P_{ij} compares a_i and a_j and puts a 0 to v_i (*resp.*, v_j) if a_i (*resp.*, a_j) is smaller.
  2. Let processors $P_{11}, P_{12}, \ldots, P_{1n}$ be associated with $v_1, v_2, \ldots, v_n$, respectively. If a processor sees its value, say v_i, being 1, it outputs a_i.
- Since the input values are distinct, there is only one 1 in v_i's and the output maximum is unique.
- The time complexity is $T(n, n(n-1)/2) = O(1)$!
- Note that there are many processors writing 0's to the same location of v_i.
- Efficiency: $E(n, n(n-1)/2) = 1/(n-1)$.
- Does "distinct input" really matter?

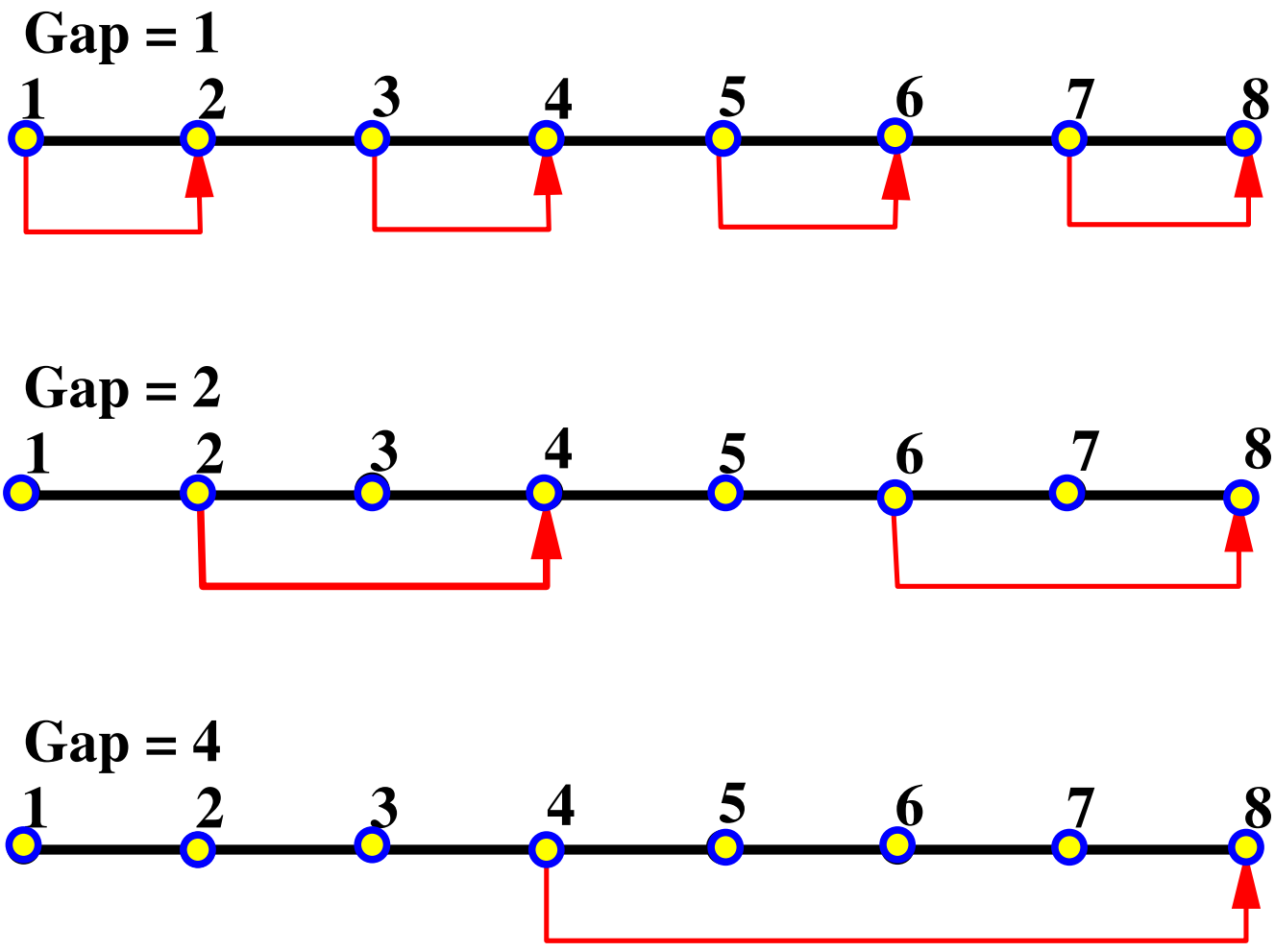
Parallel Prefix (EREW Model)

