COMP 633 - Parallel Computing

Lecture 2
August 24, 2021

PRAM (1): The PRAM model and its complexity measures

First class summary

- In this course we study how to speed up large computational problems using parallel computing
  - in theory and in practice

- We study various parallel programming models
  - Initially we consider a theoretical model, the Parallel Random Access Machine (PRAM)
    - study algorithms and their asymptotic complexity
  - Subsequently we focus on practical models and their implementation on current hardware
    - shared memory multiprocessors, accelerators, and distributed memory clusters
      - examine execution model, hardware operation, programming constructs, performance analysis
      - illustrate principles using various case studies

Topics today

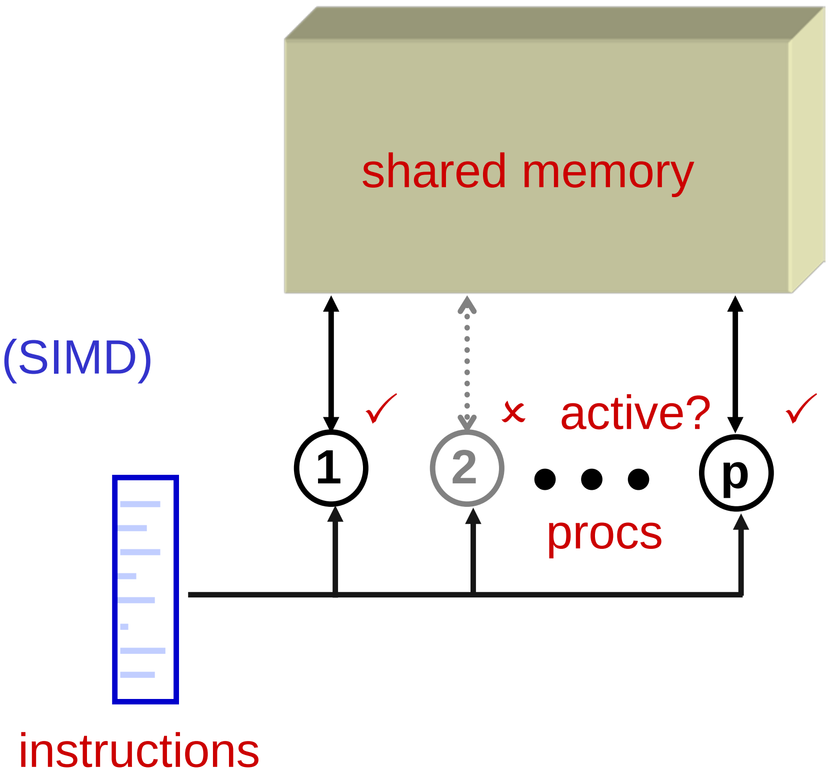
- PRAM model
  - execution model
  - programming model

- Work-Time model
  - programming model
  - complexity metrics
  - Brent’s theorem: translation to PRAM programs

- Parallel prefix algorithm
  - derivation
  - applications

PRAM model of parallel computation

- PRAM = Parallel Random Access Machine
  - *p* processors
  - shared memory
  - each processor has a unique identity $1 \le i \le p$
  - synchronous PRAM model
    - Single Instruction, Multiple Data (SIMD)
    - each processor may be active (✓) or inactive (×)
    - each instruction is executed by active processors only
    - each instruction completes in unit time



PRAM program

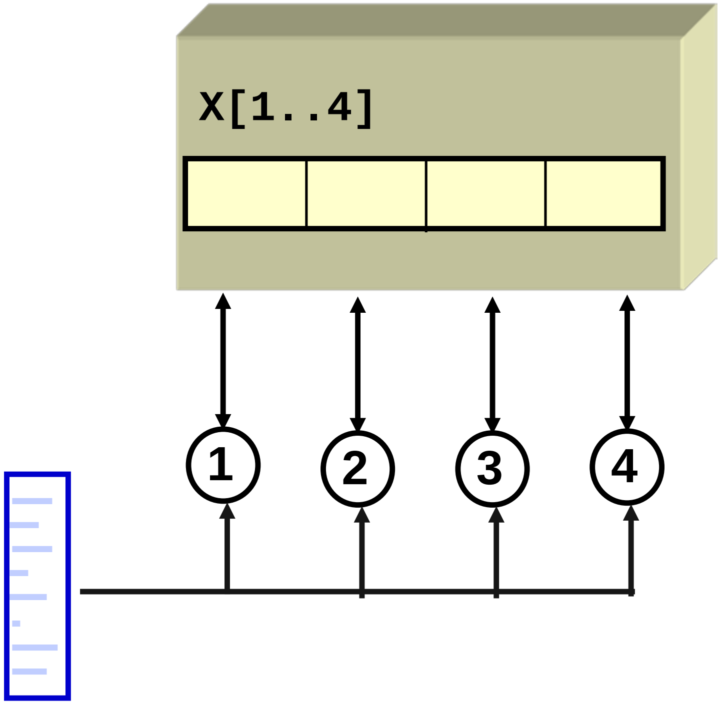
- PRAM program
  - sequential program
  - expressions involving processor id *i* have a unique value in each processor
    - *i* can be used as an array index

```
X[i] := 10 * i
```

