Parallel Programs

- **Conditions of Parallelism:**
  - **Data Dependence**
  - **Control Dependence**
  - **Resource Dependence**
  - **Bernstein's Conditions**
- **Asymptotic Notations for Algorithm 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**
- **Example Motivating Problems With high levels of concurrency**
- **Limited 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**
  - **Static Multiprocessor Scheduling Example**

Conditions of Parallelism: Data Dependence

1 **True Data or Flow Dependence: A statement S2 is data dependent on statement 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**

denoted by S1 ¾® S2

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

denoted by S1 +¾® S2

3 **Output dependence: Two statements are output dependent if they produce the same output variable**

denoted by S1 o—® S2

Conditions of Parallelism: Data Dependence

4 **I/O dependence: Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file is referenced by both I/O statements.**

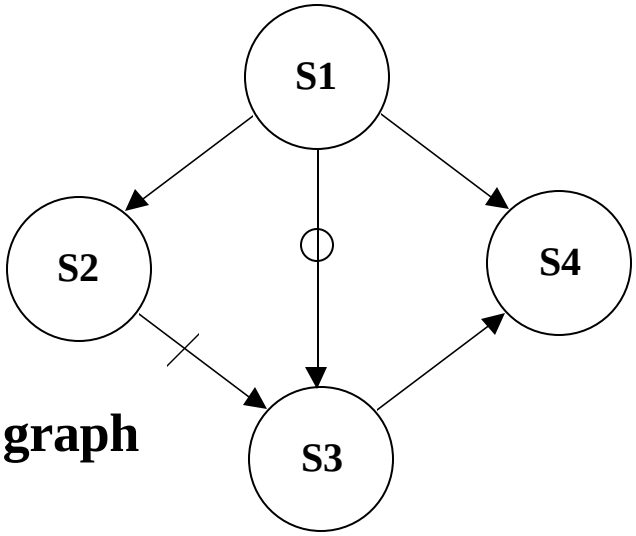
5 **Unknown dependence:**

- **Subscript of a variable is subscribed (indirect addressing)**
- **The subscript does not contain the loop index.**
- **A variable appears more than once with subscripts having different coefficients of the loop variable.**
- **The subscript is nonlinear in the loop index variable.**

Data and I/O Dependence: Examples

A -

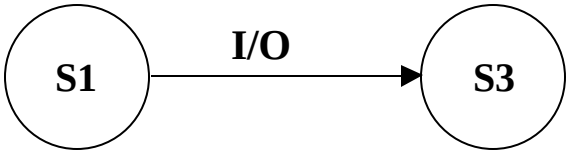
S1:	**Load R1,A**
S2:	**Add R2, R1**
S3:	**Move R1, R3**
S4:	**Store B, R1**



Dependence graph

B -

S1:	**Read (4),A(I)**	**/Read array A from tape unit 4/**
S2:	**Rewind (4)**	**/Rewind tape unit 4/**
S3:	**Write (4), B(I)**	**/Write array B into tape unit 4/**
S4:	**Rewind (4)**	**/Rewind tape unit 4/**



I/O dependence caused by accessing the same file by the read and write statements

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 including functional units (integer, floating point), memory areas, among parallel tasks.**
- **Bernstein's Conditions:**

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 in program order and
  - Instructions are in program order
  - Each instruction requires one step to execute
  - Two adders are available

$P_1 : C = D \times E$

$P_2 : M = G + C$

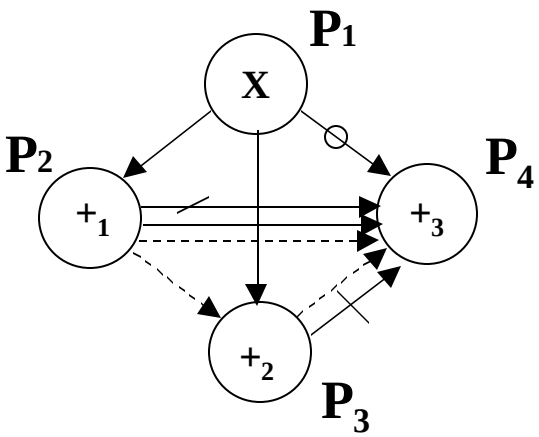
$P_3 : A = B + C$

$P_4 : C = L + M$

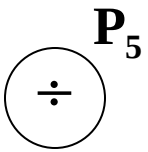
$P_5 : F = G \div E$

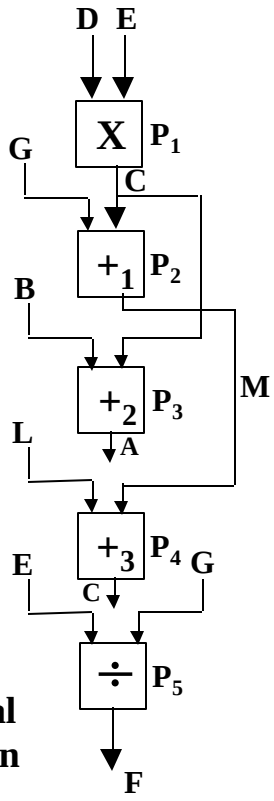
Using Bernstein's Conditions after checking statement pairs:

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

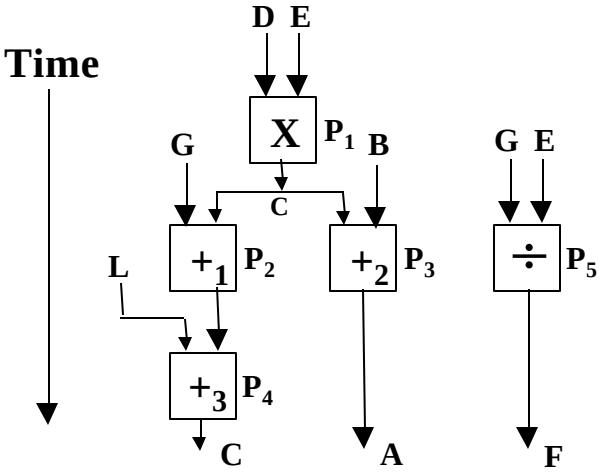


Dependence graph:
Data dependence (solid lines)
Resource dependence (dashed lines)





Sequential execution



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

Asymptotic Notations for Algorithm Analysis

Asymptotic analysis of computing time of an algorithm f(n) ignores constant execution factors and concentrates on determining the order of magnitude of algorithm performance.

