ĐẠI HỌC
BÁCH KHOA

25 YEARS ANNIVERSARY
SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Parallel Algorithm Design

References

- Michael J. Quinn. **Parallel Computing. Theory and Practice**. McGraw-Hill
- Albert Y. Zomaya. **Parallel and Distributed Computing Handbook**. McGraw-Hill
- Ian Foster. **Designing and Building Parallel Programs**. Addison-Wesley.
- Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar **. Introduction to Parallel Computing, Second Edition.** Addison Wesley.
- Joseph Jaja**. An Introduction to Parallel Algorithm.** Addison Wesley.
- Nguyễn Đức Nghĩa**. Tính toán song song.** Hà Nội 2003.

2.1 Parallel Programming Models

Parallel Programming Models

- Overview
- Shared Memory Model
- Threads Model
- Message Passing Model
- Data Parallel Model
- Other Models

Overview

- There are several parallel programming models in common use:
  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Hybrid
- Parallel programming models exist as an abstraction above hardware and memory architectures.

Overview

- Although it might not seem apparent, these models are NOT specific to a particular type of machine or memory architecture. In fact, any of these models can (theoretically) be implemented on any underlying hardware.
- Which model to use is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.
- The following sections describe each of the models mentioned above, and also discuss some of their actual implementations.

Shared Memory Model

- In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. Program development can often be simplified.
- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality.

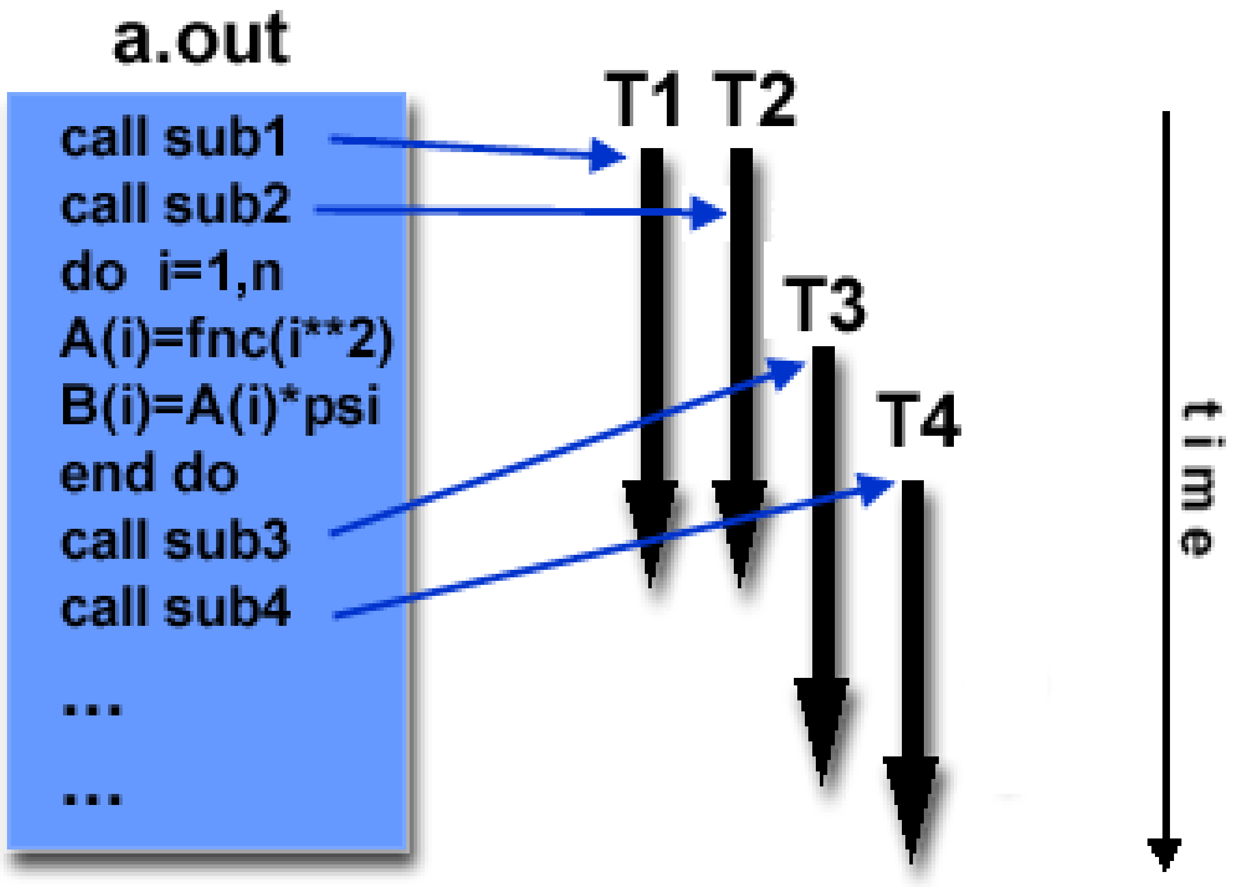
Shared Memory Model: Implementations

- On shared memory platforms, the native compilers translate user program variables into actual memory addresses, which are global.
- No common distributed memory platform implementations currently exist.

Threads Model

- In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.
- Perhaps the most simple analogy that can be used to describe threads is the concept of a single program that includes a number of subroutines:
  - The main program **a.out** is scheduled to run by the native operating system. a.out loads and acquires all of the necessary system and user resources to run.
  - a.out performs some serial work, and then **creates a number of tasks (threads)** that can be scheduled and run by the operating system concurrently.
  - **Each thread has local data**, but also, **shares the entire resources of a.out**. This saves the overhead associated with replicating a program's resources for each thread. Each thread also benefits from a global memory view because it shares the memory space of a.out.
  - A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
  - **Threads communicate** with each other **through global memory** (updating address locations). This requires synchronization constructs to insure that more than one thread is not updating the same global address at any time.
  - Threads can come and go, but a.out remains present to provide the necessary shared resources until the application has completed.
- Threads are commonly associated with shared memory architectures and operating systems.

Threads Model



```
call sub1
call sub2
do  i=1,n
A(i)=fnc(i**2)
B(i)=A(i)*psi
end do
call sub3
call sub4
...
...
```

Threads Model Implementations

- From a programming perspective, threads implementations commonly comprise:
  - A library of subroutines that are called from within parallel source code
  - A set of compiler directives imbedded in either serial or parallel source code
- In both cases, the programmer is responsible for determining all parallelism.
- Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.
- **POSIX Threads**
  - Library based; requires parallel coding
  - Specified by the IEEE POSIX 1003.1c standard (1995).
  - C Language only
  - Commonly referred to as Pthreads.
  - Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations.
  - Very explicit parallelism; requires significant programmer attention to detail.

Threads Model: OpenMP

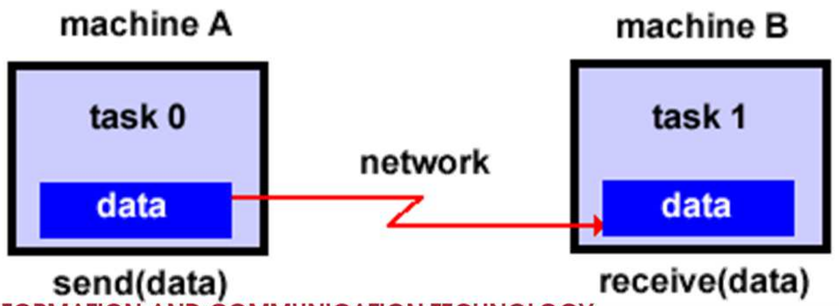
- **OpenMP**
  - Compiler directive based; can use serial code
  - Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998.
  - Portable / multi-platform, including Unix and Windows NT platforms
  - Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use - provides for "incremental parallelism"
- Microsoft has its own implementation for threads, which is not related to the UNIX POSIX standard or OpenMP.

Message Passing Model

- The message passing model demonstrates the following characteristics:
  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Message Passing Model Implementations: MPI Standard

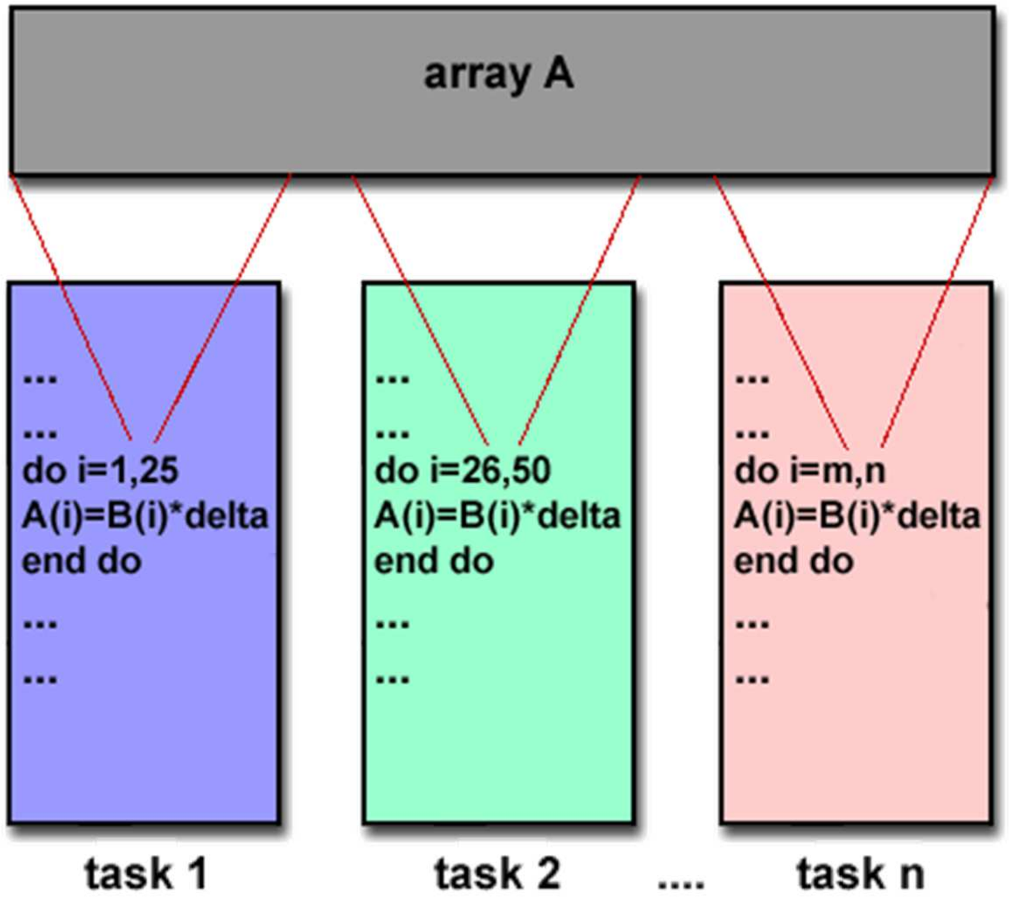
- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded in source code. The programmer is responsible for determining all parallelism.
- Part 1 of the **Message Passing Interface (MPI)** Standard was released in 1994. Part 2 (MPI-2) was released in 1996.
- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI. A few offer a full implementation of MPI-2.
- For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons.



