Parallel Algorithms

- **The PRAM MODEL OF COMPUTATION**
  - Abbreviation for Parallel Random Access Machine
  - Consists of \( p \) processors (PEs), \( P_0, P_1, P_2, \ldots, P_{p-1} \) connected to a shared *global memory*.
  - Can assume w.l.o.g. that these PEs also have local memories to store local data.
  - All processors can read or write to global memory *in parallel*.
  - All processors can perform arithmetic and logical operations *in parallel*
  - *Running time* can be measured as the number of parallel memory accesses.
  - Read, write, and logical and arithmetic operations take constant time.
- **Comments on PRAM model**
  - PRAM can be assumed to be either *synchronous* or *asynchronous*.
    - When synchronous, all operations (e.g., read, write, logical & arithmetic operations) occur in lock step.
  - The global memory support communications between the processors (PEs).
Real parallel computers

- Mapping PRAM data movements to real parallel computers communications:
  - Often real parallel computers have a communications network such as a 2D mesh, binary tree, their combinations, … .
  - To implement a PRAM algorithm on a parallel computer with a network, one must perform the PRAM data movements using this network.
  - Data movement on a network is relatively slow in comparison to arithmetic operations.
  - Network algorithms for data movement are often have a higher complexity that the same data movement on PRAM.
  - In parallel computers with shared memory, the time for PEs to access global memory locations is also much slower than arithmetic operations.
    - Some researchers argue that the PRAM model is impractical, since additional algorithms steps have to be added to perform the required data movement on real machines.
    - However, the constant-time global data movement assumptions for PRAM allow algorithms designers to ignore differences between parallel computers and to focus on parallel solutions that utilize parallelism.
Parallel Computing Performance Evaluation Measures.

- **Parallel running time** is measured in same manner as with sequential computers.
- **Work** (usually called **Cost**) is the product of the parallel running time and the number of PEs.
- **Work** often has another meaning, namely the sum of the actual running time for each PE.
  - Equivalent to the sum of the number of $O(1)$ steps taken by each PE.
- **Work Efficient** (usually **Cost Optimal**): A parallel algorithm for which the work/cost is in the same complexity class as an optimal sequential algorithm.
- To simplify things, we will treat “work” and “cost” as equivalent.

**Memory Access Types**

- PRAM Memory Access Types: Concurrent or Exclusive
  - **EREW** - one of the most popular
  - **CREW** - also very popular
  - **ERCW** - not often used.
  - **CRCW** - most powerful, also very popular.
• **Types of Parallel WRITES** includes:
  – **Common** (Assumed in CLR text): The “write” succeeds only if all processors writing to a common location are all writing the same value.
  – **Arbitrary**: An arbitrary processor writing to a global memory location is successful.
  – **Priority**: The processor with the highest priority writing to a global memory location is successful.
  – **Combination (or combining)**: The value written at a global memory location is the combination (e.g., sum, product, max, etc..) of all the values currently written to that location. This is a very strong CR.

• **Simulations** of PRAM models
  – EREW PRAM can be simulated directly on CRCW PRAM.
  – CRCW can be simulated on EREW with O(log p) slowdown (See Section 30.2).

• **Synchronization** and **Control**
  – All processors are assumed to operate in lock step.
  – A processor can either perform a step or remain idle.
  – In order to detect the termination of a loop, we assume this can be tested in O(1) using a control network.
  – CRCW can detect termination in O(1) using only concurrent writes (section 30.2)
  – A few researchers charge the EREW model O(log p) time to test loop termination (Exercise 30-1-80).
List Ranking

• Assume that we are given a linked list

![Linked List Diagram]

– Each object has a responsible PE.

