Parallel Program Issues

- Dependency Analysis:
  - Types of dependency
  - Dependency Graphs
  - Bernstein's Conditions of Parallelism
- Asymptotic Notations for Algorithm Complexity Analysis
- Parallel Random-Access Machine (PRAM)
  - Example: sum algorithm on P processor PRAM
- Network Model of Message-Passing Multicomputers
  - Example: Asynchronous Matrix Vector Product on a Ring
- Levels of Parallelism in Program Execution
- Hardware Vs. Software Parallelism
- Parallel Task Grain Size
- Software Parallelism Types: Data Vs. Functional Parallelism
- Example Motivating Problem With high levels of concurrency
- Limited Parallel Program Concurrency: Amdahl's Law
- Parallel Performance Metrics: Degree of Parallelism (DOP)
  - Concurrency Profile
- Steps in Creating a Parallel Program:
  - Decomposition, Assignment, Orchestration, Mapping
  - Program Partitioning Example (handout)
  - Static Multiprocessor Scheduling Example (handout)

(PCA Chapter 2.1, 2.2)

Parallel Programs: Definitions

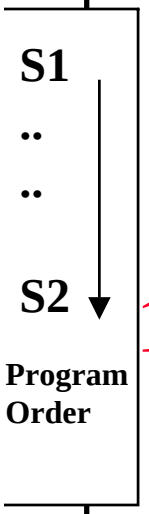
- **A parallel program is comprised of a number of tasks running as threads (or processes) on a number of processing elements that cooperate/communicate as part of a single parallel computation.**
- ***Task*:**
  - **Arbitrary piece of undecomposed work in parallel computation**
  - **Executed sequentially on a single processor; concurrency in parallel computation is only across tasks.**
- ***Parallel or Independent Tasks:***
  - **Tasks that with no dependencies among them and thus can run in parallel on different processing elements.**
- ***Parallel Task Grain Size:*** **The amount of computations in a task.**
- ***Process (thread)*:**
  - **Abstract entity that performs the computations assigned to a task**
  - **Processes communicate and synchronize to perform their tasks**
- ***Processor or (Processing Element)*:**
  - **Physical computing engine on which a process executes sequentially**
  - **Processes virtualize machine to programmer**
    - **First write program in terms of processes, then map to processors**
- **Communication to Computation Ratio (C-to-C Ratio):** **Represents the amount of resulting communication between tasks of a parallel program**

In general, for a parallel computation, a lower C-to-C ratio is desirable and usually indicates better parallel performance

Dependency Analysis & Conditions of Parallelism

- **Dependency analysis is concerned with detecting the presence and type of dependency between tasks that prevent tasks from being independent and from running in parallel on different processors and can be applied to tasks of any grain size.**
- **Dependencies between tasks can be algorithm/program related or hardware resource related.**
- **Algorithm/program Task Dependencies:**
  - **Data Dependence:**
    - **True Data or Flow Dependence**
    - **Name Dependence:**
      - **Antidependence**
      - **Output (or write) dependence**
  - **Control Dependence**
- **Hardware/Architecture Resource Dependence**

Conditions of Parallelism: Data & Name Dependence



Assume task S2 follows task S1 in sequential program order

1 **True Data or Flow Dependence:** Task S2 is data dependent on task S1 if an execution path exists from S1 to S2 and if at least one output variable of S1 feeds in as an input operand used by S2

Represented by S1 $\longrightarrow$ S2 in dependency graphs

Name Dependencies

2 **Antidependence:** Task S2 is antidependent on S1, if S2 follows S1 in program order and if the output of S2 overlaps the input of S1

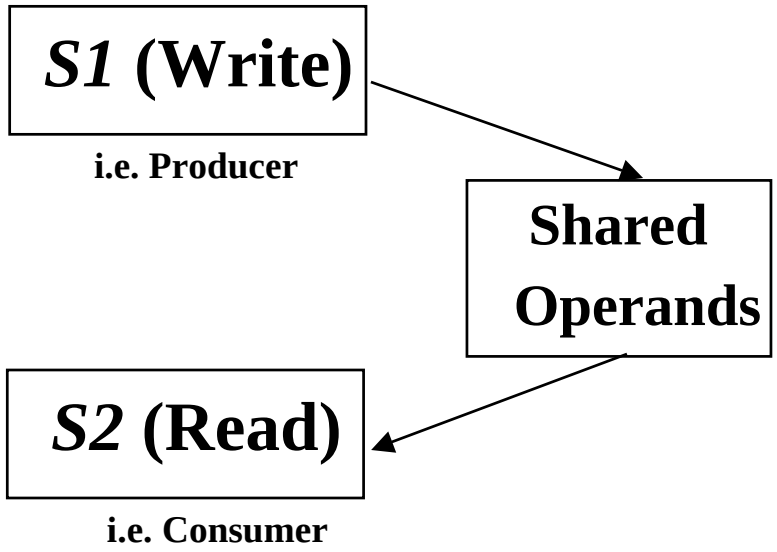
Represented by S1 $+\!\!\longrightarrow$ S2 in dependency graphs

3 **Output dependence:** Two tasks S1, S2 are output dependent if they produce the same output variable

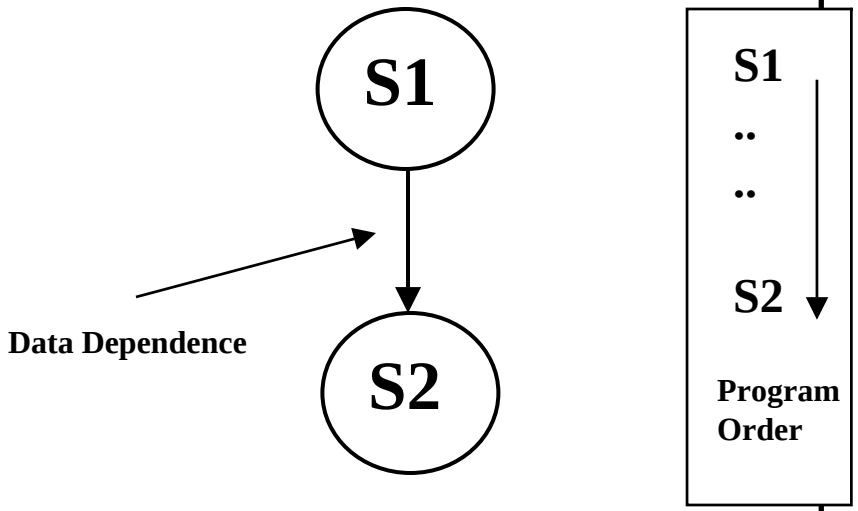
Represented by S1 $o\!\!\longrightarrow$ S2 in dependency graphs

(True) Data (or Flow) Dependence

- **Assume task S2 follows task S1 in sequential program order**
- **Task S1 produces one or more results used by task S2,**
  - Then task S2 is said to be data dependent on task S1
- **Changing the relative execution order of tasks S1, S2 in the parallel program violates this data dependence and results in incorrect execution.**



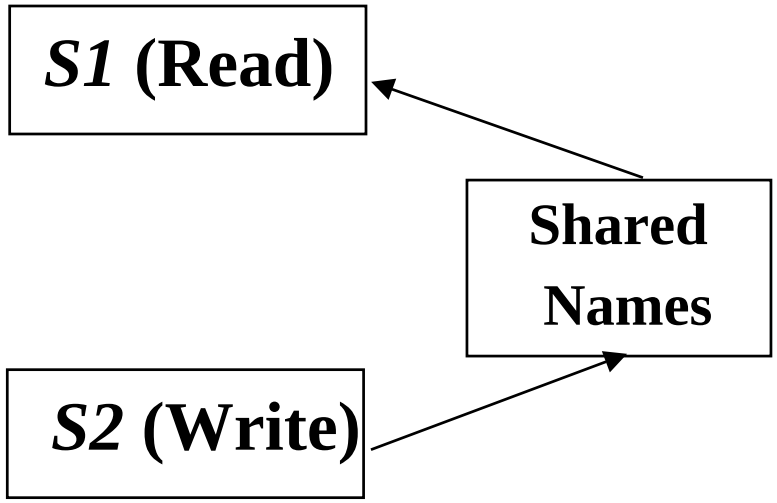
Dependency Graph Representation



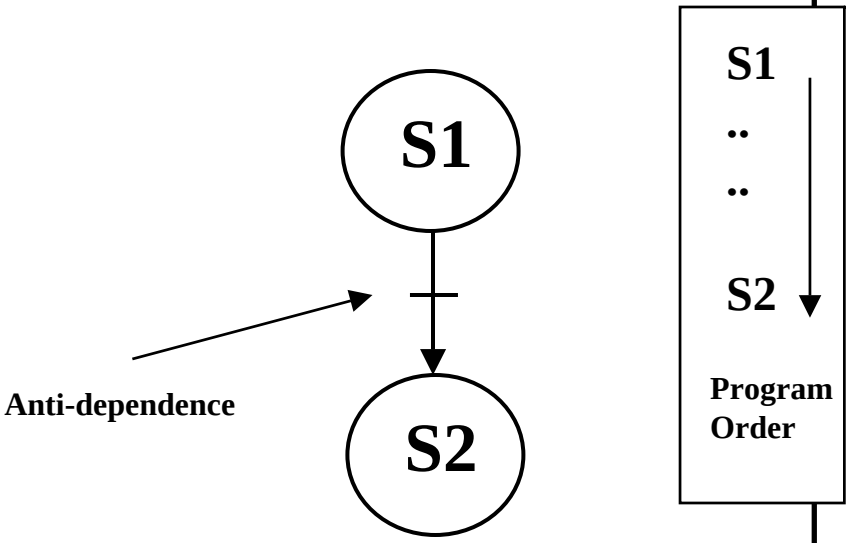
Task S2 is data dependent on task S1

Name Dependence Classification: Anti-Dependence

- **Assume task S2 follows task S1 in sequential program order**
- **Task S1 reads one or more values from one or more names (registers or memory locations)**
- **Instruction S2 writes one or more values to the same names (same registers or memory locations read by S1)**
  - Then task S2 is said to be anti-dependent on task S1
- **Changing the relative execution order of tasks S1, S2 in the parallel program violates this name dependence and may result in incorrect execution.**



