Lecture 8-9

PRAM models

Computational networks

Brent Theorem

PRAM – Reference model

- Theoretical Model of Parallel Computation

used well for performance evaluation

- The abstract model used for serial computing is known as RAM (for random-access machine, or model).

- PRAM is the counterpart for parallel computing.

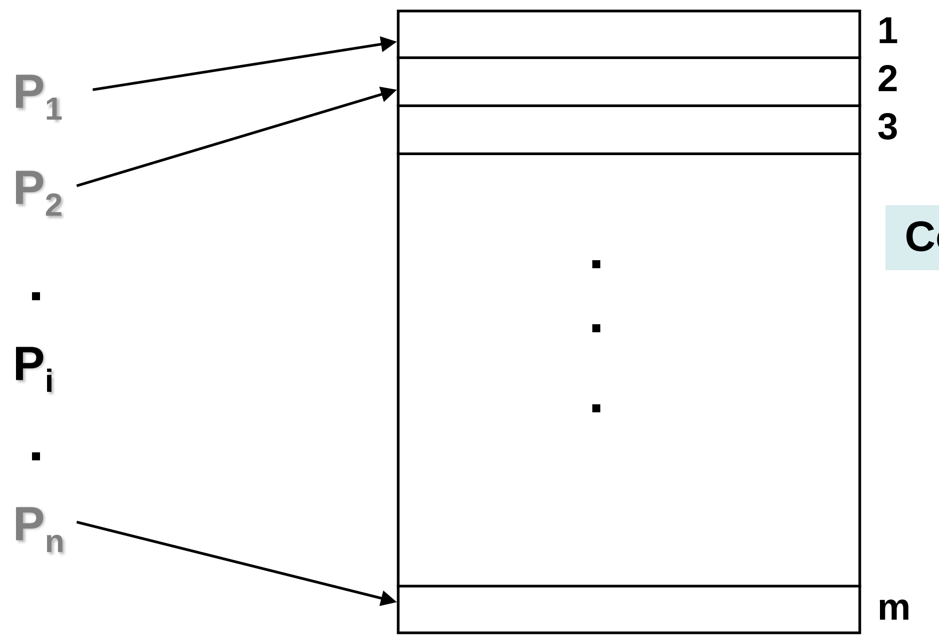
RAM model

- **Random Access Machine** is a favorite model of a sequential computer. Its main features are:
  - Computation unit with a user defined program.
  - Read-only input tape and write-only output tape.
  - **Unbounded** number of local **memory** cells.
  - Each memory cell is capable of holding an integer of unbounded size.
  - Instruction set includes operations for :
    - moving data between memory cells,
    - comparisons
    - conditional branches, and
    - simple arithmetic operations.
  - Execution starts with the first instruction and ends when a HALT/STOP instruction is executed.
  - All operations take unit time regardless of the lengths of operands.
  - **Time complexity** = the number of instructions executed.
  - **Space complexity** = the number of memory cells accessed.

Parallel RAM-> PRAM model

- Parallel Random Access Machine
- Shared-memory multiprocessor
- **unlimited number of processors,** each
  - has unlimited local memory
  - knows its ID
  - able to access the shared memory
- **unlimited shared memory**

PRAM MODEL



Common Memory

n **RAM processors connected to a common memory of** ***m*** **cells**

ASSUMPTION: at each time unit each P_i can

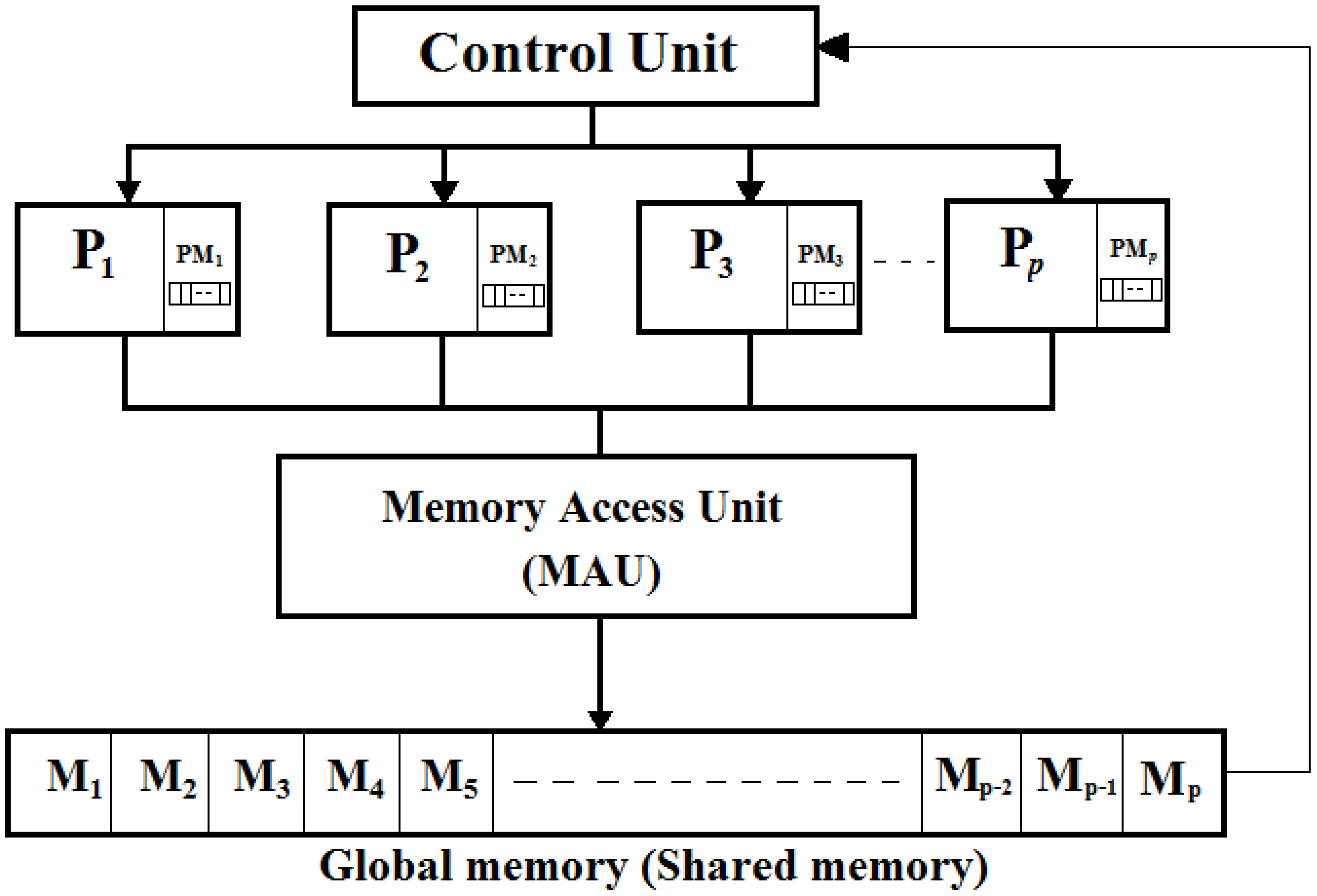
- reading from memory,
- make an internal computation and
- writing into memory.

CONSEQUENCE: any pair of processors P_i P_j can communicate in constant time!

P_i writes the message in cell x at time t

P_j reads the message in cell x at time $t+1$

PRAM architecture model



PRAM

- Inputs/Outputs are placed in the shared memory. (designated address)
- Memory cell stores an arbitrarily large integer.
- Each instruction takes a unit time.
- **Instructions are synchronized across the processors**
- A PRAM instruction executes in **3-phase cycles**.
  - **Read** (if any) from a shared memory cell.
  - **Local computation** (if any).
  - **Write** (if any) to a shared memory cell.

