## The PRAM Model

- $p$ processors (executing the same program, single instruction counter, and with a common clock)
- Processors communicate by reading and writing into a global shared memory.
- Each operation takes unit time (independent of the number of processors).
- Each processor has a unique id, $i$, $1 \leq i \leq p$. Used to distinguish the behavior of the processors.



## The PRAM Model

- Simple(st) model of parallel computation
- Running time = number of (parallel) memory accesses
- Unrealistic: impossible to access global memory in constant time (independent of $p$)
- Nevertheless,
  - Valid to a first approximation
  - Gives useful (lower) bounds
  - Performance on other models can be derived from PRAM performance



## PRAM Classification

Q: How to deal with conflicts: multiple accesses to the same memory location?

- Forbid them: EREW
- Allow concurrent reads, but not writes: CREW
- Concurrent writes but not reads: ERCW (for completeness' sake)
- Allow concurrent reads and writes: CRCW



## Concurrent writes?

Q[bis]: How to perform multiple writes to the same memory location: what value to write?

A: Sub-classification of CW

- Common
- Any
- Priority
- Combining

## Example: Vector Sum

*Input:* Vectors $v[1, \ldots n]$ and $w[1, \ldots n]$ in shared memory

*Output:* Vector $z[1, \ldots n]$ s.t. $z[i] = v[i] + w[i]$ for $i = 1 \ldots n$

Trivial solution (on $n$ processors):

```
forall i in 1 ... n do
    z[i] := v[i]+w[i]
```



## On $p$ processors

```
local integer h, idx
for h = 1 to n%p do {
    idx = (i-1)*p+h;
    if idx <= n then
        z[idx] := v[idx]+w[idx];
    }
```

Analysis: All reads and writes are exclusisve (EREW)

$$T(n,p) = \left\lceil \frac{n}{p} \right\rceil = O(n/p)$$



## Work-Time Paradigm

Disadvantages of basic PRAM model: Algorithm = program executed by processor $i$ (cumbersome details, vis à vis $n$ and $p$).

Solution: Use a more abstract model, where specific activities of individual processors are not given.

- Algorithm/Program = sequence of steps
- Step = parallel operations $\Rightarrow$ forall construct (on as many processors as needed)
- Tackle actual number of physical processors later



## Work-Time Paradigm

Two complexity measures of WT algorithms

- Step complexity, $S(n)$
- Work complexity, $W(n)$, total number of operations

$$W(n) = \sum_{i=1}^{S(n)} W_i(n)$$

## Example: Reduction Operation

*Input:* Vector $a[1, \ldots n]$, (for $n = 2^k$) of elements of type $T$, binary associative operator, $\oplus : T \times T \to T$

*Output:* scalar, $S = \oplus_{i=1}^{n} a[i]$

*Solution:* ($\oplus$ is "op" in program)

```
for h = 1 to k do
    forall i in 1 ... 2^(k-h) do
        a[i] := a[2*i] op a[2*i+1]
    enddo
enddo
S = a[1]
```



## Analysis

- EREW Algorithm
- Step Complexity: $S(n) = k + 1 = \Theta(\log n)$
- Work Complexity: $W(n) = \left( \sum_{h=1}^{k} 2^{k-h} \right) + 1 = \Theta(n)$



## PRAM Algorithms on Lists

- Each processor "responsible" for one object: $i$-th processor does not necessarily have the $i$-th element of the list
- Note: $i$ used as pointer and also index (lists à la Fortran).

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| next[i] | / | 4 | 8 | 5 | 7 | 1 | 3 | 6 |

the list: 2 → 4 → 5 → 7 → 3 → 8 → 6 → 1

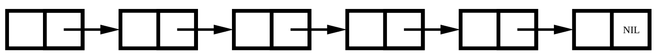


## Pointer Jumping

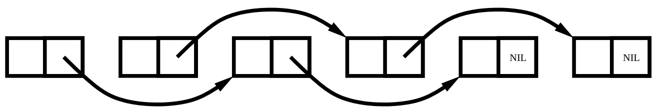
Powerful technique for many algorithms (list ranking, parallel prefix, Euler tours, etc.) Key step:

```
forall i such that next[i] /= NIL next[i] = next[next[i]]
```

- The list




- becomes



# List Ranking

- *Given* A collection of "objects" in global shared memory, arranged as a singly linked list, $L$.
- *Compute* for each object in $L$, its distance from the end of the list, i.e.,

$$d[i] = \begin{cases} 0 & \text{if next}[i] = NIL \\ 1 + d[\text{next}[i]] & \text{otherwise} \end{cases}$$

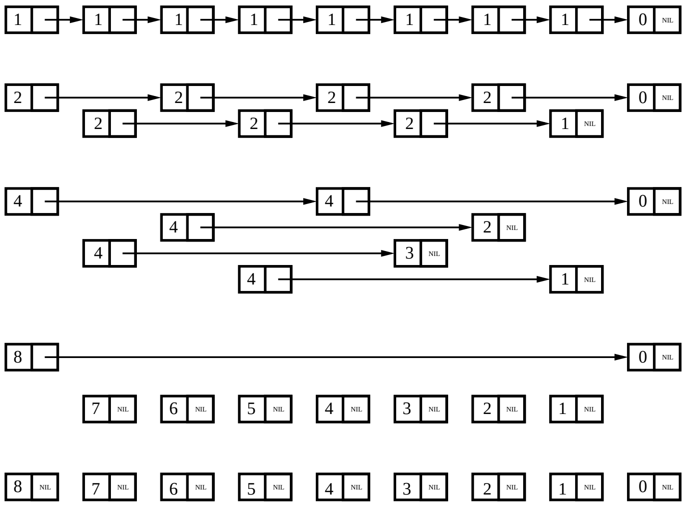


# Program List-Rank(L)

```
for each processor i in parallel
do if next[i] = nil
    then d[i] := 0
    else d[i] := 1
enddo
while there exists i such that next[i] /= nil
    for each such i do in parallel
        d[i] := d[i] + d[next[i]];
        next[i] = next[next[i]]
    endfor
endwhile
```



# Program List-Rank(L)





# Observations

- Synchronization (SIMD)
- EREW (more synchronization)
- Loop termination detection (even more synchronization)

## Correctness

- Invariant: at the beginning of every instance of the `while` loop (lines 6-11), for any element $i$, the sum of the $d[i]$ values in the sublist headed by $i$ is the distance of the $i$-th element from the end of the original list
- SIMD Synchronization (necesary to ensure that no value is overwritten before it is read by the processor that needs it).



## Analysis

- Number of procesors, $p = n$
- Running time: $T(n, n) = \lceil \lg n \rceil$
- Work: $W(n, n) = O(n \lg n)$
- Work of the sequential algorithm: $W(n, 1) = T_1(n) = n$

List-Rank(L) is not work-efficient



## Exercise: What does this program do?

```
 for each processor i in parallel
 do y[i] := x[i]
 enddo
 while there exists i such that next[i] /= nil
     for each such i do in parallel
         y[next[i]] := y[i] + y[next[i]];
         next[i] = next[next[i]]
    endfor
endwhile
```



## More Exercises (from CLR)

- Write an EREW program to reverse a list in constant time
- Select and broadcast representative to circular lists [30.1.4]
- Prefix computation on arrays [3.1.2]
- Explicit loop termination [30.1.8]
- Develop an exercise problem on your own and submit its solution