    - conditionals specify active processors

```
if odd(i) then
      X[i] := X[i] + X[i+1]
endif

if i ≤ 2 then
     X[i] := 1
else
     X[i] := -1
endif
```



Concurrent memory access - Read

- Concurrent reads (CR)
  - all readers of a given location see the same value

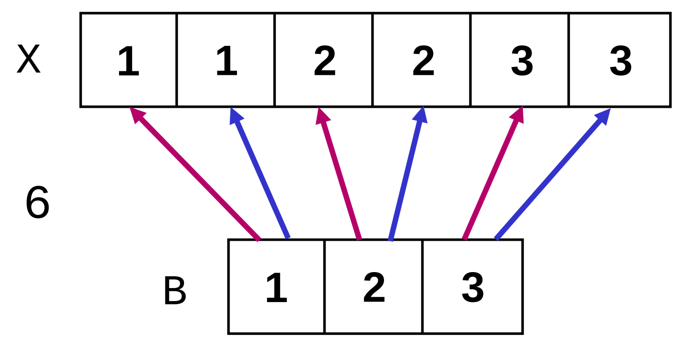
    `X[i] := y` value of `y` read concurrently by all *p* processors

    `X[i] := B[ ⌈i/2⌉ ]` the first `p/2` elements of `B` are read concurrently by two processors

- Eliminating bounded-degree concurrent reads
  - replace **`X[i] := B[ ⌈i/2⌉ ]`** with

```
if odd(i) then
    X[i] := B[ ⌈i/2⌉ ]
endif
if even(i) then
    X[i] := B[ ⌈i/2⌉ ]
endif
```

Ex.
`p = 6`



concurrent read is eliminated but number of steps is doubled

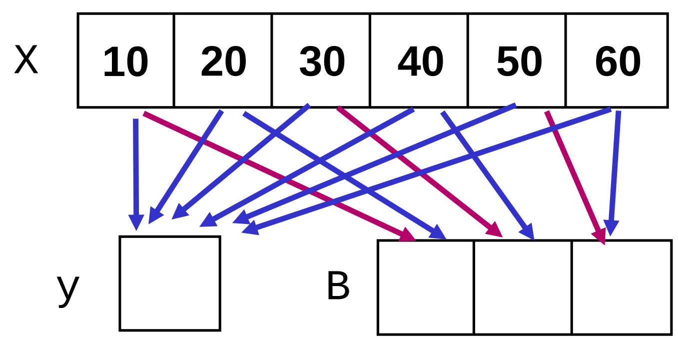
Concurrent memory access - Write

- Concurrent writes (CW)
  - final value depends on the arbitration policy among writes to the same destination:
    - Arbitrary CW
      - nondeterministic choice among values written
    - Common CW
      - processors that write a value to the same destination must write the same value, else error
    - Priority CW
      - value written by processor with lowest processor id
    - Combining Write
      - all values combined using a specified associative operation (e.g. “+”)

- Example (p = 6)

```
y := X[i]

B[ ⌈i/2⌉ ] := X[i]
```



Concurrent writes:

- Let `B[1:p]` be an array of boolean values and define $c = B_1 \vee B_2 \vee \ldots \vee B_p$
  - use p processors and concurrent writes to compute c in a constant number of steps
    a) with combining CW

    b) with a CW policy other than combining CW (which?)

Concurrent memory access

- PRAM variants
  - EREW, CREW, ERCW, CRCW
  - differ in performance, not expressive power
    - EREW < CREW < CRCW
  - loosely reflect difficulty of model implementation

- The following are considered EREW
  - references to
    - processor id i
    - number of processors p
    - problem size n
  - references to local variables