Dependency Graph Representation

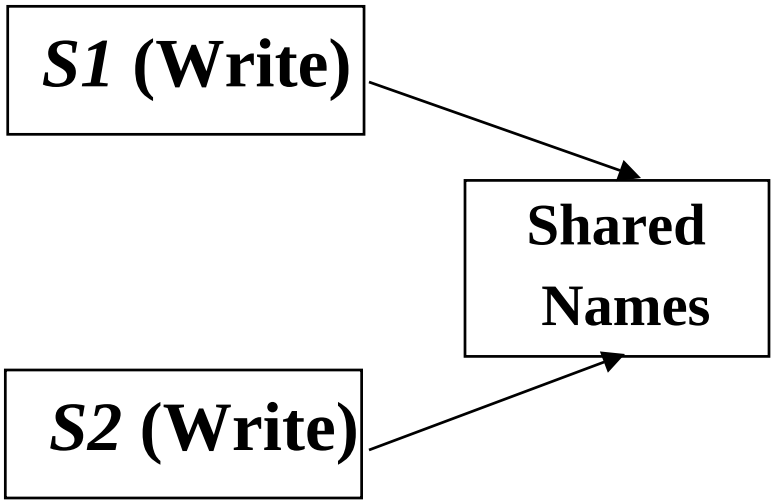


Task S2 is anti-dependent on task S1

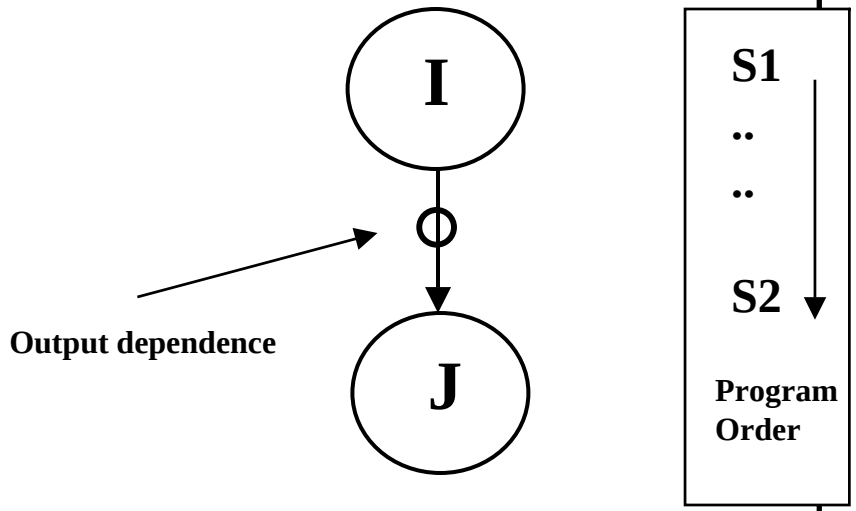
Name: Register or Memory Location

Name Dependence Classification:
Output (or Write) Dependence

- **Assume task S2 follows task S1 in sequential program order**
- **Both tasks S1, S2 write to the same a name or names (same registers or memory locations)**
  - Then task S2 is said to be output-dependent on task S1
- **Changing the relative execution order of tasks S1, S2 in the parallel program violates this name dependence and may result in incorrect execution.**



Dependency Graph Representation



Task S2 is output-dependent on task S1

Name: Register or Memory Location

Dependency Graph Example

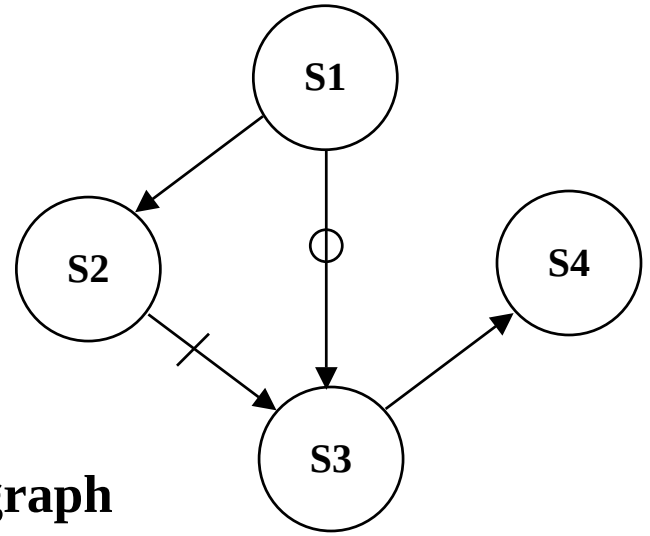
Here assume each instruction is treated as a task

S1:	**Load R1, A**	**/ R1 ← Memory(A) /**
S2:	**Add R2, R1**	**/ R2 ← R1 + R2 /**
S3:	**Move R1, R3**	**/ R1 ← R3 /**
S4:	**Store B, R1**	**/Memory(B) ← R1 /**

True Date Dependence:
(S1, S2) (S3, S4)

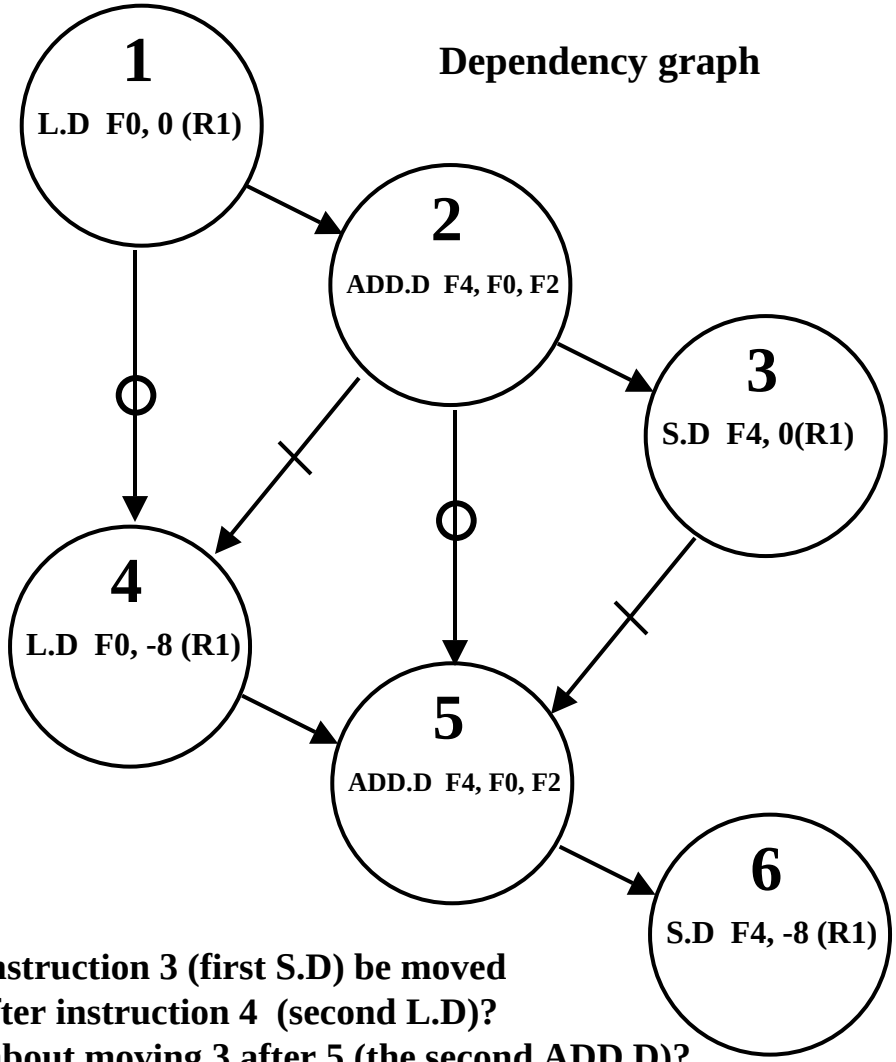
Output Dependence:
(S1, S3)

Anti-dependence:
(S2, S3)



Dependency graph

Dependency Graph Example



MIPS Code

1	L.D	F0, 0 (R1)
2	ADD.D	F4, F0, F2
3	S.D	F4, 0(R1)
4	L.D	F0, -8(R1)
5	ADD.D	F4, F0, F2
6	S.D	F4, -8(R1)

True Date Dependence:
(1, 2) (2, 3) (4, 5) (5, 6)

Output Dependence:
(1, 4) (2, 5)

Anti-dependence:
(2, 4) (3, 5)

Can instruction 4 (second L.D) be moved just after instruction 1 (first L.D)? If not what dependencies are violated?

Can instruction 3 (first S.D) be moved just after instruction 4 (second L.D)? How about moving 3 after 5 (the second ADD.D)? If not what dependencies are violated?

(From 551)

Conditions of Parallelism

- **Control Dependence:**
  - **Order of execution cannot be determined before runtime due to conditional statements.**
- **Resource Dependence:**
  - **Concerned with conflicts in using shared resources among parallel tasks, including:**
    - **Functional units (integer, floating point), memory areas, communication links etc.**
- **Bernstein's Conditions of Parallelim:**

Two processes P_1 , P_2 with input sets I_1, I_2 and output sets O_1, O_2 can execute in parallel (denoted by $P_1 \parallel P_2$) if:

$$I_1 \cap O_2 = \varnothing$$
$$I_2 \cap O_1 = \varnothing$$
$$O_1 \cap O_2 = \varnothing$$

Bernstein's Conditions: An Example

- For the following instructions P_1, P_2, P_3, P_4, P_5 :
  - Each instruction requires one step to execute
  - Two adders are available

$P_1 : C = D \times E$

$P_2 : M = G + C$

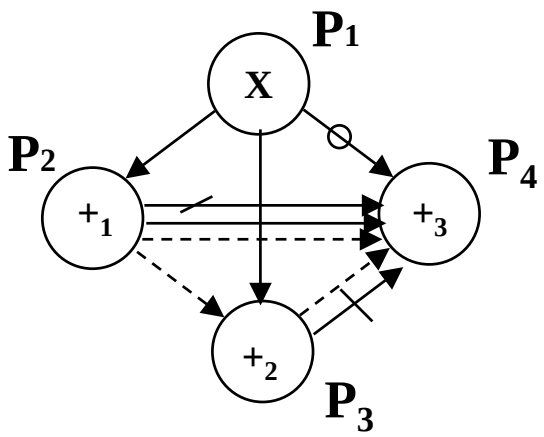
$P_3 : A = B + C$

$P_4 : C = L + M$

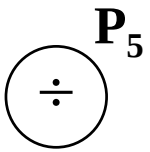
$P_5 : F = G \div E$

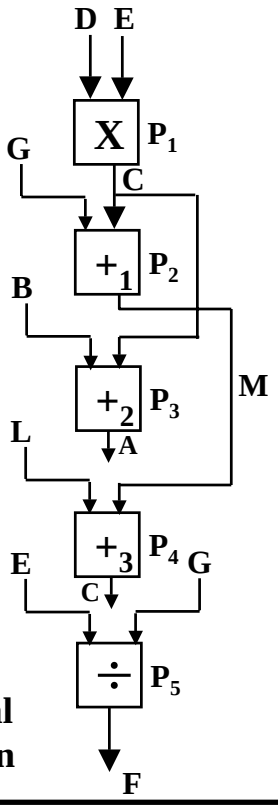
Using Bernstein's Conditions after checking statement pairs:

$P_1 \parallel P_5$, $P_2 \parallel P_3$, $P_2 \parallel P_5$, $P_3 \parallel P_5$, $P_4 \parallel P_5$

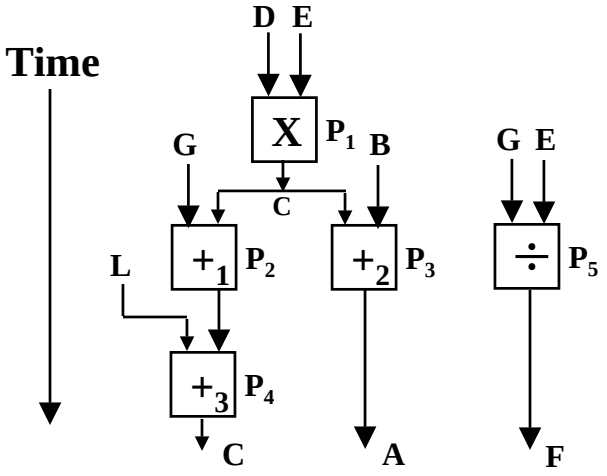


Dependence graph:
Data dependence (solid lines)
Resource dependence (dashed lines)





Sequential execution



Parallel execution in three steps assuming two adders are available per step

Asymptotic Notations for Algorithm Analysis

- Asymptotic analysis of computing time (computational) complexity of an algorithm T(n)= f(n) ignores constant execution factors and concentrates on:
  - Determining the order of magnitude of algorithm performance.
  - How quickly does the running time (computational complexity) grow as a function of the input size.
- We can compare algorithms based on their asymptotic behavior and select the one with lowest rate of growth of complexity in terms of input size or problem size n independent of the computer hardware.

