Parallel Algorithms

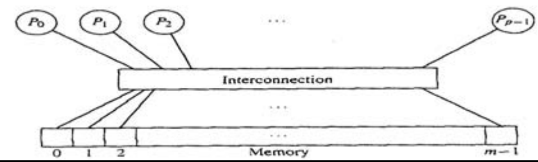
- ✈ **several operations can be executed at the same time**
- ✈ **many problems are most naturally modeled with parallelism**

- A Simple Model for Parallel Processing
- Approaches to the design of parallel algorithms
- Speedup and Efficiency of parallel algorithms
- A class of problems *NC*
- Parallel algorithms, e.g.

TECH
Computer Science

A Simple Model for Parallel Processing

- Parallel Random Access Machine (PRAM) model
  - ✈ **a number of processors all can access**
  - ✈ **a large share memory**
  - ✈ **all processors are synchronized**
  - ✈ **all processor running the same program**
    - ➢ each processor has an unique id, pid. and
    - ➢ may instruct to do different things depending on their pid



PRAM models

- PRAM models vary according
  - ✈ **how they handle write conflicts**
  - ✈ **The models differ in how fast they can solve various problems.**
- Concurrent Read Exclusive Write (CREW)
  - ✈ **only one processor are allow to write to**
  - ✈ **one particular memory cell at any one step**
- Concurrent Read Concurrent Write (CRCW)
- Algorithm works correctly for CREW
  - ✈ **will also works correctly for CRCW**
  - ✈ **but not vice versa**

Approaches to the design of parallel algorithms

- Modify an existing sequential algorithm
  - ✈ **exploiting those parts of the algorithm that are naturally parallelizable.**
- Design a completely new parallel algorithm that
  - ✈ **may have no natural sequential analog.**

- Brute force Methods for parallel processing:
  - ✈ **Using an existing sequential algorithm but**
    - ➢ each processor using differential initial conditions
  - ✈ **Using compiler to optimize sequential algorithm**
  - ✈ **Using advanced CPU to optimize code**

Speedup and Efficiency of parallel algorithms

- ✈ **Let T*(n) be the time complexity of a sequential algorithm to solve a problem P of input size n**
- ✈ **Let $T_p(n)$ be the time complexity of a parallel algorithm to solves P on a parallel computer with p processors**

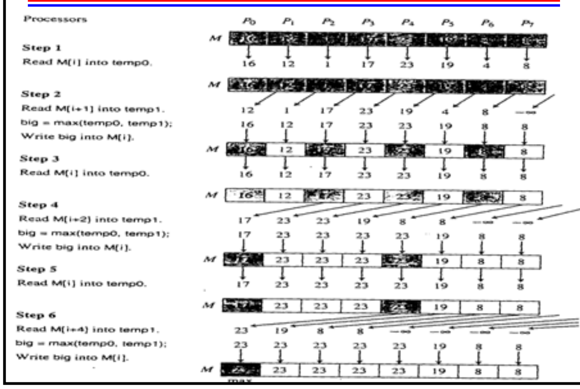
- Speedup
  - ✈ **$S_p(n) = T^*(n) / T_p(n)$**
  - ✈ **$S_p(n) <= p$**
  - ✈ **Best possible, $S_p(n) = p$**
    - ➢ when $T_p(n) = T^*(n)/p$
- Efficiency
  - ✈ **$E_p(n) = T_1(n) / (p\ T_p(n))$**
    - ➢ where $T_1(n)$ is when the parallel algorithm run in 1 processor
  - ✈ **Best possible, $E_p(n) = 1$**

A class of problems *NC*

- The class NC consists of problems that
  - ✈ **can be solved by parallel algorithm using**
    - ➢ polynomially bounded number of processors p(n)
    - ➢ $p(n) \in O(n^k)$ for problem size n and some constant k
  - ✈ **the number of time steps bounded by a polynomial in the *logarithm* of the problem size n**
    - ➢ $T(n) \in O((\log n)^m)$ for some constant m

- Theorem:
  - ✈ **$NC \subseteq P$**

Parallel algorithms, e.g. Binary Fan-In Technique



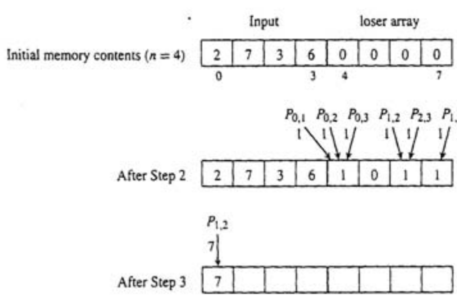
Algorithm: Parallel Tournament for Max

- **Algorithm:Parallel Tournament for Maximum**
- **Input**: Keys x[0],x[1],....x[n-1],
- initially in memory cells M[0] ,...,M[n-1], and integer n.
- **Output**:The largest key will be left in M[0].
- parTournamentMax(M,n)
- int incr
- Write -(some very small value) into M[n+pid]
- incr=1;
- while(incr<n)
- key big, temp0,temp1;
- Read M[pid] into temp0
- Read M[pid+incr] into temp1
- big=max(temp0,temp1);
- Write big into M[pid].
- incr=2*incr;

- **Analysis**: Use n processor and θ(log n) time

Algorithm: Finding Max in Constant Time

- CRCW method



Algorithm: Common-Write Max of n Keys

- Uses n^2 processors, does only three read/write steps!

fastMax(M, n)

1. Compute i and j from pid.
   if (i ≥ j) **return**;

   $P_{i,j}$ reads x_i (from M[i]).
2. $P_{i,j}$ reads x_j (from M[j]).
   $P_{i,j}$ compares x_i and x_j.
   Let k be the index of the smaller key (i if tied).
   $P_{i,j}$ writes 1 in loser[k].
   // At this point, every key other than the largest
   // has lost a comparison.
3. $P_{i,i+1}$ reads loser[i] (and $P_{0,n-1}$ reads loser[n−1]).
   The processor that read a 0 writes x_i in M[0]. ($P_{0,n-1}$ would write x_{n-1}.)
   // This processor already has the needed x in its local memory
   // from steps 1 and 2.