CSE 613: Parallel Programming

Lecture 14
(Distributed-Memory Algorithms: Sorting & Searching)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2017

Parallel QuickSort: A Shared-Memory Version

Input: An array $A[\,q : r\,]$ of distinct elements.

Output: Elements of $A[\,q : r\,]$ sorted in increasing order of value.

Par-Randomized-Looping-QuickSort ($A[\,q : r\,]$)

1. $m \leftarrow r - q + 1$
2. *if* $m > 1$ *then*
3. $\quad k \leftarrow 0$
4. $\quad$ *while* $\max\{\, r - k,\ k - q \,\} > 3m \,/\, 4$ *do*
5. $\qquad$ select a random element x from $A[\,q : r\,]$
6. $\qquad k \leftarrow$ *Par-Partition* ($A[\,q : r\,],\ x$)
7. $\quad$ *spawn* *Par-Randomized-Looping-QuickSort* ($A[\,q : k - 1\,]$)
8. $\quad$ *Par-Randomized-Looping-QuickSort* ($A[\,k + 1 : r\,]$)
9. $\quad$ *sync*

Parallel QuickSort: Distributed-Memory Version

Input: An array $A[\ q : r\ ]$ of distinct elements distributed among processing nodes $P_s, P_{s+1}, \ldots, P_t$ such that each nodes contains between $\alpha/2$ and 2α elements, where $\alpha = n\ /\ p$ = orig #elems / orig #nodes.

Output: Elements of $A[\ q : r\ ]$ sorted in increasing order of value distributed among the nodes in the following order: $P_s, P_{s+1}, \ldots, P_t$.

Distributed-Randomized-Looping-QuickSort ($A[\ q : r\ ],\ \alpha,\ s,\ t$)

1. *if* $s = t$ *then* sort $A[\ q : r\ ]$ locally on P_s using serial quicksort
2. *else*
3. $\quad m \leftarrow r - q + 1,\ \ k \leftarrow 0$
4. $\quad$ *while* $\max\{\ r - k,\ k - q\ \} > 3m\ /\ 4$ *do*
5. $\quad\quad$ select a random element x from $A[\ q : r\ ]$
6. $\quad\quad$ $k \leftarrow$ *Distributed-Rank* ($A[\ q : r\ ],\ x\ ,\ s,\ t$)
7. $\quad$ Find an i, and redistribute $A[\ q : r\ ]$ among the nodes as evenly as possible such that
   (a) all elements $\leq x$ are stored among nodes P_s to P_i ,
   (b) all elements $> x$ are stored among nodes P_{i+1} to P_t , and
   (c) no node stores fewer $\alpha/2$ or more than 2α elements
7. $\quad$ *parallel:* *Distributed-Randomized-Looping-QuickSort* ($A[\ q : k\ ],\ \alpha,\ s,\ i$)
   $\quad\quad$ *Distributed-Randomized-Looping-QuickSort* ($A[\ k + 1 : r\ ],\ \alpha,\ i + 1,\ t$)

Distributed QuickSort: Example

Image Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition

Distributed QuickSort: Example

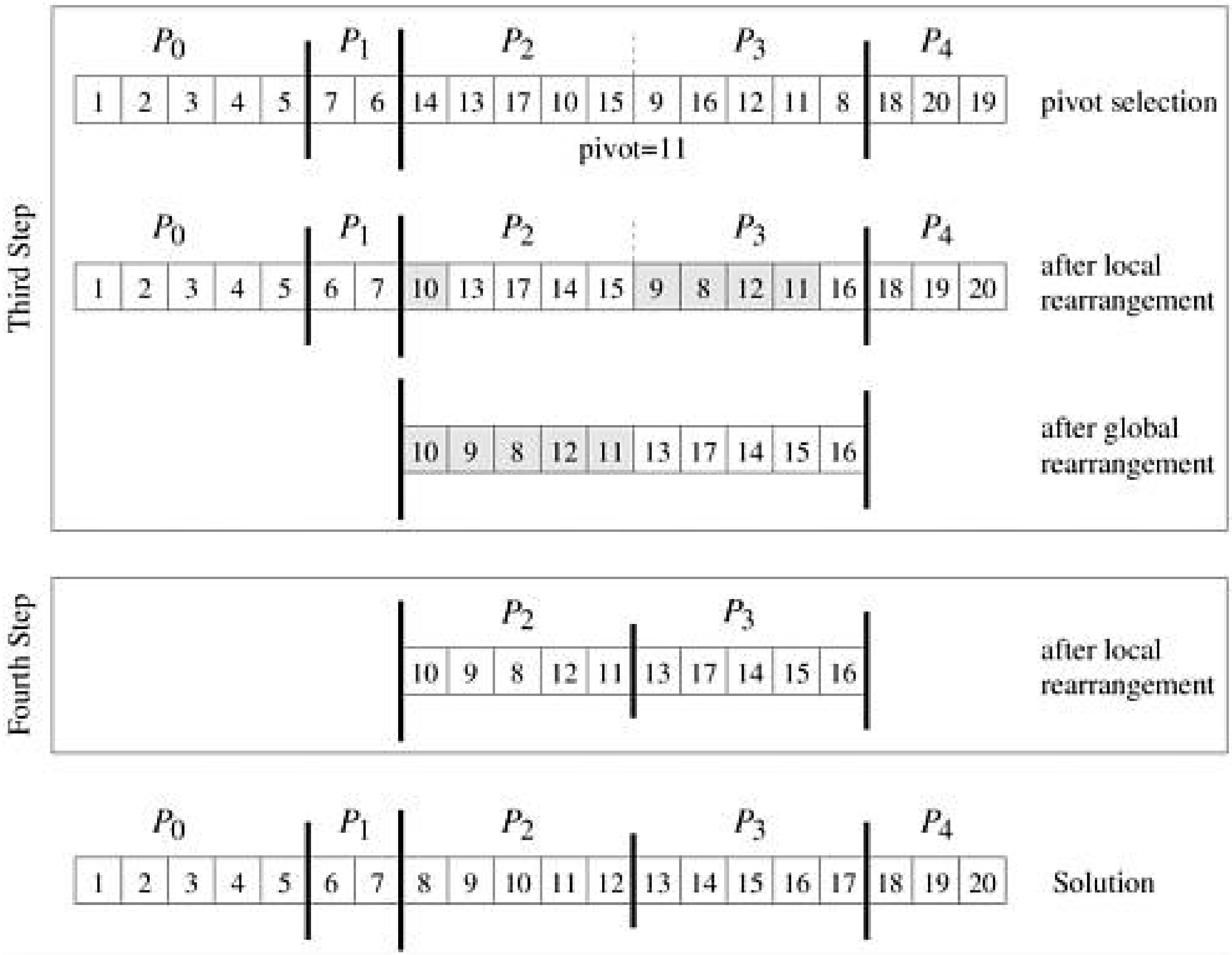

Image Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Distributed QuickSort: Distributed Rank & Partition

