



# CS575 Parallel Processing

---

## Lecture five: Performance

### Wim Bohm, Colorado State University

Material on Naïve model from:

“Speedup vs Efficiency in Parallel Systems” - Eager, Zahorjan and Lazowska  
IEEE Transactions on Computers, Vol 38 No 3, March 1989

CSU has institution license for IEEE online library

Goto <http://ieeexplore.ieee.org> using browser opened from CSU domain machine or with CSU VPN client to download

Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 license.



# Parallel Processing

---

- Divide a computation into sub tasks
- Execute sub tasks in parallel
  - Can all sub tasks run in parallel?
  - NO! Usually there is data dependence between the sub tasks
- Benefit?
  - Speedup
- Cost?
  - More resources
    - Processors, memories, network



# Notation

---

- $T_i$ : time to execute a program with  $i$  processors
- $T_\infty$ : time to execute a program with unbounded # processors
- Speedup:  $S(n) = T_1 / T_n$ 
  - Linear speedup:  $S(n) = k \cdot n$
  - Often used in the stricter sense:  $S(n) = n$
- Efficiency:  $E(n) = S(n) / n$ 
  - Average utilization of  $n$  processors
  - Range?
  - What does  $E(n) = 1$  signify?
- Does  $E(n) = 1$  happen a lot in practice?
  - Data dependence, communication, contention



# Bounds on Speedup

---

- Achievable bounds
- Amdahl's law
  - If fraction  $f$  of a program is inherently sequential, the bound on  $S(n)$ :
$$T_1 = 1$$
$$T_n = f + (1-f)/n$$
$$S(n) \leq 1 / (f + (1-f)/n)$$
  - Even simpler:  $S(n) < 1 / f$
  - What does this assume, and thus totally ignore?



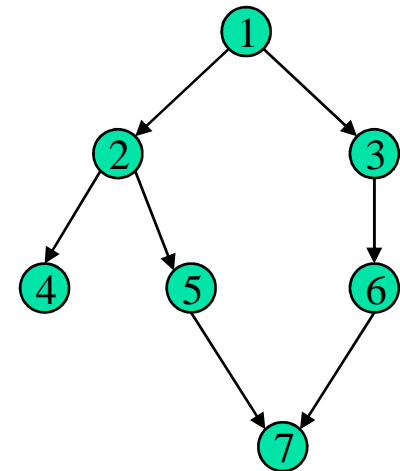
# Zahorjan et. al. slightly less naïve

---

- **Program:** Acyclic Directed graph
  - Nodes  $\equiv$  tasks, Edges  $\equiv$  precedence relations
  - Strict  $A \rightarrow B$ : B cannot begin until A is finished
  - Fixed set of tasks, no deadlock
- **Machine:** n identical processors
  - Execute each task in one time-step
  - No communication overhead
- **Scheduling:** Work conserving
  - Leaves no task idle when processor available

# Program Parallelism

- How many steps does the program take?
  - finite resources
    - sequential (1PE) = 7 steps
    - 2PEs, 3, 4, .....?
  - unbounded resources?
- Does scheduling affect this?
  - Example schedules using 2PEs
    - P0: 1 2 4 6 7
    - P1: 3 5
    - P0: 1 2 5 7
    - P1: 3 6 4



Program graph



# Average Parallelism

---

- Graph exposes all parallelism in program
  - But too detailed, need abstraction:  $\pi$
- Average parallelism  $\pi$  :
  1. Average number of processors busy given unbounded # of available processors
  2. Speedup given unbounded # of processors =  $T_1 / T_\infty$
  3.  $\pi$  = Total service / critical path length
    - Total service = number of tasks
    - Critical path length = length of longest path in graph



## Average Parallelism vs speedup

---

- $\pi = T_1 / T_\infty$   
When there is no resource constraint  
average parallelism = average speedup
- $S(n) = T_1 / T_n$   
total service / execution time with n processors =  
average number of processors busy.
- If a task can get executed immediately when enabled (unbounded resources), then the parallel execution time = the longest path through the graph.



# Parallelism vs Speedup - Questions

---

- How does (average) parallelism affect Speedup in the finite resource case?
- Is there a lower bound on efficiency and speedup given a certain average parallelism ?
  - i.e., How bad can a scheduling policy be ?
- To achieve a certain speedup, we may introduce more processors, but then: what efficiency penalty is paid ?



# Limits on Speedup

---

- Hardware limit:  $n$ 
  - Can only be achieved if all processors busy all the time
- Software limit:  $\pi$ 
  - Can be achieved when #processors = max //ism in graph
  - Adding more processors gives only more idle time



## Lower Bounds on $S(n)$ and $E(n)$

---

- Theorem:

For any program graph, for any work conserving scheduling policy:

$$S(n) \geq n \cdot \pi / (\pi + n - 1) \text{ and}$$

$$E(n) \geq \pi / (\pi + n - 1)$$

- What does the following mean?