Data Parallel Model

- The data parallel model demonstrates the following characteristics:
    - Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
    - A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
    - Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
- On shared memory architectures, all tasks may have access to the data structure through global memory. On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task.

Data Parallel Model



```
...
...
do i=1,25
A(i)=B(i)*delta
end do
...
...
```

```
...
...
do i=26,50
A(i)=B(i)*delta
end do
...
...
```

```
...
...
do i=m,n
A(i)=B(i)*delta
end do
...
...
```

Other Models

- Other parallel programming models besides those previously mentioned certainly exist, and will continue to evolve along with the ever changing world of computer hardware and software.
- Only three of the more common ones are mentioned here.
  - Hybrid
  - Single Program Multiple Data
  - Multiple Program Multiple Data

Hybrid

- In this model, any two or more parallel programming models are combined.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP). This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines.
- Another common example of a hybrid model is combining data parallel with message passing. As mentioned in the data parallel model section previously, data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data between tasks, transparently to the programmer.

Single Program Multiple Data (SPMD)

- Single Program Multiple Data (SPMD):
- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- A single program is executed by all tasks simultaneously.
- At any moment in time, tasks can be executing the same or different instructions within the same program.
- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
- All tasks may use different data

Multiple Program Multiple Data (MPMD)

- Multiple Program Multiple Data (MPMD):
- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executing the same or different program as other tasks.
- All tasks may use different data

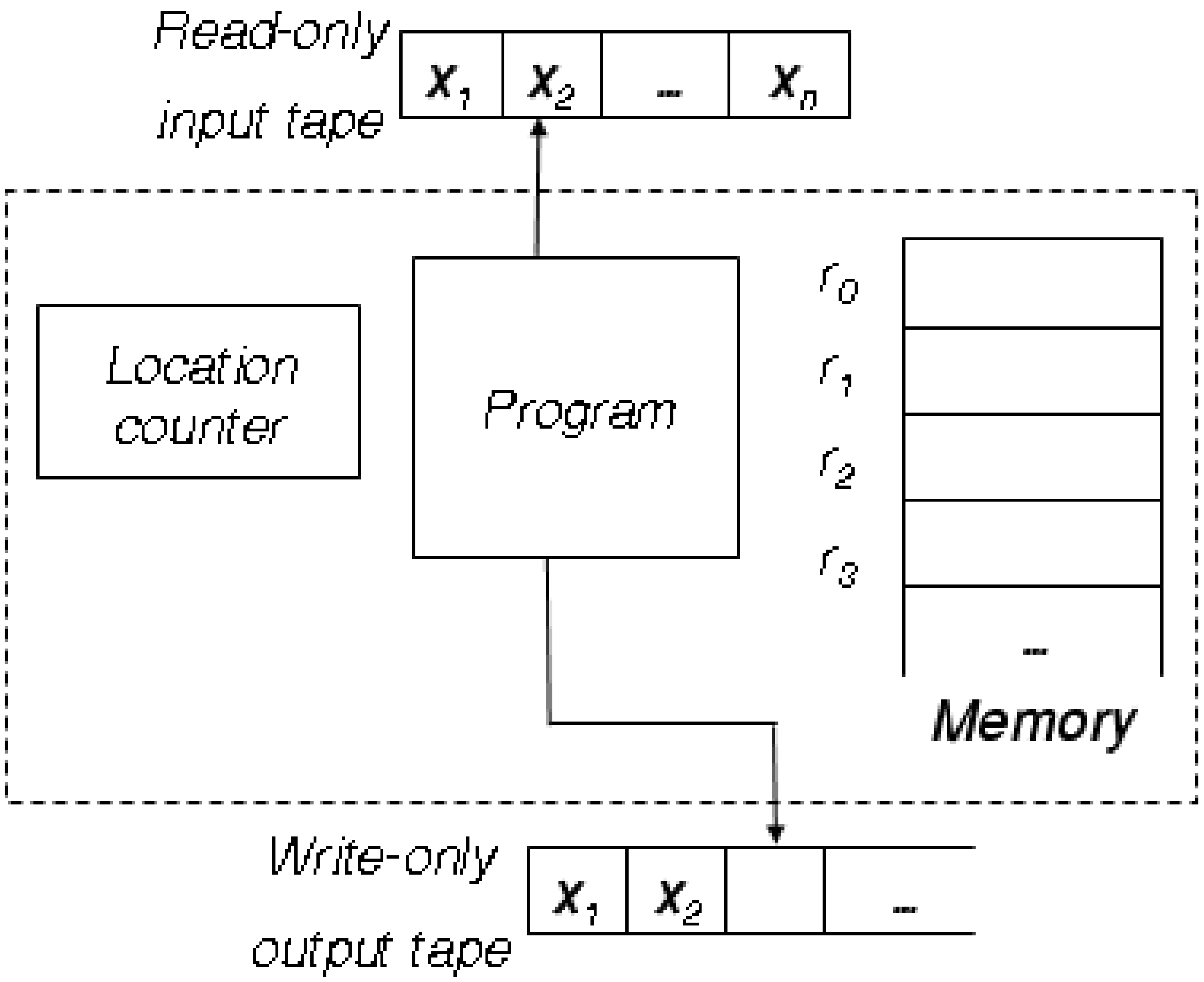
Basic Parallel Algorithms

Parallel Random Access Machine

Random Access Machine Model of Computation

- The Random Access Machine (or RAM) model for sequential computation.
- Assume that the memory has M memory locations, where M is a large (finite) number
- Accessing memory can be done in unit time.
- Instructions are executed one after another, with no concurrent operations.
- The input size depends on the problem being studied and is the number of items in the input
- The running time of an algorithm is the number of primitive operations or steps performed.

Random Access Machine



PRAM (Parallel Random Access Machine)

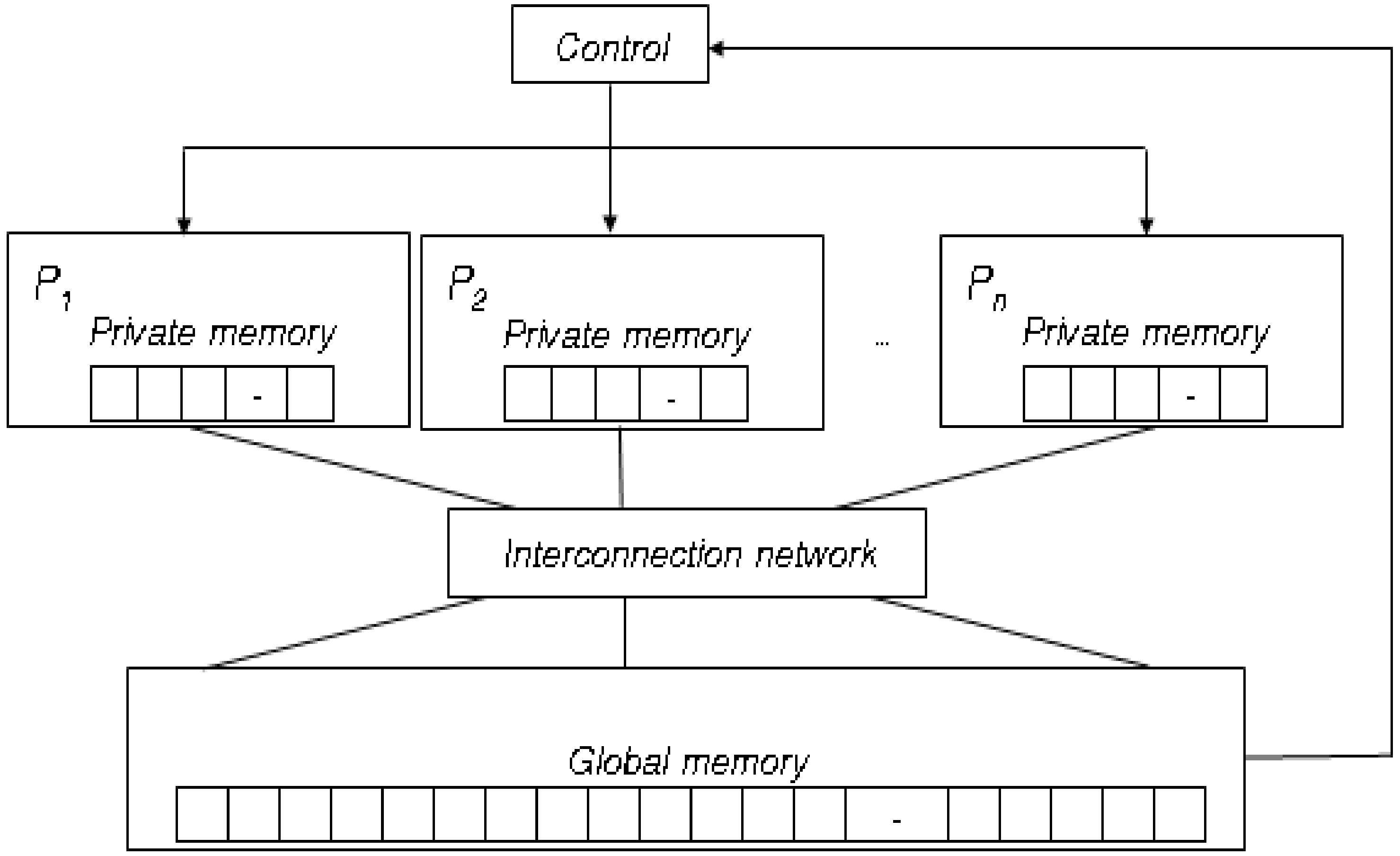
- PRAM is a natural generalization of the RAM sequential model.
- Each of the *p* processors P_0, P_1, … , P_{p-1} are identical to a RAM processor and are often referred to as *processing elements* (PEs) or simply as *processors*.
- All processors can read or write to a shared global memory in parallel (i.e., at the same time).
- The processors can also perform various arithmetic and logical operations in parallel
- Running time can be measured in terms of the number of parallel memory accesses an algorithm performs.