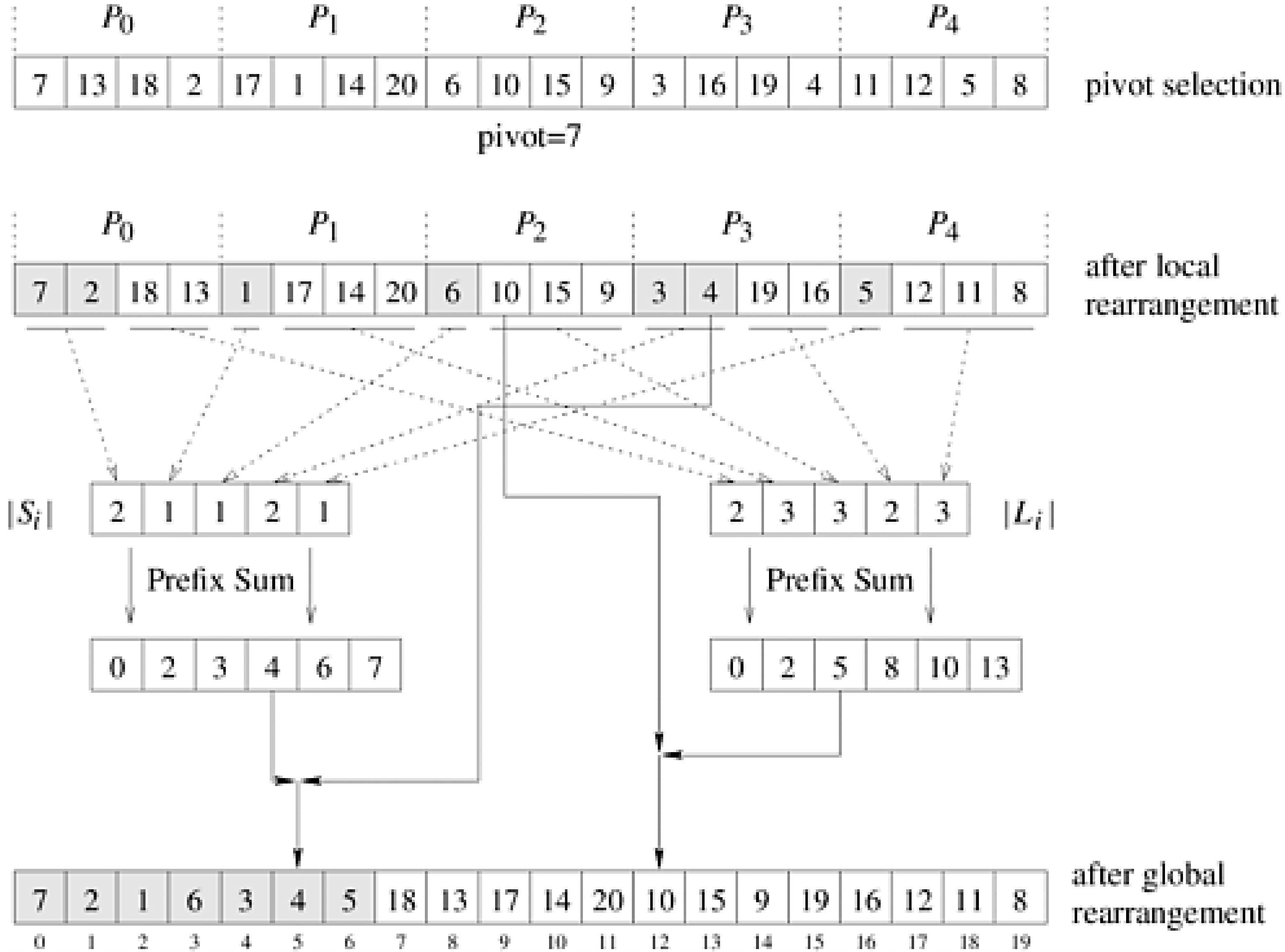

Image Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Distributed QuickSort

Distributed-Randomized-Looping-QuickSort ($A[\, q : r \,]$, α, s, t)

1. *if* $s = t$ *then* sort $A[\, q : r \,]$ locally on P_s using serial quicksort
2. *else*
3. $m \leftarrow r - q + 1$, $k \leftarrow 0$
4. *while* $\max\{\, r - k, k - q \,\} > 3m \,/\, 4$ *do*
5. select a random element x from $A[\, q : r \,]$
6. $k \leftarrow$ *Distributed-Rank* ($A[\, q : r \,]$, x , s, t)
7. Find an i, and redistribute $A[\, q : r \,]$ among the nodes as evenly as possible such that
   (a) all elements $\leq$ x are stored among nodes P_s to P_i ,
   (b) all elements > x are stored among nodes P_{i+1} to P_t , and
   (c) no node stores fewer than $\alpha/2$ or more than 2α elements
8. *parallel:* *Distributed-Randomized-Looping-QuickSort* ($A[\, q : k \,]$, α, s, i)
   Distributed-Randomized-Looping-QuickSort ($A[\, k + 1 : r \,]$, α, $i + 1$, t)

Lines 5-6 (assuming t_s and t_w to be constants)

– communication complexity $= O(p + \log p)$ (why?)