- **<u>Problem:</u> Given a_1, a_2, ..., a_n in an array, compute $p_k = \Sigma_{i=1}^{k} a_i$ for all $k = 1, 2, \ldots n$, where n is a power of 2 (for the sake of convenience).**
- The operator does not have to be "+", it could be multiplication, maximum/minimum of numbers, and other associative operators.
- **Algorithm (Doubling):** Start with *Prefix*(1).

```
procedure Prefix(Δ)
begin
  if Δ = n/2 then
    a_n := a_{n/2} + a_n;
  else
    begin
      for i := 1 to n/(2Δ) parallel do
          a_{2iΔ} := a_{(2i-1)Δ} + a_{2iΔ};
      Prefix(2Δ); /* recursion here */
      for i := 1 to n/(2Δ) - 1 parallel do
          a_{(2i+1)Δ} := a_{2iΔ} + a_{(2i+1)Δ};
    end
end
```

Parallel Prefix Computation
EREW Model

Doubling Technique



- **Complexity:**
  1. No. of processors: $n/2$
  2. No. of steps: $O(\log_2 n)$
  3. $T(n, n/2) = \log_2 n$
  4. Speedup $S(n/2) = T(n,1)/T(n,n/2) = n/\log_2 n$.
  5. Efficiency $E(n, n/2) = 2/\log_2 n$
- **Speedup using Brent's Lemma:**
  1. The total number of computations satisfies $R(n) = R(n/2) + (n-1)$ and $R(2) = 1$
  2. Solving for $R(n)$ yields $R(n) = O(n)$.
  3. By Brent's Lemma, we can modify the algorithm to use $O(n/\log_2 n)$ processors so that the running time is still $O(\log_2 n)$.
  4. The new algorithm has time complexity $T(n, n/\log_2 n) = O(\log_2 n)$ and hence speedup ratio becomes $O(1)$.