PRAM Properties

- An unbounded number of processors all can access
- All processors can access an unbounded shared memory
- All processor's execution steps are synchronized
- However, processors can run different programs.
  - Each processor has an unique id, called the pid
  - Processors can be instructed to do different things, based on their pid (if pid < 200, do this, else do that)

Parallel Random Access Machine

- PRAM



The PRAM Model

- **P**arallel **R**andom **A**ccess **M**achine
  - Theoretical model for parallel machines
  - p processors with uniform access to a large memory bank
  - UMA (uniform memory access) – Equal memory access time for any processor to any address

Types of PRAM

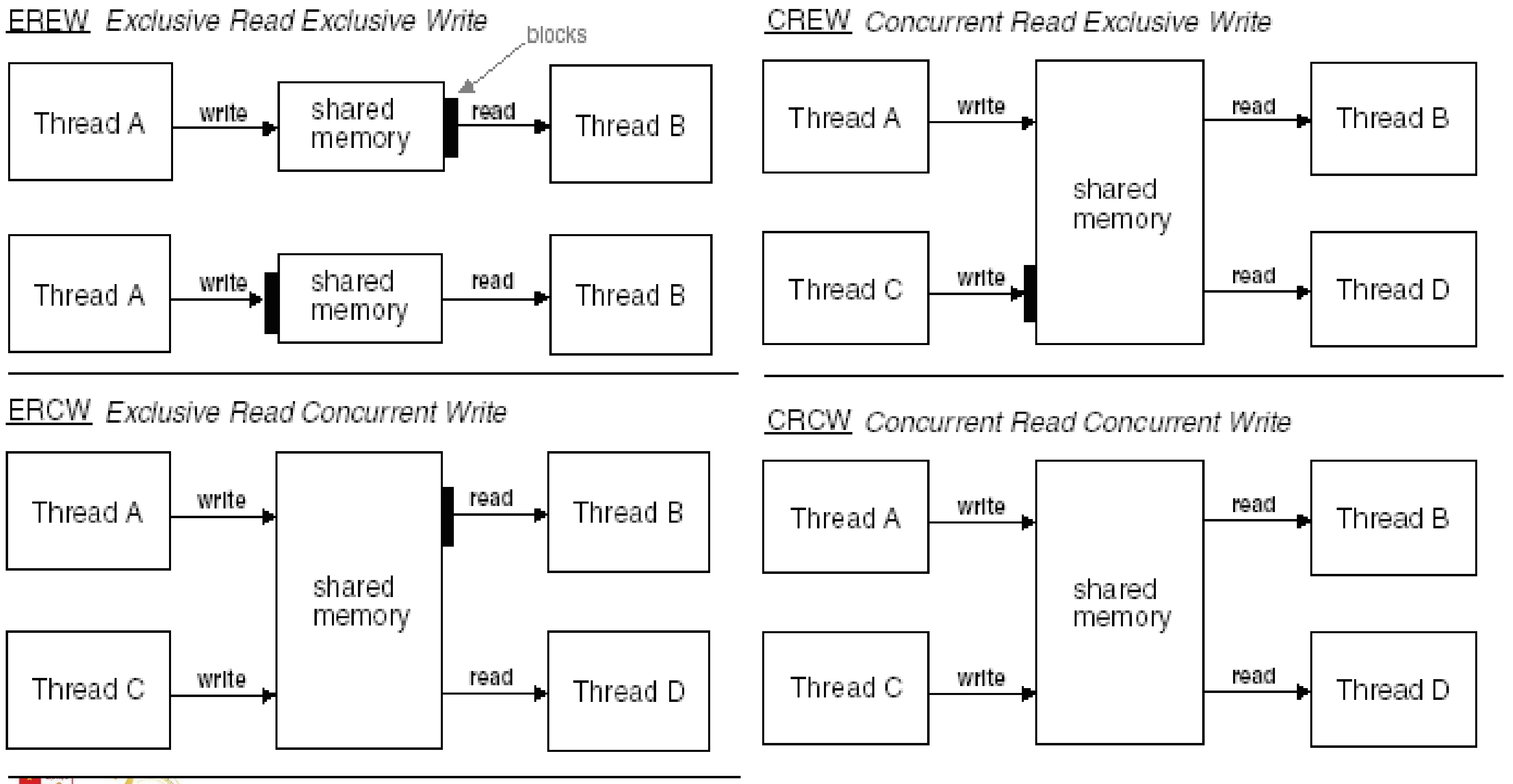
- Exclusive-Read Exclusive-Write
- Exclusive-Read Concurrent-Write
- Concurrent-Read Exclusive-Write
- Concurrent-Read Concurrent-Write

- If concurrent write is allowed we must decide which “written value” to accept

The PRAM Models

- PRAM models vary according
  - How they handle write conflicts
  - The models differ in how fast they can solve various problems.
- Exclusive Read, Exclusive Write (EREW)
  - Only one processor is allow to read or write to the same memory cell during any one step
- Concurrent Read Exclusive Write (CREW)
- Concurrent Read Concurrent Write (CRCW)
  - An algorithm that works correctly for EREW will also work correctly for CREW and CRCW, but not vice versa

Types of PRAM

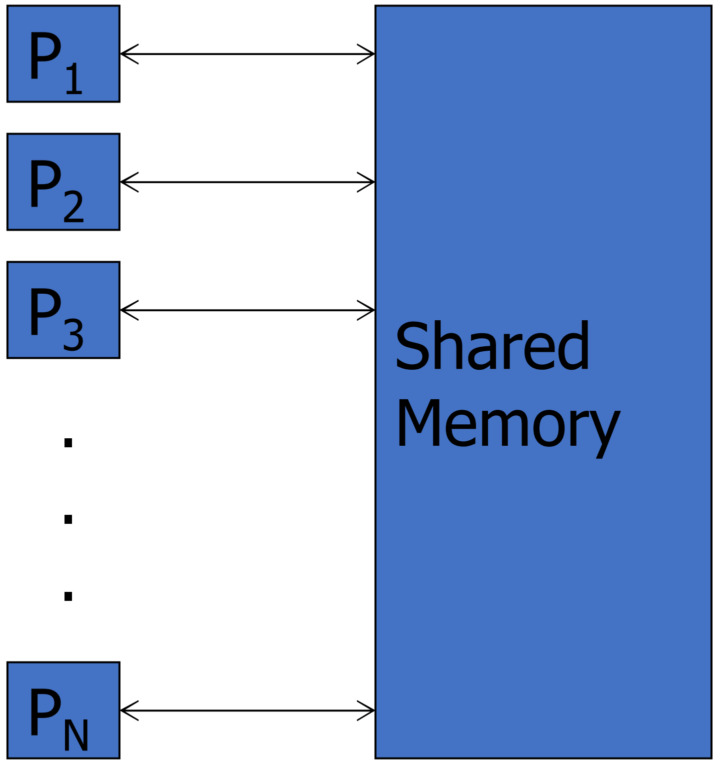


Assumptions

- There is no upper bound on the number of processors in the PRAM model.
- Any memory location is uniformly accessible from any processor.
- There is no limit on the amount of shared memory in the system.
- Resource contention is absent.
- The algorithms designed for the PRAM model can be implemented on real parallel machines, but will incur communication cost.
- Since communication and resource costs varies on real machines, PRAM algorithms can be used to establish a lower bound on running time for problems.

More Details on the PRAM Model

- Both the memory size and the number of processors are unbounded
- No direct communication between processors
  - they communicate via the memory
- Every processor accesses any memory location in 1 cycle
- Typically all processors execute the same algorithm in a synchronous fashion although each processor can run a different program.
  - READ phase
  - COMPUTE phase
  - WRITE phase
- Some subset of the processors can stay idle (e.g., even numbered processors may not work, while odd processors do, and conversely)



PRAM CW?