♦ **Upper bound: Order Notation (Big Oh)**

Used in worst case analysis of algorithm performance.

$$f(n) = O(g(n))$$

iff there exist two positive constants c and n_0 such that

$$| f(n) | \leq c | g(n) | \quad \text{for all } n > n_0$$

$\Rightarrow$ i.e. g(n) is an upper bound on f(n)

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Asymptotic Notations for Algorithm Analysis

- **Asymptotic Lower bound:** Big Omega Notation

  Used in the analysis of the lower limit of algorithm performance

$$f(n) = \Omega(g(n))$$

if there exist positive constants c, n_0 such that

$$| f(n) | \geq c | g(n) | \quad \text{for all} \quad n > n_0$$

$\Rightarrow$ i.e. g(n) is a lower bound on f(n)

- **Asymptotic Tight bound:** Big Theta Notation

  Used in finding a tight limit on algorithm performance

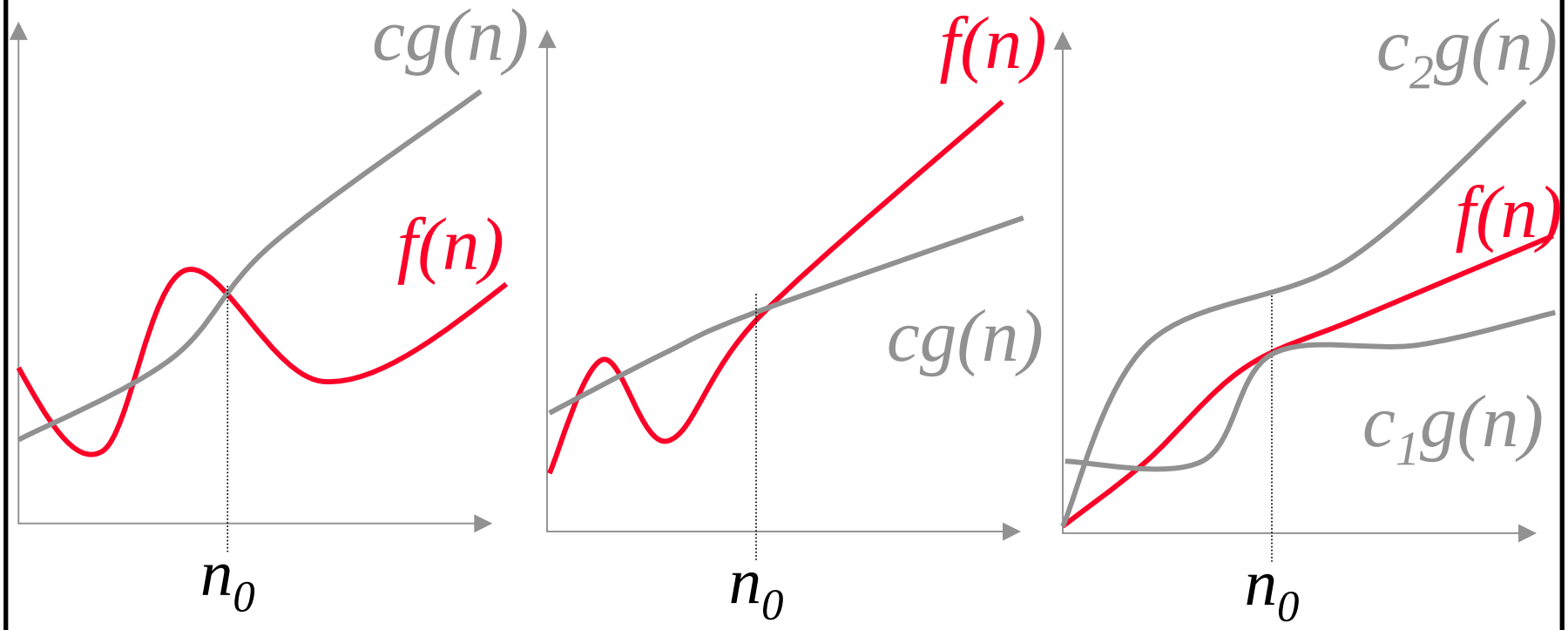
$$f(n) = \Theta (g(n))$$

if there exist constant positive integers c_1, c_2, and n_0 such that

$$c_1 | g(n) | \leq | f(n) | \leq c_2 | g(n) | \quad \text{for all} \quad n > n_0$$

$\Rightarrow$ i.e. g(n) is both an upper and lower bound on f(n)

Graphs of O, Ω, Θ



f(n) =O(g(n))
Upper Bound

f(n) = Ω(g(n))
Lower Bound

f(n) = Θ(g(n))
Tight bound

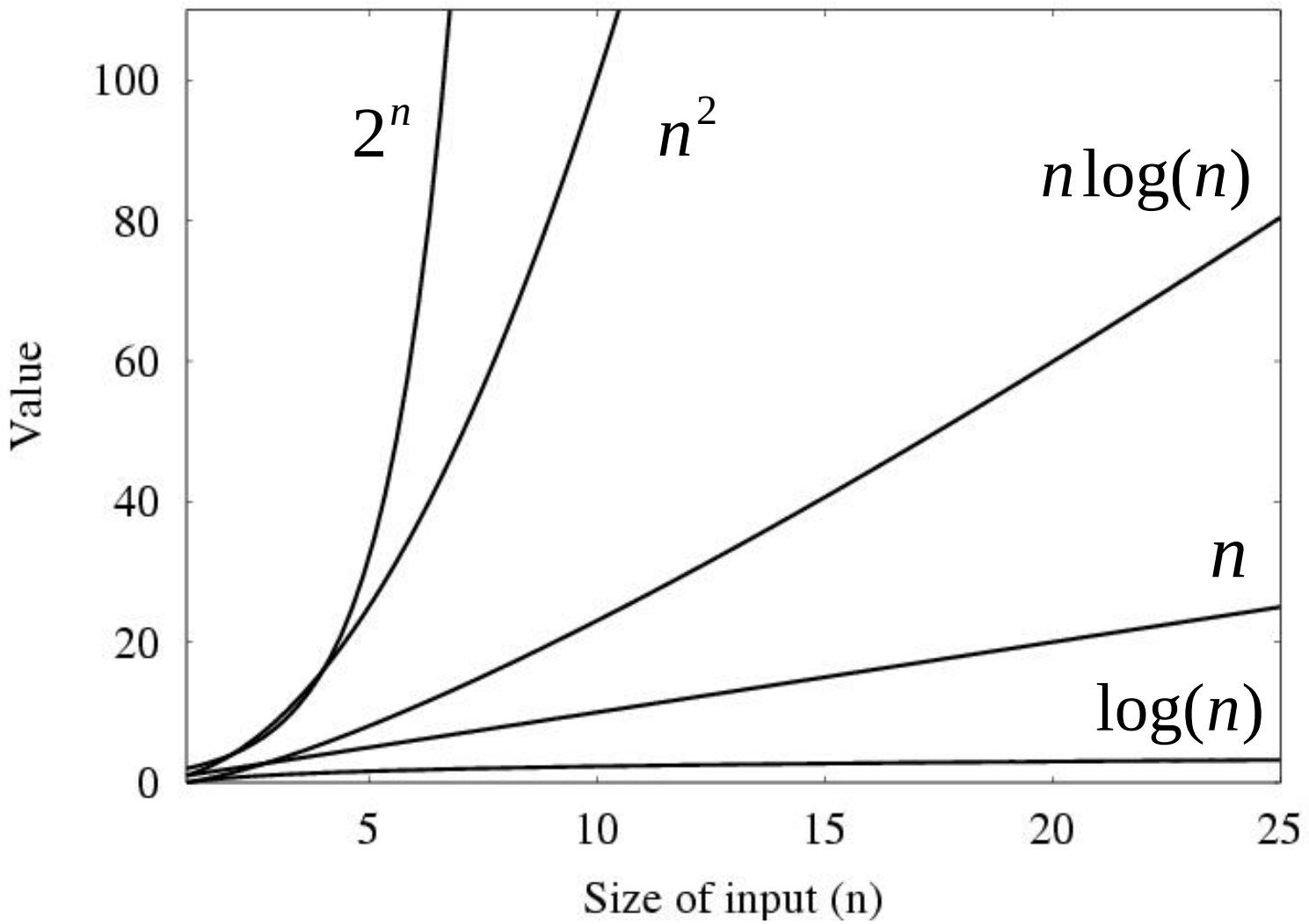
Rate of Growth of Common Computing Time Functions

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

e.g NP-Complete/Hard Algorithms
(NP – Non Polynomial)

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

Rate of Growth of Common Computing Time Functions



O(1) < O(log n) < O(n) < O(n log n) < O (n^2) < O(n^3) < O(2^n)

Theoretical Models of Parallel Computers

- **Parallel Random-Access Machine (PRAM):**
  - ***n*** **processor, global shared memory model.**
  - **Models idealized parallel computers with zero synchronization communication or memory access overhead.**
  - **Utilized in parallel algorithm development and scalability and complexity analysis.**
- **PRAM variants: More realistic models than pure PRAM**
  - **EREW-PRAM: Simultaneous memory reads or writes to/from the same memory location are not allowed.**
  - **CREW-PRAM: Simultaneous memory writes to the same location is not allowed. (Better to model SAS MIMD?)**
  - **ERCW-PRAM: Simultaneous reads from the same memory location are not allowed.**
  - **CRCW-PRAM: Concurrent reads or writes to/from the same memory location are allowed.**

Sometimes used to model SIMD since no memory is shared

Example: sum algorithm on P processor PRAM

Add n numbers

- **Input:** Array A of size $n = 2^k$ in shared memory
- **Initialized local variables:**
  - the order n,
  - number of processors $p = 2^q \leq n$,
  - the processor number s
- **Output:** The sum of the elements of A stored in shared memory

begin

1. for j = 1 to l (= n/p) do
   Set B(l(s - 1) + j): = A(l(s-1) + j)
2. for h = 1 to log n do
   2.1 if $(k- h - q \geq 0)$ then
   for $j = 2^{k-h-q}(s-1) + 1$ to $2^{k-h-q}S$ do
   Set B(j): = B(2j -1) + B(2s)
   2.2 else {if $(s \leq 2^{k-h})$ then
   Set B(s): = B(2s -1) + B(2s)}
3. if (s = 1) then set S: = B(1)

end

Compute partial Sums

Running time analysis:

- Step 1: takes O(n/p) each processor executes n/p operations
- The hth of step 2 takes $O(n / (2^h p))$ since each processor has to perform $(n / (2^h p))$ ⌉ operations
- Step three takes O(1)
- Total Running time:

$$T_p(n) = O\left(\frac{n}{p} + \sum_{h=1}^{\log n} \left\lceil \frac{n}{2^h p} \right\rceil\right) = O(\frac{n}{p} + \log n)$$