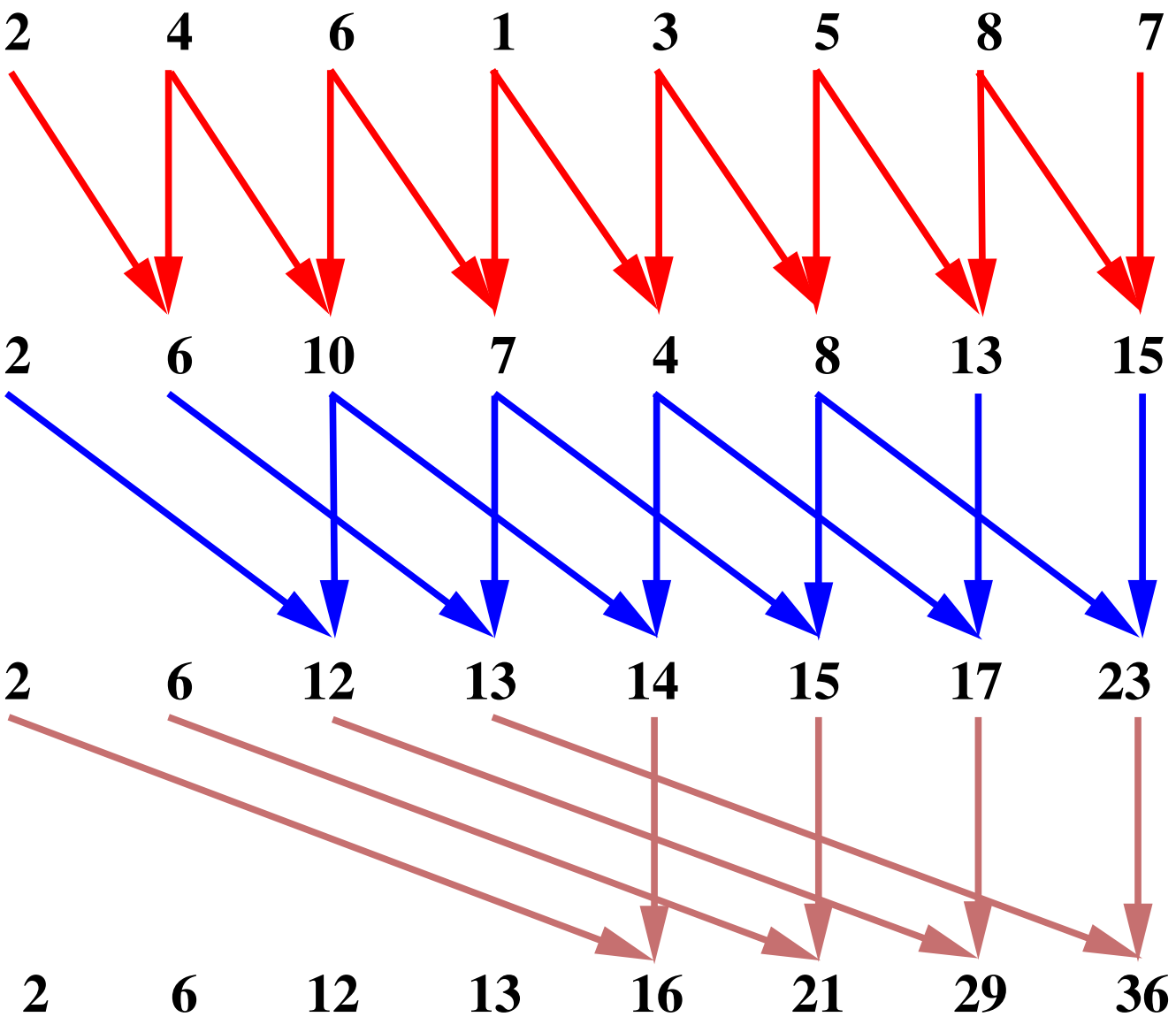
Parallel Prefix 2 (EREW Model)

- Here is another more straightforward point of view of doubling:

```
procedure Prefix
begin
  for i := 0 to n − 1 parallel do
    s_i := x_i; /* make a copy */
  for j = 0 to (log_2 n) − 1 do
    for i = 2^j to n − 1 parallel do
      s_i := s_{i−2^j} + s_i;
  for i := 0 to n − 1 parallel do
    x_i := s_i; /* write back */
end
```