**Problem:** For each element in the linked list, find its distance from the end.
– If `next` is the pointer field, then

\[
d[i] = \begin{cases} 
0 & \text{if } \text{next}\{i\} = \text{NIL}, \\
\text{d}[\text{next}\{i\}] + 1 & \text{if } \text{next}\{i\} \neq \text{NIL}.
\end{cases}
\]

• Naïve algorithm: Propagate distance from end
  all PEs set `distance = -1`
  the PE with `next = NIL` resets `distance = 0`
  while there is a PE with its `distance = -1`
    the PE with `distance = -1` and `distance(next) > -1`
    sets `distance = distance(next) + 1`

• Above algorithm processes links serially and requires \(O(n)\) time.
• Can we give a better parallel solution? (YES!)
**Algorithm List-Rank(L)**

1. **for each** processor $i$ **in parallel** do
2. \[ \text{if } \text{next}[i] = \text{NIL} \]
3. \[ \text{then } d[i] = 0 \]
4. \[ \text{else } d[i] = 1 \]
5. **while** there is an object $i$ with $\text{next}[i] \neq \text{NIL}$
6. **all** processors $i$ **(in parallel)** do
7. \[ \text{if } \text{next}[i] \neq \text{NIL} \]
8. \[ d[i] \leftarrow d[i] + d[\text{next}[i]] \]
9. \[ \text{next}[i] \leftarrow \text{next}[\text{next}[i]] \]

**O(log p) Parallel Solution**

Reads and Writes in step 8

(synchronization)

- First, all $d[i]$ fields are read in parallel
- Second, all next[i] fields are read in parallel
- Third, all $d[\text{next}[i]]$ fields are read in parallel
- Fourth, all sums are computed in parallel
- Finally, all sums are written to $d[i]$ in parallel

**Correctness**

- **Invariant:** for each $i$, if we add the $d$ values in the sublist headed by $i$, we obtain the correct distance from $i$ to the end of the original list $L$.
- Object ‘splices out’ its successor and adds successor’s $d$-value to its own.
O(log p) Parallel Solution (cont.)

• Note that the pointer fields are changed by pointer jumping, thus destroying the structure of the list. If the list structure must be preserved, then we make copies of the next pointers and use the copies to compute the distance.

• Only EREW needed, as no two objects have equal next pointer (except next[i]=next[j]=NIL).

• Technique called also “recursive doubling”.

• Θ(log n) iterations required.

• Θ(log n) running time, as each step takes Θ(1) time.

• **Cost** = (#PEs) * (running time) = Θ(n log n)

• Serial algorithm takes Θ(n) time:
  • walk list and set reverse pointer for each next pointer
  • walk from end of list backward, computing distance from end.

• Parallel algorithm is not cost optimal
Parallel Prefix Computation

• Let \((x_1, x_2, \ldots, x_n)\) be an input sequence.

• Let \(\otimes\) be a binary, associative operation defined on the sequence elements.

• Let \((y_1, y_2, \ldots, y_n)\) be the output sequence, where

\[
\begin{align*}
y_1 &= x_1 \quad \text{and} \\
y_k &= y_{k-1} \otimes x_k & \text{for} \quad k > 1.
\end{align*}
\]

**Problem:** Compute \((y_1, y_2, \ldots, y_n)\).

• If \(\otimes\) is ordinary addition and \(x_i = 1\) for each \(i\), then

\[y_k = \text{(distance from front of linked list)} + 1.\]

• Distance from end can be found by reversing links and performing same task.

**Notation:** \([i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j\)

• properties

  • the input is \([1,1], [2,2], \ldots, [n,n]\) as \([i,i]= x_i\).

  • the output is \([1,1], [1,2], \ldots, [1,n]\) as \([1,i]= y_i\).

• since operation \(\otimes\) is associative, \([i, j] \otimes [j+1, k] = [i, k]\).
Algorithm List-Prefix(L)
1. for each processor i in parallel do
2. \[ y[i] \leftarrow x[i] \]
3. while there is an object i with next[i] ≠ NIL
4. all processors i (in parallel) do
5. if next[i] ≠ NIL then
6. \[ y[next[i]] \leftarrow y[i] \odot y[next[i]] \]
7. \[ next[i] \leftarrow next[next[i]] \]

Correctness
- At each step, if we perform a prefix computation on each of the several existing lists, each object obtains its correct value.
- Object ‘splices out’ its successor and “adds” its own value to successor’s value.