    ```
    local h;  h := 2*i + 1;  X[h] := X[i]
    ```

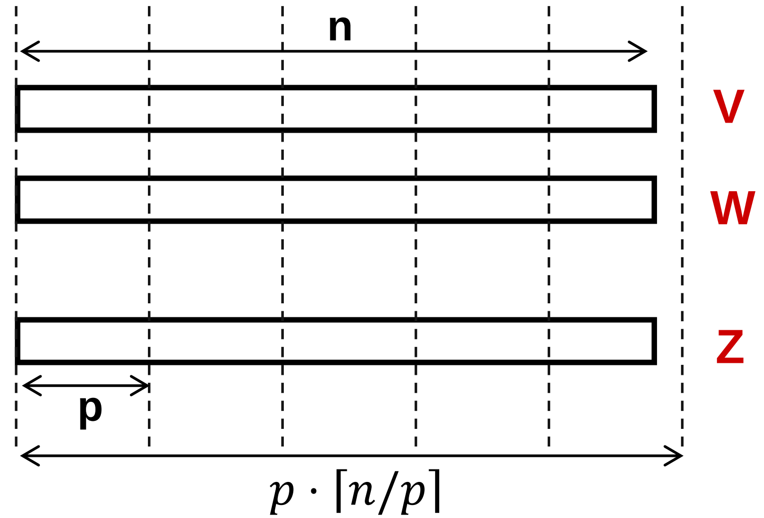
  - expression evaluation is synchronous, e.g.

    ```
     X[i] := X[i] + X[i+1]
    ```

    is EREW

A PRAM program

- Simple problem: vector addition
  - given V,W vectors of length n
  - compute Z = V + W
- PRAM program
  - constructed to operate with arbitrary
    - problem size n
    - number of processors p
  - work to be performed must explicitly be “scheduled” across processors
  - time complexity with *p* procs
    - $T_c(n,p)$ =
  - PRAM model?



Input: V[1:n], W[1:n] in shared memory

Output: Z[1:n] in shared memory

proc id

```
local integer h, k
for h := 1 to ⌈n/p⌉ do
        k := (h-1)•p + i
        if k ≤ n then
                Z[k] := V[k] + W[k]
        endif
enddo
```

Work-Time paradigm

- W-T parallel programming model
  - high-level PRAM programming model
    - specifies available parallelism
    - no explicit scheduling of parallelism over processors
  - simplifies algorithm presentation and analysis
  - W-T programs can be mechanically translated to PRAM programs

- W-T program
  - sequential program
  - `forall` construct
    - specification of available parallelism
    - number of processors is not a parameter of the model !

WT program for vector addition

Input: V[1:n], W[1:n]
Output: Z[1:n]

```
forall i in 1:n do
        Z[i] := V[i] + W[i]
enddo
```

Programming notation for the W-T framework

- standard sequential programming notation
  - statements
    - assignment
    - statement composition
    - alternative construct (if ... then ... else …)
    - repetitive construct (for, while)
  - expressions
    - arithmetic and logical functions
    - variable reference
    - (recursive) function and procedure invocation

- forall statement
  - specifies statement T may be executed simultaneously for each value of i in D
  - no restriction on T
    - can be a sequence of statements
    - can invoke (recursive) functions
    - can be another (nested) forall statement

```
forall i in D do
     statement T depending on i
enddo
```

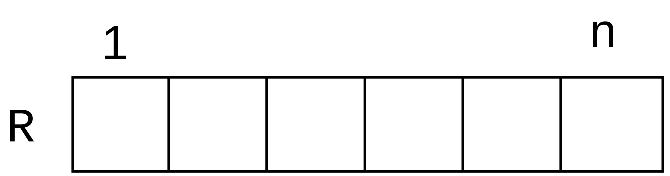
W-T complexity metrics

- Work complexity W(n)
  - total number of operations performed (as a function of input size n)
- Step complexity S(n)
  - number of parallel steps required (as a function of input size n)
  - assuming unbounded parallelism
- Inductively defined over constructs of W-T programming notation

W-T complexity measures: simple example

```
forall i in 2:n-1 do
       R[i] := (R[i-1] + R[i] + R[i+1])/3
enddo
```

```
for h := 1 to k do
       forall i in 2:n-1 do
              R[i] := (R[i-1] + R[i] + R[i+1])/3
       enddo
enddo
```



Work and Step Complexity of the forall construct

- How to define work and time complexity of the `forall` construct?

P:

```
forall i in D do
      body T depending on i
enddo
```

– assume we can determine $W(T_i)$ and $S(T_i)$ for each i in D

- W(P) =

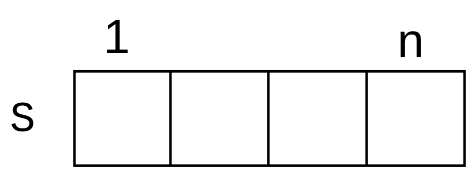
- S(P) =

W-T complexity measures: vector summation

- let $n = 2^k$

```
forall i in 1:n/2 do
        S[i] := S[2i - 1] + S[2i]
enddo
```

```
for h := 1 to k do
        forall i in 1:n/2^h do
                S[i] := S[2i - 1] + S[2i]
        enddo
enddo
```



n = 4, k = 2

W-T complexity measures: vector summation

- Vector summation (sum - reduction)
  - given V[1..n], $n = 2^k$
  - compute s = sum(V[1:n])
  - optimal sequential time $T_s(n) = \Theta(n)$

- Complexity

  W(n) =

  S(n) =

  PRAM model needed?