- What ends up being stored when multiple writes occur?
  - **Priority Conflict Resolution (PCR) CW: processors are assigned priorities and the top priority processor is the one that does writing for each group write**
  - **Equality Conflict Resolution (ECR) CW: if values are equal, write the value**
  - **Arbitary Conflic Resolution (ACR) CW: if values not equal, write a random coming value**
  - Fail common CW: if values are not equal, no write occurs
  - Collision common CW: if values not equal, write a "failure value"
  - Fail-safe common CW: if values not equal, then algorithm aborts
  - Random CW: non-deterministic choice of which value is written
  - Combining CW: write the sum, average, max, min, etc. of the values
  - etc.
- For CRCW PRAM, one of the above type CWs is assumed. The CWs can be ordered so that later type CWs can execute the earlier types of CWs.

2.2 Parallel Algorithm Complexity

Parallel algorithm evaluation

- The effectiveness of a parallel algorithm depends on 3 factors:
  - Execution time.
  - Number of processors involved.
  - Supercomputer architecture.
- The processor number and supercomputer architecture can be specified.
- The execution time depends on the working algorithm.

Complexity of Parallel Algorithm

- The parallel algorithm's computation time is the time that is calculated from the start of a processor doing its work until all work at the same time.
- Calculation time through the number of math operations to be performed during the calculation (consider that all operations are completed in the same time unit)

Complexity of Parallel Algorithm (2)

- To evaluate the proximity of complexity, we use the following asymptotic comparison symbols:
  - T(n) = O(f (n)) if we found positive numbers c and m such that T (n) <c * f (n) for all values n> m.
  - T(n) = Ω(f (n)) if we found positive numbers c and m such that T (n)> c * f (n) for all values n> m.
  - T(n) = Θ(f (n)) if we found positive numbers c1, c2 and m such that c1 * f (n) <T (n) <c2 * f (n) for all n> m.
- Usually we are interested in the upper bound O(n).

Pseudo Code

Syntax

```
FOR index  =  1 TO N DO IN PARALLEL
      ….. Paralle Task …
END PARALLEL
```

```
FOR index of S DO IN PARALLEL
            … Parallel Task …
END PARALLEL.
```

- Parallel Loop:
  - As the serial loop, add the keyword ‘In Parallel’
  - CPU *i* runs the code with corresponding value of index *i*
  - N CPU runs the code in parallel

Examples

Sum of 2 vectors

```
INPUT     : 2 array of A[1..n], B[1..n] in shared memory.
OUTPUT : array C[1..n]  =   A[1..n] + B[1..n] in shared memory.
BEGIN
    FOR i  =  1 TO n DO IN PARALLEL
        X     =     A[i];
        Y     =     B[i];
        C[i] =     X + Y;
    END PARALLEL.
END;
```

- The i_th processor read values of A[i] and B[i] from shared memory and write to local variables X,Y.
- C[i] is assigned by sum of X,Y.
- Complexity: O(1).
- PRAM EREW.

Prob. BOOLEAN - AND

- Problem:

```
INPUT        :    A[1…n] OF BOOLEAN.
OUTPUT       :    RESULT = A[1] and A[2] and … and A[n].
```

- Serial Algorithm:

```
BEGIN
    RESULT=    TRUE;
    FOR i = 1 TO n DO
        RESULT=    RESULT and A[i];
    END FOR;
END.
```

- Complexity: O(n) with 1 Processor

Prob. BOOLEAN - AND

- Analysis:
  - If all A[i] = TRUE → Result : TRUE.
  - If ∃ A[i] = FALSE → Result : FALSE.
- Algorithm for PRAM ERCW :
  - Using ECR for CW?
  - Using PCR, ACR, ECR for CW?
- Algorithm for PRAM EREW?

Prob. BOOLEAN - AND

Using ECR

Complexity: O(1)

```
BEGIN
    RESULT=   FALSE;
    FOR i = 1 TO n DO IN PARALLEL
        X         =   A[i];
        RESULT=   X;
    END FOR;
END
```

Using ECR, ACR, PCR

Complexity: O(1)

```
BEGIN
    RESULT=   TRUE;
    FOR i =  1 TO n DO IN PARALLEL
        IF A[i] = FALSE THEN
            RESULT = FALSE;
        END IF;
    END FOR;
END.
```

2.3 Basic Parallel Algorithms

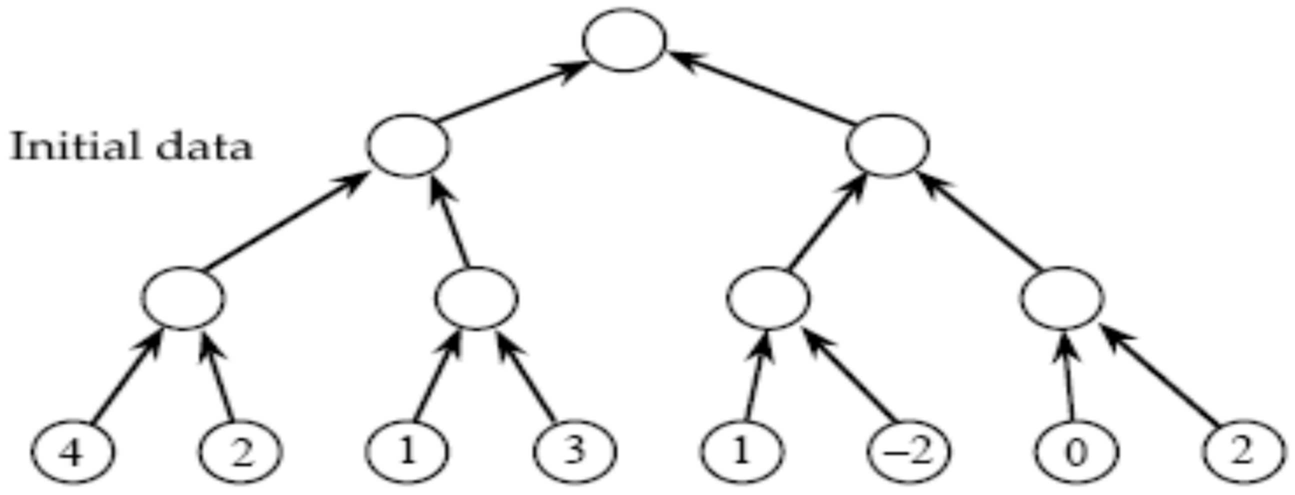
2.3.1 Binary Tree Paradigm

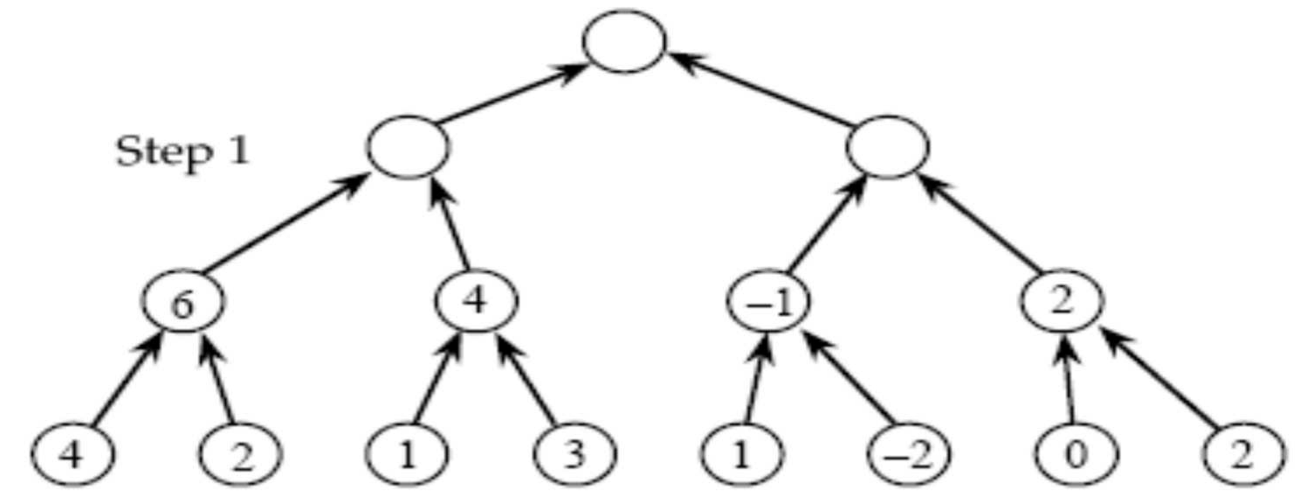
- Also known as the balanced tree model.
- Characteristics:
  - Node representing an action (math operation)
  - Nodes on the same level executed in parallel.
  - The nodes' input is the result of lower level operations and taken from shared memory.
  - After execution, each node writes the results to the shared memory.