– computation complexity $= O\left(\frac{n}{p}\right)$ (why?)

– overall $= O\left(\frac{n}{p} + p + \log p\right)$

Distributed QuickSort

Distributed-Randomized-Looping-QuickSort ($A[\,q : r\,]$, α, s, t)

1. *if* $s = t$ *then* sort $A[\,q : r\,]$ locally on P_s using serial quicksort
2. *else*
3. $m \leftarrow r - q + 1$, $k \leftarrow 0$
4. *while* $\max\{\, r - k,\ k - q \,\} > 3m / 4$ *do*
5. select a random element x from $A[\,q : r\,]$
6. $k \leftarrow$ *Distributed-Rank* ($A[\,q : r\,]$, x, s, t)
7. Find an i, and redistribute $A[\,q : r\,]$ among the nodes as evenly as possible such that
   (a) all elements $\leq$ x are stored among nodes P_s to P_i ,
   (b) all elements $>$ x are stored among nodes P_{i+1} to P_t , and
   (c) no node stores fewer than $\alpha/2$ or more than 2α elements
8. *parallel:* *Distributed-Randomized-Looping-QuickSort* ($A[\,q : k\,]$, α, s, i)
   Distributed-Randomized-Looping-QuickSort ($A[\,k + 1 : r\,]$, α, $i + 1$, t)

Line 7 (assuming t_s and t_w to be constants)

- communication complexity $= O\left(p + \log p + \frac{n}{p}\right)$ (why?)
- computation complexity $= O(1)$ (why?)
- overall $= O\left(\frac{n}{p} + p + \log p\right)$

Distributed QuickSort

Distributed-Randomized-Looping-QuickSort ($A[\,q : r\,]$, α, s, t)

1. *if* $s = t$ *then* sort $A[\,q : r\,]$ locally on P_s using serial quicksort
2. *else*
3. $m \leftarrow r - q + 1$, $k \leftarrow 0$
4. *while* $\max\{\, r - k,\ k - q \,\} > 3m / 4$ *do*
5. select a random element x from $A[\,q : r\,]$
6. $k \leftarrow$ *Distributed-Rank* ($A[\,q : r\,]$, x , s, t)
7. Find an i, and redistribute $A[\,q : r\,]$ among the nodes as evenly as possible such that
   (a) all elements $\leq$ x are stored among nodes P_s to P_i ,
   (b) all elements > x are stored among nodes P_{i+1} to P_t , and
   (c) no node stores fewer $\alpha/2$ or more than 2α elements
8. *parallel:* *Distributed-Randomized-Looping-QuickSort* ($A[\,q : k\,]$, α, s, i)
   Distributed-Randomized-Looping-QuickSort ($A[\,k + 1 : r\,]$, α, $i + 1$, t)

Depth of the shared-memory version is $O(\log n)$ w.h.p.

Same bound applies to the distributed-memory version.

Hence, $T_p = O\left(\left(\frac{n}{p} + p + \log p\right)\log n\right) = O\left(\frac{n\log n}{p} + p\log n\right)$ (w.h.p.)

Distributed Sample Sort

Task: Sort n distinct keys using p processing nodes.

Steps:

1. **Initial Distribution:** The master node scatters the n keys among p processing nodes as evenly as possible.
2. **Pivot Selection:** Each node sorts its local keys, and selects $q - 1$ evenly spaced keys from its sorted sequence. The master node gathers these *local pivots* from all nodes, locally sorts those $p(q-1)$ keys, selects $p - 1$ evenly spaced global pivots from them, and broadcasts them to all nodes.
3. **Local Bucketing:** Each node inserts the global pivots into its local sorted sequence using binary search, and thus divides the keys among p buckets.
4. **Distribute Local Buckets:** For $1 \le i \le p$, each node sends bucket i to node i.
5. **Local Sort:** Each node locally sorts the elements it received in step 4.
6. **Final Collection:** The master node collects all sorted keys from all nodes, and for $1 \le i < p$, places all keys from node i ahead of all keys from node $i + 1$.

Bound on Bucket Sizes

Theorem: If each node selects $q - 1$ evenly spaced keys in step 2, then no node will sort more than are $\frac{n}{p} + \frac{n}{q}$ keys (in the worst case) in step 5.

Proof: Will be shown in the class.

Analyzing Distributed Sample Sort

Steps: (assuming $q = \Theta(p)$, and t_s and t_w constants)

1. **Initial Distribution:** $O\left(\log p + \frac{n}{p}(p-1)\right) = O(n + \log p)$ **[comm: scatter]**

2. **Pivot Selection:** $O\left(\frac{n}{p}\log\frac{n}{p} + pq\log(pq)\right) = O\left(\frac{n}{p}\log\frac{n}{p} + p^2\log p\right)$ **[comp: sort]**

   $O(\log p + (q-1)(p-1) + (p-1)\log p) = O(p^2)$ **[comm: gather, broadcast]**

3. **Local Bucketing:** $O\left((p-1)\log\frac{n}{p}\right) = O(p\log n)$ **[comp: binary search]**

4. **Distribute Local Buckets:** $O\left(\frac{n}{p} + \left(\frac{n}{p} + \frac{n}{q}\right)\right) = O\left(\frac{n}{p}\right)$ **[comm: send, receive]**

5. **Local Sort:** $O\left(\left(\frac{n}{p} + \frac{n}{q}\right)\log\left(\frac{n}{p} + \frac{n}{q}\right)\right) = O\left(\frac{n}{p}\log\frac{n}{p}\right)$ **[comp: sort]**

6. **Final Collection:** $O\left((p-1)\left(\frac{n}{p} + \frac{n}{q}\right)\right) = O(n)$ **[comm: receive]**

<u>Analyzing Distributed Sample Sort</u>

Overall:

$$t_{comp} = O\left(\frac{n}{p}\log\frac{n}{p} + p^2\log p + p\log n\right)$$

$$t_{comm} = O(n + p^2)$$

$$T_p = t_{comp} + t_{comm} = O\left(n + \frac{n}{p}\log\frac{n}{p} + p^2\log p + p\log n\right)$$

Overall (excluding steps 1 and 6):

$$t_{comp} = O\left(\frac{n}{p}\log\frac{n}{p} + p^2\log p + p\log n\right)$$

$$t_{comm} = O\left(\frac{n}{p} + p^2\right)$$

$$T_p = t_{comp} + t_{comm} = O\left(\frac{n}{p}\log\frac{n}{p} + p^2\log p + p\log n\right)$$

Depth-First Search (DFS)

Input: A directed/undirected graph $G = (V, E)$ with vertex set $V = \{1,2, \dots, n\}$ and edge set E. For each $v \in V$, the adjacency list of v is given by the ordered set $adj[v]$. Vertex 1 is the root of G.