Input: V[1:n] vector of integers, $n = 2^k$

Output: s = sum(V[1:n])

```
P1: forall i in 1:n do
        B[i] := V[i]
    enddo

P2: for h := 1 to k do
        forall i in 1:n/2^h do
            B[i] := B[2i-1]+B[2i]
        enddo
    enddo

P3: s := B[1]
```

Brent's theorem and $T_c(n,p)$

- Brent's theorem schedules a W-T program for a *p*-processor PRAM
  - idea
    - simulate each parallel step in W-T program using *p* processors
    - the work $W_i(n)$ to be performed in step i can be completed using p processors in time

$$\left\lceil \frac{W_i(n)}{p} \right\rceil$$

  - bound concurrent runtime $T_C(n,p)$ of resultant PRAM program
    - by summing over all S(n) steps

$$T_c(n,p) = \sum_{i=1}^{S(n)} \left\lceil \frac{W_i(n)}{p} \right\rceil \le \sum_{i=1}^{S(n)} \left(\left\lfloor \frac{W_i(n)}{p} \right\rfloor + 1 \right) \le \left\lfloor \sum_{i=1}^{S(n)} \frac{W_i(n)}{p} \right\rfloor + S(n) = \left\lfloor \frac{W(n)}{p} \right\rfloor + S(n)$$

$$\left\lceil \frac{W(n)}{p} \right\rceil = \left\lceil \sum_{i=1}^{S(n)} \frac{W_i(n)}{p} \right\rceil \le \sum_{i=1}^{S(n)} \left\lceil \frac{W_i(n)}{p} \right\rceil = T_c(n,p)$$

Scheduling W-T vector summation algorithm

W-T vector summation algorithm

Input: V[1:n] vector of integers, $n = 2^k$
Output: s = sum(V[1:n])

```
P1: forall i in 1:n do
          B[i] := V[i]
    enddo

P2: for h := 1 to k do
       forall i in 1:n/2^h do
          B[i] := B[2i-1]+B[2i]
       enddo
    enddo

P3: s := B[1]
```

PRAM vector summation algorithm

Input: V[1:n] vector of integers, $n = 2^k$
Output: s = sum(V[1:n])
p > 0 processor *PRAM*; processor index *i*

```
local integer j, r;
P1: for j := 1 to ⌈n/p⌉ do
       r := (j-1)•p + i
       if r ≤ n then B[r] := V[r] endif
    enddo

P2: for h := 1 to k do
       for j := 1 to ⌈(n/2^h)/p⌉ do
          r := (j-1)•p + i
          if  r ≤ n/2^h then
                B[r] := B[2r-1]+B[2r]
          endif
       enddo
    enddo

P3: if i ≤ 1 then s := B[1] endif
```

Performance of translated W-T program

- Count steps needed to perform the additions
  - Brent's theorem predicts

$$T_c(n,p) = O\left(\left\lfloor \frac{n-1}{p} \right\rfloor + \lg n\right)$$

  - counts for various p

p	$T_c(n,p)$
$p=1$	$(n-1)/p$
$p>n$	$\lg n$
$p=3, n=2^k, k$ even	$\approx \lfloor (n-1)/p \rfloor + \frac{1}{2}\lg n$

- Upper bound is tight (for this program)
- translation retains EREW model

PRAM vector summation algorithm

Input: V[1:n] vector of integers, n = 2^k
Output: s = sum(V[1:n])
$p > 0$ processor *PRAM*; processor index i

```
local integer j, r;
P1: for j := 1 to ⌈n/p⌉ do
        r := (j-1)•p + i
        if r ≤ n then B[r] := V[r] endif
    enddo

P2: for h := 1 to k do
        for j := 1 to ⌈(n/2^h)/p⌉ do
            r := (j-1)•p + i
            if  r ≤ n/2^h then
                    B[r] := B[2r-1] + B[2r]
            endif
        enddo
    enddo

P3: if i ≤ 1 then s := B[1] endif
```