Parallel Prefix Computation

Doubling Technique 2

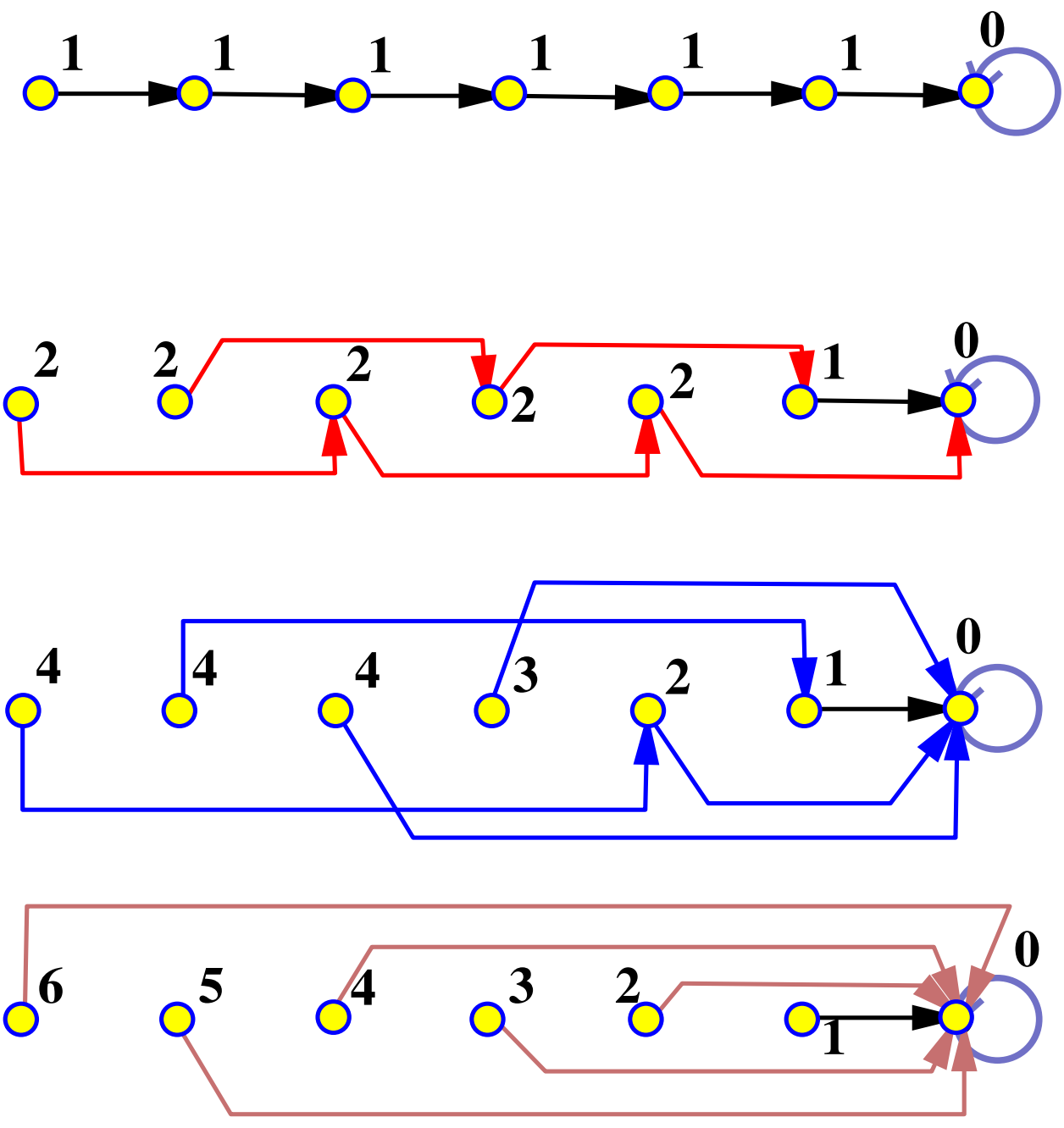


List Ranking

- **List Ranking: Given a singly linked-list L, find the distance from each node to the tail (or head) node.**
- We assume the tail node points to itself.

```
procedure ListRanking
begin
  for all k ∈ L parallel do
    begin
      P(k) := next(k); /* make a local copy */
      if P(k) ≠ k then
        distance(k) := 1 /* non-tail */
      else
        distance(k) := 0 /* tail */
    end
  repeat log2 n times
    for all k ∈ L parallel do
      if P(k) ≠ P(P(k)) then
        begin
          distance(k) := distance(k) + distance(P(k))
          P(k) := P(P(k))
        end
  for all k ∈ L parallel do
    rank(k) := distance(k)
end
```