- **Upper bound:**

  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) 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

- **Lower bound:**

  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)

- **Tight bound:**

  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)

The Growth Rate of Common Computing 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

Theoretical Models of Parallel Computers

- **Parallel Random-Access Machine (PRAM):**
  - ***n*** **processor, global shared memory model.**
  - **Models idealized parallel computers with zero synchronization or memory access overhead.**
  - **Utilized 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.**
  - **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.**

Example: sum algorithm on P processor PRAM

- **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 ³ 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 £ 2^(k-h)) then
           Set B(s): = B(2s -1 ) + B(2s)}
3. if (s = 1) then set S: = B(1)
end
```

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:

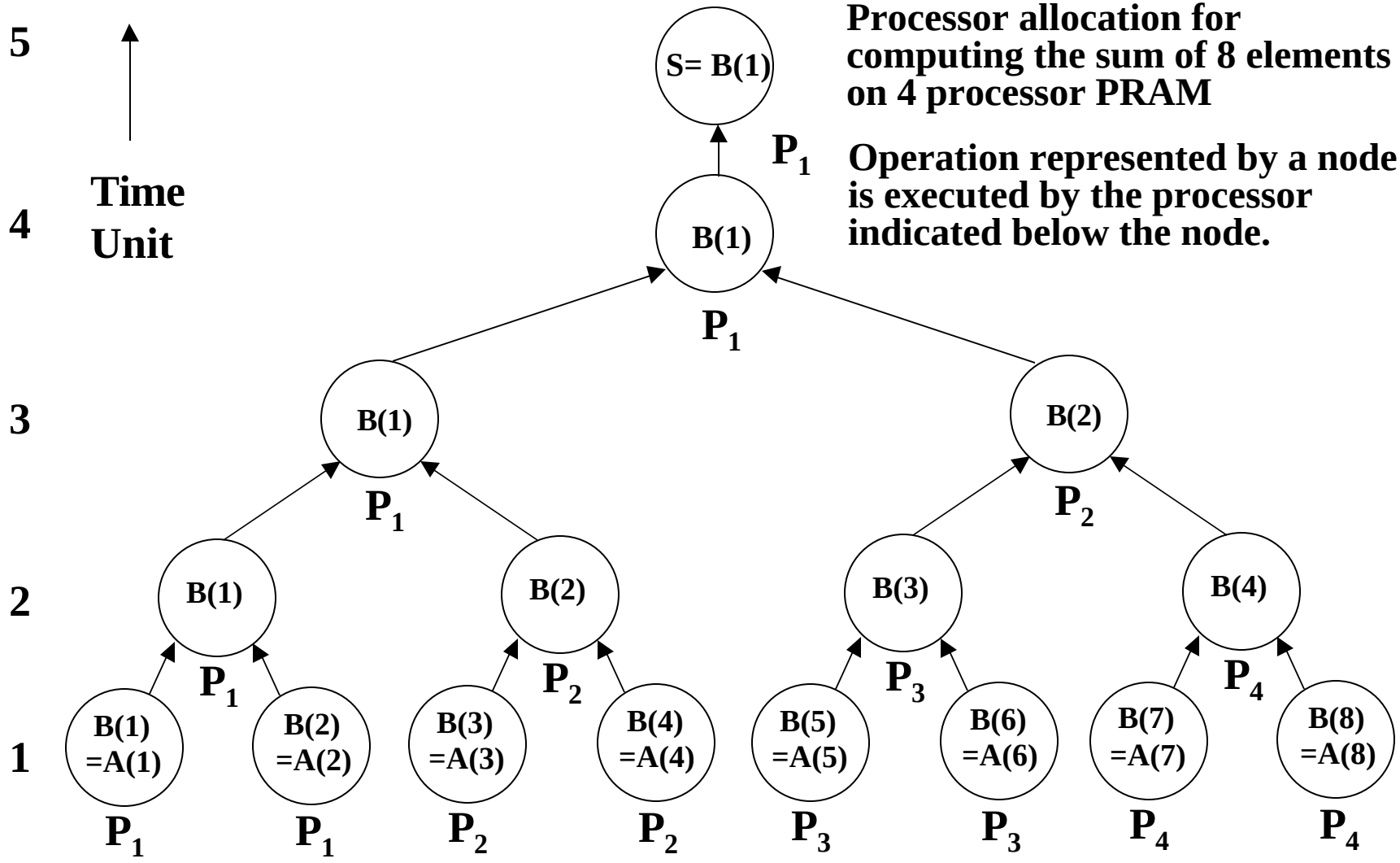
$$T_p(n) = O\left(\frac{n}{p} + \sum_{h=1}^{\log n} \left\lceil \frac{n}{2^h p} \right\rceil\right) = O\left(\frac{n}{p} + \log n\right)$$

Example: Sum Algorithm on P Processor PRAM

For $n = 8$ $p = 4$

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.



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, concentrating on the structural properties of the problem.**