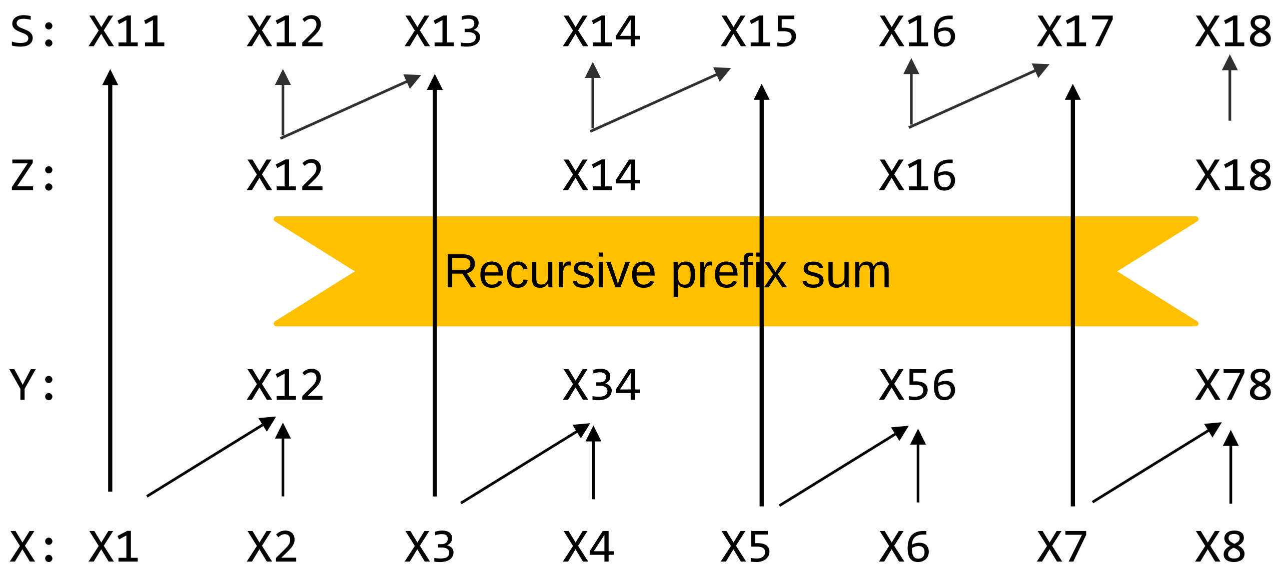
Parallel prefix-sum

- Prefix sum
  - Input
    - Sequence X of $n = 2^k$ elements, binary associative operator +
  - Output
    - Sequence S of $n = 2^k$ elements, with $S_i = x_1 + \ldots + x_i$
  - Example:
    - X = [1, 4, 3, 5, 6, 7, 0, 1]
    - S = [1, 5, 8, 13, 19, 26, 26, 27]
  - $T_S(n) = \Theta(n)$

- Uses of prefix sum
  - efficient parallel implementation of sequential "scan" through consecutive actions
    - ex: Given series of bank transactions T[1:n], with T[i] positive or negative, and T[1] the opening deposit > 0
      - Was the account ever overdrawn?
  - explicit or implicit component of many parallel algorithms

Prefix sum algorithm

- Recursive solution
  - `Xi` stands for `X[i]` and `Xij` stands for `X[i]+X[i+1]+… +X[j]`

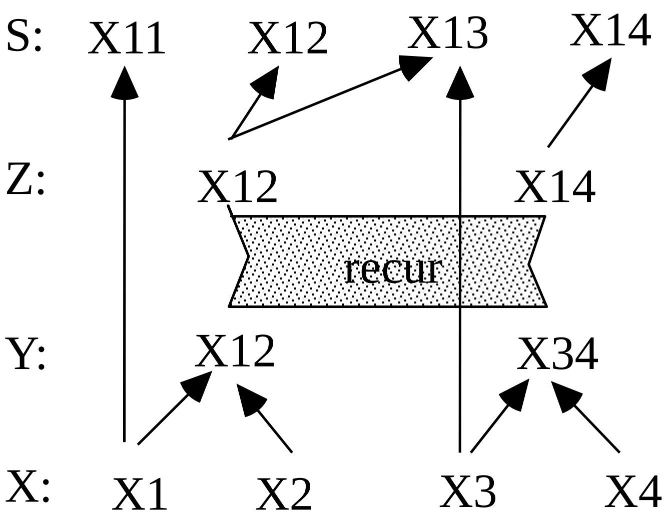


- W-T complexity
  - $W(n) = W\left(\frac{n}{2}\right) + O(n),\ W(1) = O(1) \ \Rightarrow ?$
  - $S(n) = S\left(\frac{n}{2}\right) + O(1), S(1) = O(1) \ \Rightarrow ?$

Parallel prefix sum algorithm – WT model

Input: X[1..n] vector of integers

Output: S[1..n]