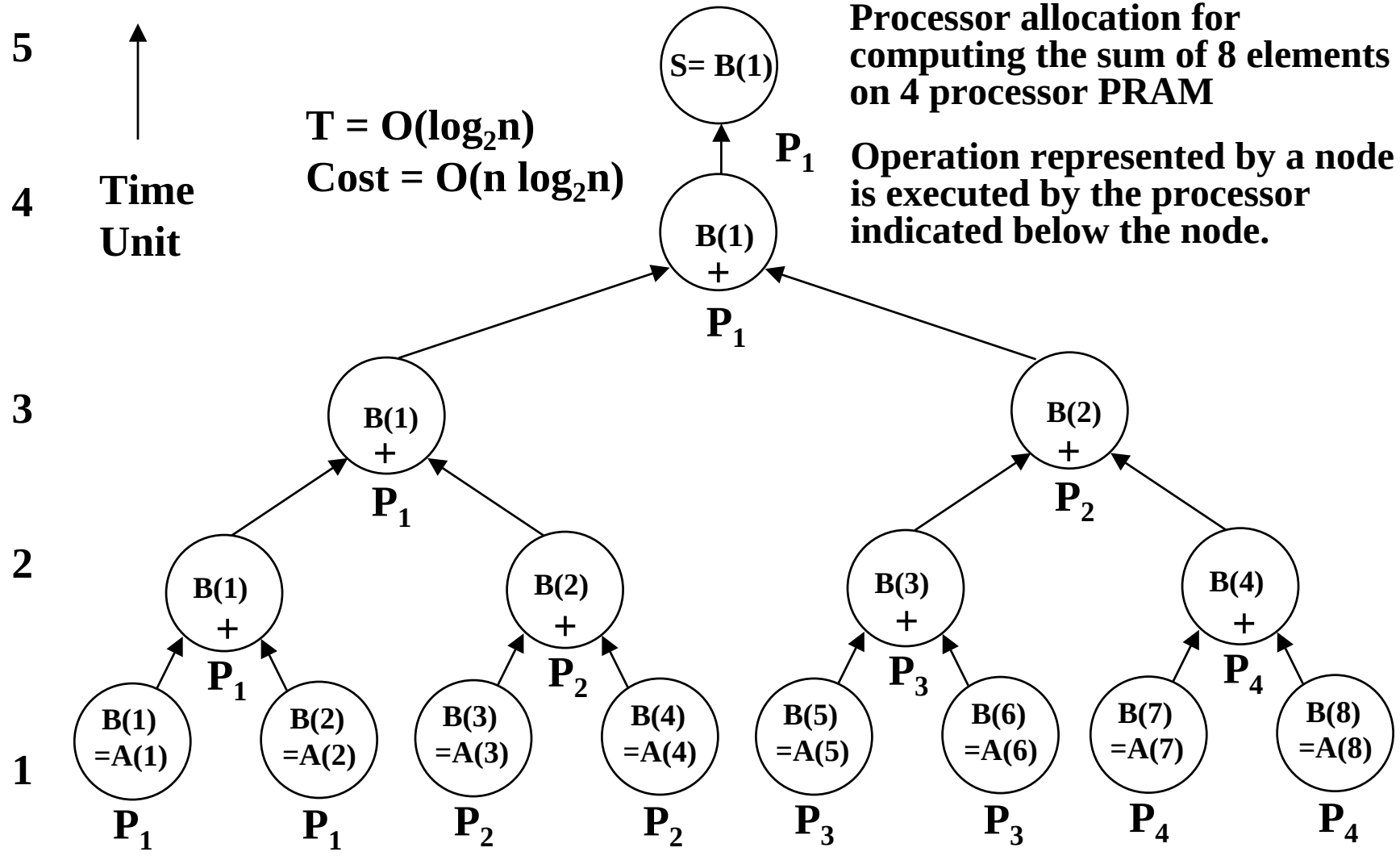
Example: Sum Algorithm on P Processor PRAM

$T = 2 + \log_2(8) = 2 + 3 = 5$

For $n = 8$ $p = 4 = n/2$

Processor allocation for computing the sum of 8 elements on 4 processor PRAM

Operation represented by a node is executed by the processor indicated below the node.



For $n >> p$ $O(n/p + \text{Log}_2 n)$

Performance of Parallel Algorithms

- Performance of a parallel algorithm is typically measured in terms of worst-case analysis.
- For problem Q with a PRAM algorithm that runs in time T(n) using P(n) processors, for an instance size of *n*:
  - The time-processor product C(n) = T(n) . P(n) represents the cost of the parallel algorithm.
  - For $P \leq P(n)$, each of the of the T(n) parallel steps is simulated in O(P(n)/p) substeps. Total simulation takes O(T(n)P(n)/p)
  - The following four measures of performance are asymptotically equivalent:
    - P(n) processors and T(n) time
    - C(n) = P(n)T(n) cost and T(n) time
    - O(T(n)P(n)/p) time for any number of processors $p \leq P(n)$
    - O(C(n)/p + T(n)) time for any number of processors.

Matrix Multiplication On PRAM

- **Multiply matrices A x B = C of sizes n x n**
- **Sequential Matrix multiplication:**

For i=1 to n {

For j=1 to n { $C(i,j) = \sum_{t=1}^{n} a(i,t) \times b(t,j)$ **}}**

Dot product O(n) sequentially on one processor

Thus sequential matrix multiplication time complexity $O(n^3)$

- **Matrix multiplication on PRAM with p = $O(n^3)$ processors.**
  - **Compute in parallel for all i, j, t = 1 to n**

c(i,j,t) = a(i, t) x b(t, j) O(1)

All product terms computed in parallel in one time step using n^3 processors

  - **Compute in parallel for all i;j = 1 to n:**

$$C(i,j) = \sum_{t=1}^{n} c(i,j,t) \qquad O(\log_2 n)$$

All dot products computed in parallel Each taking $O(\log_2 n)$

Thus time complexity of matrix multiplication on PRAM with n^3 processors = $O(\log_2 n)$ Cost(n) = $O(n^3 \log_2 n)$

  - **Time complexity of matrix multiplication on PRAM with n^2 processors = $O(n \log_2 n)$**
  - **Time complexity of matrix multiplication on PRAM with n processors = $O(n^2 \log_2 n)$**

From last slide: O(C(n)/p + T(n)) time complexity for any number of processors.

The Power of The PRAM Model

- **Well-developed techniques and algorithms to handle many computational problems exist for the PRAM model.**
- **Removes algorithmic details regarding synchronization and communication cost, concentrating on the structural and *fundamental data dependency properties* (dependency graph) of the parallel computation/algorithm.**
- **Captures several important parameters of parallel computations. Operations performed in unit time, as well as processor allocation.**
- **The PRAM design paradigms are robust and many parallel network (message-passing) algorithms can be directly derived from PRAM algorithms.**
- **It is possible to incorporate synchronization and communication costs into the shared-memory PRAM model.**

Network Model of Message-Passing Multicomputers

- A network of processors can viewed as a graph G (N,E)
  - Each node $i \in N$ represents a processor
  - Each edge $(i,j) \in E$ represents a two-way communication link between processors i and j.
  - Each processor is assumed to have its own local memory.
  - No shared memory is available.
  - Operation is synchronous or asynchronous(message passing).
  - Basic message-passing communication primitives:
    - send(X,i) a copy of data X is sent to processor P_i, execution continues.
    - receive(Y, j) execution of recipient processor is suspended (blocked) until the data from processor P_j is received and stored in Y then execution resumes.

Graph represents network topology

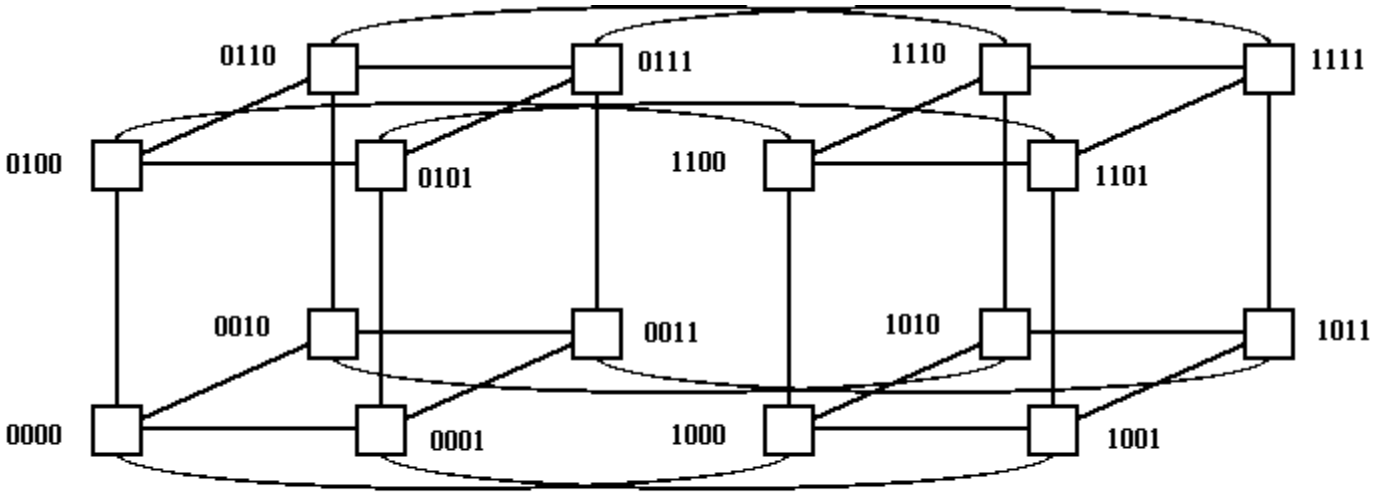
e.g Point-to-point interconnect

Network Model of Multicomputers

- **Routing** is concerned with delivering each message from source to destination over the network.
- Additional important network topology parameters:
  - The **network diameter** is the maximum distance between any pair of nodes (in links or hops).
  - The **maximum degree** of any node in G
    - Directly connected to how many other nodes
- Example:
  - **Linear array:** P processors P_1, …, P_p are connected in a linear array where:
    - Processor P_i is connected to P_{i-1} and P_{i+1} if they exist.
    - Diameter is p-1; maximum degree is 2 (1 or 2).
  - **A ring** is a linear array of processors where processors P_1 and P_p are directly connected. Degree = 2, Diameter = p/2

A Four-Dimensional Hypercube

- **In a d-dimensional binary hypercube, each of the 2^d nodes is assigned a d-bit address.**
- **Two processors are connected if their binary addresses differ in one bit position.**
- **Degree = Diameter = d**



Here d = Degree = Diameter = 4
Number of nodes = $2^d = 2^4$ = 16 nodes

Binary tree computations map directly to the hypercube topology

Example: Asynchronous Matrix Vector Product on a Ring

$Y = Ax$

- **Input:**
  - $n \times n$ matrix A ; vector x of order n
  - The processor number i. The number of processors
  - The ith submatrix $B = A(1:n, (i-1)r+1 ; ir)$ of size $n \times r$ where $r = n/p$
  - The ith subvector $w = x(i-1)r + 1 : ir)$ of size r
- **Output:**
  - Processor P_i computes the vector $y = A_1x_1 + \ldots. A_ix_i$ and passes the result to the right
  - Upon completion P_1 will hold the product Ax