- **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 network algorithms can be directly derived from PRAM algorithms.**
- **It is possible to incorporate synchronization and communication into the shared-memory PRAM model.**

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.**

Network Model of Message-Passing Multicomputers

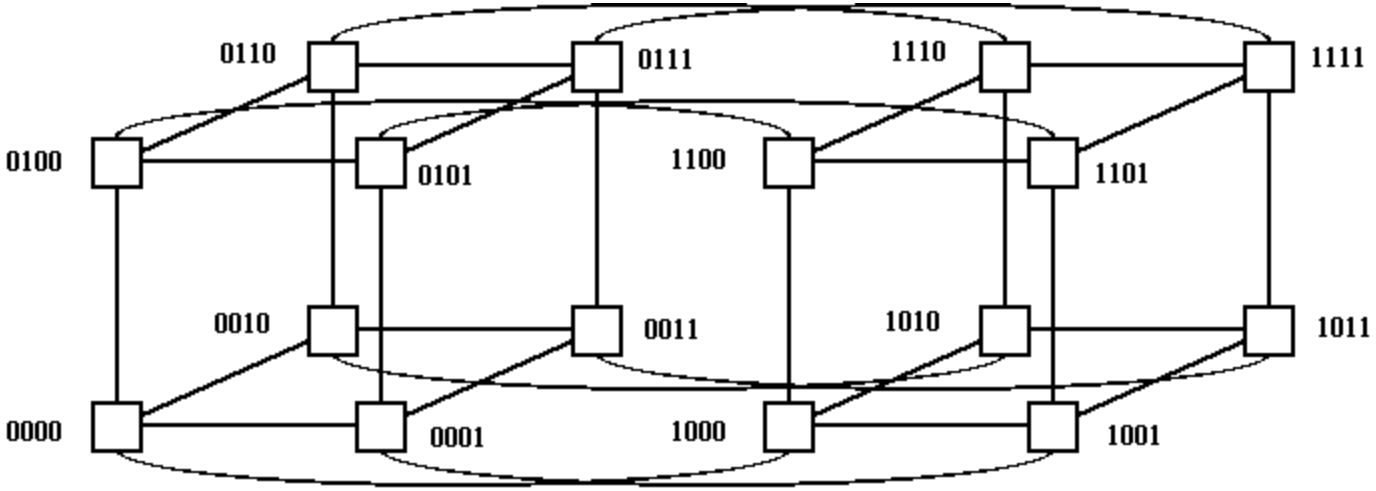
- A network of processors can viewed as a graph G (N,E)
  - Each node i Î N represents a processor
  - Each edge (i,j) Î E represents a two-way communication link between processors i and j.
  - Each processor is assumed to have its own local memory.
  - No shared memory is available.
  - Operation is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

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.
  - The maximum degree of any node in G
- 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
  - A ring is a linear array of processors where processors P_1 and P_p are directly connected.

A Four-Dimensional Hypercube

- **Two processors are connected if their binary indices differ in one bit position.**



Example: Asynchronous Matrix Vector Product on a Ring

- **Input:**
  - **n x 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 x 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**
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

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