Example

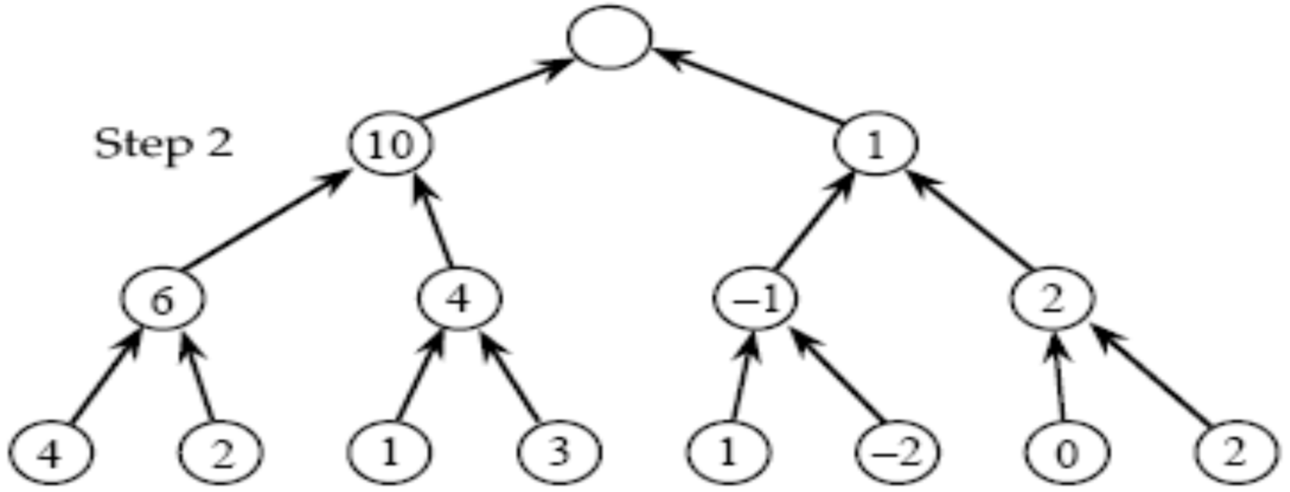
- Sum of n-number problem. (n = 2^k).
- If performed in serial algorithm: O(n) with 1 processor
- Parallelization with multiple processors?
  - Independent data → parallel data → use processor indicators to partition data.
    - The plus operation executed with only 2 numbers → need up to n/2 processors for parallel execution of n/2 pairs.
    - Where is the result written to in order to perform the repetition as above?