PRAM Complexity Measures

- for each individual processor
  - ***time***: number of instructions executed
  - ***space***: number of memory cells accessed

- PRAM machine
  - ***time***: time taken by the longest running processor
  - ***space***: total number of memory cells accessed
  - ***hardware***: maximum number of active processors

Technical Issues for PRAM

- How processors are activated
- How shared memory is accessed

- Too many processors gives problems with synchronization

PRAM execution

- During computation steps processors have a special activation register that specifying the maximum index of an active processor.
- Initially, only the first processor is in active mode, it computes the number of required active processors and loads this register, and then the other corresponding processors start executing their programs.
- The computation will continue until the first processor halts, at which time all other active processors are halted.

Processor Activation – 2 variants

- P_0 places the number of processors (p) in the designated shared-memory cell
  - each active P_i (where $i < p$), starts executing
  - $O(1)$ time to activate
  - all processors halt when P_0 halts
- Active processors explicitly activate additional processors via FORK instructions
  - tree-like activation
  - $O(\log p)$ time to activate

Parallel time complexity = *the time elapsed for P_0's computation.*

PRAM

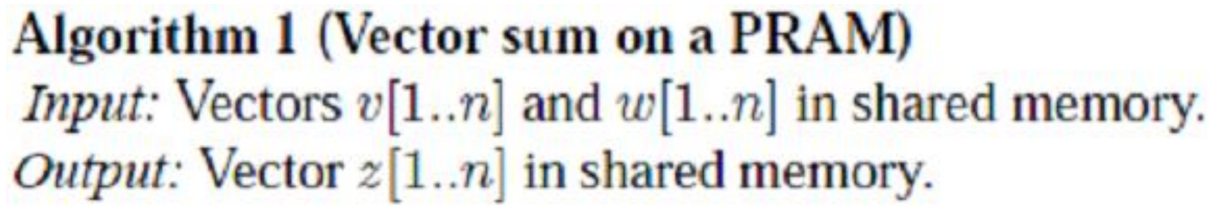
- To describe a program in PRAM model we need to specify **n** programs (n=no of processors)
  - 1 program defined based on the ID => n programs
- Pros: It is a very good <u>**conceptual model**</u> for designing efficient parallel algorithms
  - due to simplicity and possibility of simulating efficiently PRAM algorithms on more realistic parallel architectures

simplification

basic parallel statement

```
for all x in X  do in parallel
   instruction(x)
```

Vectors sum – PRAM



Algorithm 1 (Vector sum on a PRAM)
Input: Vectors $v[1..n]$ and $w[1..n]$ in shared memory.
Output: Vector $z[1..n]$ in shared memory.

PRAM with ***n*** processors

Processors: $P_0 \ldots P_{n-1}$

Algorithm of P_i

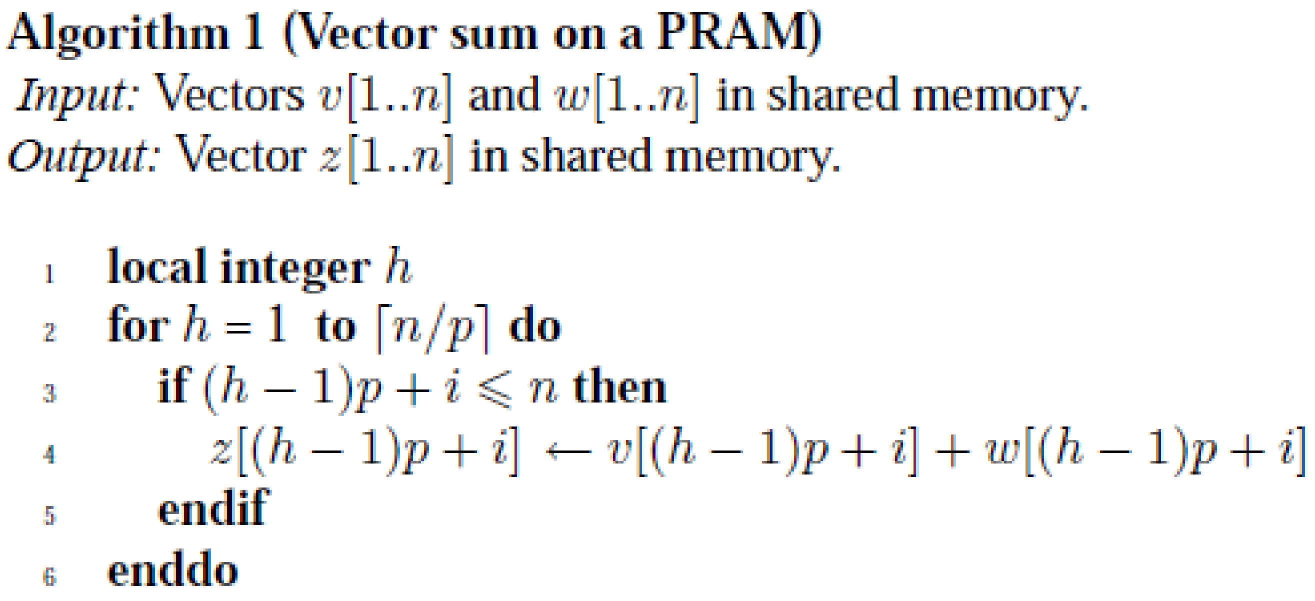
```
Read v[i]
Read w[i]
z[i] ← v[i]+w[i]
Write z[i]
```

One step

Vectors sum

PRAM with ***p*** processors

Processors: $P_0 \ldots P_{p-1}$



Algorithm 1 (Vector sum on a PRAM)

Input: Vectors $v[1..n]$ and $w[1..n]$ in shared memory.

Output: Vector $z[1..n]$ in shared memory.

```
local integer $h$
for $h = 1$ to $\lceil n/p \rceil$ do
    if $(h-1)p + i \leqslant n$ then
        $z[(h-1)p+i] \leftarrow v[(h-1)p+i] + w[(h-1)p+i]$
    endif
enddo
```

Algorithm (WT paradigm)

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

The Work-Time paradigm

- **Work-Time** = A higher-level abstraction than PRAM.
- In the PRAM model, ***algorithms are presented as a program to be executed by all the processors****;* in each step an operation is performed simultaneously by all active processors.
- In the WT model, each step may contain an arbitrary number of operations to be performed simultaneously, and the *scheduling of these operations over processors is left implicit.*
  - we will use the **forall** construct to denote such concurrent operations, and we drop explicit mention of the processor *id* and *p*, the number of processors.
- In fact the **forall** construct is the only construct that distinguishes a WT algorithm from a sequential algorithm.

Example

Algorithm 2 (Sequence reduction, WT description)
Input: Sequence a with $n = 2^k$ elements of type T, binary associative operator $\oplus : T \times T \rightarrow T$.
Output: $S = \oplus_{i=1}^{n} a_i$.

```
$T$ REDUCE(**sequence**⟨T⟩ $a$, $\oplus : T \times T \rightarrow T$)

$T\ B[1..n]$
**forall** $i \in 1 : n$ **do**
    $B[i] \leftarrow a_i$
**enddo**
**for** $h = 1$ **to** $k$ **do**
    **forall** $i \in 1 : n/2^h$ **do**
        $B[i] \leftarrow B[2i-1] \oplus B[2i]$
    **enddo**
**enddo**
$S \leftarrow B[1]$
**return** $S$
```

The *step complexity* of the algorithm, denoted by $S(n)$, is the number of steps that the algorithm executes. (**Depth**)
If $W_i(n)$ is the number of simultaneous operations at parallel step i, then

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

$$\begin{array}{cc}
S_{2-4}(n) = \Theta(1) & W_{2-4}(n) = \Theta(n) \\
S_{6-8}(n) = \Theta(1) & W_{6-8}(n,h) = \Theta(\frac{n}{2^h}) \\
S_{5-9}(n) = kS_{6-8}(n) = \Theta(\lg n) & W_{5-9}(n) = \sum_{h=1}^{k} W_{6-8}(n,h) = \Theta(n) \\
S_{10}(n) = \Theta(1) & W_{10}(n) = \Theta(1) \\
S(n) = S_{2-4}(n) + S_{5-9}(n) + S_{10}(n) = \Theta(\lg n) & W(n) = W_{2-4}(n) + W_{5-9}(n) + W_{10}(n) = \Theta(n)
\end{array}$$