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. The best solution may differ from that suggested by existing sequential algorithms.**

One must:

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

Main goal: Speedup (plus low programming effort and resource needs)

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

For a fixed problem:

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

Some Important Concepts

***Task*:**

- **Arbitrary piece of undecomposed work in parallel computation**
- **Executed sequentially on a single processor; concurrency is only across tasks**
- **E.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace**
- **Fine-grained versus coarse-grained tasks**

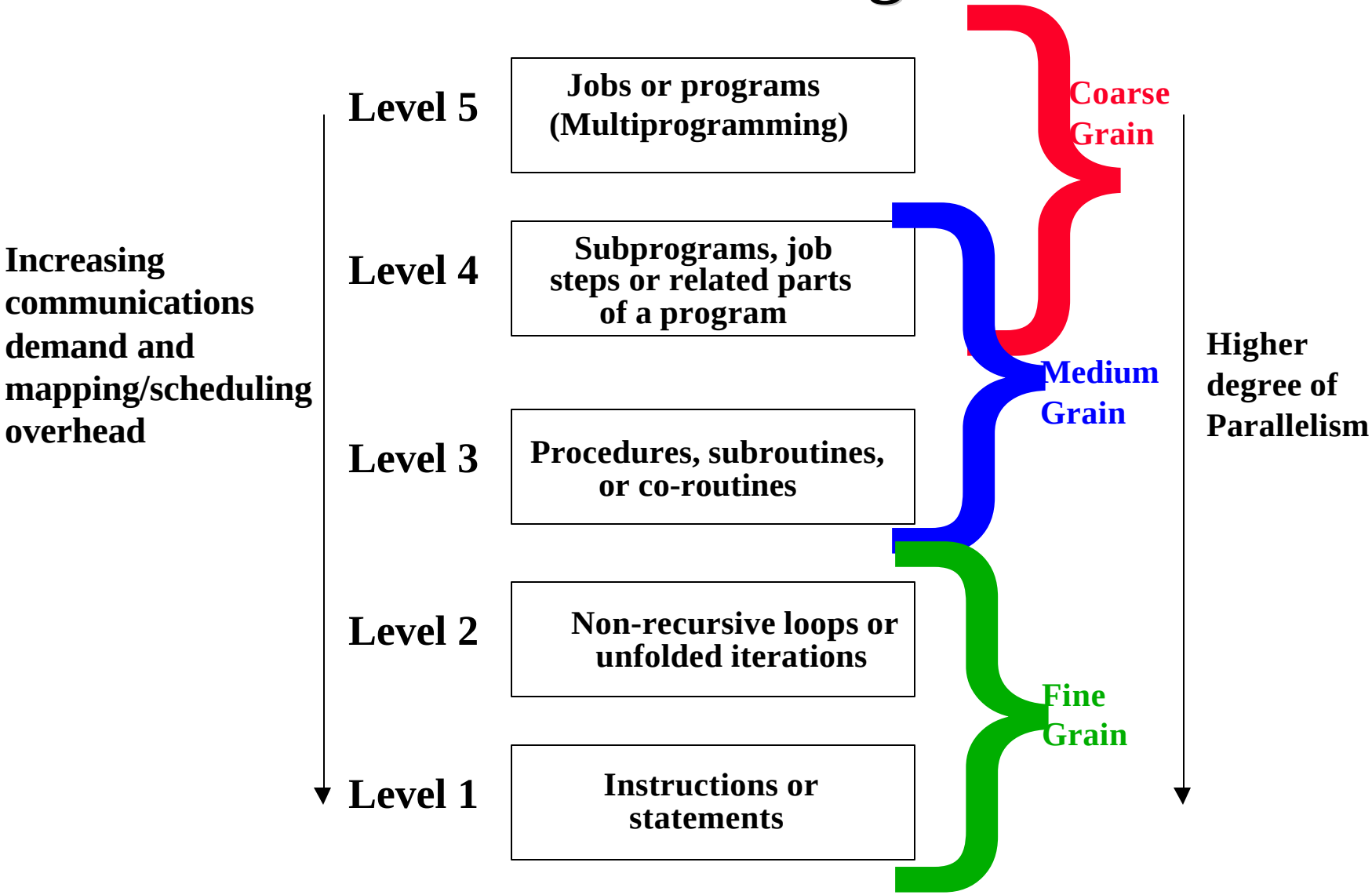
***Process (thread)*:**

- **Abstract entity that performs the tasks assigned to processes**
- **Processes communicate and synchronize to perform their tasks**

***Processor*:**

- **Physical engine on which process executes**
- **Processes virtualize machine to programmer**
  - **first write program in terms of processes, then map to processors**

Levels of Parallelism in Program Execution



Hardware and Software Parallelism

- **Hardware parallelism:**
  - **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.**
- **Software parallelism:**
  - **Defined by the control and data dependence of programs.**
  - **Revealed in program profiling or program flow graph.**
