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.     $k \leftarrow 0$
4.     while max{ $r - k$, $k - q$ } > $3m / 4$ do
5.         select a random element $x$ from $A[q : r]$
6.         $k \leftarrow Par-Partition ( A[q : r], x )$
7.     spawn Par-Randomized-Looping-QuickSort ( A[q : k - 1] )
9.     sync
```
Parallel QuickSort: Distributed-Memory Version

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

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

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

1. **if** $s = t$ **then** sort $A[q:r]$ locally on $P_s$ using serial quicksort
2. **else**
3. \[m \leftarrow r - q + 1, \ k \leftarrow 0\]
4. **while** $\max\{r - k, k - q\} > 3m/4$ **do**
5. \[\text{select a random element } x \text{ from } A[q:r]\]
6. \[k \leftarrow \text{Distributed-Rank} \ (A[q:r], x, s, t)\]
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 $2\alpha$ elements
7. **parallel:** Distributed-Randomized-Looping-QuickSort $\ (A[q:k], \alpha, s, i)$

\[
\text{Distributed-Randomized-Looping-QuickSort} \ (A[k+1:r], \alpha, i+1, t)
\]
Distributed QuickSort: Example

Distributed QuickSort: Example

Distributed QuickSort: Distributed Rank & Partition

<table>
<thead>
<tr>
<th></th>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>13</td>
<td>18</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>20</td>
<td>6</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3</td>
<td>16</td>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>12</td>
<td>5</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

pivot selection

pivot = 7

<table>
<thead>
<tr>
<th></th>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>2</td>
<td>18</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>14</td>
<td>20</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>5</td>
<td>12</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

after local rearrangement

<table>
<thead>
<tr>
<th></th>
<th>Sᵢ</th>
<th>Lᵢ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 1 1 2 1 2 3 3 2 3</td>
<td></td>
</tr>
</tbody>
</table>

Prefix Sum

|   | 0 2 3 4 6 7 |
|   | 0 2 5 8 10 13 |

after global rearrangement

Distributed QuickSort

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. \( m \leftarrow r - q + 1, \ k \leftarrow 0 \)
4. while \( \max\{ r - k, k - q \} > 3m / 4 \) do
5. select a random element \( x \) from \( A[q : r] \)
6. \( k \leftarrow \text{Distributed-Rank} (\ A[q : r], x, s, t ) \)
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 \( 2\alpha \) elements
8. parallel: Distributed-Randomized-Looping-QuickSort ( \( A[q : k], \alpha, s, i \) )

\[ \text{Distributed-Randomized-Looping-QuickSort} (A[k + 1 : r], \alpha, i + 1, t) \]

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** \( (A[q:r], \alpha, s, t) \)

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. \( k \leftarrow \text{Distributed-Rank} \ (A[q:r], x, s, t) \)
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 \( 2\alpha \) elements
8. *parallel:* \text{Distributed-Randomized-Looping-QuickSort} \ (A[q:k], \alpha, s, i)
   \text{Distributed-Randomized-Looping-QuickSort} \ (A[k+1:r], \alpha, 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

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

1. \textbf{if} \ s = t \textbf{ then} sort \(A[\ q\ :\ r\ ]\) locally on \(P_s\) using serial quicksort
2. \textbf{else}
3. \(m \leftarrow r - q + 1,\ k \leftarrow 0\)
4. \textbf{while} \(\max\{r - k, k - q\} > 3m / 4\) \textbf{do}
5. \textbf{select a random element} \(x\) from \(A[\ q\ :\ r\ ]\)
6. \(k \leftarrow \text{Distributed-Rank} \ (A[\ q\ :\ r\ ],\ x,\ s,\ t)\)
7. \textbf{Find an} \(i\), \textbf{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 \(P_t\), \textbf{and}
   \(c\) no node stores fewer \(\alpha/2\) or more than \(2\alpha\) elements
8. \textbf{parallel:} \text{Distributed-Randomized-Looping-QuickSort} \ (A[\ q\ :\ k\ ],\ \alpha,\ s,\ i)\quad \text{Distributed-Randomized-Looping-QuickSort} \ (A[\ k\ +\ 1\ :\ r\ ],\ \alpha,\ 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 \( 1 \leq i \leq p \), 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 \( 1 \leq i < p \), 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:** Homework.