Comments on algorithm
- Almost the same as link ranking.
- Primary difference is in steps 5-7
- List-Rank: \( P_i \) updates \( d_i \), its own value.
- List-Prefix: \( P_i \) updates \( y[next[i]] \), its successors value.
- The running time is \( \Theta(\log n) \) on EREW PRAM, and the total work is \( \Theta(n \log n) \). (again not cost optimal)
**CREW is More Powerful Than EREW**

- **PROBLEM**: Consider a forest $F$ of binary trees in which each node $i$ has a pointer to its parents. For each node, find the identity of the root node of its tree.

- **ASSUME**:
  - Each node has two pointer fields,
    - $\text{pointer}[i]$, which is a pointer to its parent (is $\text{NIL}$ for a root node).
    - $\text{root}[i]$, which we have to compute (initially has no value).
  - An individual PE is assigned to each node.

- **CREW ALGORITHM: FIND_ROOTS($F$)**

  1. for each processor $i$, do (in parallel)
  2. if $\text{parent}[i] = \text{NIL}$
  3. then $\text{root}[i] = i$
  4. while there is a node $i$ with $\text{parent}[i] \neq \text{NIL}$, do
  5. for each processor $i$ (in parallel)
  6. if $\text{parent}[i] \neq \text{NIL}$,
  7. then $\text{root}[i] \leftarrow \text{root}[\text{parent}[i]]$
  8. $\text{parent}[i] \leftarrow \text{parent}[\text{parent}[i]]$

- **Comments on algorithm**
  - WRITES in lines 3,7,8 are EW as PE$_i$ writes to itself.
  - READS in lines 7-8 are CR, as several nodes may have pointers to the same node.
  - The while loop in lines 4-8 performs pointer jumping and fills in the root fields.

See an example on the next two slides
Example
Figure 30.5  Finding the roots in a forest of binary trees on a CREW PRAM. Node numbers are next to the nodes, and stored root fields appear within nodes. The links represent parent pointers. (a)–(d) The state of the trees in the forest each time line 4 of FIND-ROOTS is executed. Note that path lengths are halved in each iteration.
– **Invariant** (for while): If parent[i] = NIL, then root[i] has been assigned the identity of the node’s root.

– **Running time:** If $d$ is the maximum depth of a tree in the forest, then $FIND_ROOTS$ runs in $O(\log d)$. If the maximum depth is $\log n$, then the running time is $O(\log(\log n))$, which is “essentially constant” time.

– The next theorem shows that no EREW algorithm for this problem can run faster than $FIND_ROOTS$. *(concurrent reads help in this problem)*

**Theorem** (Lower Bound): Any EREW algorithm that finds the roots of the nodes of a binary tree of size $n$ takes $\Omega(\log n)$ time.

– The proof must apply to all EREW algorithms for this problem.

– At each stage, one piece of information can be only copied to one additional location using ER.

– The number of locations that can contain a piece of information at most doubles at each ER step.

– For each tree, after $k$ steps at most $2^{k-1}$ locations can contain the root.

– If the largest tree is $\Theta(n)$, then for each node of this tree to know the root requires that for some constant $c$,

$$2^{k-1} \geq c \cdot n \Rightarrow k-1 \geq \log(c) + \log(n) \Rightarrow$$

or equivalently, $k$ is $\Omega(\log n)$.

**Conclusion:** When the largest tree of forest has $\Theta(n)$ nodes and depth $o(n)$, then $FIND_ROOTS$ is faster than the fastest EREW algorithm for this problem.

– In particular, if the forest consists of one fully balanced binary tree, the $FIND_ROOTS$ runs in $O(\log \log n)$ while any EREW algorithm runs in $\Omega(\log n)$. 
**CRCW is More Powerful Than CREW**

**PROBLEM:** Find the maximum element in an array of real numbers.

**Assume:**
- The input is an array \( A[0 \ldots n-1] \).
- There are \( n^2 \) CRCW PRAM PEs labeled \( P(i,j) \), with \( 0 \leq i,j \leq n-1 \).
- \( m[0 \ldots n-1] \) is an array used in the algorithm.