Work-Depth analysis…

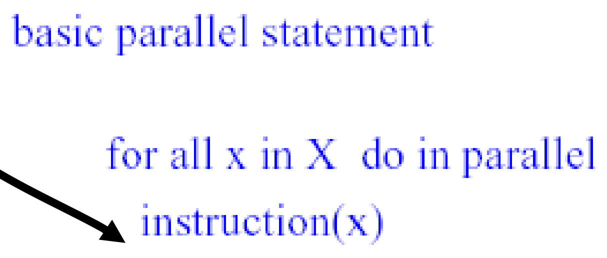
Variant…

```
for h =1 to k do
    forall  i  $\in$ 1,n do
        if ( i % $2^h$ =0 and i+$2^{h-1}$<=n ) then
            B[i] $\leftarrow$ B[i]+ B[i+$2^{h-1}$]
        endif
    enddo
enddo
```

Shared-Memory Access

Concurrent (C) means, many processors can do the operation simultaneously in the same memory

Exclusive (E) not concurent

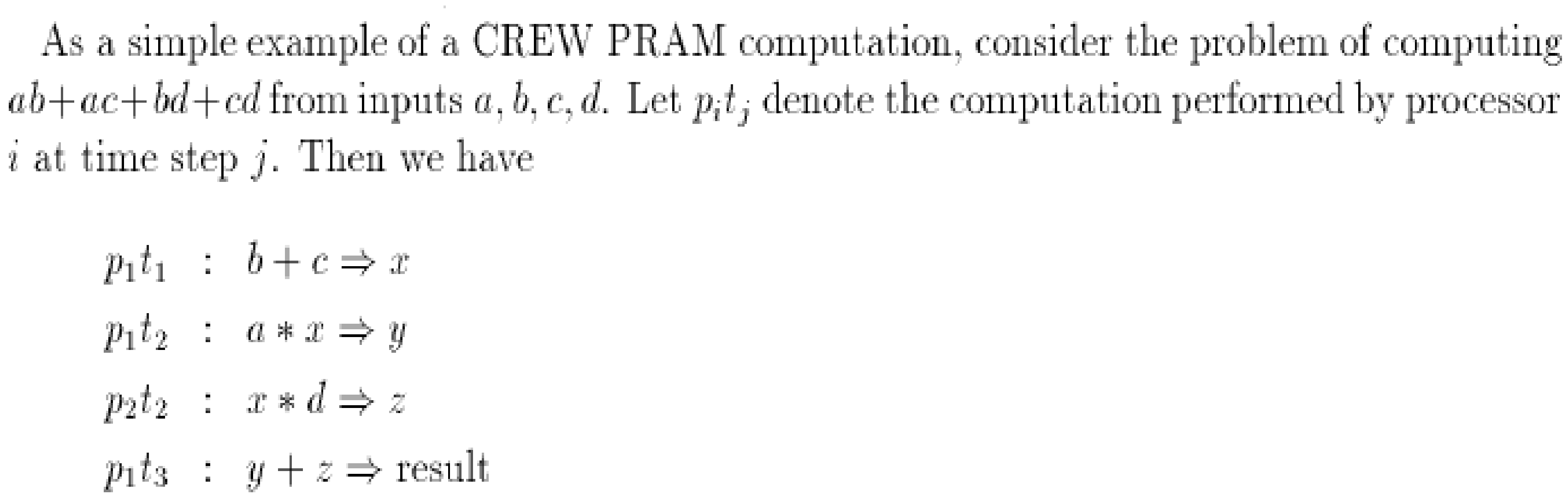


basic parallel statement

```
for all x in X  do in parallel
    instruction(x)
```

- EREW (Exclusive Read Exclusive Write)
- CREW (Concurrent Read Exclusive Write
  - Many processors can read simultaneously
    the same location, but only one can attempt to write to a given location
- ERCW
- CRCW
  - **Many processors can write**/read **at**/from **the same memory location**

example



As a simple example of a CREW PRAM computation, consider the problem of computing $ab+ac+bd+cd$ from inputs a, b, c, d. Let $p_i t_j$ denote the computation performed by processor i at time step j. Then we have

$$\begin{aligned}
p_1 t_1 &: \quad b + c \Rightarrow x \\
p_1 t_2 &: \quad a * x \Rightarrow y \\
p_2 t_2 &: \quad x * d \Rightarrow z \\
p_1 t_3 &: \quad y + z \Rightarrow \text{result}
\end{aligned}$$

Basic submodels for CRCW PRAM

- **PRIORITY CRCW:** the processors are assigned fixed distinct priorities and the processor with the highest priority is allowed to complete WRITE.
- **ARBITRARY CRCW:** one randomly chosen processor is allowed to complete WRITE. The algorithm may make no assumptions about which processor was chosen.
- **COMMON CRCW:** all processors are allowed to complete WRITE **iff** all the values to be written are equal. Any algorithm for this model has to make sure that this condition is satisfied. If not, the algorithm is illegal and the machine state will be undefined.

Example - searching

- Assume *p*-processor PRAM, ***p<n*.** Assume that shared memory contains ***n*** distinct items and *P0* owns value ***x***.
  The task is to let *P0* know whether ***x*** occurs within the input array.

- <u>EREW PRAM algorithm:</u>
  - *P0* broadcasts *x* to *P1,...,Pp* in *log p* steps using **binary broadcast tree**.
  - All processors perform local searches, each on *[n/p]* items in *[n/p]* steps.
  - Every processor defines a flag *Found* and all processors perform a **parallel reduction**.

    T(n,p) = O(log p + n/p)

- <u>CREW PRAM algorithm:</u> A similar approach, but *P1,...,Pp* can read *x* simultaneously in *O(1)* time. But the final reduction takes *O(log p)* time anyway, so
  T(n,p) = O(log p + n/p)

- <u>COMMON CRCW PRAM algorithm:</u> The final step takes now also *O(1)* time, those processors with the flag *Found* set can write simultaneously into *P0*'s cell
  T(n,p) = O(n/p).

Example CRCW-PRAM

- Initially
  - table **A** contains values 0 and 1
  - **output** contains value 0

```
for each $1 \le i \le 5$ do in parallel
    if $A[i] = 1$ then output=1;
```

- The program computes the "Boolean OR" of A[1], A[2], A[3], A[4], A[5]

Example CREW-PRAM

- Assume initially table **A** contains [0,0,0,0,0,1] and we have the following parallel program

```
for each $1 \le i \le 5$ do in parallel
  $A[i] ;= A[i] + A[i+1]$
```