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]
Analyzing Distributed Sample Sort

Overall:

\[ t_{\text{comp}} = 0 \left( \frac{n}{p} \log \frac{n}{p} + p^2 \log p + p \log n \right) \]

\[ t_{\text{comm}} = 0(n + p^2) \]

\[ T_p = t_{\text{comp}} + t_{\text{comm}} = 0 \left( n + \frac{n}{p} \log \frac{n}{p} + p^2 \log p + p \log n \right) \]

Overall (excluding steps 1 and 6):

\[ t_{\text{comp}} = 0 \left( \frac{n}{p} \log \frac{n}{p} + p^2 \log p + p \log n \right) \]

\[ t_{\text{comm}} = 0 \left( \frac{n}{p} + p^2 \right) \]

\[ T_p = t_{\text{comp}} + t_{\text{comm}} = 0 \left( \frac{n}{p} \log \frac{n}{p} + p^2 \log p + p \log n \right) \]
Depth-First Search (DFS)

Input: A directed/undirected graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$ 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** ($G = (V, E)$, $dfn[1:n]$)

1. $k \leftarrow 0$
2. for $v \leftarrow 1$ to $n$ do $dfn[v] \leftarrow 0$
3. for $v \leftarrow 1$ to $n$ do
4. \hspace{1em} DFS ($v$) \hspace{1em} \{ DFS is a local function \}

**DFS** ($v$)

1. if $dfn[v] = 0$ then
2. \hspace{1em} $k \leftarrow k + 1$
3. \hspace{1em} $dfn[v] \leftarrow k$
4. \hspace{1em} for each $u \in adj[v]$ in given order do
5. \hspace{2em} 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.
Static partitioning of the search space among processing nodes may lead to significant load imbalance.

Dynamic partitioning leads to better load balancing.

*Image Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition*
A Generic Scheme for Dynamic Load Balancing

DFS Maintains a Stack of Unvisited Vertices

Work-Splitting Strategies

— 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.

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 \textit{target} initialized to \((\text{label} + 1) \mod p\), where \textit{label} is the local processors label.
- When the processor runs out of work, it attempts to get work from the processor with label \textit{target}, and increments \textit{target} to \((\text{target} + 1) \mod p\).

Global Round Robin (GRR)
- All processors access a single global variable called \textit{target}.
- Whenever a processor needs work it gets hold of the variable \textit{target}, and tries to get work from a processor whose label is the value of \textit{target}.
- \textit{target} is set to \((\text{target} + 1) \mod p\) 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.
Assumptions

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

— **$\alpha$-splitting:** If work $w$ is partitioned into two parts of size $\delta w$ and $(1 - \delta)w$ for some $0 \leq \delta \leq 1$, then there exists an arbitrarily small constant $\alpha$ ($0 < \alpha \leq 0.5$), 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 $\epsilon$ 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, $V(p) \leq (p - 1) + (p - 2)(p - 2) = O(p^2)$.

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 $V(p)$. 
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 $X = \sum_{1 \leq i \leq p} X_i$.

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, $E[X] = E[\sum_{1 \leq i \leq p} X_i] = \sum_{1 \leq i \leq p} E[X_i]$

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

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

Hence, the expected value of $V(p)$ is $p \ln p + \Theta(p)$. \[ \leq 2p \ln p \text{ w.h.p.} \]
Termination Detection (Tree-Based)

How do we know when all processes have become idle?

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.
Termination Detection (Tree-Based)

How do we know when all processes have become idle?

**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 \( w_i \).