**CRCW Algorithm \( FAST\_MAX(A) \)**

1. \( n \leftarrow \text{length}[A] \)
2. **for** \( i \leftarrow 0 \) **to** \( n-1 \), **do** (in parallel)
3. \( P(i,0) \) sets \( m[i] = \text{true} \)
4. **for** \( i \leftarrow 0 \) **to** \( n-1 \) **and** \( j \leftarrow 0 \) **to** \( n-1 \), **do** (in parallel)
5. \( \text{if} \ A[i] < A[j] \)
6. \( \text{then} \ P(i,j) \) sets \( m[i] \leftarrow \text{false} \)
7. **for** \( i \leftarrow 0 \) **to** \( n-1 \), **do** (in parallel)
8. \( \text{if} \ m[i] = \text{true} \)
9. \( \text{then} \ P(i,0) \) sets \( \text{max} \leftarrow A[i] \)
10. **return** \( \text{max} \)

**Example:**

<table>
<thead>
<tr>
<th>( A[j] )</th>
<th>( m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>F T F T T</td>
</tr>
<tr>
<td>6</td>
<td>F F F F T</td>
</tr>
<tr>
<td>2</td>
<td>T T F T F</td>
</tr>
<tr>
<td>6</td>
<td>F F F F T</td>
</tr>
</tbody>
</table>

**Algorithms Analysis:**
- Note that Step 5 is a CR and Steps 6 and 9 are CW. When a concurrent write occurs, all PEs writing to the same location are writing the same value.
- Since all loops are in parallel, \( FAST\_MAX \) runs in \( O(1) \) time. The cost = \( O(n^2) \) since there are \( n^2 \) PEs. The sequential time is \( O(n) \), so **not cost optimal**.
It is interesting to note (key to the algorithm):

A CRCW model can perform a Boolean AND of \( n \) variables in \( O(1) \) time using \( n \) processors. Here the \( n \) ANDs were

\[
m[i] = \bigwedge_{j=0}^{n-1} (A[i] \geq A[j])
\]

- This \( n \)-AND evaluation can be used by CRCW PRAM to determine in \( O(1) \) time if a loop should terminate.

- **Concurrent writes help in this problem**

**Theorem** (lower bound): Any CREW algorithm that computes the maximum of \( n \) values requires \( \Omega(\log n) \) time.

- Using the CREW property, after one step each element \( x \) can know how it compares to at most one other element.
- Using the EW property, after one step each element \( x \) can obtain information about at most one other element.

  - No matter how many elements know the value of \( x \), only one can write information to \( x \) in one step.
- Thus, each \( x \) can have comparison information about at most two other elements after one step.
- Let \( k(i) \) denote the largest number of elements that any \( x \) can have comparison information about after \( i \) steps.
Above reasoning gives the recurrence relation
\[ k(i+1) = 3k(i) + 2 \]
since after step \( i \), each element \( x \) could know about a maximum of \( k(i) \) other elements and can obtain information about at most two other elements \( x \) and \( y \), each of which knows about \( k(i) \) elements.

The solution to this recurrence relation is
\[ k(i) = 3^i - 1 \]

In order for the a maximal element \( x \) to be compared to all other elements after step \( i \), we must have
\[
\begin{align*}
k(i) & \geq n - 1 \\
3^i - 1 & \geq n - 1 \\
\log(3^i - 1) & \geq \log(n - 1)
\end{align*}
\]

Since there exists a constant \( c > 0 \) with
\[ \log(x) = c \log_3(x) \]
for all positive real values \( x \),
\[
\begin{align*}
\log(n-1) &= c \log_3(n-1) \\
&\leq c \log_3(3^i - 1) \\
&< c \log_3(3^i) \\
&= c \ i
\end{align*}
\]

Above shows any CREW algorithm takes \( i \) steps
\[ i \geq \frac{\log(n-1)}{c} \]
so \( \Omega(\log n) \) steps are required.