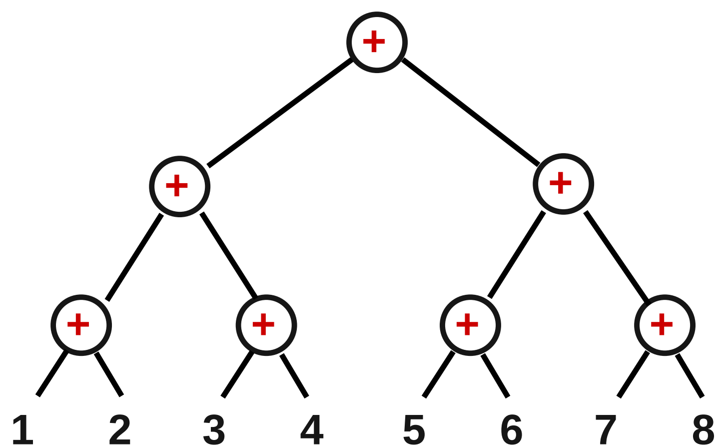


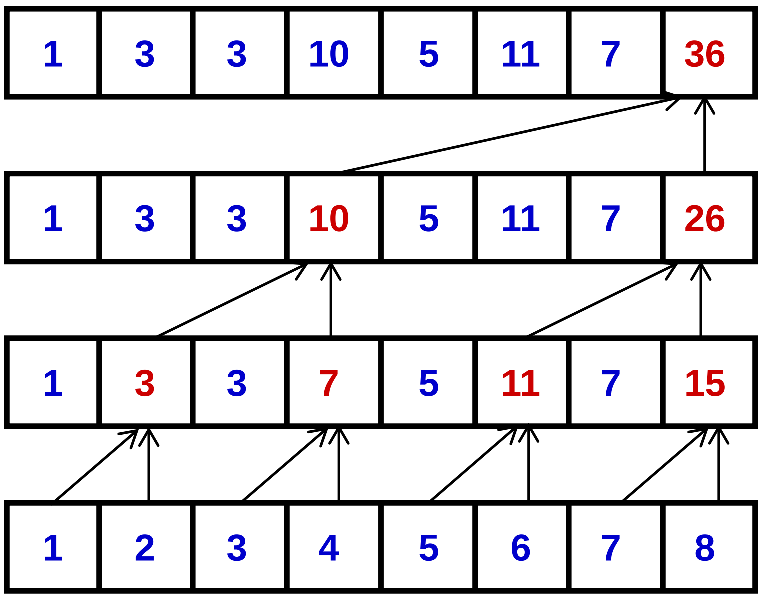

```
par_prefix_sum( X[1..n] ) =
   var Y[1..n/2], Z[1..n/2], S[1..n];
   S[1] := X[1];
   if n > 1 then
      forall  1 ≤ i ≤ n/2  do
         Y[i] := X[2i-1] + X[2i]
      enddo
      Z[1..n/2] := par_prefix_sum(Y[1..n/2]);
      forall 2 ≤ i ≤ n do
         if even(i) then
            S[i] := Z[i/2]
         else
            S[i] := Z[(i-1)/2] + X[i]
         endif
      enddo
   endif
   return S[1..n]
```