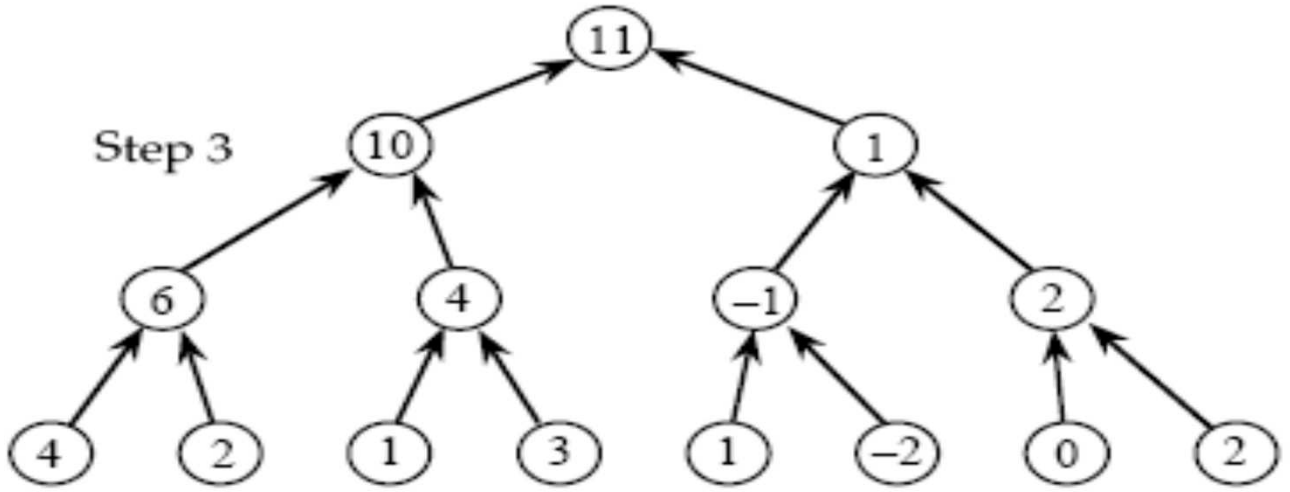
Example: Sum of 8 numbers





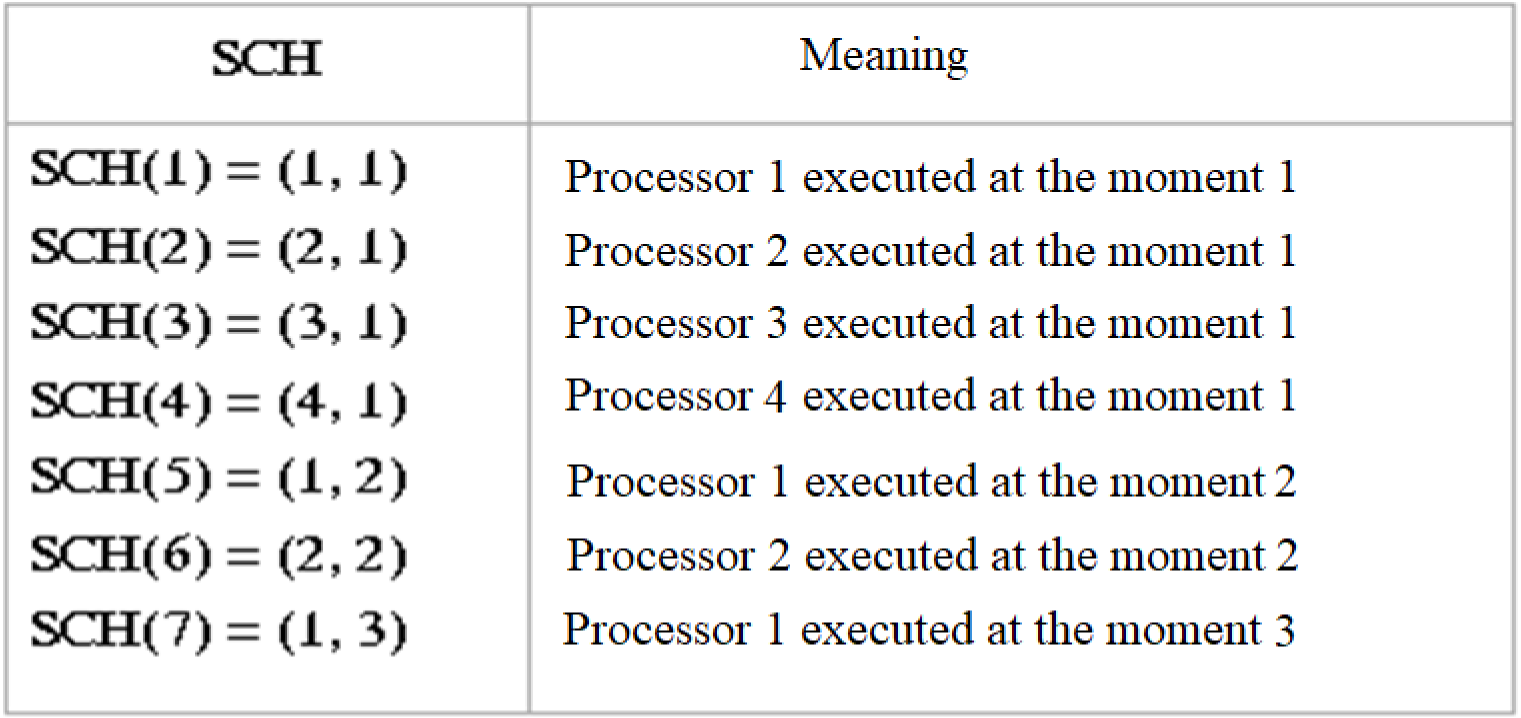
Example: Sum of 8 numbers





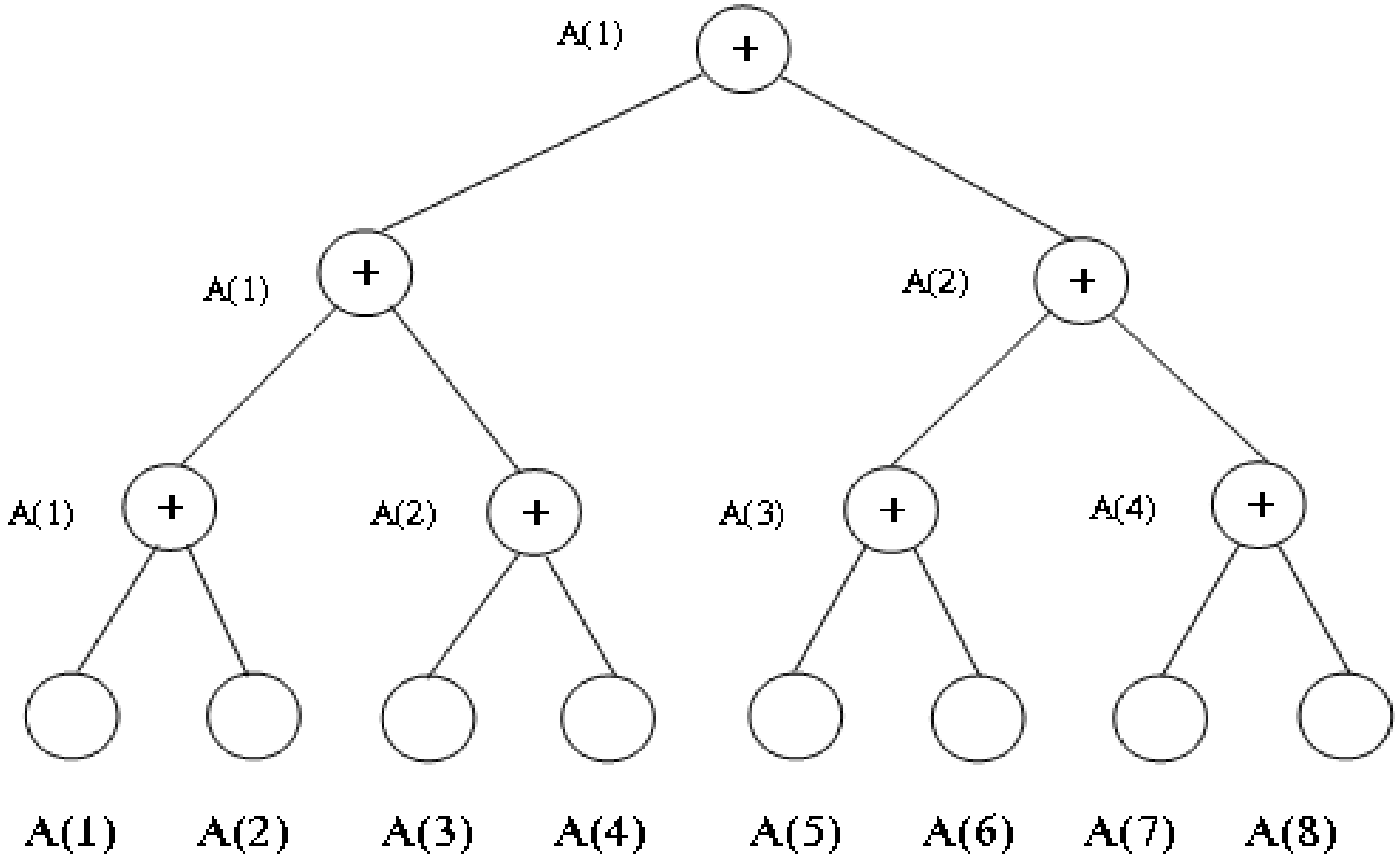
Sum of n-number problem

- Idea: A[i] = A[2*i-1] + A[2*i] (i>0)
- Maximum number of steps : log(n)



SCH	Meaning
SCH(1) = (1, 1)	Processor 1 executed at the moment 1
SCH(2) = (2, 1)	Processor 2 executed at the moment 1
SCH(3) = (3, 1)	Processor 3 executed at the moment 1
SCH(4) = (4, 1)	Processor 4 executed at the moment 1
SCH(5) = (1, 2)	Processor 1 executed at the moment 2
SCH(6) = (2, 2)	Processor 2 executed at the moment 2
SCH(7) = (1, 3)	Processor 1 executed at the moment 3

Sum of n-numbers



Sum of n-number problem

```
INPUT       :   A[1…n];
OUTPUT      :   SUM = ∑ A[i];
BEGIN
    p    =    n/2;
    WHILE p > 0 DO
        FOR i = 1 TO p DO IN PARALLEL
            A[i]  =  A[2i-1] + A[2i];
        END PARALLEL;
        p    =    p/2;
    END WHILE;
END.
```

Performance evaluation:

- Complexity: O(log(n)) with O(n) processor.
- Machine used: PRAM EREW.

More examples

- Problem Boolean-AND:
  - Replace + operation with AND operation
  - Done on ERCW : O(1).
  - Done on EREW : O(log(n)).
- Scalar product of 2 vectors problem:
  - There are 2 parallel steps:
    - Multiply parallel each pair with n processors.
    - Sum the obtained results according to the balanced tree model.

Scalar product of 2 vectors problem

- Algorithm:

```
INPUT       :   A[1..n], B[1..n];
OUTPUT      :   RESULT=   ∑(A[i]*B[i]);
BEGIN
    FOR i = 1 TO n DO IN PARALLEL
        C[i] =   A[i] * B[i];
    END PARALLEL;
    FOR i = 1 TO log(n) DO
        FOR j = 1 TO n/2^i DO IN PARALLEL
            C[j] =   C[j] + C[j +  n/2^i];
        END PARALLEL;
    END FOR ;
END;
```

- Performance evaluation:
  - Complexity: O(logn) with n processor
  - Machine PRAM EREW.

2.3.2 Growing by Doubling

- As opposed to the previous technique:
  - Balancing tree: processor number decreased by 1/2 after each step.
  - Double development: number of processors increased by 2 after each step.
- Common formula for both algorithms:
  - Specify the number of repetitive steps (logn).
  - Determine the number of processors and specific indicators on each iteration step.
  - Define work-by-processor in each serial step

Example

- Problem “Broadcast" in PRAM.
- Problem as follows:
  - Machine PRAM EREW with n processor.
  - P1 contains value x in its private memory.
  - Write an algorithm that copy value x to the rest of the processors

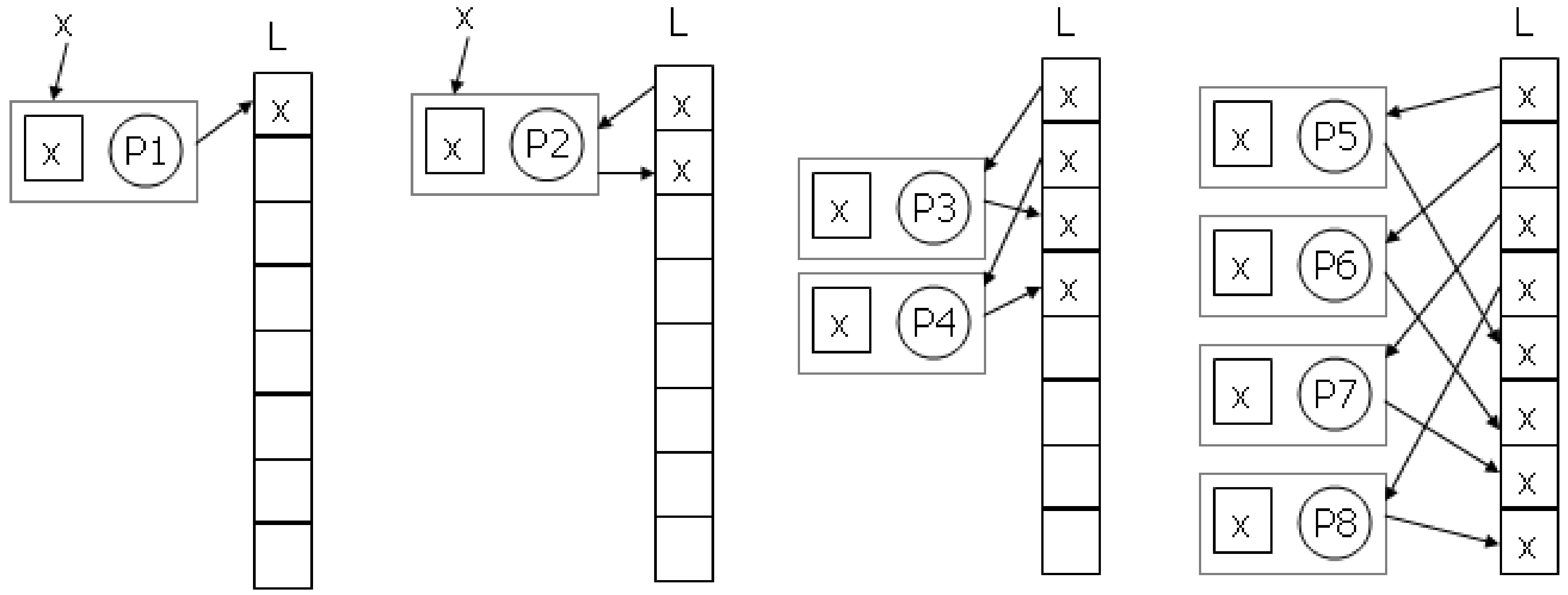
“Broadcast” in PRAM

- The concept of information transmission in PRAM:
  - Step1: processor A records value x to shared memory cell M
  - Step2: processor B reads x from memory cell M
- The simplest algorithm:
  - P1 processor that records value x in memory cell M.
  - PRAM EREW → at a specific moment only 1 processor is reading data from 1 memory cell.
  - → the processors read data in turn → O(n).

"Broadcast" in PRAM

- Parallel algorithm ideas:
  - B1: P1 writes value x to the memory cell m1.
  - B2: divided into 2 phases:
    - P2 reads data from m1 → P2 also has x.
    - P2 records x in memory cell m2.
  - B3: divided into 2 phases:
    - P3, P4 reads data from : m1, m2.
    - P3,P4 writes data to : m3, m4.
  - B4: divided into 2 phases:
    - P5,..P8 reads data from: m1,..m4.
    - P5,..P8 writes data to: m5,..m8.
  - After each step the number of participating processors doubled.

"Broadcast" in PRAM



Khởi tạo Bước 1 Bước 2 Bước 3

"Broadcast" in PRAM

- Parallel algorithm:

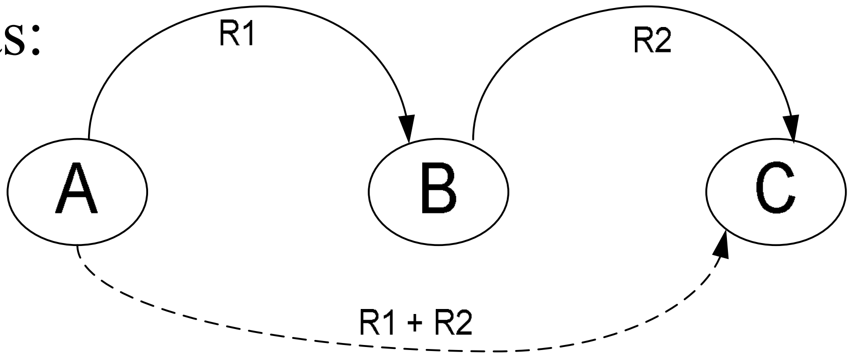
```
INPUT       :    P1 enabled.
OUTPUT      :    P1.P2,.. Pn contains an x value.
BEGIN
    P1: y     =    x;
        L[1] =    y;
    FOR  k = 0 TO log(n) -1 DO
        FOR i = $2^k$ + 1 TO $2^{k+1}$ DO IN PARALLEL
            Pi:  y     =    L[i – $2^k$];
                 L[i]  =    y;
        END PARALLEL;
    END FOR;
END.
```

- Performance evaluation:
  - Complexity: O(logn) with O(n) processor.
  - Machine PRAM EREW.