- When  $n = \pi$ ,

$$S(n) \geq \pi^2 / (2\pi - 1) > \pi/2 \quad \text{and} \quad E(n) \geq \pi / (2\pi - 1) > 1/2$$



## Proof of Theorem

---

$$\pi = T_1 / T_\infty \quad \text{or} \quad T_1 = \pi \cdot T_\infty$$

For  $n$  processors

$$\text{total **busy time**} = T_1 = \pi \cdot T_\infty$$

$$\text{Let total **idle time**} = I(n)$$

Execution time

$$T_n = (T_\infty \cdot \pi + I(n)) / n$$

Speedup

$$S(n) = T_1 / T_n = n \cdot \pi / (\pi + I(n) / T_\infty)$$

So we need to prove

$$I(n) / T_\infty \leq n - 1$$



## Proof of $I(n) \leq T_{\infty} \cdot (n-1)$

---

At time step  $t$ ,

$W(t)$ : portion of the graph not executed yet

$L(t)$  : length of critical path: by definition:  $L(t) \leq T_{\infty}$

$L(t)$  is either decreasing or NOT

$L(t)$  NOT decreasing:

task at head of critical path is not executing

but that task is enabled, hence (work conserving scheduling) all processors BUSY

$L(t)$  decreasing:

Processors **can be** idle (at most  $T_{\infty}$  time steps)

Max #processors idle =  $n-1$

$\therefore I(n) \leq T_{\infty} \cdot (n-1)$                       QED



## Corollaries

---

For any work conserving scheduling policy:

$$\text{Cor 1: } E(n) + S(n)/\pi > 1$$

efficiency plus attained fraction of speedup  $> 1$

$$\text{Cor 2: } E(n) > (\pi - S(n)) / \pi$$

In any program with, e.g.,  $\pi = 50$ , a speedup of  
2 can be achieved with 96% efficiency  
10 can be achieved with 80% efficiency



## Main conclusions

---

- Given a certain average parallelism, there are lower bounds on Speedup and Efficiency
- Small Speedup can be achieved with high Efficiency
- All this assuming work conserving scheduling ignoring

**Scheduling, Communication and Latency**



# Back to book: Cost and Optimality

---

- **Cost** =  $p \cdot T_p$ 
  - $p$ : number of processors
  - $T_p$ : Time complexity for parallel execution
  - Also referred to as **processor-time product**
  - Time can take communication into account
    - Problem with mixing processing time and communication time
    - Example: add: 1 time unit  
communicate with direct neighbor: 1 time unit
- **Cost optimal**
  - if Cost =  $O(T_1)$



## E.g. - Add $n$ numbers on hypercube

---

- $n$  numbers on  $n$  processor cube
  - Cost =  $O(n \cdot \log(n))$ , not cost optimal
- $n$  numbers on  $p$  ( $<n$ ) processor cube
  - $T_p = n/p + 2 \cdot \log(p)$
  - Cost =  $O(n + p \cdot \log(p))$ , cost optimal if  $n = O(p \cdot \log(p))$
  - $S = n \cdot p / (n + 2 \cdot p \cdot \log(p))$
  - $E = n / (n + 2 \cdot p \cdot \log(p))$
  - Build a table:  $E$  as function of  $n$  and  $p$ 
    - Rows:  $n = 64, 192, 512$       Cols:  $p = 1, 4, 8, 16$
    - larger  $n \rightarrow$  higher  $E$ ,      larger  $p \rightarrow$  lower  $E$



# Scalability

---

- Ability to keep the efficiency fixed, when  $p$  is increasing, provided we also increase  $n$
- e.g. Add  $n$  numbers on  $p$  processors (cont.)
  - Look at the  $(n,p)$  efficiency table
  - Efficiency is fixed (at 80%) with  $p$  increasing
    - only if  $n$  is increased as  $8 \cdot p \cdot \log p$



# Iso- efficiency Terminology

---

- Input size  $n$ 
  - Characterizes size of input e.g.
    - $n$  numbers to sort,  $2$   $n \times n$  matrices to multiply,  $n$  bit number to factorize
- Workload  $W$ 
  - Characterizes **sequential** time complexity in  $n$ , e.g.
    - sorting:  $n \cdot \log(n)$ , matrix multiply:  $n^3$ , factorization:  $2^n$
- Overhead  $T_o$  (was  $I(n)$  in Zahorjan et.al.s terminology)
  - Operations (or busy waiting) performed by parallel algorithm AND NOT BY THE SEQUENTIAL ALGORITHM
  - $T_o = \text{Parallel complexity} - \text{workload} = p \cdot T_p - W$
  - e.g. add  $n$  numbers of  $p$  processors cube:  $T_o = 2 \cdot p \cdot \log p$



# Iso-efficiency metric

---

- Iso-efficiency of a scalable system
  - measures degree of scalability of parallel system
  - **The workload  $W$ , in terms of number of processors  $p$ , to keep efficiency fixed**
  - e.g.  $W$  grows as exponential function of  $p$  to keep  $E$  fixed, non-scalable
  - e.g.  $W$  grows linearly wrt  $p$  to keep  $E$  fixed, scalable



## Overhead $T_o$ vs. Workload $W$

---

- $T_o = p \cdot T_p - W$

$$T_p = (T_o + W)/p$$

$$S = T_1/T_p = W / T_p = W \cdot p / (T_o + W)$$

$$E = S/p = W / (W + T_o) = 1/(1 + T_o/W)$$

rewrite to get

$$T_o = (1-E)/E \cdot W = K \cdot W$$

(Keeping  $E$  fixed implies  $(1-E)/E$  is some constant  $K$ )

- **Conclusion:**

To achieve scalability, overhead must not have a larger order of magnitude complexity than workload.



# Sources of Overhead

---

- Communication
  - PE - PE
  - PE – memory
  - And the busy waiting associated with this
- Load imbalance
  - (Barrier) synchronization causes **idle processors**
  - Program parallelism does not match machine parallelism all the time
    - Extreme: sequential components in computation
- Extra computation
  - Easily parallelizable but poor sequential algorithm
  - consider the difference in  $W$  between good and poor sequential algorithm in overhead