The maximum can actually be determined in \( \Theta(\log n) \) steps using a binary tree comparison, so this lower bound is sharp.
Simulation of CRCW PRAM With EREW PRAM

- Simulation results assume an optimal EREW PRAM sort.
  - An optimal general purpose sequential sort requires $O(n \log n)$ time.
  - A cost optimal parallel sort using $n$ PEs must run in $O(\log n)$ time
  - Parallel sorts running in $O(\log n)$ time exist, but these algorithms vary from complex to extremely complex.
  - The CLR textbook claims to have proved this result [see pgs. 653, 706, 712] are admittedly exaggerated.
    - Presentation depends on results mentioned but not proved in chapter notes on pg. 653.
  - The earlier $O(\log n)$ sort algorithms had an extremely large constant hidden by the $O$-notation (see pg 653)
  - The best-known PRAM search is Cole’s merge-sort algorithm, which runs on EREW PRAM in $O(\log n)$ time with $O(n)$ PEs and has a smaller constant.
    - A proof for CREW model can be found in “An Introduction to Parallel Parallel Algorithms” by Joseph JaJa, Addison Wesley, 1992, pg. 163-173.
  - The best-known parallel sort for practical use is the bitonic sort, which runs in $O(lg^2 n)$ and is due to Professor Kenneth Batcher (See CLR, pg. 642-6).
    - Developed for network, but later converted to run on several different parallel computer models.
• The simulation of CRCW using EREW establishes how much more power CRCW has than EREW.
• Things requiring simulation:
  – Local execution of CRCW PE commands. ← \{easy\}
  – A CR from global memory
  – A CW into global memory ← \{will use arbitrary CW\}

**Theorem:** An \(n\) processor EREW model can simulate an \(n\) processor CRCW model with \(m\) global memory locations in \(O(\log n)\) time using \(m+n\) memory locations.

• Proof:
  – Let \(Q_0, Q_1, \ldots, Q_{n-1}\) be the \(n\) EREW PEs
  – Let \(P_0, P_1, \ldots, P_{n-1}\) be the \(n\) CRCW PEs.
  – We can simulate a concurrent write of the CRCW PEs with each \(P_i\) writing the datum \(x_i\) to location \(m_i\).
  – If some \(P_i\) does not participate in CW, let \(x_i = m_i = -1\)
  – Since \(Q_i\) simulates \(P_i\), first each \(Q_i\) writes \((m_i, x_i)\) into an auxiliary array \(A\) in global memory.

| \(Q_0\) | \(\rightarrow\) | \(A[0]\) |
|\(Q_1\) | \(\rightarrow\) | \(A[1]\) |
| \(\ldots\) | \(\ldots\) | \(\ldots\) |
| \(Q_{n-1}\) | \(\rightarrow\) | \(A[n-1]\) |

\[(m_0, x_0), (m_1, x_1), \ldots, (m_{n-1}, x_{n-1})\]

See an example on the next slide
Figure 30.7 Simulating a concurrent write on an EREW PRAM. (a) A step of a common-CRCW algorithm in which 6 processors write concurrently to global memory. (b) Simulating the step on an EREW PRAM. First, ordered pairs containing location and data are written to an array $A$. The array is then sorted. By comparing adjacent elements in the array, we ensure that only the first of each group of identical writes into global memory is implemented. In this case, processors $P_0$, $P_2$, and $P_5$ perform the write.
• Next, we use an $O(\log n)$ EREW sort to sort $A$ by its first coordinate.
  – All data to be written to the same location are brought together by this sort.
  – All EREW PEs $Q_i$ for $1 \leq i \leq n$ inspect both
    $$A[i] = (m_i, x_i^*) \quad \text{and} \quad A[i-1] = (m_{i-1}, x_{i-1}^*)$$
    where the “*” is used to denote sorted values in $A$.
  – All processors $Q_i$ for which either $m_i \neq m_{i-1}$ or $i=0$ write $x_i^*$ into location $m_i$ (in parallel).
  – This performs an arbitrary value concurrent write.
  – The CRCW PRAM model with combining CW can also be simulated by EREW PRAM in $O(\log n)$ time according to Problem 30.2-9.
  – Since “arbitrary” is weakest CW and combining is the “strongest”, all CWs mentioned in CLR can be simulated by EREW PRAM in $O(\log n)$ time.
  – The simulation of the concurrent read in CRCW PRAM is similar and is Problem 30.2-8.