Balanced trees in arrays

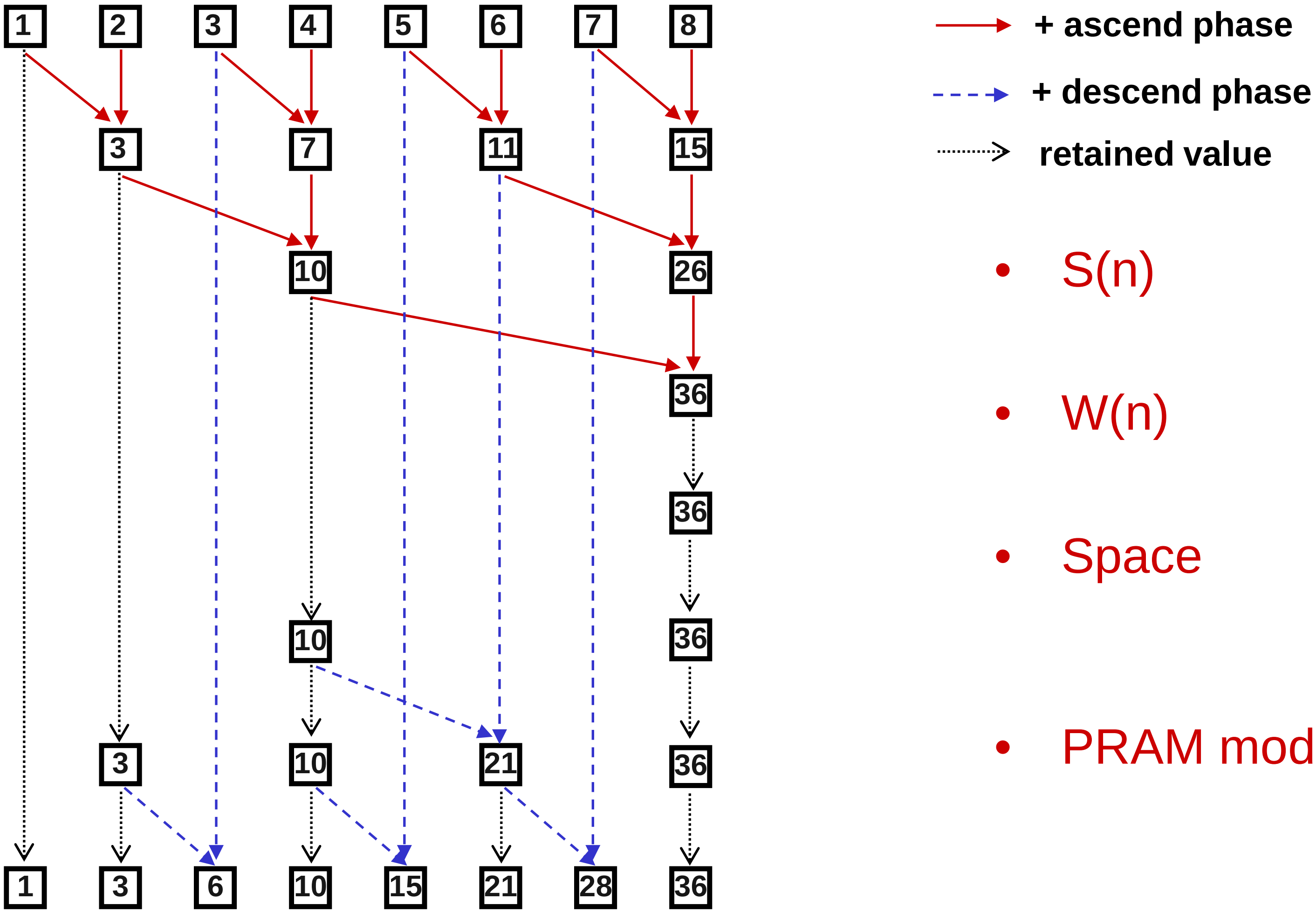
- Balanced Tree Ascend / Descend
  - Key idea
    - view input data as balanced binary tree
    - sweep tree up and/or down
  - “Tree” not a data structure but a control structure (e.g., recursion)

- Example
  - vector summation



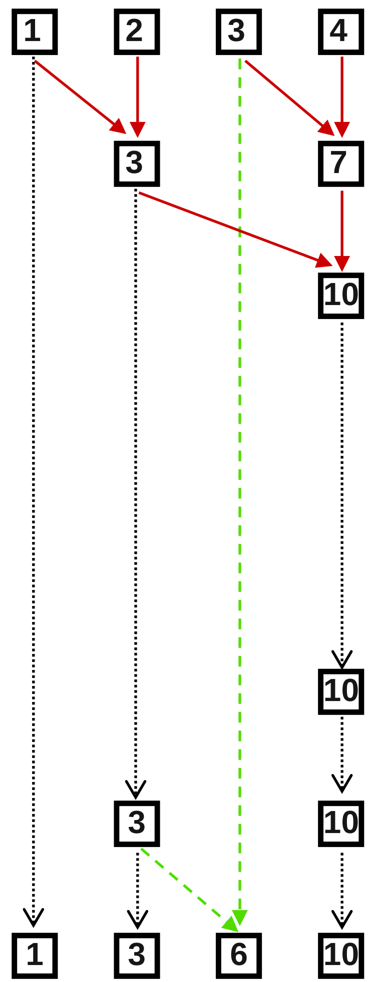


In-place prefix sum



- S(n)
- W(n)
- Space
- PRAM model

In-place prefix-sum algorithm – WT model



Input: X[1..n] vector of values, $n = 2^k$

Output: S[1..n] vector of prefix sums

```
parallel_prefix_sum( X[1..n] ) =
   forall  i in 1:n do
      S[i] := X[i]
   enddo

   for  h = 1 to k do
      forall  i in  1:n/2^h  do
         S[2^h i] := S[2^h i – 2^(h-1)] + S[2^h i]
      enddo
   enddo

   for  h = k downto 1
      forall i in 2:n/2^(h-1) do
         if odd(i) then
            S[2^(h-1) i] := S[2^(h-1) i – 2^(h-1)] + S[2^(h-1) i]
         endif
      enddo
   enddo
```