  - **A function of algorithm, programming style and compiler optimization.**

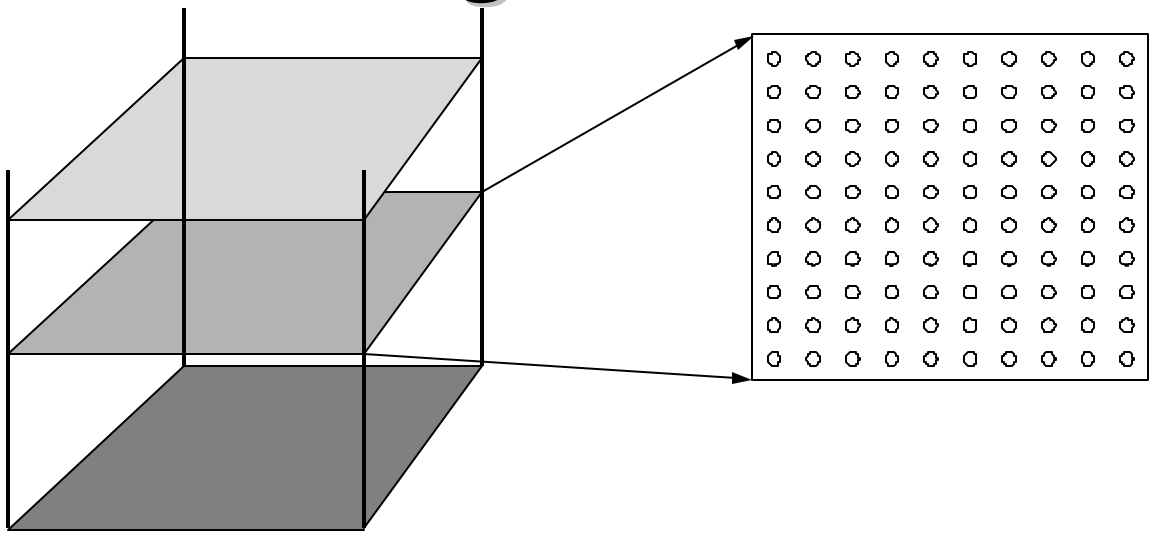
Computational Parallelism and Grain Size

- **Grain size (granularity) is a measure of the amount of computation involved in a task in parallel computation :**
  - **Instruction Level:**
    - **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:**
    - **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-size grain; task, 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:**
  - **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.**

Example Motivating Problems:
Simulating Ocean Currents



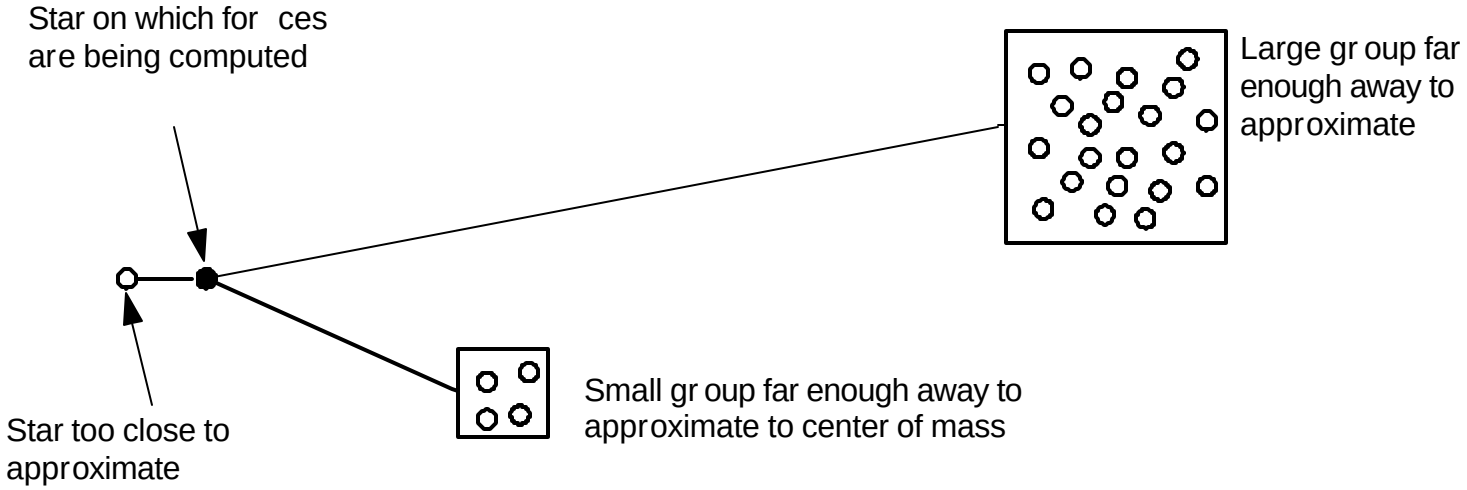
(a) Cross sections (b) Spatial discretization of a cross section

- **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**
- **Concurrency across and within grid computations**

Example Motivating Problems:

Simulating Galaxy Evolution

- **Simulate the interactions of many stars evolving over time**
- **Computing forces is expensive**
- **$O(n^2)$ brute force approach**
- **Hierarchical Methods take advantage of force law: $G \frac{m_1 m_2}{r^2}$**



- Many time-steps, plenty of concurrency across stars within one

Example Motivating Problems: Rendering Scenes by Ray Tracing

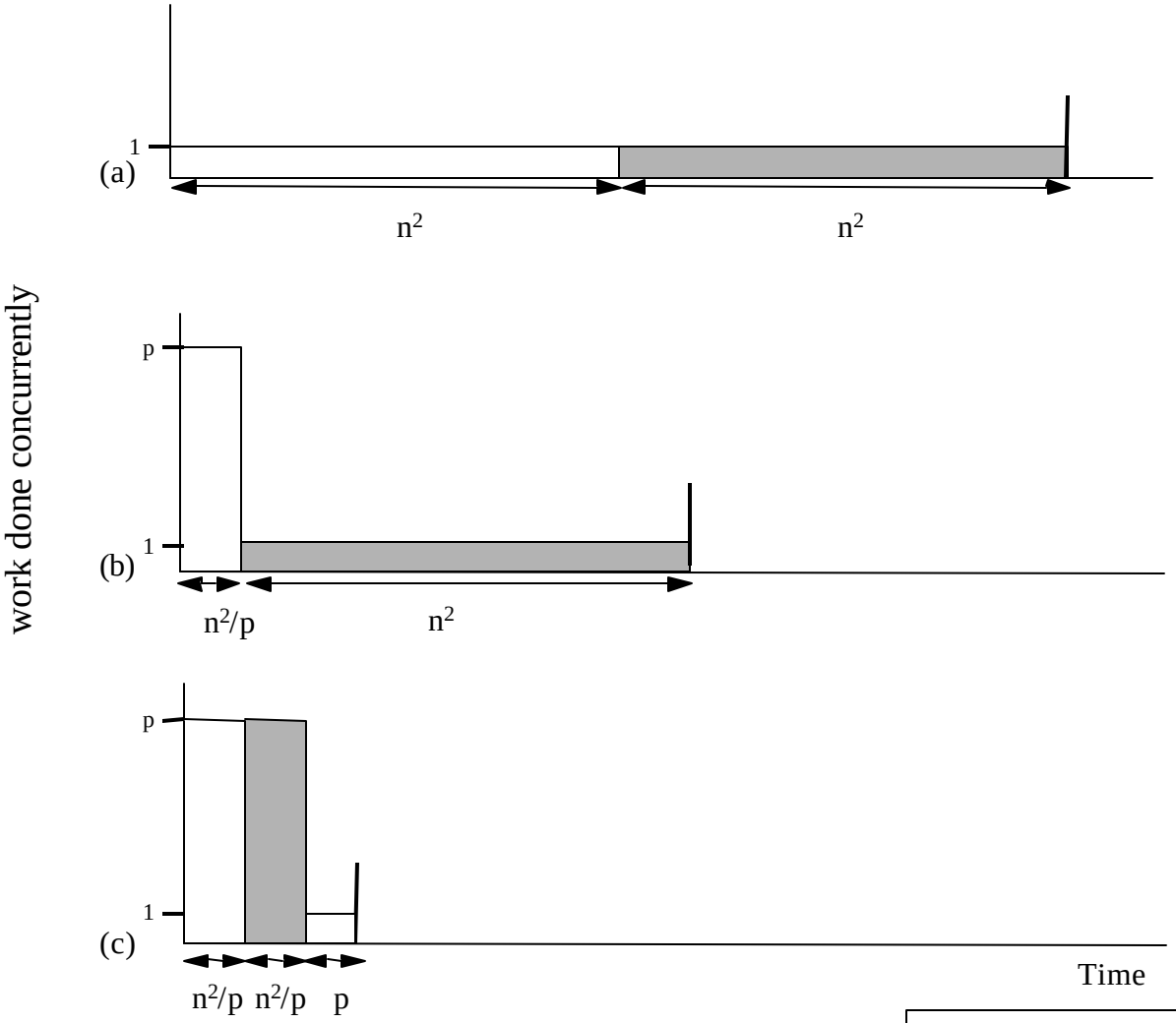
- Shoot rays into scene through pixels in image plane
- Follow their paths
  - They bounce around as they strike objects