Output: An array $dfn[1:n]$, where for each $v \in V$, $dfn[v]$ gives the rank of v in the order the algorithm visits the vertices of G.

```
DFS-Numbering ( G = ( V, E ), dfn[ 1: n ] )
    k ← 0
    for v ← 1 to n do dfn[ v ] ← 0
    for v ← 1 to n do
        DFS ( v )        { DFS is a local function }

DFS ( v )
    if dfn[ v ] = 0 then
        k ← k + 1
        dfn[ v ] ← k
        for each u ∈ adj[ v ] in given order do
            DFS ( u )
```

Depth-First Search (DFS)

```
DFS-Numbering ( G = ( V, E ), dfn[ 1: n ] )
  1.  k ← 0
  2.  for v ← 1 to n do dfn[ v ] ← 0
  3.  for v ← 1 to n do
  4.          DFS ( v )        { DFS is a local function }

DFS ( v )
  1.  if dfn[ v ] = 0 then
  2.          k ← k + 1
  3.          dfn[ v ] ← k
  4.          for each u ∈ adj[ v ] in given order do
  5.              DFS ( u )
```

Producing DFS numbers (i.e., $dfn[1: n]$) can be shown to be an *inherently sequential* process.

We will see how to perform distributed parallel DFS on a tree when the DFS numbering is not required.

Parallel DFS on a Tree

Static partitioning of the search space among processing nodes may lead to significant load imbalance.

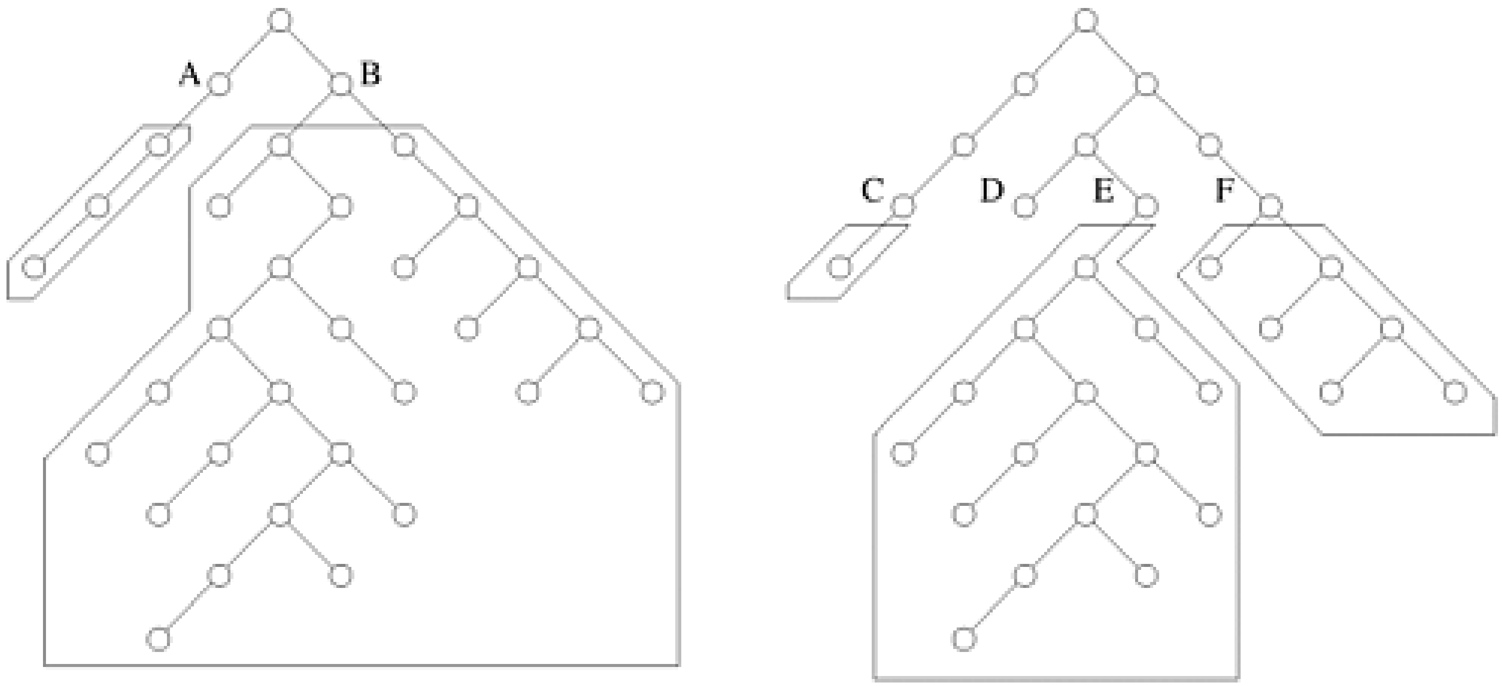

Image Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition

Dynamic partitioning leads to better load balancing.