Begin

1. Compute the matrix vector product $z = Bw$

   For all processors, $O(n2/p)$

2. If i = 1 then set y: = 0

   else receive(y,left)

3. Set y: = y +z
4. send(y, right)
5. if i =1 then receive(y,left)

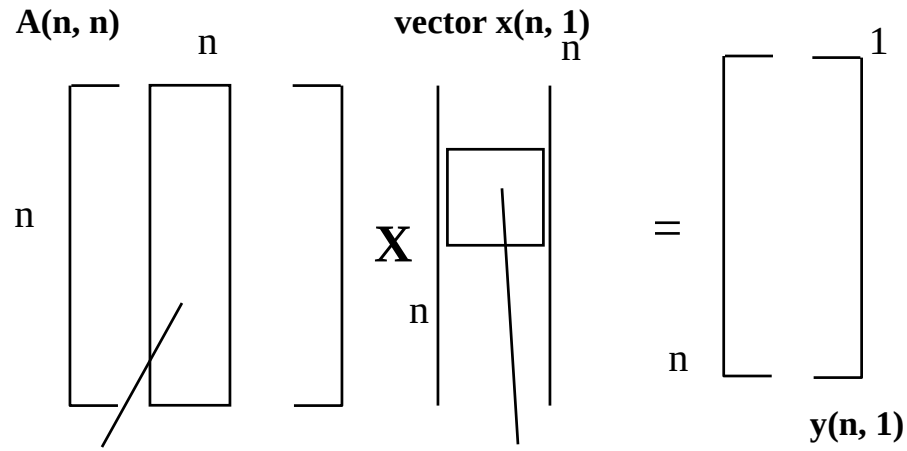
End

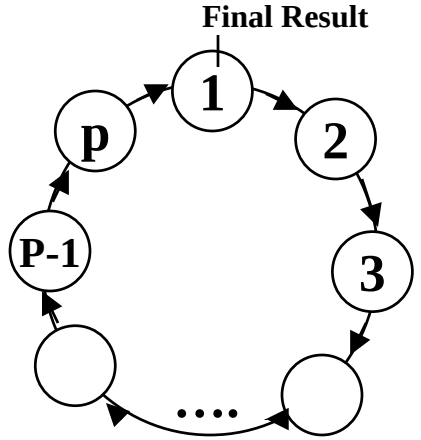
$T_{comp} = k(n^2/p)$
$T_{comm} = p(l+ mn)$
$T = T_{comp} + T_{comm}$
$= k(n^2/p) + p(l+ mn)$

k, l, m constants

Sequential time complexity of Y = Ax : $O(n^2)$

Matrix Vector Product y = Ax on a Ring





Submatrix B_i of size n x n/p

Subvector W_i of size (n/p,1)

(n,1) $Z_i = B_i w_i$ Compute in parallel

$$Y = Ax = Z_1 + Z_2 + \ldots Zp = B_1 w_1 + B_2 w_2 + \ldots B_p w_p$$

- For each processor i compute in parallel

Computation: $O(n^2/p)$ $Z_i = B_i w_i$ $O(n^2/p)$

Communication-to-Computation Ratio = $= pn / (n^2/p) = p^2/n$

- For processor #1: set y =0, y = y + Zi, send y to right processor
- For every processor except #1

  Receive y from left processor, y = y + Zi, send y to right processor
- For processor 1 receive final result from left processor p
- $T = O(n^2/p + pn)$

Communication O(pn)

Computation

Communication

Creating a Parallel Program

- **Assumption: Sequential algorithm to solve problem is given**
  - **Or a different algorithm with more inherent parallelism is devised.**
  - **Most programming problems have several parallel solutions or algorithms. The best solution may differ from that suggested by existing sequential algorithms.**

One must:

- **Identify work that can be done in parallel (dependency analysis)**
- **Partition work and perhaps data among processes (Tasks)**
- **Manage data access, communication and synchronization**
- ***Note*: work includes computation, data access and I/O**

Main goal: Maximize Speedup

$$\textbf{\textit{Speedup (p)}} = \frac{Performance(p)}{Performance(1)}$$

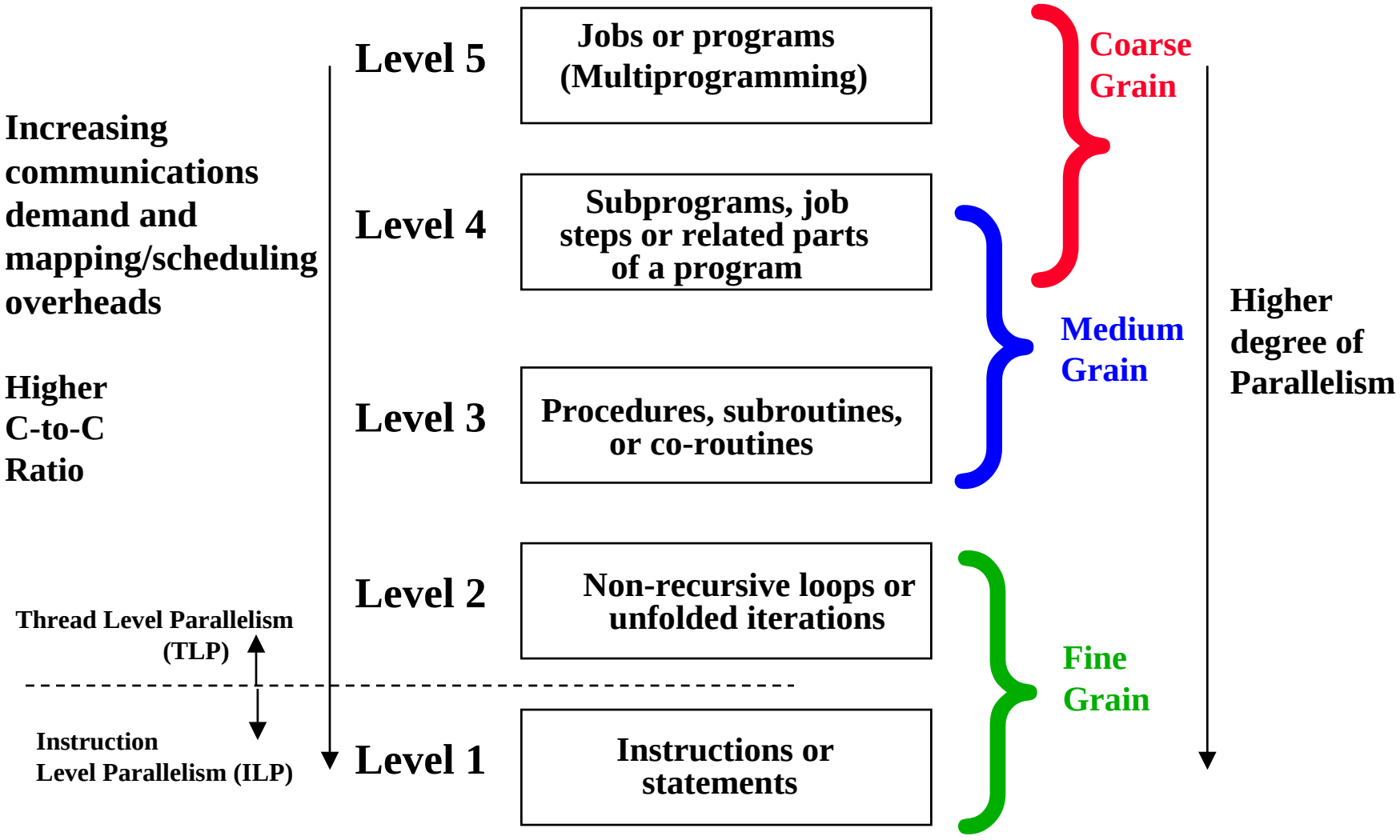
By:
Minimizing parallelization overheads
Balancing workload on processors

For a fixed size problem:

$$\textbf{\textit{Speedup (p)}} = \frac{Time(1)}{Time(p)}$$

Levels of Parallelism in Program Execution

According to task (grain) size



Task size affects Communication-to-Computation ratio (C-to-C ratio) and communication overheads

Hardware and Software Parallelism

- **Hardware parallelism:** **e.g Number of processors**
  - **Defined by machine architecture, hardware multiplicity (number of processors available) and connectivity.**
  - **Often a function of cost/performance tradeoffs.**
  - **Characterized in a single processor by the number of instructions *k* issued in a single cycle (*k*-issue processor).**
  - **A multiprocessor system with *n* *k*-issue processor can handle a maximum limit of *nk* parallel instructions (at ILP level) or n parallel threads at thread-level parallelism (TLP) level.**

- **Software parallelism:** **e.g Degree of Parallelism (DOP)**
  - **Defined by the control and data dependence of programs.**
  - **Revealed in program profiling or program dependency (data flow) graph.**
  - **A function of algorithm, parallel task grain size, programming style and compiler optimization.**

Computational Parallelism and Grain Size

- **Task grain size (granularity) is a measure of the amount of computation involved in a task in parallel computation:**
  - **Instruction Level (Fine Grain Parallelism):**
    - **At instruction or statement level.**
    - **20 instructions grain size or less.**
    - **For scientific applications, parallelism at this level range from 500 to 3000 concurrent statements**
    - **Manual parallelism detection is difficult but assisted by parallelizing compilers.**
  - **Loop level (Fine Grain Parallelism):**
    - **Iterative loop operations.**
    - **Typically, 500 instructions or less per iteration.**
    - **Optimized on vector parallel computers.**
    - **Independent successive loop operations can be vectorized or run in SIMD mode.**

Computational Parallelism and Grain Size

- **Procedure level (Medium Grain Parallelism): :**
  - **Procedure, subroutine levels.**
  - **Less than 2000 instructions.**
  - **More difficult detection of parallel than finer-grain levels.**
  - **Less communication requirements than fine-grain parallelism.**
  - **Relies heavily on effective operating system support.**
- **Subprogram level (Coarse Grain Parallelism): :**
  - **Job and subprogram level.**
  - **Thousands of instructions per grain.**
  - **Often scheduled on message-passing multicomputers.**
- **Job (program) level, or Multiprogrammimg:**
  - **Independent programs executed on a parallel computer.**
  - **Grain size in tens of thousands of instructions.**

Software Parallelism Types: Data Vs. Functional Parallelism

1 - Data Parallelism:

- Parallel (often similar) computations performed on elements of large data structures
  - (e.g numeric solution of linear systems, pixel-level image processing)
- Such as resulting from parallelization of loops.
- Usually easy to load balance.
- Degree of concurrency usually increases with input or problem size. e.g $O(n^2)$ in equation solver example (next slide).

2- Functional Parallelism:

- Entire large tasks (procedures) with possibly different functionality that can be done in parallel on the same or different data.
  - Software Pipelining: Different functions or software stages of the pipeline performed on different data:
    - As in video encoding/decoding, or polygon rendering.
- Concurrency degree usually modest and does not grow with input size
  - Difficult to load balance.
  - Often used to reduce synch wait time between data parallel phases.