List Ranking
Doubling Technique 3



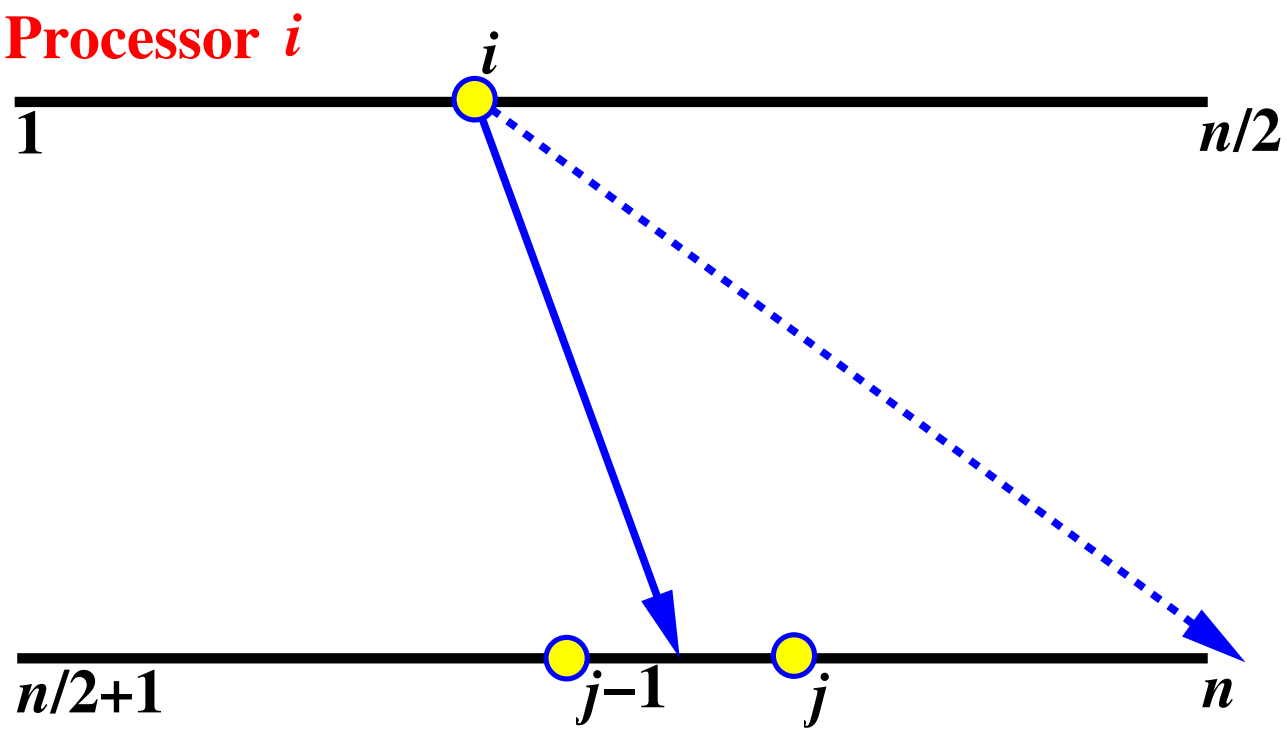
Parallel Merging (EREW Model)

- **<u>Problem:</u> Merge two sorted arrays a_1, a_2, ..., $a_{n/2}$ and $a_{n/2+1}$, $a_{n/2+2}$, ..., a_n.**

procedure *Merge*
begin
  for $i := 1$ **to** n **parallel do**
    processor P_i reads a_i into local memory x
  for $i := 1$ **to** n **parallel do**
    if $i \leq n/2$ **then**
      begin
        processor P_i does a binary search in $a_{n/2+1}$, ..., a_n;
        if a smallest j such that $x < a_{n/2+j}$, let $p_i = i + j - 1$
        otherwise, let $p_i = n/2 + i$
      end
    else
      begin
        processor P_i does a binary search in a_1, ..., $a_{n/2}$;
        if a smallest j such that $x < a_j$, let $p_i = i + j - 1$
        otherwise, let $p_i = n/2 + i$
      end
  for $i := 1$ **to** n **parallel do**
    processor i writes its x to location a_{p_i}
end