$p_i : 1 \le i < 5$

then the consecutive values of the tables A (in parallel step 0, 1, 2, 3, 4, 5) correspond to the Pascal triangle, the nonzero elements in the n-th row are

$$\binom{n}{0},\ \binom{n}{1},\ \binom{n}{2},\ \ldots \binom{n}{n}$$

for $n \ = \ 0, 1, 2, 3, 4, 5, 6.$

Pascal triangle

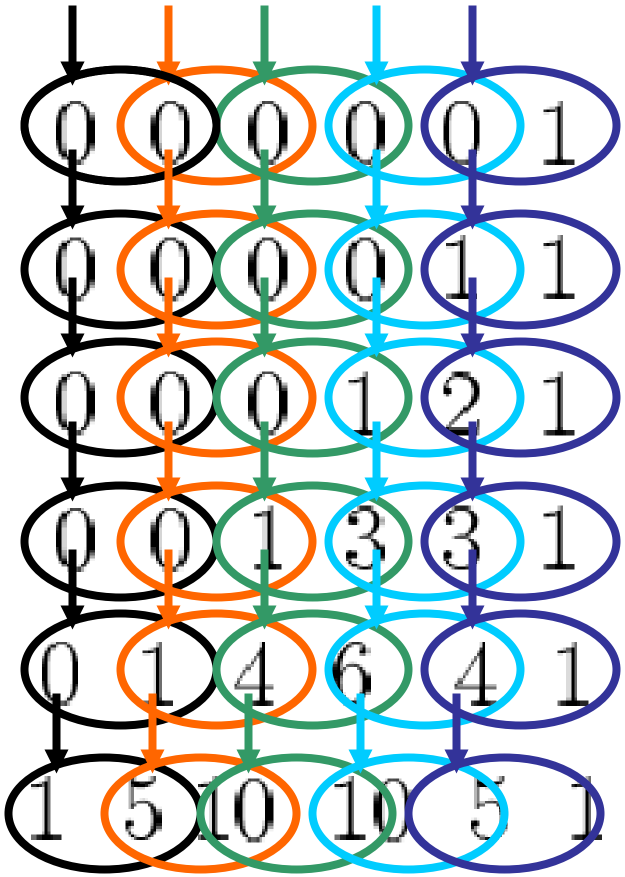
$$\binom{n}{0},\ \binom{n}{1},\ \binom{n}{2},\ \dots \binom{n}{n}$$

for $n\ =\ 0,1,2,3,4,5,6.$

$p_i : 1 \leq i < 5$

PRAM CREW (WT)

```
for each $1 \leq i \leq 5$ do in parallel
    $A[i] ;= A[i] + A[i+1]$
```



PRAM Variants =>Synchronization

- The standard PRAM posits a rigid execution pattern in which all processors are synchronized by a global clock.
- Several variants ease this restriction:
  - APRAM
  - Asynchronous PRAM (irregular synchronization points)
  - XPRAM (Periodic synchronization between intervals of asynchronous execution)
- **While these models incorporate synchronization, they do not charge an explicit cost for it !!!**

PRAM Variants => Latency

- The cost of non local memory accesses has a severe effect on the performance in massively parallel computers.

=>Several PRAMs were designed to remedy the unit memory cost idealization.

- LPRAM has been augmented by charging a cost of ***l*** units to access global memory.
- An elaboration of this model = BPRAM, augmented this by charging ***l*** units for the first message from global memory and a variable cost ***b*** for each additional memory access in a contiguous block.
- Thus the BPRAM provides incentives for one level of reference locality and for block transfer a form of data parallelism.

PRAM Variants => Bandwidth

- DRAM - A PRAM variant which assumes two classes of memory and includes a mechanism for assigning a non unit cost to a remote access
  - The DRAM is important because it eliminates the paradigm of global shared memory and replaces it with only private distributed memory.
  - While the topology of the communication network is ignored the DRAM incorporates the notion of limited bandwidth.
  - This model proposes a cost function for a nonlocal memory access which is based on the maximum possible congestion for a given data partition and execution sequence.
  - While the function is somewhat complicated it attempts to provide scheduling incentives to respect limited access to nonlocal data .
- Another PRAM variant proposed in the PRAMm incorporates bandwidth limitations by restricting the size of global shared memory to m memory locations The model is a CRCW PRAM except that in any given step only *m* accesses can be serviced.

Constrained PRAM models

- **Bounded** number of shared **memory cells.** This may be called a **small memory PRAM**.
  - If the input data set exceeds the capacity of the shared memory, the input and/or output values can be distributed evenly among the processors.
- **Bounded** size of a **machine word** and/or **memory cell**. This parameter is usually called **word size of PRAM.**
- **Bounded** number of **processors**.
  This may be called a **small PRAM**. If the number of threads of execution is higher, processors may interleave several threads.
- Constraints on simultaneous access to shared memory cells: handling **access conflicts(EREW, CRCW…)**.

Communication constraint

- Limiting the amount of PRAM shared memory corresponds to restricting the *amount of information that can be communicated between processors in one step*.

- For example, a distributed memory machine with processors interconnected by a shared bus can be modeled as a PRAM with one shared memory cell.

Computational power

- Having this range of submodels, we must ask how they compare as to the ability to execute parallel algorithms. Various submodels may have different **computational power**.

Definition

PRAM submodel A is **computationally stronger** than submodel B, written A>=B, if any algorithm written for B will run **unchanged** on A in the same parallel time, assuming the same basic properties.

Lemma

PRIORITY >= ARBITRARY >= COMMON >= CREW >= EREW

Simulation of large PRAMs on small PRAMs

- Small PRAMs can simulate larger PRAMs.
- Even though relatively simple, the following two simulations are very useful and notoriously used.
- The first result says that
  - **if we decrease the number of processors, the time complexity of a PRAM algorithm does not change, up to a multiplicative constant.**

Lemma 1

Assume $p' < p$.

Any problem that can be solved on a p-processor PRAM in t steps can be solved on a p'-processor PRAM in

$t' = O(t * p/p')$

steps assuming the same size of shared memory.

(see Lecture 6 – Building granularity)

Proof:

- Partition p simulated processors into p' groups of size p/p' each.
- Associate each of the p' simulating processors with one of these groups.
- Each of the simulating processors simulates one step of its group of processors by:
  - executing all their READ and local computation substeps first,
  - executing their WRITE substeps then.

Vectors sum

PRAM with p processors

Processors: $P_0 \ldots P_{p-1}$

Algorithm 1 (Vector sum on a PRAM)

Input: Vectors $v[1..n]$ and $w[1..n]$ in shared memory.

Output: Vector $z[1..n]$ in shared memory.

```
local integer $h$
for $h = 1$ to $\lceil n/p \rceil$ do
    if $(h-1)p + i \leqslant n$ then
        $z[(h-1)p+i] \leftarrow v[(h-1)p+i] + w[(h-1)p+i]$
    endif
enddo
```

•We assume that *h, I, n* and *p* are in the local memory.

•Under this assumption, all references to shared memory in Algorithm 1 are exclusive, and the algorithm requires only an EREW PRAM (Exclusive Read Exclusive Write).

•Algorithm 1 requires on the order of *n/p* steps to execute, so the parallel (concurrent) running time $T_c(n, p) = O(n/p)$.

Lemma 2

Assume $m' < m$.

Any problem that can be solved on a p-processor and m-cell PRAM in t steps can be solved on a $max(p, m')$-processor m'-cell PRAM in $O(t * m/m')$ steps.

Proof:

- Partition m simulated shared memory cells into m' continuous segments S_i of size m/m'.
- Each simulating processor P'_i, $1 <= i <= p$, will simulate processor P_i of the original PRAM.
- Each simulating processor P'_i, $1 <= i <= m'$, stores the initial contents of S_i into **its local memory** and will use $M'[i]$ as an auxiliary memory cell for simulation of accesses to cells of S_i.
- Simulation of one original READ operation:
  each P'_i, $i=1,...,max(p, m')$ repeats for $k=1,...,m/m'$:
  - write the value of the k-th cell of S_i into $M'[i]$, $i=1,...,m'$,
  - read the value which the simulated processor P_i, $i=1,...,p$, would read in this simulated substep, if it appeared in the shared memory.
- The local computation substep of P_i, $i=1,...,p$, is simulated in one step by P'_i.
- Simulation of one original WRITE operation is analogous to that of READ.

Model Equivalence

- given two models M_1 and M_2, and a problem Π of size n
- if M_1 and M_2 are equivalent then solving Π requires:
  - $T(n)$ time and $P(n)$ processors on M_1
  - $T(n)^{O(1)}$ time and $P(n)^{O(1)}$ processors on M_2

Simulation of stronger PRAM models on weaker ones

- It is very useful to know efficient simulations of stronger PRAM models on weaker ones, since a stronger model is more convenient for the design of algorithms, whereas weaker models, such as EREW, are closer to real parallel computers.
- Since it is technologically difficult to build full massively parallel CREW or CRCW PRAM computers, it is important to understand the costs of simulating the CREW or CRCW machines on EREW.
- Any multiple access has to be converted into a series of exclusive accesses.
- The most important are simulations of the strongest PRIORITY CRCW on the weakest EREW.