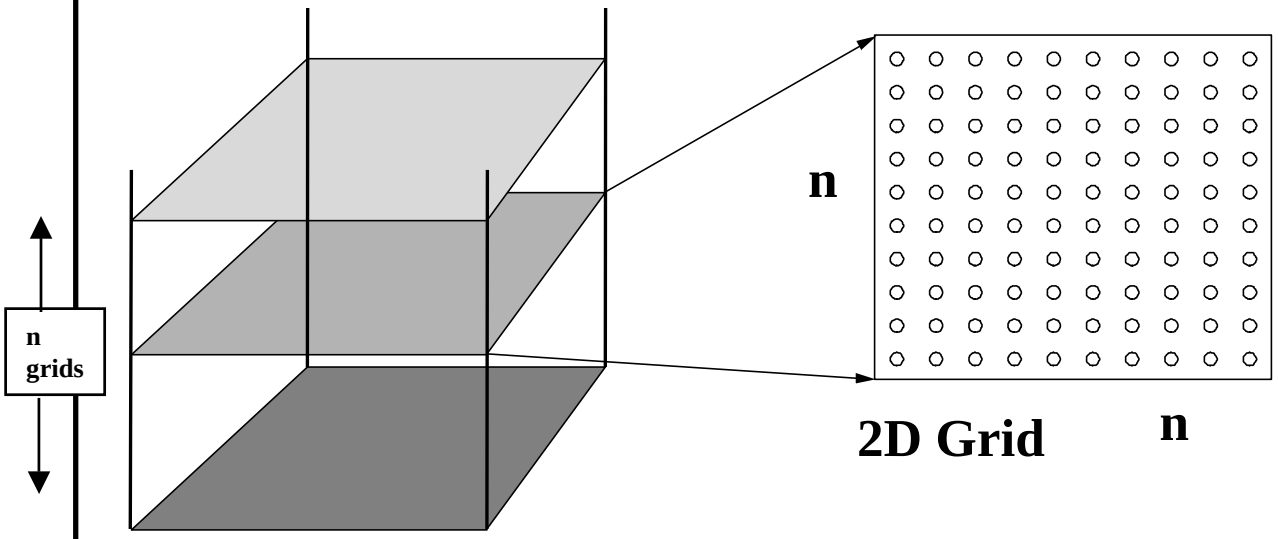
Most scalable parallel programs:

(more concurrency as problem size increases) parallel programs:

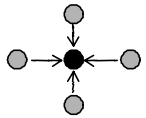
Data parallel programs (per this loose definition)

- Functional parallelism can still be exploited to reduce synchronization wait time between data parallel phases.

Example Motivating Problem: Simulating Ocean Currents/Heat Transfer ...



(a) Cross sections (b) Spatial discretization of a cross section



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

Degree of Parallelism (DOP) or concurrency: $O(n^2)$ data parallel computations per grid

Total $O(n^3)$ Computations Per iteration

- **Model as two-dimensional grids**
- **Discretize in space and time**
  - **finer spatial and temporal resolution => greater accuracy**
- **Many different computations per time step**
  - **set up and solve equations iteratively (Gauss-Seidel) .**
- **Concurrency across and within grid computations per iteration**
  - **n^2 parallel computations per grid x number of grids**

Covered next lecture (PCA 2.3)

Limited Concurrency: Amdahl's Law

- Most fundamental limitation on parallel speedup.
- Assume a fraction s of sequential execution time runs on a single processor cannot be parallelized.
- Assuming that the problem size remains fixed and that the remaining fraction (1-s) can be parallelized without any parallization overheads to run on p processors and thus reduced by a factor of p.
- The resulting speedup for p processors:

e.g perfect speedup for parallelizable portion

$$\text{Speedup}(p) = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}$$

$$\text{Parallel Execution Time} = (S + (1-S)/P) \times \text{Sequential Execution Time}$$

$$\text{Speedup}(p) = \frac{\text{Sequential Execution Time}}{((1 - F) + F/S) \times \text{Sequential Execution Time}} = \frac{1}{s + (1-s)/p}$$

- Thus for a fixed problem size, if fraction *s* of sequential execution is inherently serial, speedup <= *1/s*

Amdahl's Law Example

- **Example: 2-phase calculation**
  - **sweep over *n*-by-*n* grid and do some independent computation**
  - **sweep again and add each value to global sum sequentially**
  - **Time for first phase = n^2/p**

P = number of processors

  - **Second phase serialized at global variable, so time = n^2**

Sequential time = Time Phase1 + Time Phase2 = $n^2 + n^2 = 2n^2$

  - **Speedup <=** $\dfrac{2n^2}{\frac{n^2}{p} + n^2} = \dfrac{1}{\frac{0.5}{p} + 0.5}$ **or at most 1/s = 1/.5= 2**

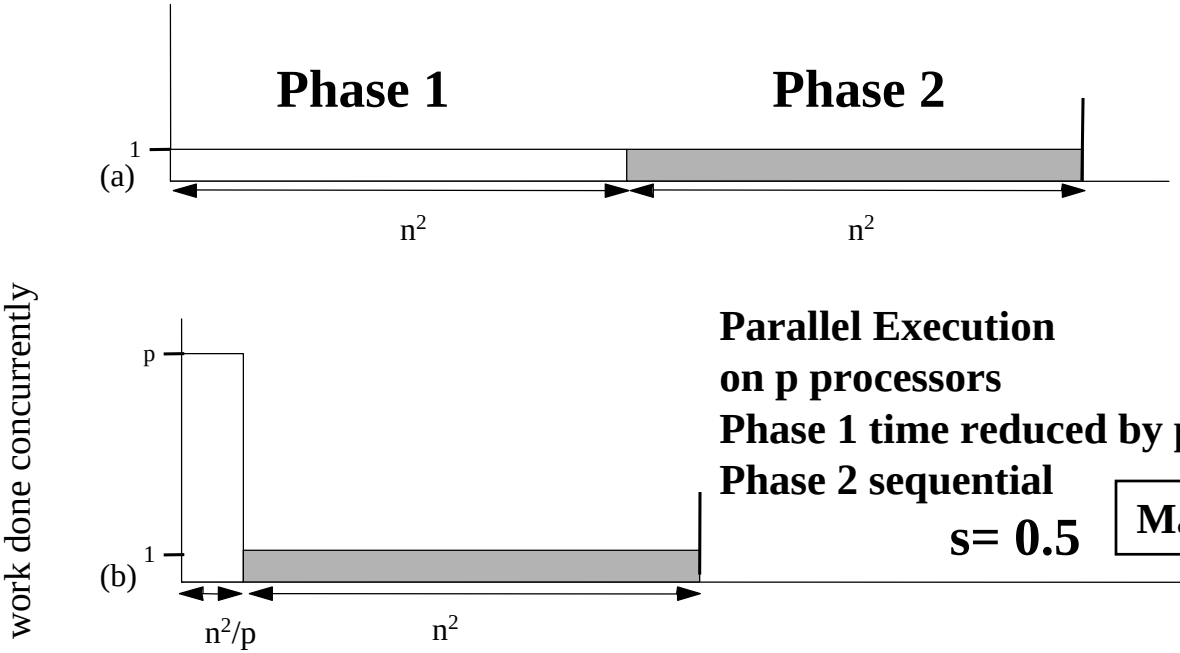
Phase1 (parallel) **Phase1 (sequential)**

  - **Possible Trick: divide second phase into two**
    - **Accumulate into private sum during sweep**
    - **Add per-process private sum into global sum**
  - **Parallel time is $n^2/p + n2/p$ + p, and speedup:**

$$\frac{2n^2}{2n^2/p + p} = \frac{1}{1/p + p/2n^2}$$

For large n Speedup ~ p

Amdahl's Law Example: A Pictorial Depiction



Sequential Execution

Parallel Execution on p processors
Phase 1 time reduced by p
Phase 2 sequential
s= 0.5

$$\text{Speedup} = \frac{1}{\frac{0.5}{p} + 0.5}$$

Maximum Possible Speedup = 2

Parallel Execution on p processors
Phase 1 time reduced by p
Phase 2 divided into two steps

$$\text{Speedup} = \frac{2n^2}{2n^2 + p^2} = \frac{1}{1/p + p/2n^2}$$

For large n Speedup ~ p

Parallel Performance Metrics
Degree of Parallelism (DOP)

- For a given time period, DOP reflects the number of processors in a specific parallel computer actually executing a particular parallel program.
- **Average Parallelism A:**
  - given maximum parallelism = *m*
  - *n* homogeneous processors
  - computing capacity of a single processor Δ
  - Total amount of work W (instructions, computations):

Computations/sec

$$W = \Delta \int_{t1}^{t2} DOP(t)dt \quad \text{or as a discrete summation} \quad W = \Delta \sum_{i=1}^{m} i.t_i$$

Where t_i is the total time that DOP = i and $\sum_{i=1}^{m} t_i = t_2 - t_1$

The average parallelism A:

$$A = \frac{1}{t_2 - t_1} \int_{t1}^{t2} DOP(t)dt$$

In discrete form

$$A = \left(\sum_{i=1}^{m} i.t_i \right) \Big/ \left(\sum_{i=1}^{m} t_i \right)$$

DOP Area

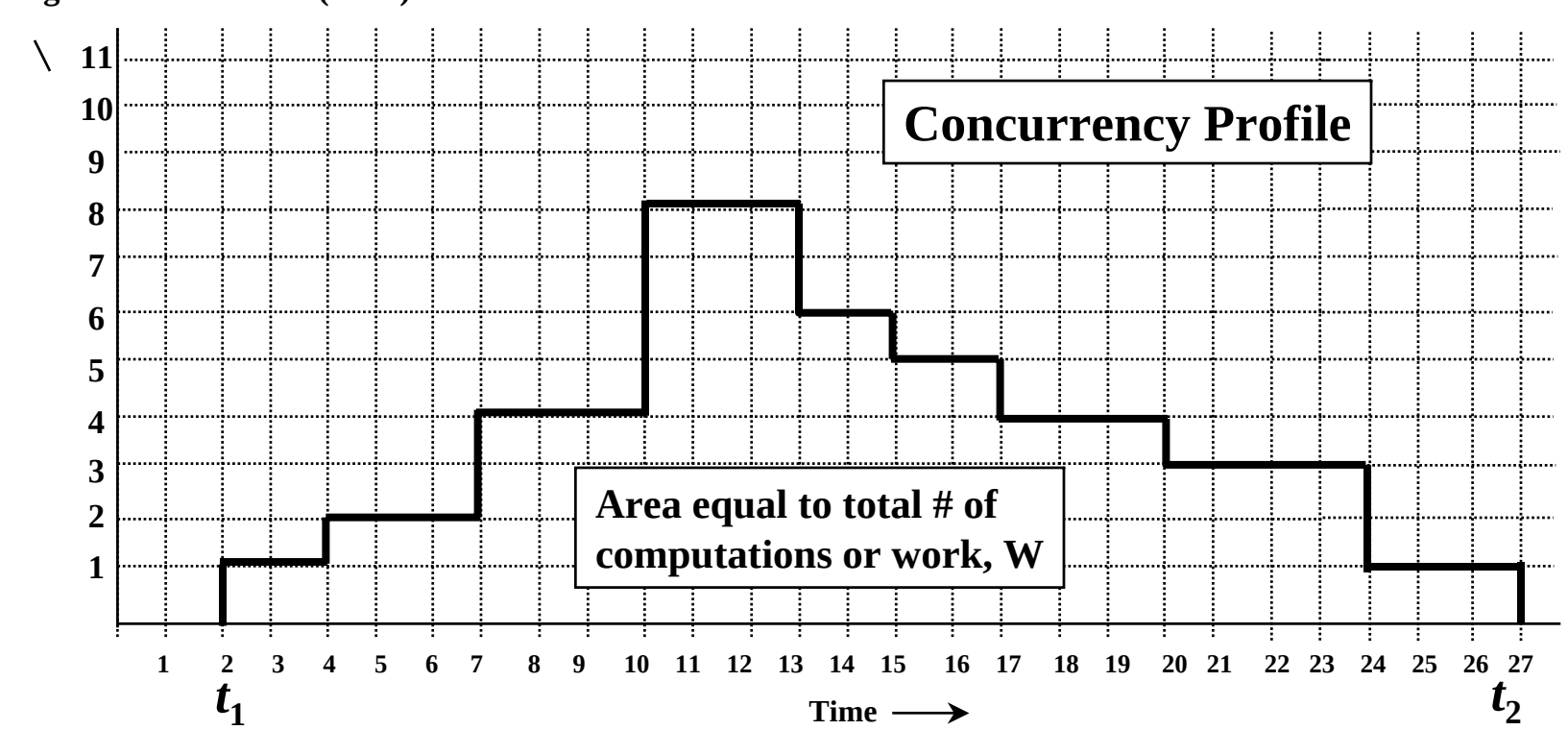
Execution Time

Example: Concurrency Profile of A Divide-and-Conquer Algorithm

- Execution observed from $t_1 = 2$ to $t_2 = 27$
- Peak parallelism $m = 8$
- A = (1x5 + 2x3 + 3x4 + 4x6 + 5x2 + 6x2 + 8x3) / (5 + 3+4+6+2+2+3)
  = 93/25 = 3.72