2.3.3 Pointer Jumping

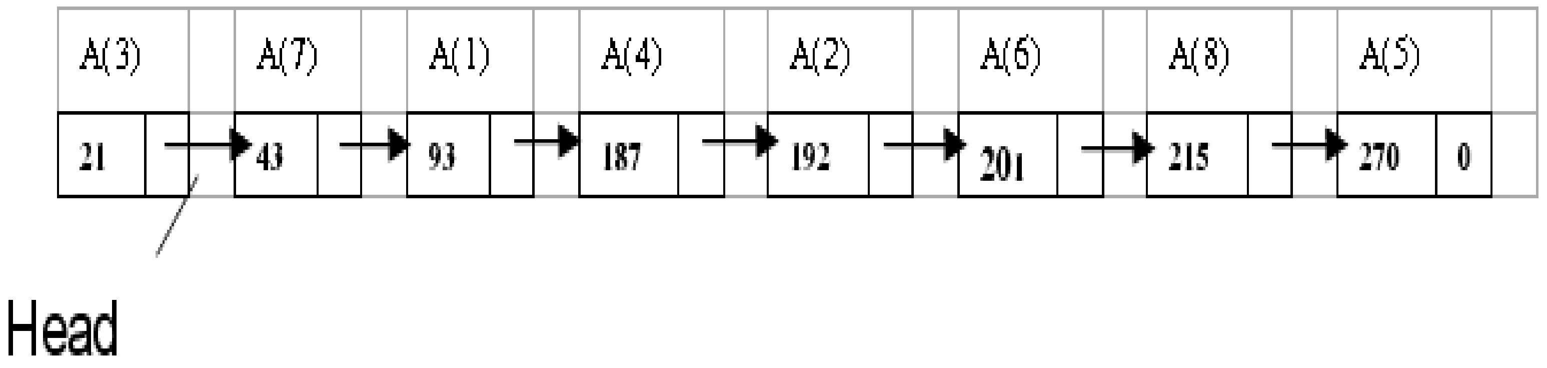
- Used in many applications with dynamic structures (lists, trees,,..).
- Method ideas:
  - Consider 3 nodes in 1 list: A → B → C.
  - Call R1, R2 is the 'job' function from A → B and from B → C.
  - Then A → C explained as:

    R3 = R1 + R2



Example

- Let's assume that the input is a list of linked elements in some order. Ranking of elements in arrays should be calculated.



- Show data as arrays:
    - Values stored at the node.
    - Index of the next node that the node points to.

The problem of determining the rank

- Consider for example with 8 elements as drawing in the previous.
- We call LINK[i] as the index of the next node of A[i]. For example, LINK[3] = 7 means that in the list of links, A[7] is behind A[3].
- LINK[i]= 0 means that A[i] is the last element of the list of links.
- HEAD variable that contains the index of the first element in the list.
- We call the rank of an array's element in the list is the distance from it to the end.

The problem of determining the rank

HEAD = 3		
i	A	LINK
1	93	4
2	192	6
3	21	7
4	187	2
5	270	0
6	201	8
7	43	1
8	215	5

Serial Algorithm

```
INPUT       :   A[1..n], LINK[1..n], HEAD.
OUTPUT      :   RANK[1..n].
BEGIN
    p    =   HEAD;
    r    =   n;
    RANK[p]     =   r;
    REPEAT
        p    =   LINK(p);
        r    =   r – 1;
        RANK[p]     =   r;
    UNTIL LINK(p) = 0;
END.
```

- Performance evaluation:
  - O(n) complexity with 1 processor

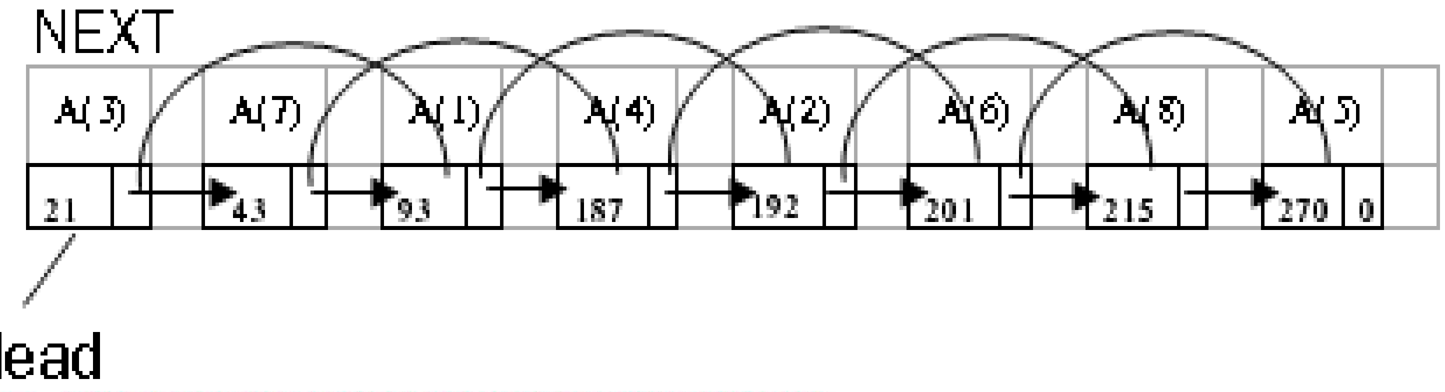
Parallel idea

- We originally set NEXT[i]= LINK[i]; that is, each point sees only its upcoming jumping destination.
- The rank of the nodes is determined by the distance it can jump to.
- In the next steps we take: NEXT[i] = NEXT[NEXT[i]];
- Update ranking values of elements.
- At the same time with it the jumping distance will be doubled. After log(n) times, NEXT[i] will reach the end of the list.
- When all NEXT[i] reach to the last element, the algorithm ends.