Lemma

Assume PRIORITY CRCW with the priority scheme based trivially on indexing: lower indexed processors have higher priority.

One step of *p*-processor *m*-cell PRIORITY CRCW can be simulated by a *p*-processor *mp*-cell EREW PRAM in *O(log p)* steps.

Proof:(Constructive.)

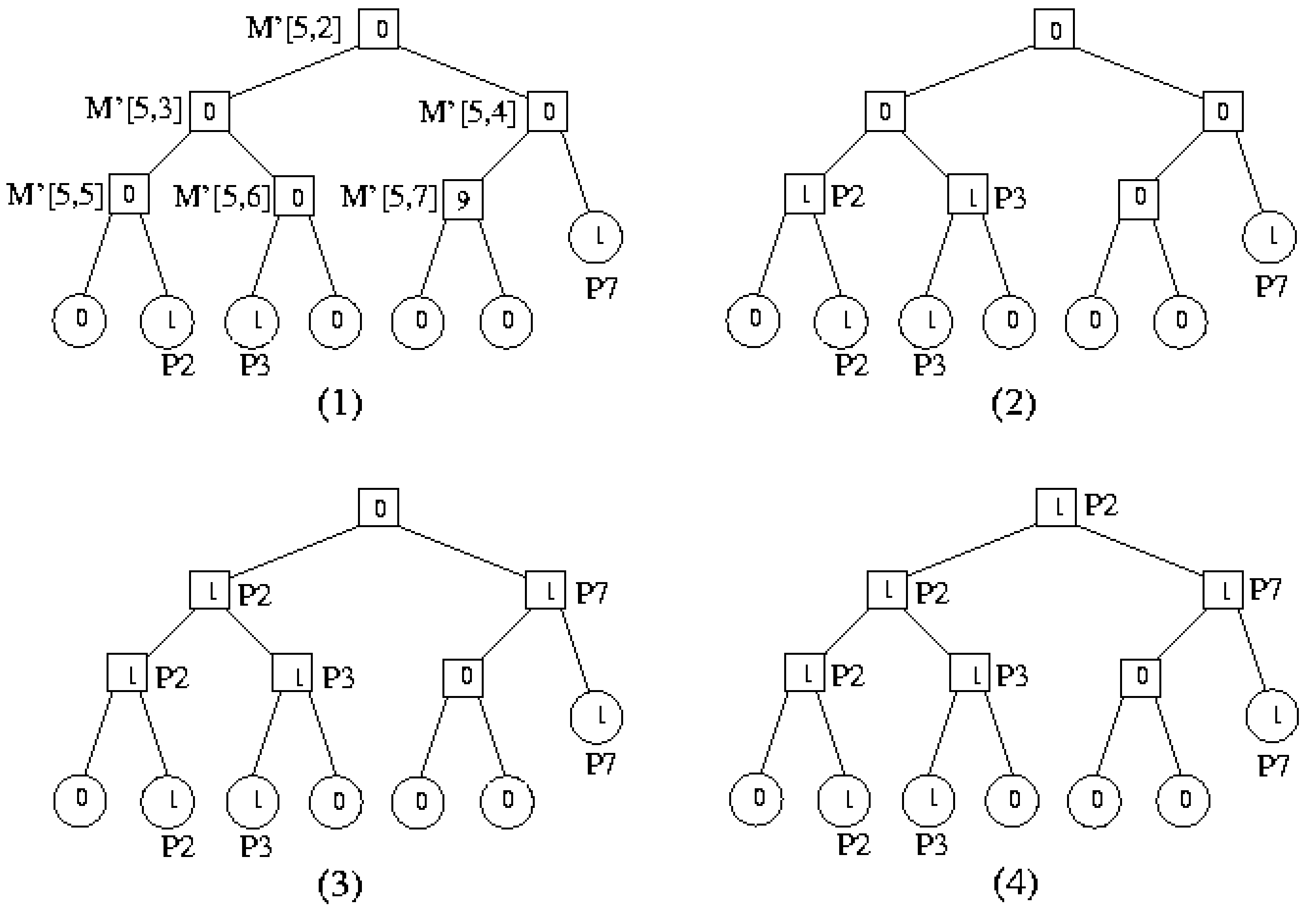
- Each PRIORITY processor *Pk* is simulated by EREW processor *P'k*.
- Each shared memory cell *M[i]*, *i=1,...,m*, of PRIORITY is simulated by an **array** of *p* shared memory cells *M'[i,k]*, *k=1,...,p*, in the EREW PRAM.
  - *M'[i,1]* plays the role of *M[i]*.
  - *M'[i,2],...,M'[i,p]* are **auxiliary cells** used for resolving conflicts, initially **empty**, and organized as **internal nodes(p-1) of a complete binary tree** *Ti* with *p* leaves. The height of every *Ti* is *[log p]*.

- **Simulation of a PRIORITY WRITE substep:** Each EREW processor must find out whether it is the processor with the lowest index within the group of processors asking to write to the same cell, and if so, it must become the group winner and perform the WRITE operation. The other processors in the group will fail. This is done as follows:
  - If *Pk* wants to write into *M[i]*, processor *P'k* turns **active** and becomes *k*-th leaf of *Ti*. It knows whether it is the right or left child of its parent.
  - Each active **left** processor stores its ID into the parent cell in its tree, marks it as **occupied**, and remains active.
  - Each active **right** processor checks its parent cell. If this is empty, it stores its ID into it, and remains active. If it is occupied, it becomes idle (he lost).
  - This is repeated *log p* times at further levels of the trees.
  - The processor who managed to proceed to the root of *Ti*, becomes the winner who can write into *M[i]*. Processors which used *Ti* must then sweep down *Ti* in the reverse order to reset the cells *M'[i,2],...,M'[i,p]* to empty.

- **Simulation of a PRIORITY READ substep:** is similar.
  - The same sweep-ups of the auxiliary trees are performed in parallel to determine the winners in the groups.
  - The winners will read the values from the cells *M'[*,1]*
  - During the cleaning sweep-down, the read value is distributed to the losers.

Example:

p=7, processors *P2, P3*, and *P7* wish to write into *M[5]*.



Lemma

One step of PRIORITY CRCW with *p* processors and *m* shared memory cells can be simulated by an EREW PRAM in *O(log p)* steps with *p* processors and *m+p* shared memory cells.

Proof:(Constructive.)

- Each PRIORITY processor *Pk* is simulated by EREW processor *P'k*.
- Each PRIORITY cell *M[i]* is simulated by EREW cell *M[i]*.
- EREW uses an **auxiliary** array *A* of *p* cells.

- If *Pk* wants to access *M[i]*, processor *P'k* writes pair *(i,k)* into *A[k]*. If *Pk* does not want to access any PRIORITY cell, processor *P'k* writes *(0,k)* into *A[k]*.
- All *p* processors sort the array *A* into lexicographic order using *(log p)*-time parallel sort.
  - Each *P'k* appends to cell *A[k]* a flag *f*:

  f=0, if the first component of A[k] is either 0 or it is the same as the first component of A[k-1]

  f=1 otherwise.

  Further steps differ for simulation of WRITE or READ.

- **PRIORITY WRITE:**
  - Each *P'k* reads the triple *(i,j,f)* from cell *A[k]* and writes it into *A[j]*.
  - Each *P'k* reads the triple *(i,k,f)* from cell *A[k]* and writes into *M[i]* iff *f=1*.
- **PRIORITY READ:**
  - Each *P'k* reads the triple *(i,j,f)* from cell *A[k]*.
  - If *f=1*, it reads value *vi* from *M[i]* and overwrites the third component in *A[k]* the flag *f*, with *vi*.
  - In at most *log p* steps, this third component is then distributed in subsequent cells of *A* until it reaches either the end or an element with a different first component.
  - Each *P'k* reads the triple *(i,j,vi)* from cell *A[k]* and writes it into *A[j]*.
  - Each *P'k* who asked for a READ reads the value *vi* from the triple *(i,k,v)* in cell *A[k]*.