$$A = \left(\sum_{i=1}^{m} i.t_i\right) \Big/ \left(\sum_{i=1}^{m} t_i\right)$$

Average Parallelism

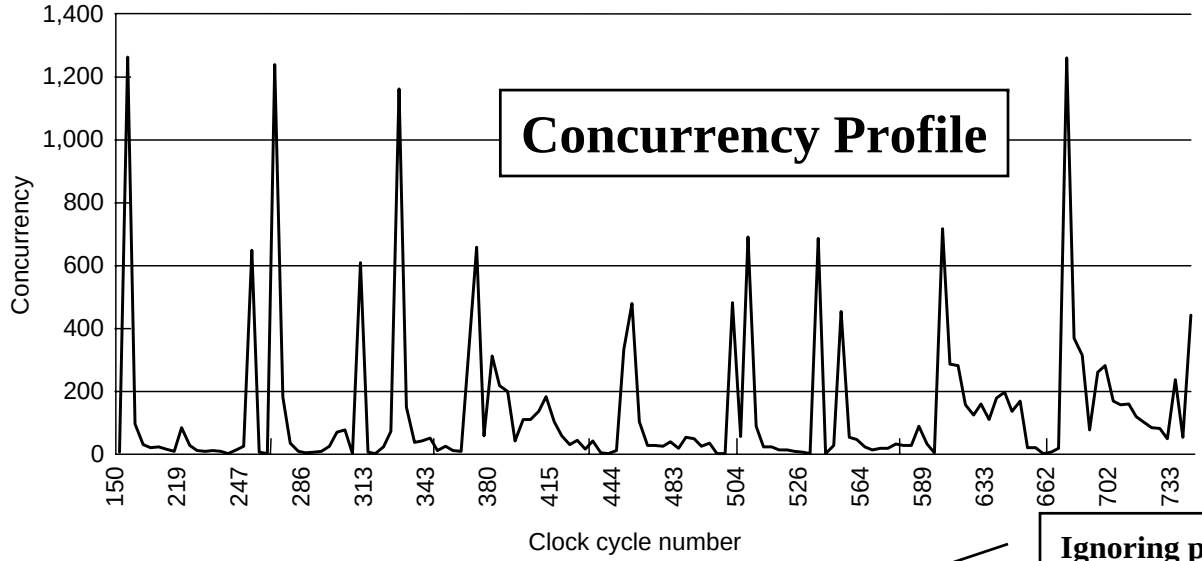


Ignoring parallelization overheads, what is the speedup?

(i.e total work on parallel processors equal to work on single processor)

Concurrency Profile & Speedup

For a parallel program DOP may range from 1 (serial) to a maximum *m*



Ignoring parallelization overheads

- **Area under curve is total work done, or time with 1 processor**
- **Horizontal extent is lower bound on time (infinite processors)**
- **Speedup is the ratio:** $\frac{\sum_{k=1}^{\infty} f_k k}{\sum_{k=1}^{\infty} f_k \left\lceil \frac{k}{p} \right\rceil}$ **, base case:** $\frac{1}{s + \frac{1-s}{p}}$

  (for p processors)

  k **= degree of parallelism** f_k **= Total time with degree of parallelism k**

- **Amdahl's law applies to any overhead, not just limited concurrency.**

Ignoring parallelization overheads

Amdahl's Law with Multiple Degrees of Parallelism

- **Assume different fractions of sequential execution time of a problem running on a single processor have different degrees of parallelism (DOPs) and that the problem size remains fixed.**
  - **Fraction F_i of the sequential execution time can be parallelized without any parallization overheads to run on S_i processors and thus reduced by a factor of S_i.**
  - **The remaining fraction of sequential execution time cannot be parallelized and runs on a single processor.**
- **Then**

$$Speedup = \frac{\text{Original Execution Time}}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right) X \text{Original Execution Time}}$$

$$Speedup = \frac{1}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right)}$$

+ overheads?

Note: All fractions F_i refer to original sequential execution time on one processor.

How to account for parallelization overheads in above speedup?

Amdahl's Law with Multiple Degrees of Parallelism : Example

- **Given the following degrees of parallelism in a sequential program**

i.e on 10 processors

$DOP_1 = S_1 = 10$ $Percentage_1 = F_1 = 20\%$

$DOP_2 = S_2 = 15$ $Percentage_1 = F_2 = 15\%$

$DOP_3 = S_3 = 30$ $Percentage_1 = F_3 = 10\%$

- **What is the parallel speedup when running on a parallel system without any parallization overheads ?**

$$Speedup = \frac{1}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i} \right)}$$

- **Speedup = 1 / [(1 - .2 - .15 - .1) + .2/10 + .15/15 + .1/30)]**

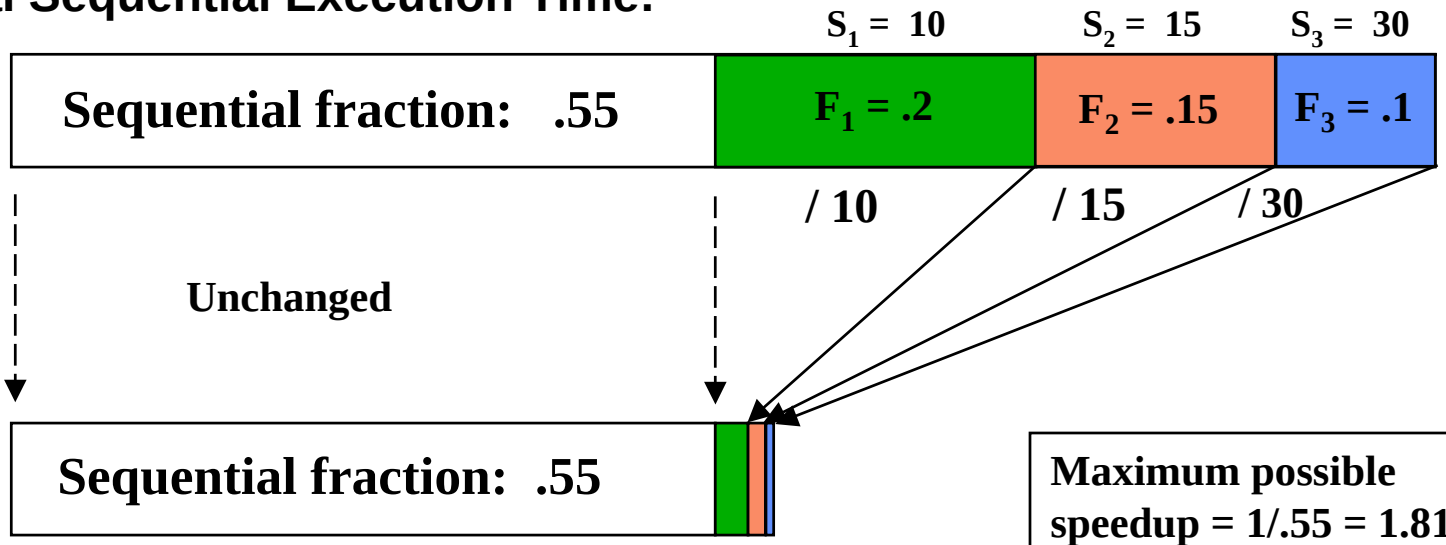
= 1 / [.55 + .0333]

= 1 / .5833 = 1.71

From 551

Pictorial Depiction of Example

Before:
Original Sequential Execution Time:



Maximum possible speedup = 1/.55 = 1.81

After:
Parallel Execution Time: .55 + .02 + .01 + .00333 = .5833

Parallel Speedup = 1 / .5833 = 1.71

Limited by sequential portion

Note: All fractions (F_i, i = 1, 2, 3) refer to original sequential execution time.

Parallel Performance Example

- The execution time T for three parallel programs is given in terms of processor count P and problem size N (or three parallel algorithms for a problem)
- In each case, we assume that the total computation work performed by an optimal sequential algorithm scales as $N+N^2$.

1 For first parallel algorithm: $T = N + N^2/P$

This algorithm partitions the computationally demanding $O(N^2)$ component of the algorithm but replicates the O(N) component on every processor. There are no other sources of overhead.