A Generic Scheme for Dynamic Load Balancing

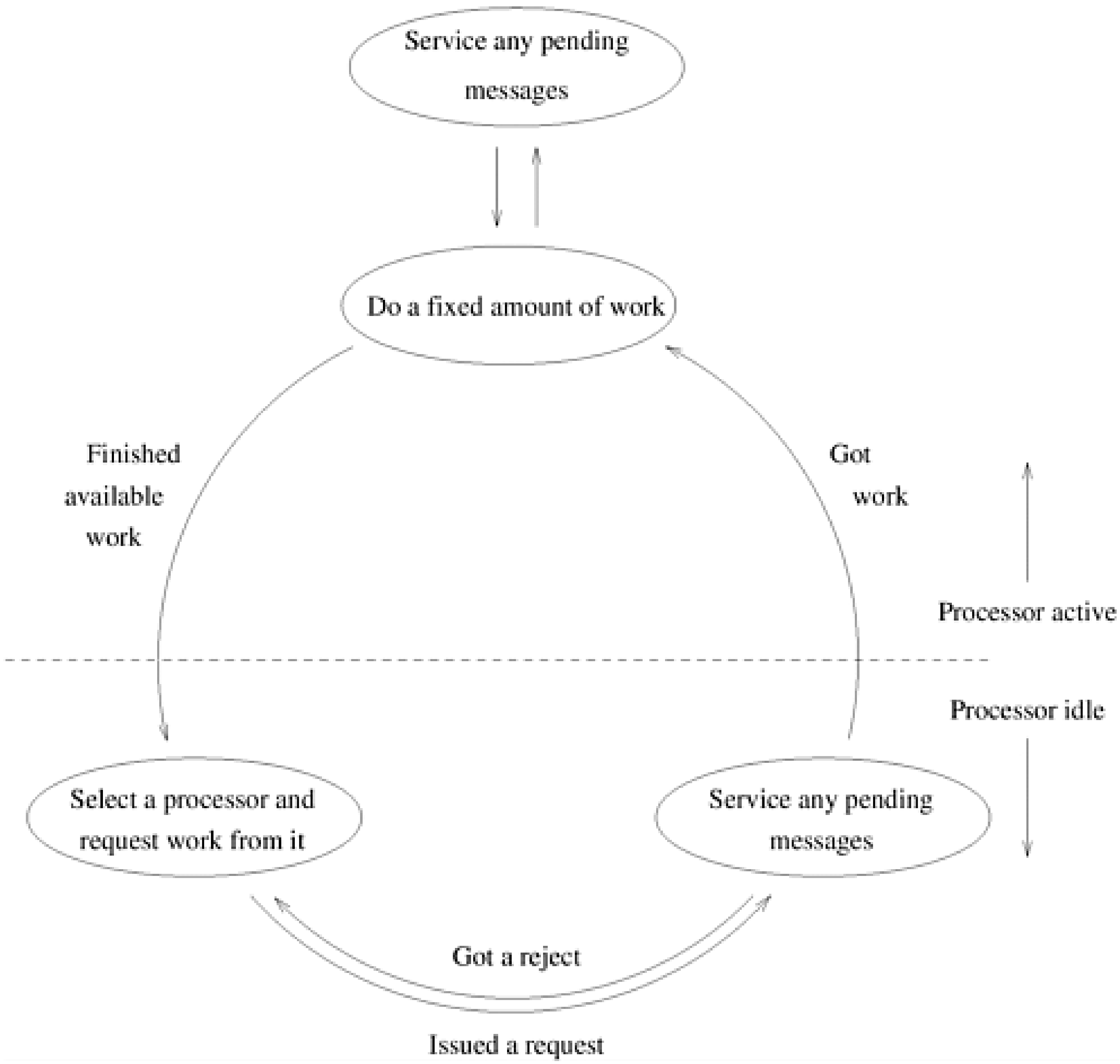

Image Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition

DFS Maintains a Stack of Unvisited Vertices

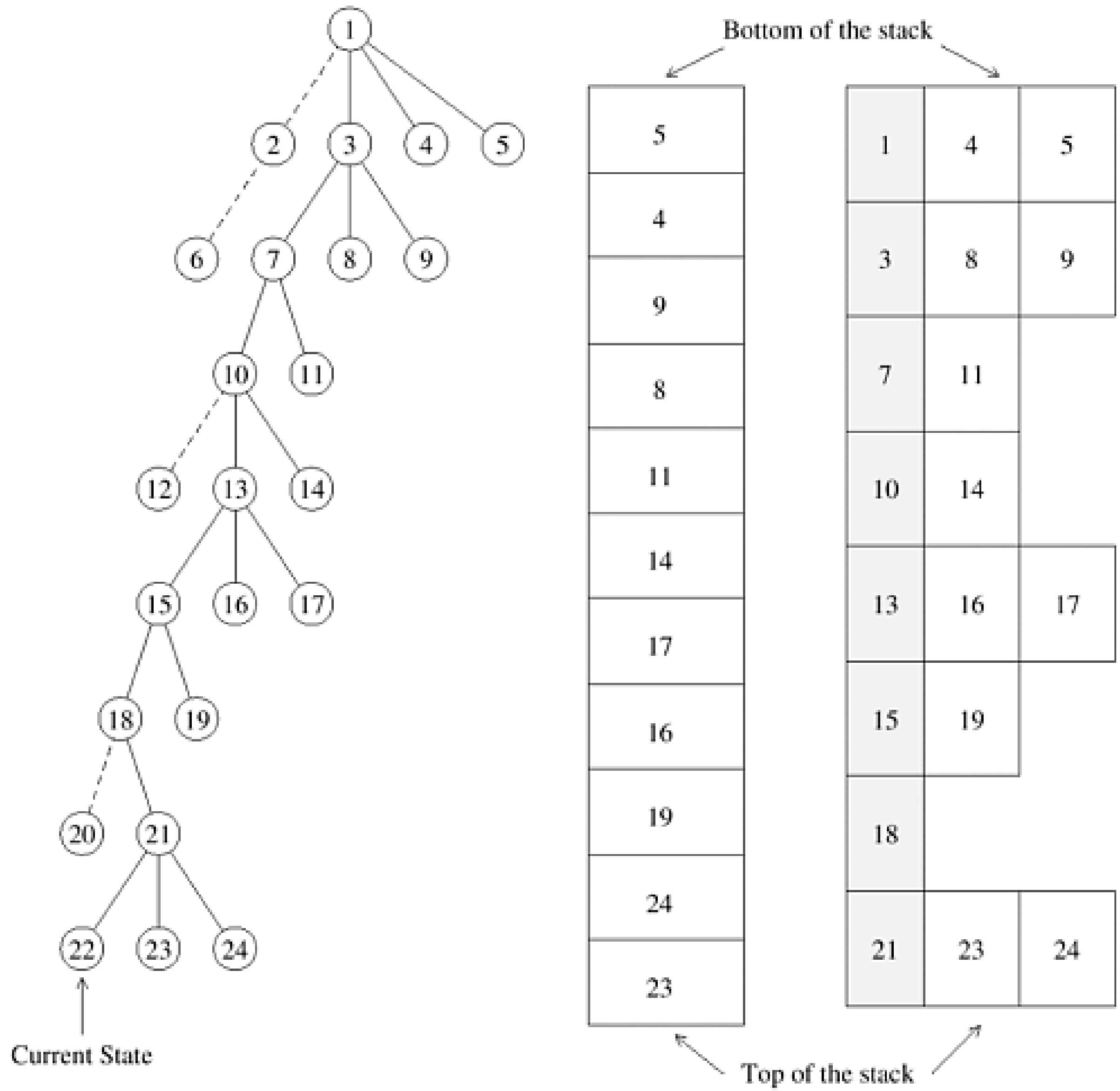

Image Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Work-Splitting Strategies

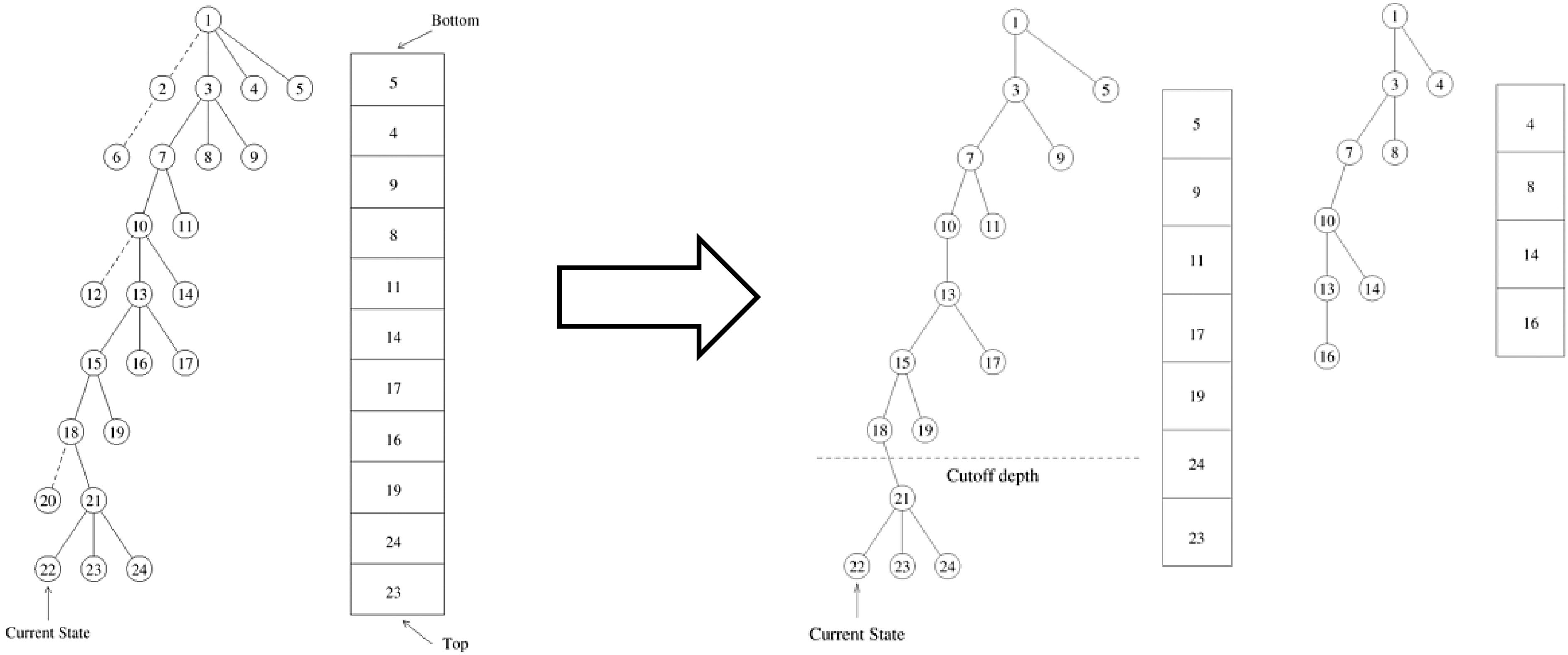

Image Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition

— Ideally the donors stack is split into two pieces such that the search space represented by each is the same. The recipient gets one piece.

— Vertices near the bottom of the stack tend to have bigger trees rooted at them while those near the top tend to have smaller trees.

— To avoid sending very small amounts work, vertices beyond a specified stack depth (called *cutoff depth*) are not given away.