Example

Assume *p=7*, *m=4*, and

- *P1* wants to access *M[2]*,
- *P2* wants to access *M[4]*,
- *P3* wants to access *M[2]*,
- *P4* wants to access *M[1]*,
- *P5* wants to access *M[4]*,
- *P6* wants to access *M[2]*,
- *P7* wants to access no cell at all.

- Array *A* in the first three steps of simulation:

(2,1,)	(4,2,)	(2,3,)	(1,4,)	(4,5,)	(2,6,)	(0,7,)
(0,7,)	(1,4,)	(2,1,)	(2,3,)	(2,6,)	(4,2,)	(4,5,)
(0,7,0)	(1,4,1)	(2,1,1)	(2,3,0)	(2,6,0)	(4,2,1)	(4,5,0)

- Array *A* in simulation of WRITE:

(2,1,1)	(4,2,1)	(2,3,0)	(1,4,1)	(4,5,0)	(2,6,0)	(0,7,0)

- Array *A* in simulation of READ:

(0,7,0)	$(1,4,v_1)$	$(2,1,v_2)$	(2,3,0)	(2,6,0)	$(4,2,v_4)$	(4,5,0)
(0,7,0)	$(1,4,v_1)$	$(2,1,v_2)$	$(2,3,v_2)$	(2,6,0)	$(4,2,v_4)$	$(4,5,v_4)$
(0,7,0)	$(1,4,v_1)$	$(2,1,v_2)$	$(2,3,v_2)$	$(2,6,v_2)$	$(4,2,v_4)$	$(4,5,v_4)$
$(2,1,v_2)$	$(4,2,v_4)$	$(2,3,v_2)$	$(1,4,v_1)$	$(4,5,v_4)$	$(2,6,v_2)$	(0,7,0)

Conclusion

- Any polylog-time PRAM algorithm is robust with respect to PRAM models.

PRAM is an attractive and important model for designers of parallel algorithms. Why?

- It is **natural**: the number of operations executed per one cycle on *p* processors is at most *p*.
- It is **strong**: any processor can read or write **any** shared memory cell in unit time.
- It is **simple**: it abstracts from any communication or synchronization overhead, which makes the complexity and correctness analysis of PRAM algorithms easier. Therefore,
- It can be used as a **benchmark**: If a problem has no feasible/efficient solution on PRAM, it has no feasible/efficient solution on any parallel machine.
- It is **useful**: it is an idealization of existing (and nowaday more and more abundant) shared memory parallel machines.

THE PRAM IS A THEORETICAL (UNFEASIBLE) MODEL

- The interconnection network between processors and memory would require a very large amount of area .
- The message-routing on the interconnection network would require time proportional to network size (i. e. the assumption of a constant access time to the memory is not realistic).

WHY THE PRAM IS A REFERENCE MODEL?

- Algorithm's designers can forget the communication problems and focus their attention on the parallel computation only.
- There exist algorithms simulating any PRAM algorithm on bounded degree networks.

E. G. A PRAM algorithm requiring time T(n), can be simulated in a mesh of tree in time $T(n)\log_2 n/\log\log n$, that is each step can be simulated with a slow-down of $\log_2 n/\log\log n$.

- Instead of design ad hoc algorithms for bounded degree networks, design more general algorithms for the PRAM model and simulate them on a feasible network.

- For the PRAM model there exists a well developed body of techniques and methods to handle different classes of computational problems.

- The discussion on parallel model of computation is still HOT

The actual trend:

COARSE-GRAINED MODELS

- The degree of parallelism allowed is independent from the number of processors.

- The computation is divided in supersteps, each one includes

  - local computation
  - communication phase
  - syncronization phase

Brent's Theorem

- Relates the work and time complexities of a parallel algorithm described in the WT formalism to its running time on a *p*-processor PRAM.

Theorem 1 (Brent 1974) *A WT algorithm with (depth) step complexity S(n) and work complexity W(n) can be simulated on a p-processor PRAM in no more than* $\lfloor \frac{W(n)}{p} \rfloor + S(n)$ *parallel steps.*

Brent's Theorem allows conversion of a WT algorithm into a bounded PRAM algorithm.

Proof:

For each time step i, for $1 \le i \le S(n)$, let $W_i(n)$ be the number of operations in that step. We simulate each step of the WT algorithm on a p-processor PRAM in $\lceil \frac{W_i(n)}{p} \rceil$ parallel steps, by scheduling the $W_i(n)$ operations on the p processors in groups of p operations at a time.

The last group may not have p operations if p does not divide $W_i(n)$ evenly. In this case, we schedule the remaining operations among the smallest-indexed processors.

Given this simulation strategy, the time to simulate step $W_i(n)$ of the WT algorithm will be $\lceil \frac{W_i(n)}{p} \rceil$

and the total time for a p processor PRAM to simulate the algorithm is

$$\sum_{i=1}^{S(n)} \lceil \frac{W_i(n)}{p} \rceil \leqslant \sum_{i=1}^{S(n)} (\lfloor \frac{W_i(n)}{p} \rfloor + 1) \leqslant \lfloor \frac{W(n)}{p} \rfloor + S(n).$$

Remark

- There are a number of complications that this simple sketch of the simulation strategy does not address.
- For example, to preserve the semantics of the **forall** construct,
  - we should generally not update any element of the left-hand side of a WT assignment until we have evaluated all the values of the right-hand side expression.
  - This can be accomplished by the introduction of a temporary result that is subsequently copied into the left hand side.

Computational Networks

A *Computational Network* is a directed acyclic graph whose vertices are subdivided into three sets:

Input vertices: These vertices have no incoming edges.

Output vertices: These vertices have no outgoing edges.

Interior vertices: These vertices have incoming edges and a single outgoing edge.

- Each interior vertex is labeled with an elementary operation.
- The number of incoming edges to an interior vertex is called ***fan-in***, and
- the number of outgoing edges is called ***fan-out****.*
- The maxima of these two quantities over entire graph is called, respectively, **the fan-in and the fan-out of the graph.**

- The length of the longest path from any input vertex to any output vertex is called ***depth*** of the computational network.

A *computation* on a computational network

The *computation* performed by a computation network, on a given set of inputs is defined to be the data that appears on the output vertices as a result of the following procedure:

- Apply the input data to the input vertices.
- Transmit data along directed edges. Whenever an interior vertex is encountered, wait until data arrives along all of its incoming edges, and then perform the indicated elementary operation. Transmit the result of the computation along all of the outgoing edges.
- The procedure terminates when there is no data at the interior vertices.

Brent's Theorem (with CN formulation)

Let N be a computational network with n interior nodes and depth d, and bounded fan-in.

Then the computations performed by N can be carried out by a CREW-PRAM computer with p processors in time O(n/p+d).

(The total time depends upon the fan-in of *N* -- we consider it as a constant of proportionality.)

->This result is normally applied to modifying parallel algorithms to use fewer processors.

Proof:

- In order to simulate the computation of *N* we define the ***depth*** of each vertex - v, *n,* of *N*, to be the maximum length of any path from an input vertex to *v*
  - clearly this is the greatest distance that any amount of input-data has to travel to reach *v*.

- We assume that that we have a data-structure in the memory of PRAM for encoding a vertex of *N*, and that has a field with
  - a pointer to the vertex that receives the output of its computation, if it is an interior vertex.
  - This field is *null* if the vertex is an output vertex.

- The simulation is done inductively – >

  we simulate all of the computations that take place at vertices of depth $\leq k$-1 before simulating the computation on vertices of depth k-1.