  - They generate new rays: ray tree per input ray
- Result is color and opacity for that pixel
- Parallelism across rays

- All above case studies have abundant concurrency

Limited Concurrency: Amdahl's Law

- Most fundamental limitation on parallel speedup.
- If fraction *s* of seqeuential execution is inherently serial, speedup <= *1/s*
- Example: 2-phase calculation
  - sweep over *n*-by-*n* grid and do some independent computation
  - sweep again and add each value to global sum
- Time for first phase = n^2/p
- Second phase serialized at global variable, so time = n^2
- Speedup <= $\frac{2n^2}{\frac{n^2}{p} + n^2}$ or at most 2
- 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 at best $\frac{2n^2}{2n^2 + p^2}$

Amdahl's Law Example: A Pictorial Depiction



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:
  - given maximum parallelism = *m*
  - *n* homogeneous processors
  - computing capacity of a single processor Δ
  - Total amount of work W (instructions, computations):

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

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:

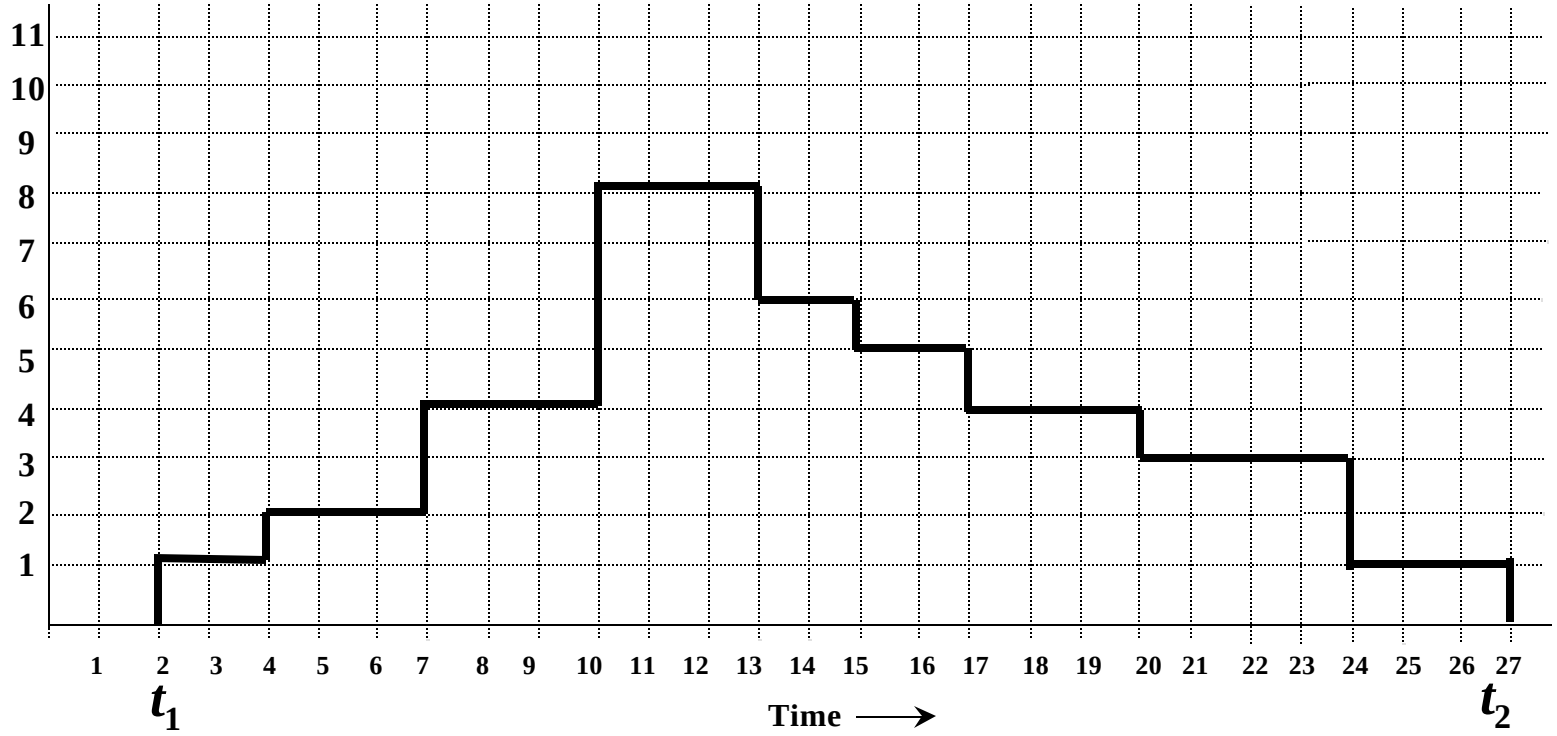
$$A = \frac{1}{t_2 - t_1} \int_{t1}^{t2} DOP(t)dt$$ In discrete form $$A = \left(\sum_{i=1}^{m} i.t_i \right) \Big/ \left(\sum_{i=1}^{m} t_i \right)$$

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

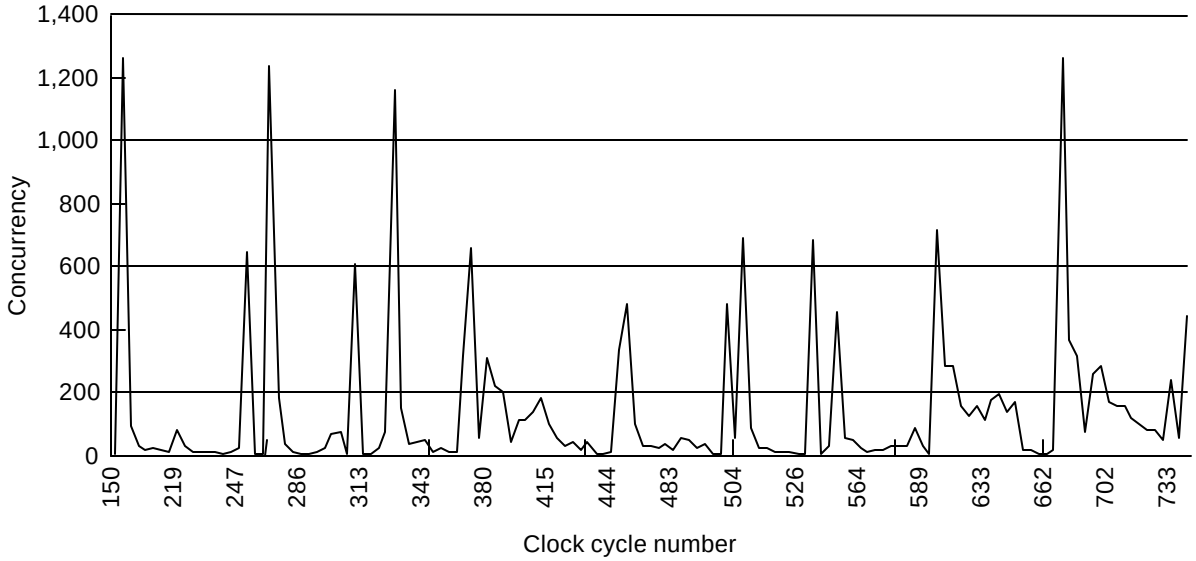