Work-Splitting Strategies

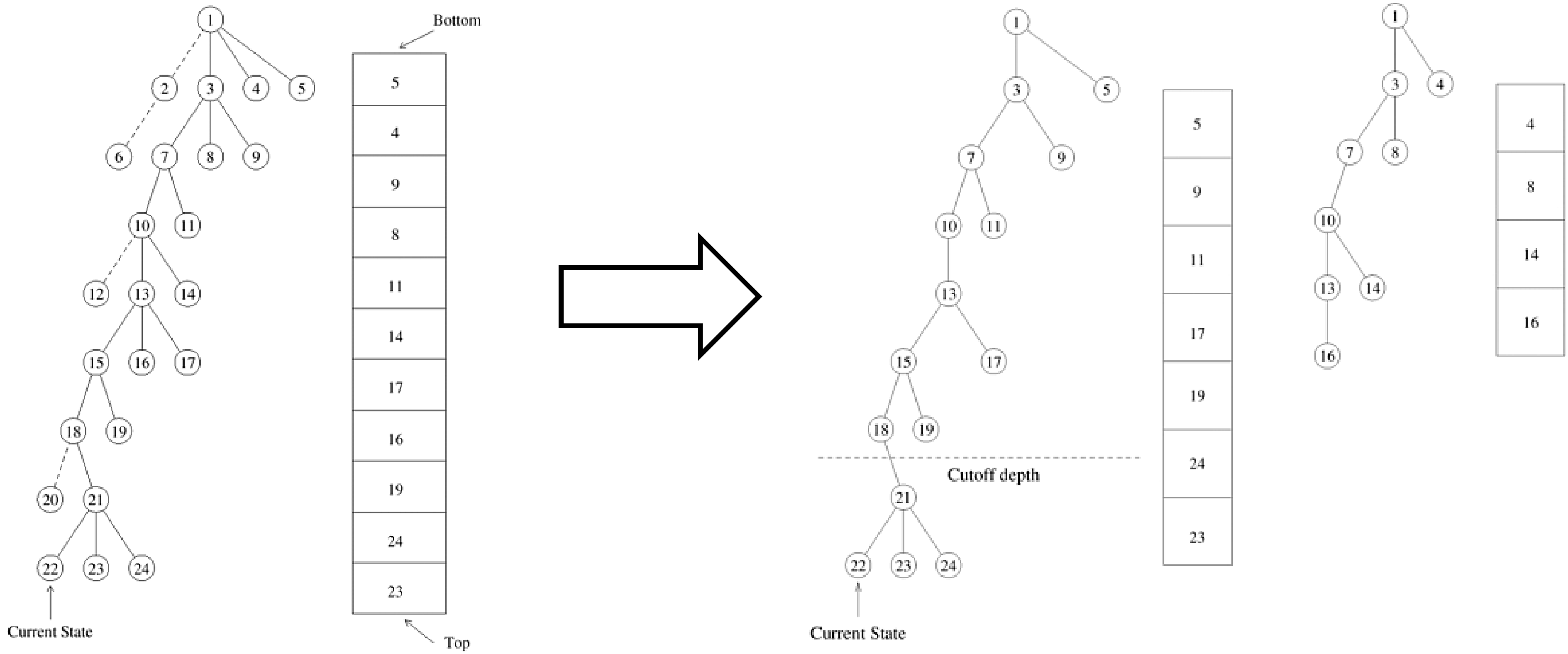

Image Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition

Some possible splitting strategies.

1. Send vertices near the bottom of the stack.
2. Send vertices near the cutoff depth.
3. Send half the vertices between the bottom of the stack and the cutoff depth.

Load Balancing Schemes

Asynchronous Round Robin (ARR)

— Each processor has an independent variable $target$ initialized to $(label + 1) \bmod p$, where $label$ is the local processors label.

— When the processor runs out of work, it attempts to get work from the processor with label $target$, and increments $target$ to $(target + 1) \bmod p$.

Global Round Robin (GRR)

— All processors access a single global variable called $target$.

— Whenever a processor needs work it gets hold of the variable $target$, and tries to get work from a processor whose label is the value of $target$.

— $target$ is set to $(target + 1) \bmod p$ before another processor gets hold of it.

Random Polling (RP)

— When a processor becomes idle it tries to get work from a processor selected uniformly at random.

Communication Overhead of Load Balancing

Assumptions

— **Too small to partition:** Work of size $\leq \epsilon$ are not partitioned.

— **α-splitting:** If work w is partitioned into two parts of size δw and $(1-\delta)w$ for some $0 \leq \delta \leq 1$, then there exists an arbitrarily small constant α $(0 < \alpha \leq 0.5)$, such that $\delta w > \alpha w$ and $(1-\delta)w > \alpha w$.

After such a split neither processor (donor and recipient) has more than $(1-\alpha)w$ work.

Communication Overhead of Load Balancing

Analysis

- Suppose after every $V(p)$ work requests each processor receives at least one work request.
- Suppose initially, only one processor has W amount of work, and all other processors are idle.
- Then after $V(p)$ requests no processor will have more than $(1-\alpha)W$ work.
- After $kV(p)$ requests no processor will have more than $(1-\alpha)^k W$ work.
- So, no processor will have more than ϵ work, after $\left(\log_{\frac{1}{1-\alpha}} \frac{W}{\epsilon}\right) V(p) = O(V(p) \log W)$ work requests.
- Number of work transfers $\leq$ number of work requests.
- For simplicity assume that the data associated with a work request and work transfer is constant.
- If t_c is the time required to communicate a piece of work, then the communication overhead, $T_o = O(t_c V(p) \log W)$.

Computation of $V(p)$

Asynchronous Round Robin (ARR)

— Worst case when all request work from the same processor simultaneously.

— **Worst-case scenario:** Processor $p-1$ has all work, and all other processors are pointing to processor 0. Then processor $p-1$ will get its first work request after one processor issues $p-1$ requests and the remaining $p-2$ processors issue $p-2$ requests each.

— Hence, $V(p) \leq (p-1) + (p-2)(p-2) = O\left(p^2\right)$.

Global Round Robin (GRR)

— All processors receive requests in sequence. Hence, $V(p) = p$.

Random Polling (RP)

— **Need to solve the following balls & bins problem:** Suppose there are p bins, and balls are being thrown into random bins (chosen independently and uniformly at random). How many balls need to be thrown to make sure that each bin gets at least one ball? This number is $V(p)$.

Computation of $V(p)$

Random Polling (RP)

— **Need to solve the following balls & bins problem:** Suppose there are p bins, and balls are being thrown into random bins (chosen independently and uniformly at random). How many balls need to be thrown to make sure that each bin gets at least one ball? This number is $V(p)$.

Let X be #balls thrown until each bin received at least one ball.

Also let X_i be #balls thrown when there were exactly $i-1$ nonempty bins.

Then $X = \sum_{1 \le i \le p} X_i$.

When there are $i-1$ nonempty bins, the probability that a ball will fall into an empty bin is $p_i = 1 - \frac{i-1}{p}$.