  Suppose that there are $\boldsymbol{n_i}$ interior vertices of N whose depth is precisely i; then (sum i: $1 \leq i \leq d : n_i) = n$.

  After simulating the computations on vertices of depth k-1, the computations on nodes of depth k could be simulated, since the inputs of these nodes are now available.

Proposition

When performing the computations on nodes of depth k, the order of the computations is irrelevant.

This is due to the definition of depth – it implies that the output of any vertex of depth k is input to a vertex of strictly higher depth (since depth is the length of the longest path from an input vertex to the vertex in question).

The simulation of the computations at depth *k*

- Processors read the data from the output areas of the data-structures for vertices at depth k-1.
- Processors perform required computations.
- Since there are n_i nodes of depth i, and the computations can be performed in any order,
  the execution-time of this phase is $[n_i / p] \leq n_i / p + 1$.
- The total execution-time is thus

(sum i : $1 \leq i \leq d$: $[n_i / p]$) $\leq$ (sum i: : $1 \leq i \leq d$: n_i / p +1) = n / p + d

Corollary

If a computational network has

- bounded fan-out and
- bounded fan-in

we have:

Let N be a computational network with n interior vertices and depth d, and bounded fan-in and fan-out.

Then the computation can be performed by a CREW-PRAM computer with p processors in time $O(n/p+d)$.

Brent Scheduling Principle

- This principle specifies that it is possible to reduce the number of processors used in parallel algorithms, without increasing the asymptotic execution time. In general, the execution time increases somewhat when the number of processors is reduced.

- Suppose algorithm A has the property that its computations can be expressed in terms of a computational network with *n* vertices, and depth *d* and has a bounded fan-in. Then algorithm A can be executed on a CREW-PRAM computer with *p* processors in time $O(n/p+d)$.

Example

- Adding the numbers given in an array of length n.
  A possible computational network is a binary tree with n leaves.

  - If $n=2^k$, then the binary tree has the depth equal to k, and the interior nodes are equal to 2^{k-1}.
  - If we use 2^{k-1} processors the time complexity is equal to k.
  - If we use $(2^{k-1})/$ k, the time complexity remains $O(k)$.

- Brent’s theorem has interesting implications related to *work efficiency* of an algorithm.

Remind…

- The amount of work performed by a parallel algorithm is the total number of computations executed by that algorithm

- A parallel algorithm is said to be *cost-optimal* if $(pT_P) = \Theta(T_S)$.
  $E = T_S / (pT_P)$ => for cost optimal systems $E = O(1)$.

scaling down…again

- Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called *scaling down* a parallel system.
- A naive way of scaling down is to think of each processor in the original case as a virtual processor and to assign virtual processors equally to scale down processors.
- Since the number of processing elements decreases by a factor of ***n*** / ***p***, the computation at each processing element increases by a factor of ***n*** / ***p***.
- The communication cost should not increase by this factor since some of the virtual processors assigned to a physical processor might talk to each other. This is the basic reason for the improvement from building granularity.

Example

Algorithm 2 (Sequence reduction, WT description)
Input: Sequence a with $n = 2^k$ elements of type T, binary associative operator $\oplus : T \times T \to T$.
Output: $S = \oplus_{i=1}^{n} a_i$.

```
$T$ REDUCE(**sequence**⟨T⟩ $a$, $\oplus : T \times T \to T$)

$T\ B[1..n]$
**forall** $i \in 1:n$ **do**
    $B[i] \leftarrow a_i$
**enddo**
**for** $h = 1$ **to** $k$ **do**
    **forall** $i \in 1:n/2^h$ **do**
        $B[i] \leftarrow B[2i-1] \oplus B[2i]$
    **enddo**
**enddo**
$S \leftarrow B[1]$
**return** $S$
```

Algorithm 3 (Sequence reduction, PRAM description)
Input: Sequence a with $n = 2^k$ elements of type T, binary associative operator $\oplus : T \times T \to T$, and processor id i.
Output: $S = \oplus_{i=1}^{n} a_i$.

```
$T$ PRAM-REDUCE(**sequence**⟨T⟩ $a$, $\oplus : T \times T \to T$)

T B[1..n]
local integer $h$, $j$, $\ell$
for $\ell = 1$ to $\lceil n/p \rceil$ do
    if $(\ell - 1)p + i \leqslant n$ then
        $B[(\ell - 1)p + i] \leftarrow a_{(\ell-1)p+i}$
    endif
enddo
for $h = 1$ to $k$ do
    for $\ell = 1$ to $\lceil \frac{n/2^h}{p} \rceil$ do
        $j \leftarrow (\ell - 1)p + i$
        if $j \leqslant n/2^h$ then
            $B[j] \leftarrow B[2j - 1] \oplus B[2j]$
        endif
    enddo
enddo
if $i = 1$ then
    $S \leftarrow B[1]$
endif
return $S$
```

$$\boxed{T_C(n,p) = \lceil \frac{n}{p} \rceil \Theta(1) + \sum_{h=1}^{k} \lceil \frac{n/2^h}{p} \rceil \Theta(1) + \Theta(1) = O(\frac{n}{p} + \lg n)}$$

Brent Theorem - interpretation

$$T = d + (W-d)/p$$

- Brent's theorem specifies for a sequential algorithm with d time steps, and a total of W operations, that a runtime T is definitely possible on a shared memory machine (PRAM) with p processors.

Proof:

$(\text{sum } i : 1 \leq i \leq d : \lceil n_i / p \rceil) \leq (\text{sum } i : 1 \leq i \leq d : (n_i + p - 1)/ p)$
$\quad = (n-d) / p + d$

- There may be an algorithm that solves this problem faster, or it may be possible to implement this algorithm faster (by scheduling instruction differently to minimize idle processors, for instance), but it is definitely possible to implement this algorithm in this time given p processors.

Sum – again… (1)

Variant 1: step through the array, adding each value in turn to an accumulator.

```
for (i=0;i < length(a); i++) { sum = sum + a(i); }
```

There are n operations, so $W = n$.

$T = n + 0/p$.

$(W-s) = 0$,

so no matter how many processors are available, this algorithm will take time n.

Sum – again… (2)

sum(a) = ((A0 + A1) + (A2 + A3)) + ((A4 + A5) + (A6 + A7)) etc...

For an array of length n, the longest chain(s) will be of length **log(n).**
s = log(n).
W = n (as before).

T = log(n) + (n - log(n))/p.

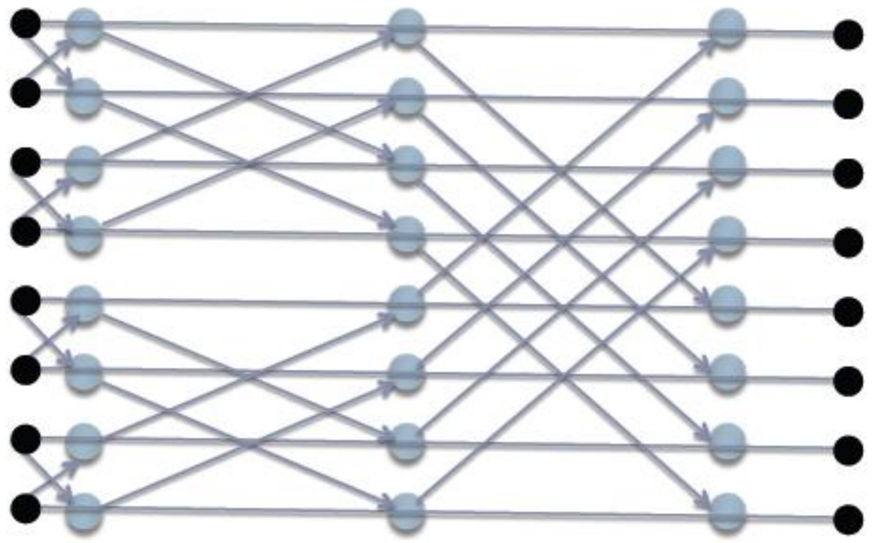
Remarks:

- No matter how many processors are used, there can be no implementation of this algorithm that runs faster than O (log(n)).
- If we have n processors, the algorithm can be implemented in log(n) time
- If we have one processor, the algorithm can be implemented in n time.
- If we consider the amount of work done in each case, with one processor, we do n work, with log(n) processors we do n work, but with n processors we do nlog(n) work.
  - The implementations with 1 or log(n) processors, therefore are cost optimal, while the implementation with n processors is not.