**Corollary:** A $n$-processor CRCW algorithm can be no more than $O(\log n)$ faster than the best $n$-processor EREW algorithm for the same problem.

**READ** (in CLR) pp. 687-692 and Ch(s) 30.1.1- 30.1.2 and 30.2.
• Suppose that we have a PRAM algorithm that uses at most $p$ processors, but we have a PRAM with only $p' < p$ processors.

• We would like to be able to run the $p$-processor algorithm on the smaller $p'$-processor PRAM in a work-efficient fashion.

**Theorem:** If a $p$-processor PRAM algorithm $A$ runs in time $t$, then for any $p' < p$, there is a $p'$-processor PRAM algorithm $A'$ for the same problem that runs in time $O(pt/p')$.

**Proof:** Let the time steps of algorithm $A$ be numbered $1, 2, ..., t$. Algorithm $A'$ simulates the execution of each time step $i=1, 2, ..., t$ in time $O(\lceil p/p' \rceil)$. There are $t$ steps, and so the entire simulation takes time $O(\lceil p/p' \rceil t) = O(pt/p')$, since $p' < p$.

• The work performed by algorithm $A$ is $pt$, and the work performed by algorithm $A'$ is $(pt/p') = pt$; the simulation is therefore work-efficient.

• If algorithm $A$ is itself work efficient, so is algorithm $A'$.

• When developing work-efficient algorithms for a problem, one needs not necessarily create a different algorithm for each different number of processors.
Deterministic Symmetry Breaking (Overview)

- **Problem:** Several processors wish to acquire mutually exclusive access to an object.
- **Symmetry Breaking:**
  - Method of choosing just one processor.
- **A Randomized Solution:**
  - All processors flip coins - (i.e., random nr.)
  - Processors with tails lose - (unless all tails)
  - Flip coins again until only 1 PE is left.
- **An Easier Deterministic Solution:**
  - Pick the PE with lowest ID number.
- **An Earlier Symmetry Breaking Problem** (See 30.4)
  - Assume an $n$-object subset of a linked list.
  - Choose as many objects as possible randomly from this sublist without selecting two adjacent objects.
- **Deterministic Version of Problem in Section 30.4:**
  - Choose a constant fraction of the objects from a subset of a linked list without selecting two adjacent objects.
    - First, compute a 6-coloring of the linked list in $O(\log^* n)$ time
    - Convert the 6-coloring of linked list to a “maximal independent set” in $O(1)$ time
    - Then the maximal independent set will contain a constant fraction of the $n$ objects, with no two objects adjacent.
Definitions and Preliminary Comments

• Definition:
  \[ \log^{(i)} n = \begin{cases} 
  n, & \text{if } i=0 \\
  \log (\log^{(i-1)} n), & \text{if } i>0 \text{ and } \log^{(i-1)} n > 0 \\
  \text{undefined}, & \text{otherwise.} 
\end{cases} \]

• Observe that \( \log^{(i)} \neq \log^i n \)

• Definition:
  \[ \log^* n = \min \{ i \geq 0 \mid \log^{(i)} n \leq 1 \} \]

• Observe that
  - \( \log^* 2 = 1 \)
  - \( \log^* 4 = 2 \)
  - \( \log^* 16 = 3 \)
  - \( \log^* 65536 = 4 \)
  - \( \log^* (2^{65536}) = 5 \)

• Note: \( 2^{65536} > 10^{80} \), which is approximately how many atoms exist in the observable universe.

• Definition [Coloring of an undirected graph \( G = (V, E) \)]:
  A function \( C: V \rightarrow \mathbb{N} \) such that if \( C(u) = C(v) \) then \((u,v) \notin E\).
  – Note, no two adjacent vertices have the same color.