- Number of processors: n; time complexity: $O(\log_2 n)$

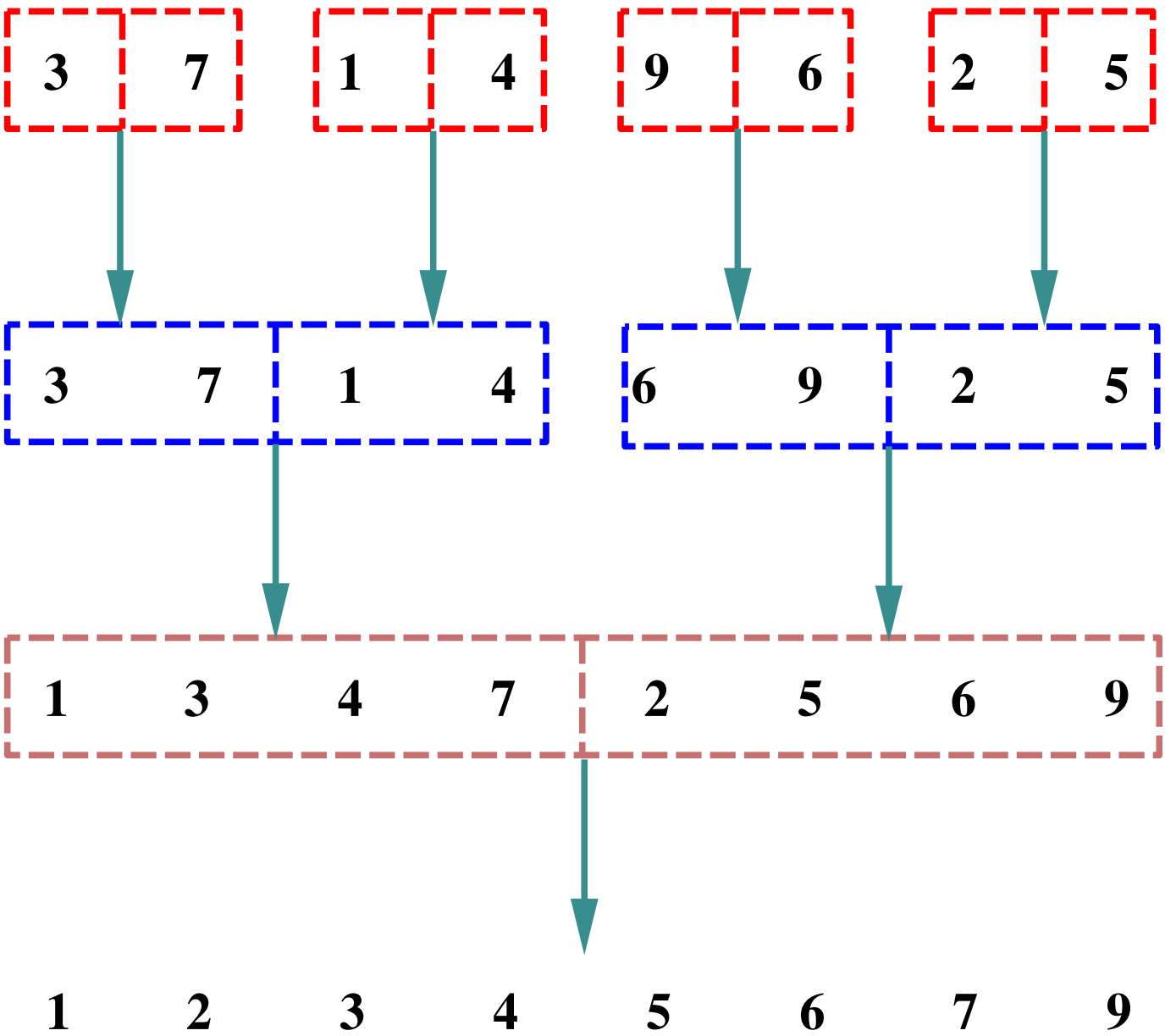
Parallel Merging
EREW Model



Parallel Sorting (EREW Model)

- **Given a list of n keys a_1, a_2, ..., a_n, sort them in nondecreasing order.**
- The list should be "expanded" by appending enough number of maximum values so that n is a power of 2.

```
procedure ParallelMergeSort
begin
  for i := 1 to ⌈log2 n⌉ do
    begin
      k := 2^(i-1); /* size of sublists to be merged */
      processors P_i, ..., P_(i+2k-1) merge the two
          lists of size k at a_i;
    end
end
```

- **Complexity:** Since each merge phase needs $O(\log_2 n)$ steps and sorting requires $O(\log_2 n)$ steps, the time complexity is $T(n, n) = O((\log_2 n)^2)$.
- The current best, actually optimal, algorithm is due to Richard Cole. Its time complexity is $T(n, n) = O(\log_2 n)$ and hence optimal.

Parallel Merge Sort