Conclusion

- It is important to remember that Brent's theorem does not tell us how to implement any of these algorithms in parallel;
  - it merely tells us what is possible.
- The Brent's theorem implementation may be hideously ugly compared to the naive implementation.
- Key to understanding Brent's theorem is understanding time steps.
  In a single time step every instruction that has no dependencies is executed, and therefore **t** is equal to the length of the longest chain of instructions that depend on the results of other instructions (as any shorter chains will be finished executing by (or before) the time the longest chain has).

Radix 2 (FFT Graph)



- N inputs, N outputs, Size = NlgN, Depth = lgN
- The reverse of an FFT graph is an FFT graph

FFT Graph Formula

- At stage i, node $a_n \ldots a_1$ has
  - inputs $a_n \ldots a_{i+1}0\ a_{i-1} \ldots a_1$ and $a_n \ldots a_{i+1}1\ a_{i-1} \ldots a_1$
  - outputs an…$a_{i+2}\ 0\ a_i \ldots a_1$ and $a_n \ldots a_{i+2}1\ a_i \ldots a_1$
- Reverse FFT: Same structure, with address bits reversed

FFT Computation

- At stage i, node $a_n \ldots a_1$ has
  - inputs $a_n \ldots a_{i+1} 0 a_{i-1} \ldots a_1$ and $a_n \ldots a_{i+1} 1 a_{i-1} \ldots a_1$
  - outputs $a_n \ldots a_{i+2} 0 a_i \ldots a_1$ and an$\ldots a_{i+2} 1 a_i \ldots a_1$

- Assume p = 2k, n≥2k (p≤√N)

1. Each processor computes on a block of contiguous

inputs $a_n \ldots a_{n-k+1} x \ldots x$ (p distinct blocks) – n-k stages

2. Transpose: $a_n \ldots a_{n-k+1} a_{n-k} \ldots a_1$ □ $a_{n-k} \ldots a_1$ $a_n \ldots a_{n-k+1}$

3. Each processor computes on a block of contiguous inputs – k stages.

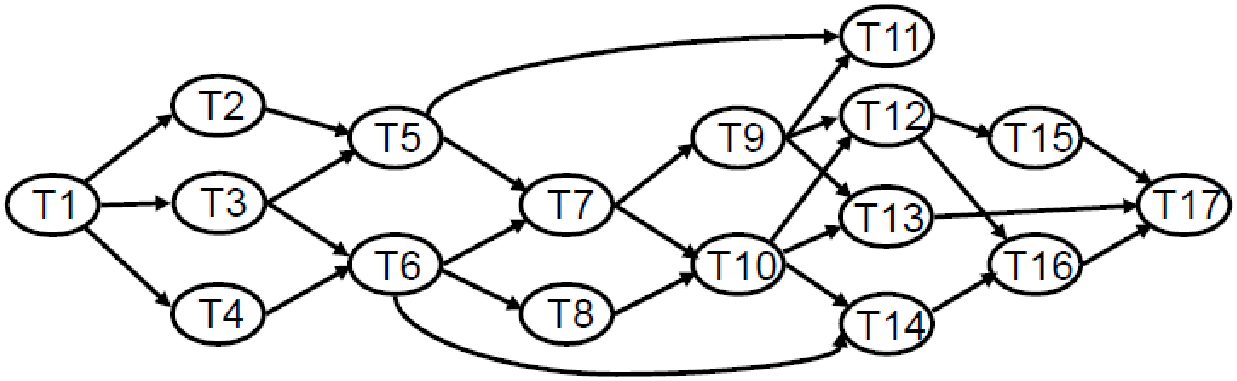
- Communication volume N, communication depth 1

Task decomposition -Task graph

- Divide work into tasks that can be executed concurrently
- Many different decompositions possible for any computation
- Tasks may be same, different, or even indeterminate sizes
- Tasks may be independent or have non-trivial orde
- Conceptualize tasks and ordering as task dependency DAG

DAG - example

—node = task
—edge = control dependency



Different task decompositions may lead to significant differences with respect to their eventual parallel performance.

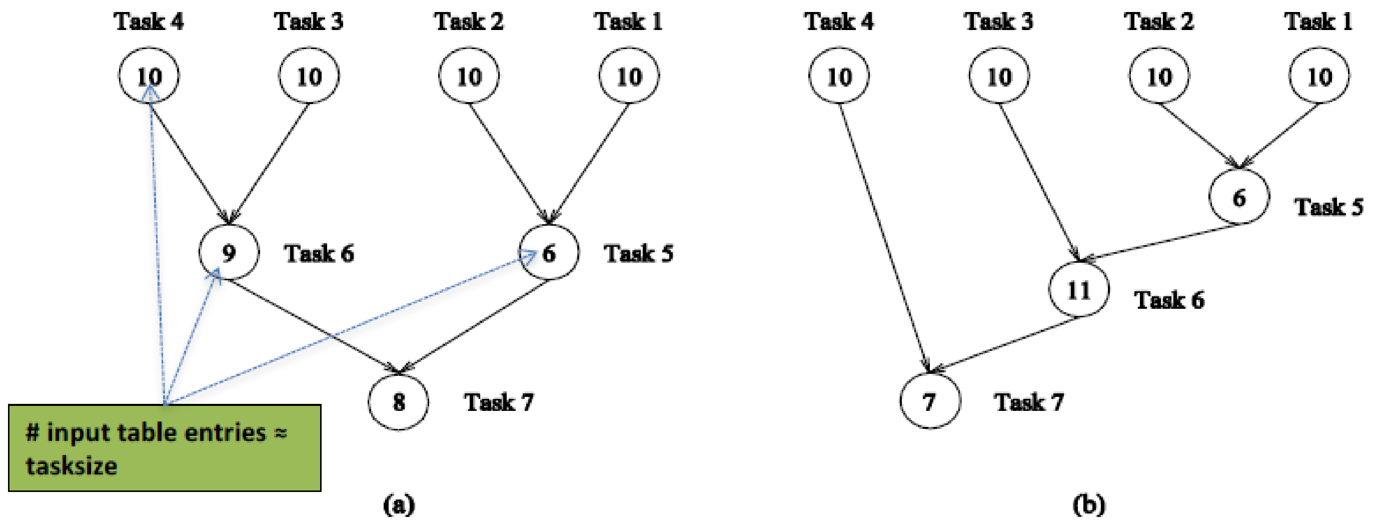
Data Dependencies

- **True(Flow)-Dependence:** Task 1 computes a value stored at A, and Task 2 retrieves a value stored at A.
- **Anti-Dependence:** Task 1 retrieves a value stored at A, and Task 2 computes a value stored at A
- **Input-Dependence:** Task 1 and Task 2 both retrieve a value stored at A
- **Output-Dependence:** Task 1 and Task 2 both compute a value stored at A

Critical Path Length

- A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.
- The longest path determines the shortest time in which the program can be executed in parallel.
- The length of the longest path in a task dependency graph is called the critical path length.

Examples- two database query decompositions:



Interaction vs Dependency graphs

- Subtasks generally exchange data with others in a decomposition.
- The graph of tasks (nodes) and their interactions/data exchange (edges) is referred to as a task interaction graph.
- Task interaction graphs represent input- and output-dependencies, whereas task dependence graphs represent true-dependencies or anti-dependencies.