• In a 6-coloring, all colors are in the range \{0,1,2,3,4,5\}. 

Definitions and Preliminary Comments (cont.)

• Example: A linked list has a two coloring.
  – Let even objects have color 0 and odd ones color 1.
  – We can compute a 2-coloring in $O(\log n)$ time using a prefix sum.
• Current Goal: Show a 6-coloring can be computed in $O(\log^* n)$ time without using randomization.

• Definition: An independent set of a graph $G = (V,E)$ is a subset $V' \subseteq V$ of the vertices such that each edge in $E$ is incident on at most one vertex in $V'$.

• Example: Two independent sets are shown above, one with vertices of type ◊ and the other of type ◯.

• Definition: A maximal independent set (MIS) is an independent set $V'$ such that the set $V' \cup \{v\}$ is not independent for any vertex $v$ in $V-V'$.

• Observe, that one independent set in above example is of maximum size and the other is not.

• Comment: Finding a maximum independent set is a NP-complete problem, if we interpret “maximum” as finding a set of largest cardinality. Here, “maximal” means that the set can not be enlarged and remain independent.
Efficient 6-coloring for a Linked List

**Theorem:** A 6-coloring for a linked list of length \( n \) can be computed in \( O(\log^* n) \) time.

- Note that \( O(\log^* n) \) time can be regarded as “almost constant” time.

**Proof:**

- Initial Assumption: Each object \( x \) in a linked list has a distinct processor number \( P(x) \) in \( \{0,1,2, \ldots, n-1\} \).

- We will compute a sequence
  
  \[ C_0(x), C_1(x), \ldots, C_{n-1}(x) \]

  of coloring for each object \( x \) in the linked list.

- The first coloring \( C_0 \) is an \( n \)-coloring with \( C_0(x) = P(x) \) for each \( x \).
  - This coloring is legal since no two adjacent objects have the same color.
  - Each color can be described in \( \lceil \log n \rceil \) bits, so it can be stored in an ordinary computer word.

- Assume that
  - colorings \( C_0, C_1, \ldots, C_k \) have been found
  - each color in \( C_k(x) \) can be stored in \( r \) bits.

- The color \( C_{k+1}(x) \) will be determined by looking at \( \text{next}(x) \), as follows:
  - Suppose \( C_k(x) = a \) and \( C_k(\text{next}(x)) = b \) are \( r \)-bit colors, represented as follows:
    - \( a = (a_{r-1}, a_{r-2}, \ldots, a_0) \)
    - \( b = (b_{r-1}, b_{r-2}, \ldots, b_0) \)
Since $C_k(x) \neq C_k(\text{next}(x))$, there is a least index $i$ for which $a_i \neq b_i$

- Since $0 \leq i \leq r-1$, we can write $i$ with only $\lceil \log r \rceil$ bits as follows:
  \[ i = (i_{\lceil \log r \rceil 1}, i_{\lceil \log r \rceil 2}, \ldots, i_0) \]

- We recolor $x$ with the value of $i$ concatenated with the bit $a_i$ as follows:
  \[ C_{k+1}(x) = < i, a_i > = (i_{\lceil \log r \rceil 1}, i_{\lceil \log r \rceil 2}, \ldots, i_0, a_i) \]

- The tail must be handled separately. (Why?) If $C_k(\text{tail}) = (d_{r-1}, d_{r-2}, \ldots, d_0)$
  then the tail is assigned the color, $C_{k+1}(\text{tail}) = < 0, d_0 >$

- Observe that the number of bits in the $k+1$ coloring is at most $\lceil \log r \rceil +1$.

- **Claim:** Above produces a legal coloring.
  - We must show that $C_{k+1}(x) \neq C_{k+1}(\text{next}(x))$
  - We can assume inductively that
    \[ a = C_k(x) \neq C_k(\text{next}(x)) = b. \]
  - If $i \neq j$, then $< i, a_i > \neq < j, b_j >$
  - If $i = j$, then $a_i \neq b_i = b_j$, so $< i, a_i > \neq < j, b_j >$
  - Argument is similar for tail case and is left to students to verify.