So, X_i is a geometric random variable with parameter p_i, and $E[X_i] = \frac{1}{p_i} = \frac{p}{p-i+1}$.

Hence, $E[X] = E\left[\sum_{1 \le i \le p} X_i\right] = \sum_{1 \le i \le p} E[X_i]$

$$= \sum_{1 \le i \le p} \left(\frac{p}{p-i+1}\right) = p \sum_{1 \le i \le p} \left(\frac{1}{i}\right) = pH(p) = p \ln p + \Theta(p)$$

Here, $H(p) = \sum_{1 \le i \le p} \left(\frac{1}{i}\right) = \ln p + \Theta(1)$ is known as the *harmonic number*.

Hence, the expected value of $V(p)$ is $p \ln p + \Theta(p)$. [$< 2p \ln p$ w.h.p.]

Termination Detection (Tree-Based)

How do we know when all processes have become idle?

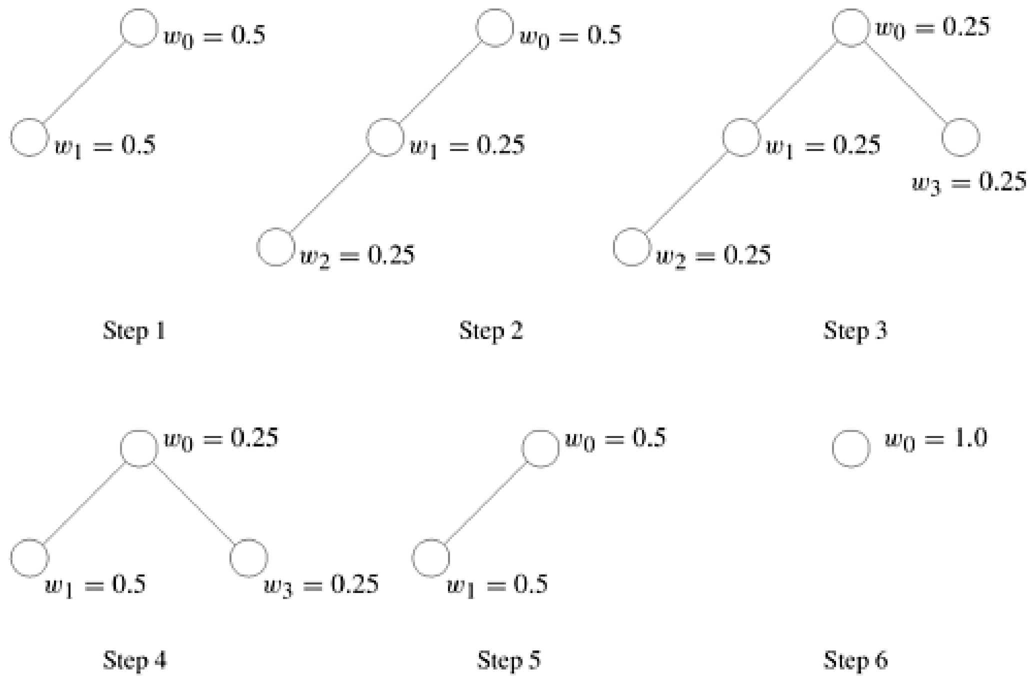

Image Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Suppose, initially, only processor 0 has any work. We set $w_0 = 1$, and $w_i = 0$ for $i > 0$.

When processor i's work is partitioned it retains half of w_i, and gives the other half to the recipient processor.

When a processor completes its work it returns its weight from which it received its work.

Termination is signaled when $w_0 = 1$ and processor 0 is idle.

<u>Termination Detection (Tree-Based)</u>

How do we know when all processes have become idle?

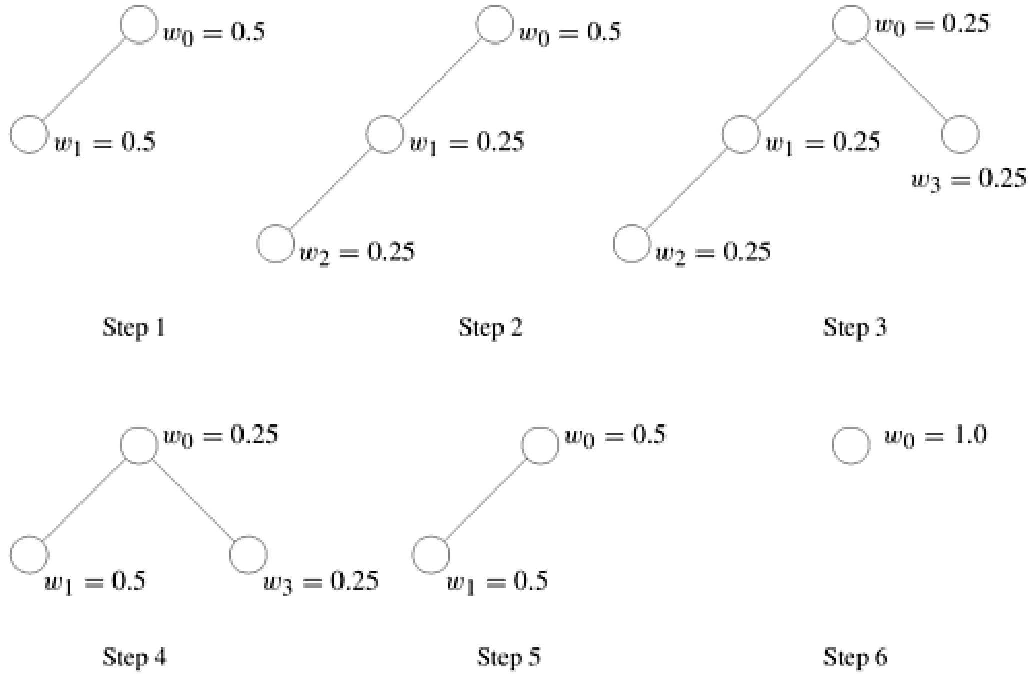

Image Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Drawback: Due to the finite precision of computers, repeated halving of the weight may make the weight so small that it becomes 0.

Solution: Manipulate $\frac{1}{w_i}$ instead of w_i.