2 For the second parallel algorithm: $T = (N+N^2)/P + 100$

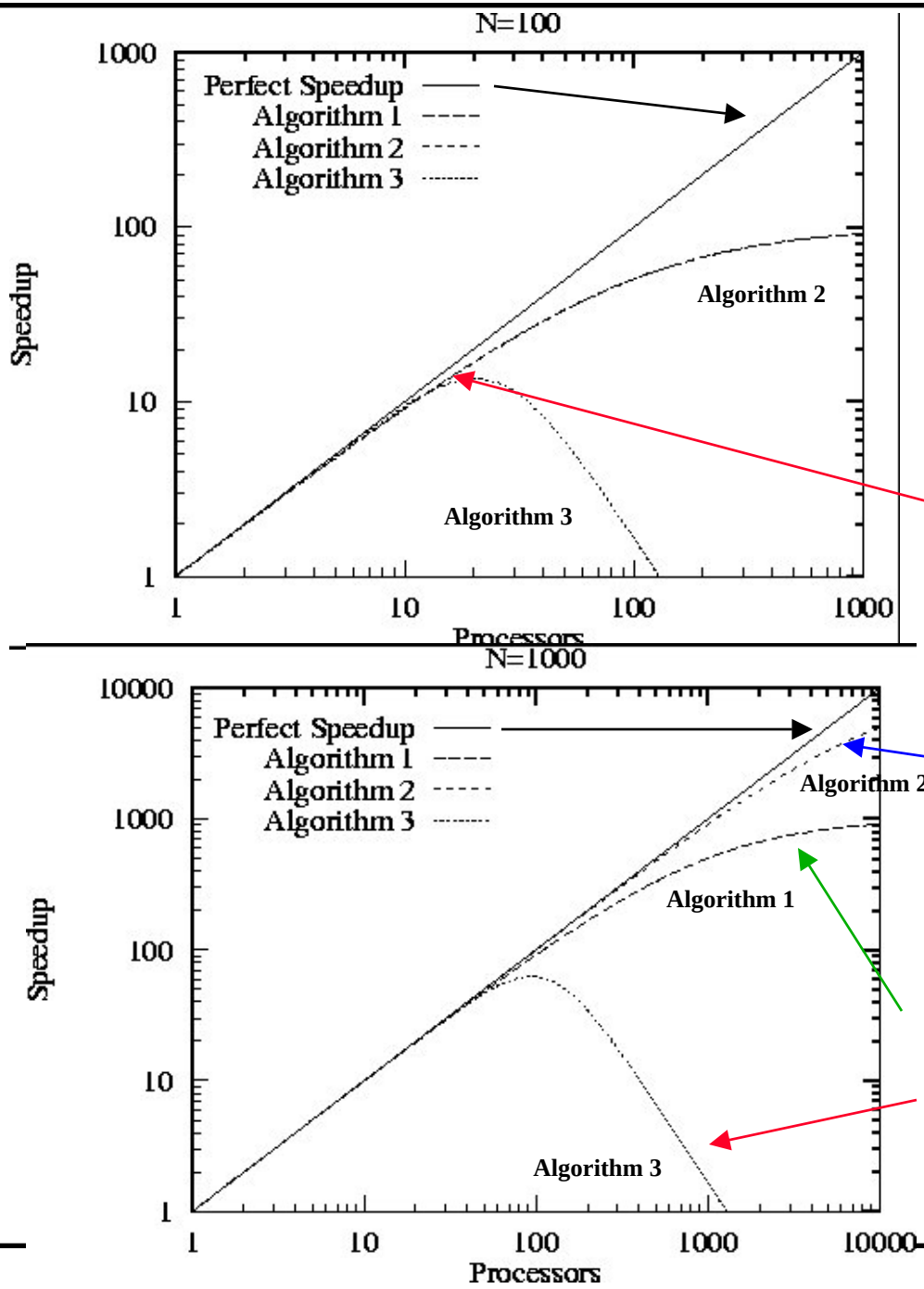
This algorithm optimally divides all the computation among all processors but introduces an additional cost of 100.

3 For the third parallel algorithm: $T = (N+N^2)/P + 0.6P^2$

This algorithm also partitions all the computation optimally but introduces an additional cost of $0.6P^2$.

- All three algorithms achieve a speedup of about 10.8 when P = 12 and N=100 . However, they behave differently in other situations as shown next.
- With N=100 , all three algorithms perform poorly for larger P , although Algorithm (3) does noticeably worse than the other two.
- When N=1000 , Algorithm (2) is much better than Algorithm (1) for larger P .

Parallel Performance Example (continued)



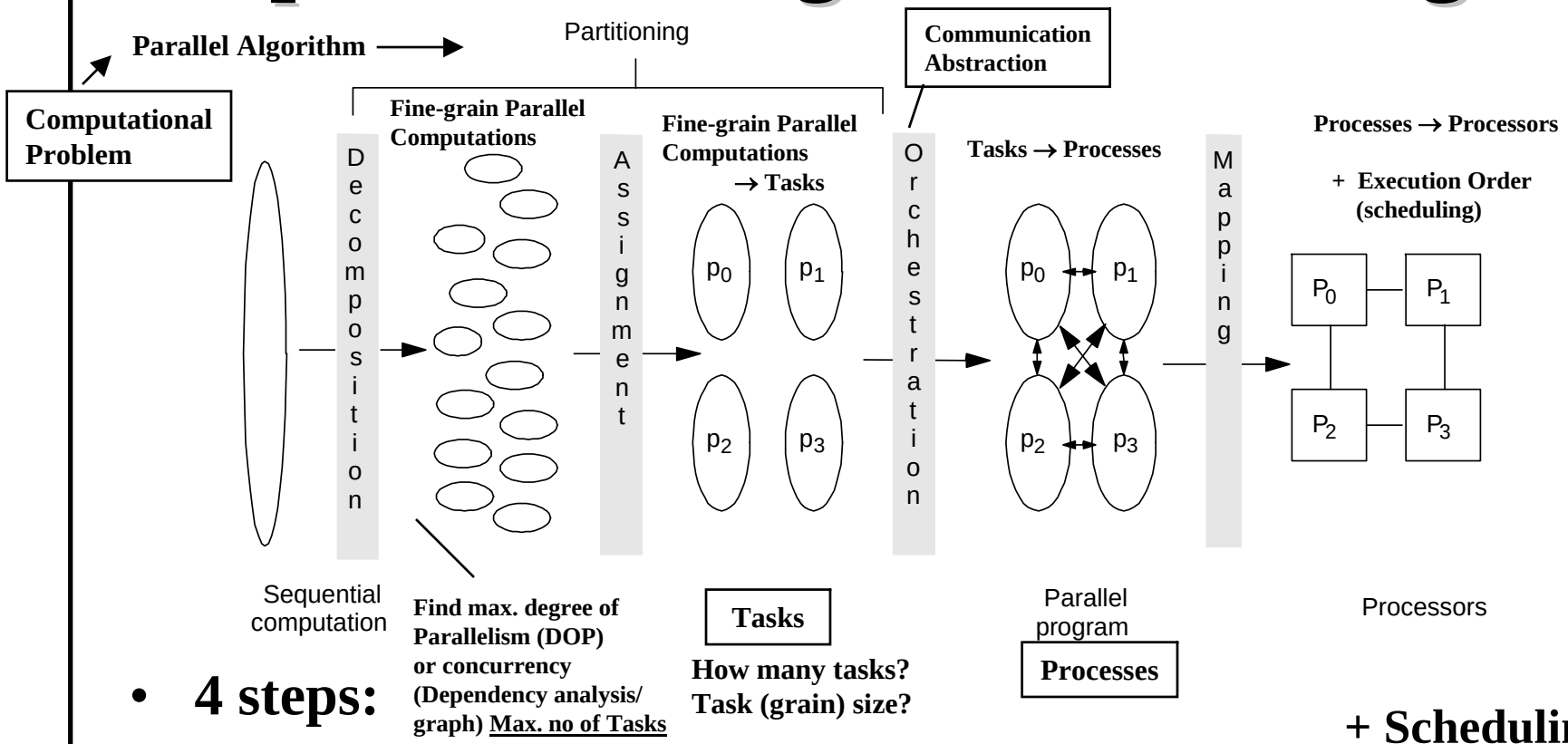
All algorithms achieve:
Speedup = 10.8 when P = 12 and N=100

N= Problem Size
P = Number of processors

N=1000 , Algorithm (2) performs much better than Algorithm (1) for larger P .

Algorithm 1: $T = N + N^2/P$
Algorithm 2: $T = (N+N^2)/P + 100$
Algorithm 3: $T = (N+N^2)/P + 0.6P^2$

Steps in Creating a Parallel Program



- **4 steps:**

+ Scheduling

Decomposition, Assignment, Orchestration, Mapping

- **Done by programmer or system software (compiler, runtime, ...)**
- **Issues are the same, so assume programmer does it all explicitly**

Partitioning: Decomposition & Assignment

Decomposition

- **Break up computation into maximum number of small concurrent tasks that can be combined into fewer/larger tasks in assignment step:**
  - **Tasks may become available dynamically.**
  - **No. of available tasks may vary with time.**
  - **Together with assignment, also called *partitioning.***

Grain (task) size Problem (Assignment)

i.e. identify concurrency (dependency analysis) and decide level at which to exploit it.

i.e Find maximum software concurrency or parallelism (Decomposition)

Assignment

- **Grain-size problem:**
  - **To determine the number and size of grains or tasks in a parallel program.**
  - **Problem and machine-dependent.**
  - **Solutions involve tradeoffs between parallelism, communication and scheduling/synchronization overheads.**
- **Grain packing:**
  - **To combine multiple fine-grain nodes (parallel computations) into a coarse grain node (task) to reduce communication delays and overall scheduling overhead.**

Goal: **Enough tasks to keep processors busy, but not too many (too much overheads)**

- **No. of tasks available at a time is upper bound on achievable speedup**

Assignment

Fine-Grain Parallel Computations → Tasks

- Specifying mechanisms to divide work up among tasks/processes:
  - Together with decomposition, also called ***partitioning***.
  - **Balance workload, reduce communication and management cost**
    - May involve duplicating computation to reduce communication cost.

To Maximize Speedup

- **Partitioning problem:**
  - To partition a program into parallel tasks to give the shortest possible execution time on a specific parallel architecture.
    - **Determine size and number of tasks in parallel program**
- Structured approaches usually work well:
  - Code inspection (parallel loops) or understanding of application.
  - Well-known heuristics.
  - *Static* versus *dynamic* assignment.
- As programmers, we worry about partitioning first:
  - *Usually* independent of architecture or programming model.
  - But cost and complexity of using primitives may affect decisions.

Communication/Synchronization Primitives used in orchestration

Orchestration

Tasks → Processes

- For a given parallel programming environment that realizes a parallel programming model, orchestration includes:
  - Naming data.
  - Structuring communication (using communication primitives)
  - Synchronization (ordering using synchronization primitives).
  - Organizing data structures and scheduling tasks temporally.

Execution order

- **Goals**

overheads

  - Reduce cost of communication and synchronization as seen by processors
  - Preserve locality of data reference (includes data structure organization)
  - Schedule tasks to satisfy dependencies as early as possible
  - Reduce overhead of parallelism management.

- **Closest to architecture (and programming model & language).**
  - Choices depend a lot on communication abstraction, efficiency of primitives.
  - Architects should provide appropriate primitives efficiently.

Mapping/Scheduling

Processes → Processors
+ Execution Order (scheduling)

- **Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs.**
- **Mapping can be specified statically or determined at runtime by load-balancing algorithms (dynamic scheduling).**
- **Two aspects of mapping:**
  - **Which processes will run on the same processor, if necessary**
  - **Which process runs on which particular processor**
    - **mapping to a network topology/account for NUMA**
- **One extreme: *space-sharing***
  - **Machine divided into subsets, only one app at a time in a subset**
  - **Processes can be pinned to processors, or left to OS.**
- **Another extreme: complete resource management control to OS**
  - **OS uses the performance techniques we will discuss later.**
- **Real world is between the two.**
  - **User specifies desires in some aspects, system may ignore**

May also involve duplicating tasks to reduce communication costs

Program Partitioning Example

Example 2.4 page 64
Fig 2.6 page 65
Fig 2.7 page 66
In Advanced Computer Architecture, Hwang
(see handout)

Static Multiprocessor Scheduling

Dynamic multiprocessor scheduling is an NP-hard problem.

Node Duplication: to eliminate idle time and communication delays, some nodes may be duplicated in more than one processor.

Fig. 2.8 page 67

Example: 2.5 page 68
In Advanced Computer Architecture, Hwang
(see handout)

Table 2.1 Steps in the Parallelization Process and Their Goals

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no ?	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchronization cost as seen by the processor Reduce serialization at shared resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary Exploit locality in network topology

Determine size and number of tasks

Tasks → Processes

Processes → Processors

+ Execution Order (scheduling)