- **Note:** This coloring method takes an $r$-bit color and replaces it with an $(\lceil \log r \rceil +1)$-bit coloring.
• The length of the color representation strictly drops until \( r = 3 \).
  • Each statement below is implied by the one following it. Hence, Statement (3) implies (1):
    \[
    \begin{align*}
    (1) \quad r &> \lceil \log r \rceil + 1 \\
    (2) \quad r &> \log r + 2 \\
    (3) \quad 2^{r-2} &> r
    \end{align*}
    \]
  • The last statement can be proved by induction for \( r \geq 5 \) easily; (1) can be checked out for the case of \( r = 4 \) separately.
    – When \( r = 3 \), two colors can differ in the bit in position \( i = 0, 1, \) or 2.
    – The possible colors are
      \[
      < i, a_i > = \{ 00, 01, 10 \}, \{ 0, 1 \}
      \]
    – Note that only six colors out of the possible 8 which can be formed from three digits are used.
    – Assume that each PE can determine the appropriate index \( i \) in \( O(1) \) time.
      • These operations are available on many machines.
    – Observe this algorithm is EREW, as each PE only accesses \( x \) and then \( \text{next}[x] \).
    – **Claim:** Only \( O(\log^* n) \) iterations are needed to bring initial \( n \)-coloring down to a 6-coloring.
      • Details are assigned reading in text.
      • Proof is a bit technical, but intuitively OK, as \( r_{i+1} = \lceil \log r_i \rceil + 1 \)
Computing a Maximal Independent Set

- **Algorithm Description:**
  - The previous algorithm is used to find a 6-coloring.
  - This MIS algorithm iterates through the six colors, in order.
  - At each color, the processor for each object $x$ evaluates $C(x) = i$ and $alive(x)$
  - If true, mark $x$ as being in the MIS being constructed.
  - Each object adjacent to a newly added MIS object has its $alive$-bit set to false.
  - After iterating through each color, the algorithms stops.

- **Claim:** The set $M$ constructed is independent.
  - Suppose $x$ and $next(x)$ are in set $M$.
  - Then $C(x) \neq C(next(x))$
  - Assume w.l.o.g. $C(x) < C(next(x))$.
  - Then $next(x)$ is killed before its color is considered, so it is not in $M$

- **Claim:** The set $M$ constructed is maximal.
  - Suppose $y$ is a new object not in $M$ and the set $M \cup \{y\}$ is independent
  - By independence of $M \cup \{y\}$, the objects $x$ and $z$ that precede and follow $y$ are not in $M$.
  - Then, $y$ will be selected by that algorithm to be a member of $M$, a contradiction.
• Above two claims complete the proof of correctness of this algorithm.
• The algorithm is EREW, since PEs only access their own object or its successor and these read/write accesses occur synchronously (i.e., in lock step).
• Running Time is \( O(\log^* n) \).
  – Time to compute the 6-coloring is \( O(\log^* n) \)
  – Additional time required to compute MIS is \( O(1) \).

**Application to Cost Optimal Algorithm in 30.4**

– In Section 30.4, each PE services \( O(\log n) \) objects initially.
– After each PE selects an object to possibly delete, the randomized selection of the objects to delete is replaced by the preceding deterministic algorithm.
– The altered algorithm runs in \( O((\log^* n) (\log n)) \).
  • The first factor represents the work done at each level of recursion. It was formerly \( O(1) \) and was the time used to make a randomized selection.
  • The second factor represents the levels of recursion required.
– This deterministic version of this algorithm deletes more objects at each level of the recursion, as a maximal independent set is deleted each time.
– This method of deleting is fast!!!
  • Recall \( \log^* n \) is essentially constant or \( O(1) \).
  • Expect a small constant for \( O(\log^* n) \) time.
  • A maximal independent set is deleted each time.

**READ (in CLR) pp. 712, 713 and Ch 30.5.**