$$A = \left(\sum_{i=1}^{m} i.t_i\right) \Big/ \left(\sum_{i=1}^{m} t_i\right)$$

Degree of Parallelism (DOP)



Concurrency Profile & Speedup

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



- **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:** $\frac{1}{s + \frac{1-s}{p}}$
- **Amdahl's law applies to any overhead, not just limited concurrency.**

Parallel Performance Example

- The execution time T for three parallel programs is given in terms of processor count P and problem size N
- 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: $T = (N+N^2)/P + 100$

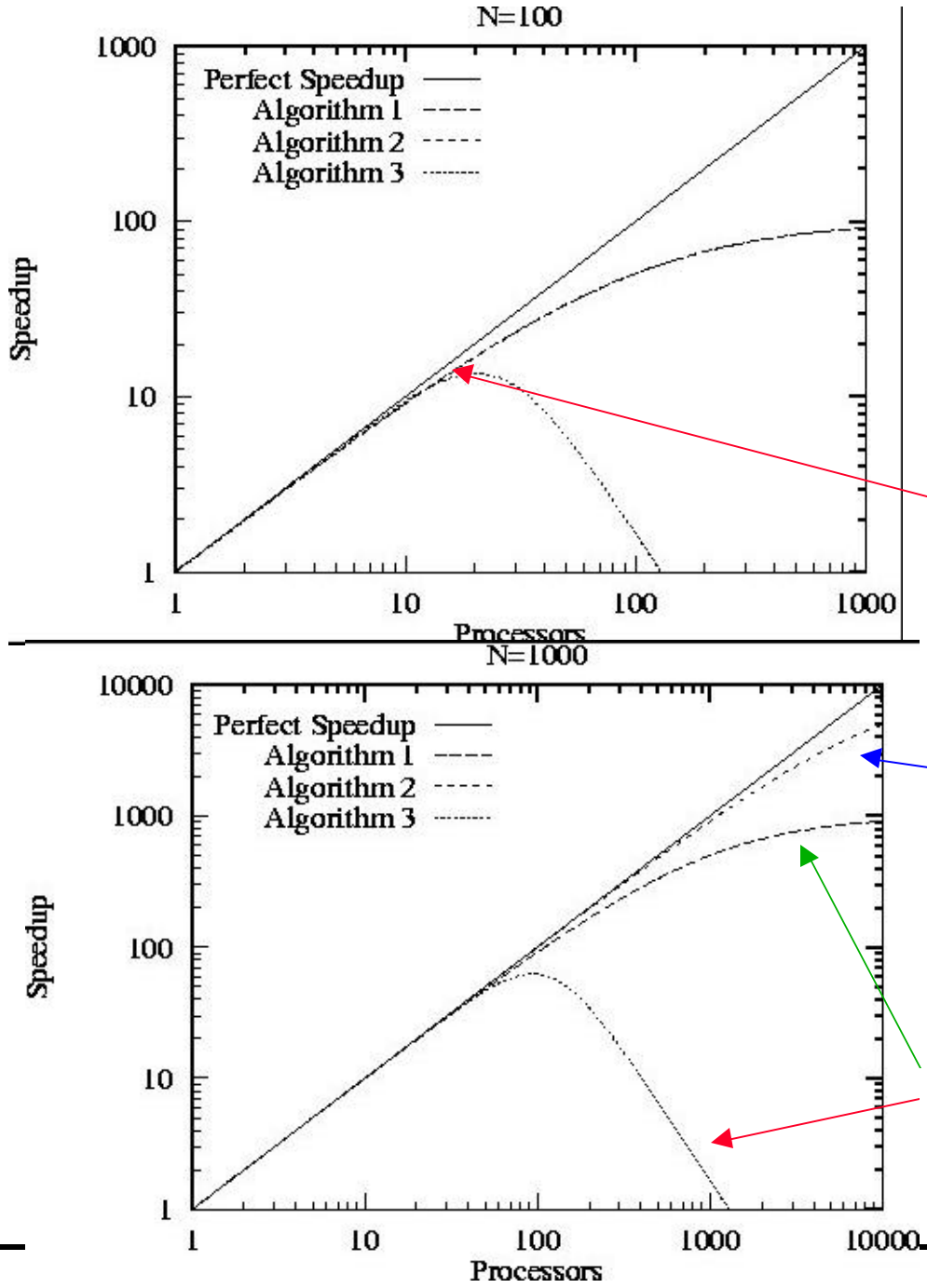
This algorithm optimally divides all the computation among all processors but introduces an additional cost of 100.

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)

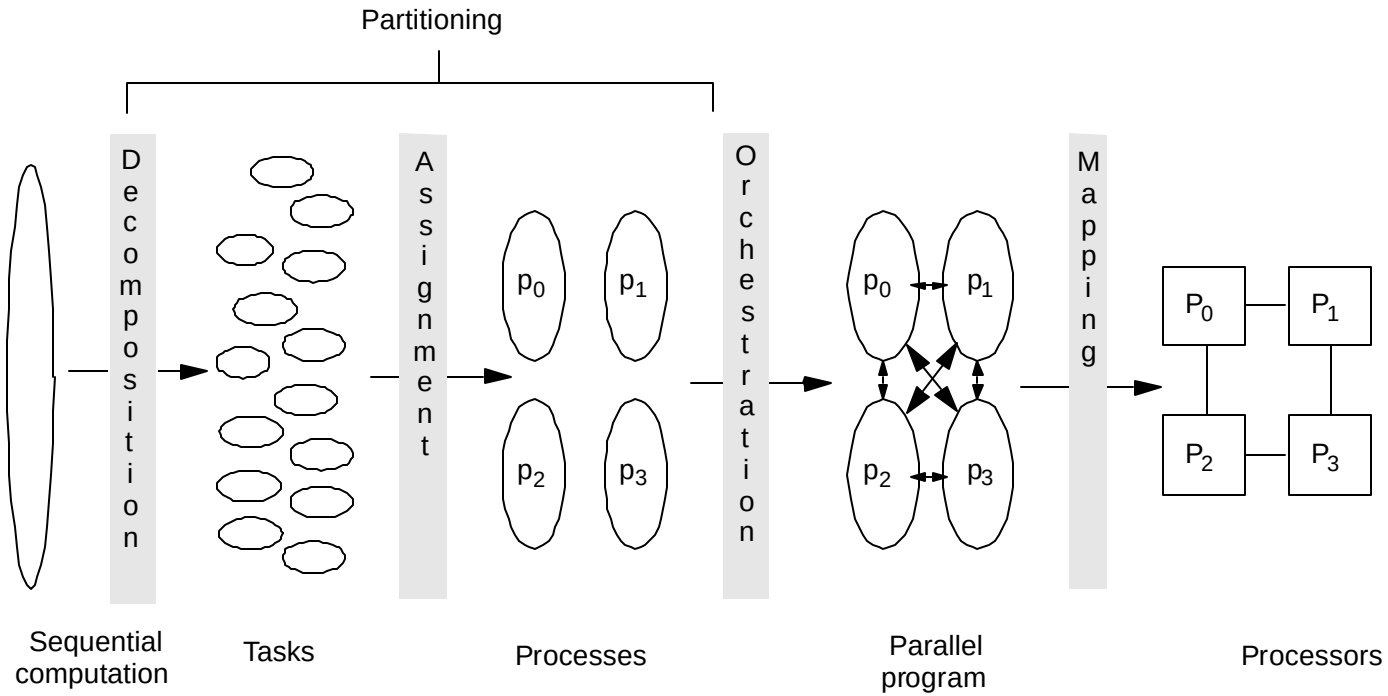


All algorithms achieve:
Speedup = 10.8 when P = 12 and N=100

N=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:**

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

Decomposition

- **Break up computation into concurrent tasks to be divided among processes:**
  - **Tasks may become available dynamically.**
  - **No. of available tasks may vary with time.**
  - **Together with assignment, also called *partitioning*.**

i.e. identify concurrency and decide level at which to exploit it.

- **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 overhead.**
- **Grain packing:**
  - **To combine multiple fine-grain nodes into a coarse grain node (task) to reduce communication delays and overall scheduling overhead.**

Goal: Enough tasks to keep processes busy, but not too many

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

Assignment

- **Specifying mechanisms to divide work up among processes:**
  - **Together with decomposition, also called *partitioning*.**
  - **Balance workload, reduce communication and management cost**
- **Partitioning problem:**
  - **To partition a program into parallel branches, modules to give the shortest possible execution on a specific parallel architecture.**
- **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.**

Orchestration

- Naming data.
- Structuring communication.
- Synchronization.
- Organizing data structures and scheduling tasks temporally.

- **Goals**
  - Reduce cost of communication and synch. as seen by processors
  - Reserve locality of data reference (incl. data structure organization)
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management

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

Mapping

- **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**
- **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**

Program Partitioning Example

Example 2.4 page 64
Fig 2.6 page 65
Fig 2.7 page 66
In Advanced Computer Architecture, Hwang

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

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