Scan-based primitives

- Scan operations (parallel prefix operations) can be used to implement many useful primitives
  - Suppose we are given SCAN to compute prefix sum of integer sequences

    ```
    seq<int> SCAN(seq<int>)
    ```

  - step complexity is $\Theta(\lg n)$
  - work complexity is $\Theta(n)$
  - PRAM model is EREW

- The next three examples have the same complexity as `SCAN`

COPY (or DISTRIBUTE)

```
seq<int> COPY(int v, int n) ){

  seq<int> V[1:n];
  V[1] = v;
  forall i in 2 : n do
     V[i] := 0;
  enddo
  return SCAN(V);
}
```

v = 5
n = 7
V = 5 0 0 0 0 0 0
Res = 5 5 5 5 5 5 5

ENUMERATE

```
seq<int> ENUMERATE(seq<bool> Flag){

  seq<int> V[1:#Flag];
  forall i in 1 : #Flag do
      V[i] := Flag[i] ? 1 : 0;
  enddo
  return SCAN(V);
}
```

Flag =	T	T	F	T	F	F	T
V =	1	1	0	1	0	0	1
Res =	1	2	2	3	3	3	4

PACK

```
seq<T> PACK(seq<T> A, seq<bool> Flag){

  seq<T> R[1:#A];
  P := ENUMERATE(Flag);
  forall i in 1 : #Flag do
     if Flag[i] then R[P[i]] := A[i] endif;
  enddo
  return R[1:P[#Flag]];
}
```

A =	!	@	#	$	%	^	&
Flag=	T	T	F	T	F	F	T
P =	1	2	2	3	3	3	4
R =	!	@	$	&			

Radix Sort

Input: `A[1:n]` with *b*-bit integer elements
Output: `A[1:n]` sorted
Auxiliary: `FL[1:n], FH[1:n], BL[1:n], BH[1:n]`

```
for h := 0 to b-1 do
     forall i in 1:n do
          FL[i] := (A[i] bit h) == 0
          FH[i] := (A[i] bit h) != 0
     enddo
     BL := PACK(A,FL)
     BH := PACK(A,FH)
     m  := #BL
     forall i in 1:n do
            A[i] := if (i ≤ m) then BL[i] else BH[i–m]endif
     enddo
enddo
```

S(n) =

W(n) =

Complexity measures for W-T algorithms

- Asymptotic time complexity measures
  - (optimal) sequential time complexity $T_s(n)$
  - parallel time complexity $T_c(n,p)$

- Speedup
  - definition

$$SP(n,p) = \frac{T_S(n)}{T_c(n,p)}$$

  - limitation

$$SP(n,p) = \frac{T_S(n)}{T_c(n,p)} \le \frac{T_S(n)}{W(n)/p} = \frac{pT_S(n)}{W(n)} = O(p)$$

- Average available parallelism
  - definition

$$AAP(n) = \frac{W(n)}{S(n)}$$

Objectives in the design of W-T algorithms

- Goal 1: construct work efficient algorithms
  - a W-T algorithm is work efficient if $W(n) = \Theta(T_s(n))$
  - work-inefficient parallel algorithms have limited appeal on a PRAM with a fixed number of processors p

$$\lim_{n\to\infty} SP(n,p) \quad \le \quad \lim_{n\to\infty} \frac{pT_s(n)}{W(n)} \quad = \quad p \lim_{n\to\infty} \frac{T_s(n)}{W(n)} \quad = \quad 0$$

Objectives in the design of W-T algorithms

- Goal 2: minimize step complexity
  - get optimal speedup using $AAP(n) = T_s(n) / S(n)$ processors

$$SP(n, AAP(n)) = \Theta\left(\frac{T_s(n)}{T_c(n, AAP(n))}\right) = \Omega\left(\frac{T_s(n)}{\frac{T_s(n)}{AAP(n)} + S(n)}\right)$$

$$= \Omega\left(\frac{T_s(n)}{S(n) + S(n)}\right) = \Omega(AAP(n))$$

  - when $S(n)$ is decreased, $AAP(n)$ is increased
    - with fixed problem size
      - can use more processors to get greater speedup
    - with fixed number of processors
      - reach optimal speedup at smaller problem size

W-T model advantages

- Widely developed body of techniques
- Ignores scheduling, communication and synchronization
  - “easiest” parallel programming
- Source-level complexity metrics
  - Work and step complexity
  - related to running time via Brent’s theorem
- Good place to start
  - many “real-world” algorithms can be derived starting from W-T algorithms