Parallel idea

i	3	7	1	4	2	6	8	5
LINK	7	1	4	2	6	8	5	0
NEXT	7	1	4	2	6	8	5	0
RANK	1	1	1	1	1	1	1	1

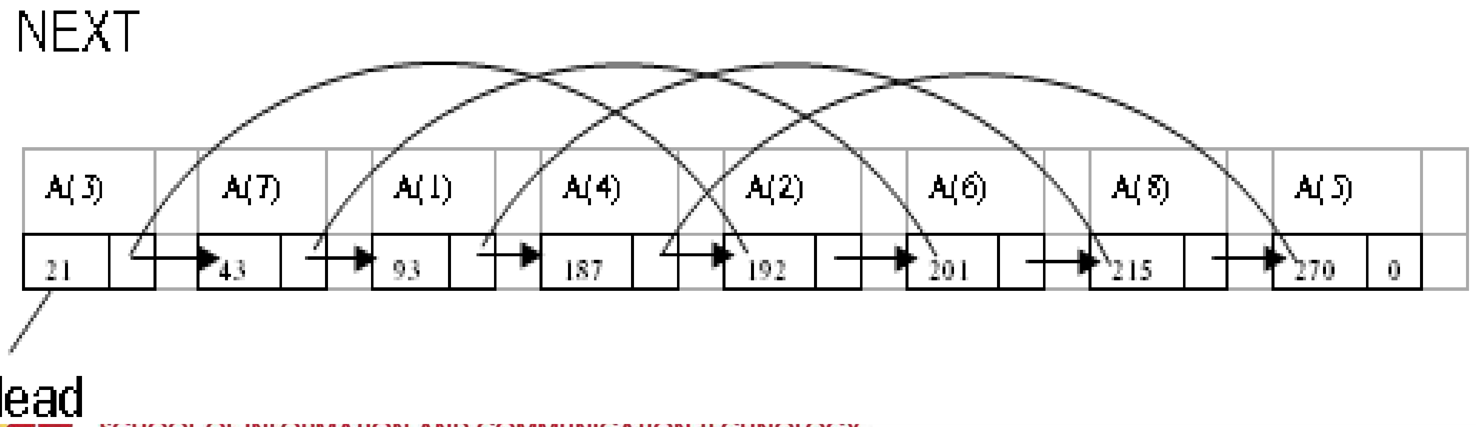
(*a*) Initialization



Parallel idea

i	3	7	1	4	2	6	8	5
LINK	7	1	4	2	6	8	5	0
NEXT	1	4	2	6	8	5	0	0
RANK	2	2	2	2	2	2	2	1

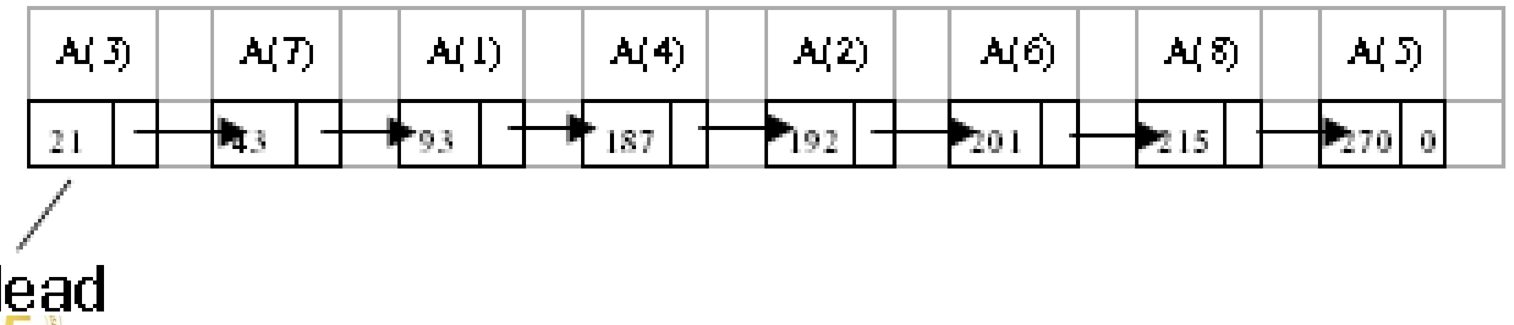
(*b*) Stage 1



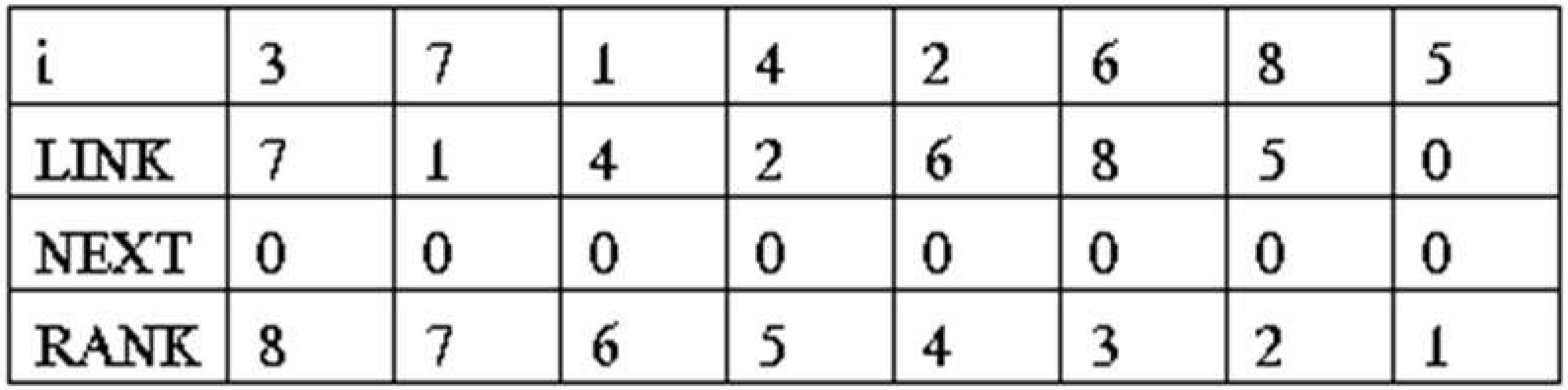
Parallel solved arts

i	3	7	1	4	2	6	8	5
LINK	7	1	4	2	6	8	5	0
NEXT	2	6	8	5	0	0	0	0
RANK	4	4	4	4	4	3	2	1

(c) Stage 2



Parallel idea



i	3	7	1	4	2	6	8	5
LINK	7	1	4	2	6	8	5	0
NEXT	0	0	0	0	0	0	0	0
RANK	8	7	6	5	4	3	2	1

(d) Stage 3

- Step 3 ends when all values Next[i] = 0
- RANK[i] defines the rank of elements in the linked list.

Parallel algorithm

```
INPUT       :    A[1..n], LINK[1..n], HEAD;
OUTPUT      :    RANK[1..n];
BEGIN
    FOR i = 1 TO n DO IN PARALLEL
        RANK[i]    =    1;
        NEXT[i] =    LINK[i];
    END PARALLEL;
    FOR k = 1 TO log(n) DO
        FOR i = 1 TO n DO IN PARALLEL
            IF NEXT[i] <> 0 THEN
                RANK[i] = RANK[i] + RANK[NEXT[i]];
                NEXT[i] = NEXT[NEXT[i]];
            END IF;
        END PARALLEL;
    END FOR;
END;
```

Parallel algorithm

- Performance Evaluation:
  - Algorithm divided into 2 serial parts
    - Part one: O(1) time unit.
    - Part two: O(logn) time unit.
  - Computer architecture:
    - Command: RANK[i]= RANK[i]+ RANK[NEXT[i]] can be split into 3 single sentences as follows:
      - B1: X = RANK[i];
      - B2: Y = RANK[NEXT[i]];
      - B3: RANK[i] = X+Y.
    - Because processors perform in parallel, no memory cells are read or written simultaneously → PRAM EREW architecture with O(n) processor.

2.3.4 Partitioning

- General ideas:
  - Divide a large problem into p small problems, where p is the number of processors allowed to execute simultaneously.
- Example: **Merge Sorting**
- Let A[1..n] and B[1..n] be the two sorted arrays. Performing merging these two arrays into an array C[1..2n] that will be also sorted.

Serial algorithm

- Idea:
  - 3 indicators i, j, k slide on 3 arrays A, B, C respectively.
  - Values of array C are determined by passing the smallest value of the 2 values A_i and B_j.
  - Moving these indexes appropriately to be able to pass through all elements of the 2 arrays A, B.

Serial algorithm

```
INPUT       :   A1 ≤ A2 ≤ …. ≤ An  và  B1 ≤ B2 ≤ … ≤ Bn .
OUTPUT  :   C[1..2n] = A[1..n] U B[1..n] : C1 ≤ C2 ≤ …≤ C2n
BEGIN
    A[n+1] =  ∞; B[n+1] = ∞ ;
    i = 1; j = 1; k = 1;
    WHILE k ≤ 2n DO
        IF A[i] < B[j] THEN
            C[k]=   A[i];
            i    =   i + 1;
        ELSE
            C[k]=   B[j] ;
            j    =   j + 1;
        END IF;
        k    =   k + 1;
    END WHILE;
END.
```

- Complexity: O(n).

Parallel algorithm

- Using division technique :
  - Divide array A into r = n / log(n);
  - Each group has k = log(n) elements. Let's assume k, r are integer numbers.
- So we have groups like this:
  - Group NA_1: A_1, A_2 ,……………………………..….A_k;
  - Group NA_2: A_{k+1}, A_{k+2},……………………………A_{2k};
  - …………………………………………….
  - Group NA_r: $A_{(r-1)k+1}$, $A_{(r-1)k+2}$, ………………..A_n;

Parallel algorithm

- Find r integers j[1], j[2],....j[r] so that:
  - j[1] is the largest index that satisfies $A_k \geq B_{j[1]}$;
  - j[2] is the largest index that satisfies $A_{2k} \geq B_{j[2]}$;
  - ………………
  - j[r] is the largest satisfying index $A_n \geq B_{j[r]}$;
- Divide array B[1..n] into r+1 groups:
  - Group NB_1: B_1, B_2 , ….. ………………………..$B_{j[1]}$;
  - Group NB_2: $B_{j[1]+1}$, $B_{j[1]+2}$,………………………$B_{j[2]}$;
  - ……………………………………….
  - Group NB_r: $B_{j[r-1]+1}$, $B_{j[r-1]+2}$……………………..$B_{j[r]}$;
  - Group NB_{r+1}: $B_{j[r]+1}$, ……………………………..B_n;

Parallel algorithm

- Now we realize that :
  - Elements in the NA_i group are no smaller than NB_{i-1} group's elements and no larger than NB_i group's elements
  - We separately mix the elements in NA_i and NB_i, the ordering order of the elements in this new set remains unchanged in array C.

A = (1, 5, 15, 18, 19, 21, 23, 24, 27, 29, 30, 31, 32, 37, 42, 49),

B = (2, 3, 4, 13, 15, 19, 20, 22, 28, 29, 38, 41, 42, 43, 48, 49).

A	1, 5, 15, 18	19, 21, 23, 24	27, 29, 30, 31	32, 37, 42, 49
Group	Group 1	Group 2	Group 3	Group 4

Group	Group 1	Group 2	Group 3	Group 4
A	1, 5, 15, 18	19, 21, 23, 24	27, 29, 30, 31	32, 37, 42, 49
B	2, 3, 4, 13, 15	19, 20, 22	28, 29	38, 41, 42, 43, 48, 49

Group 1 for C[1..9] = (1, 2, 3, 4, 5, 13, 15, 15, 18);
Group 2 for C[10..16] = (19, 19, 20, 21, 22, 23, 24);
Group 3 for C[17..22] = (27, 28, 29, 30, 31);
Group 4 for C[23..32]   = (32, 37, 38, 41, 42, 43, 48, 49, 49);

Parallel algorithm

```
INPUT      :   $A1 \le A2 \le .... \le An$  và  $B1 \le B2 \le ... \le Bn$ .
OUTPUT     :   $C[1..2n] = A[1..n] \cup B[1..n] : C1 \le C2 \le ... \le C2n$
BEGIN
    FOR i = 1 TO r DO IN PARALLEL
        Pi :
            READ($A_{(i-1)k+1}, \dots A_{ik}$);
            $j[i] = MAX\{ t : A_{ik} \ge B_t )$ : BINARY_SEARCH.
            S_MERGE($A_{(i-1)k+1}, \dots A_{ik}, B_{j[i-1]+1}, \dots B_{j[i]}$);
    END PARALLEL;
END.
```

- Function BINARY_SEARCH: binary search on sorted array.
- Function S_MERGE: mix 2 arrays in the order as the previous algorithm presented.

Complexity evaluation

- 2 child programs used:
  - Performing a binary search on array B consisting of n elements will cost O(log(n)) time units.
  - The work of merging NA and NB arrays depends on the size of the NB array because the number of NB elements is not pre-defined, while the number of NA elements is pre-defined k = log(n).
    - If the size of NB_i array is also k, this step can be performed with a time of O(log(n)).
    - If the size of the NB_i array is greater than k, we can recursively repeat the division operation a few times with NB. Then step 3 can also be done with O(log(n)) time units.

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions !

soict.hust.edu.vn/ fb.com/groups/soict