Multicore Computing
Lecture08 - Performance

SUNG KYUN KWAN
UNIVERSITY

남 범 석
bnam@skku.edu

Performance

- *Performance* is a measure of how well (in terms of latency, throughput, etc) computational requirements can be satisfied

- Performance is the *raison d'être (reason of existence)* for parallelism.
  - Parallel performance versus sequential performance
  - If the performance is not better, parallelism is not necessary

Parallel Performance Expectation

- Q: If each processor is rated at k MFLOPS and there are p processors, should we see k*p MFLOPS performance?
- Q: If it takes 100 seconds on 1 processor, shouldn't it take 10 seconds on 10 processors?

- True for Embarrassingly Parallel Computations
  - Can be obviously divided into completely independent parts that can be executed simultaneously
  - There is no interaction between separate processes
    - E.g.,) Monte Carlo Simulations
  - If it takes *T* time sequentially, there is the potential to achieve *T/P* time running in parallel with *P* processors

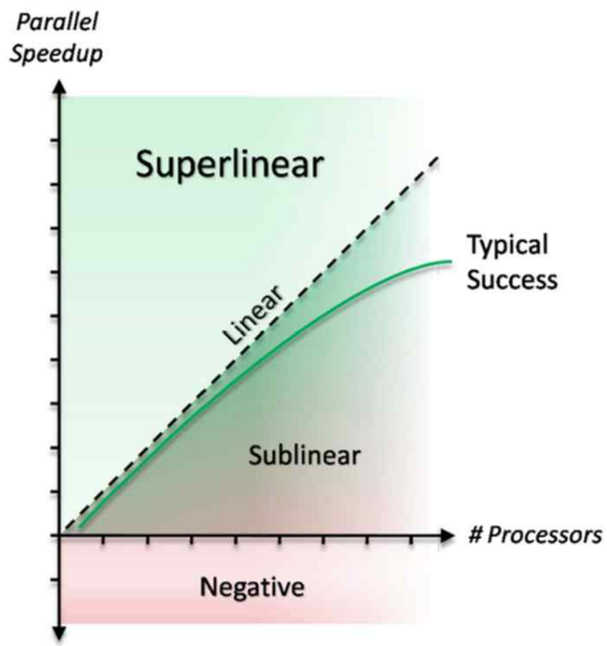
- False for numerous algorithms

Performance Metrics and Formulas

- T_1 is the execution time on a single processor
- T_p is the execution time on a p processor system

- $S(p)$ (S_p) is the *speedup*

$$S(p) = \frac{T_1}{T_p}$$



  - E.g) Theoretical Performance of Pairwise sum
    - $T_1 = O(n)$
    - $T_p = O(\log n)$
    - $S(p) = O(n/\log n)$

Performance Metrics and Formulas

- T_1 is the execution time on a single processor
- T_p is the execution time on a p processor system

- $E(p)$ (E_p) is the *efficiency*

$$Efficiency = \frac{S_p}{p} = \frac{T_s}{pT_p}$$

  - *E.g.) Pairwise sum*

$$E = O\left(\frac{n}{\log n} \cdot \frac{1}{n}\right) = O\left(\frac{1}{\log n}\right)$$

- Efficiency measures the fraction of time for which a processor is usefully utilized

Performance Metrics and Formulas

- T_1 is the execution time on a single processor
- T_p is the execution time on a p processor system

- *Cost(p)* (C_p) is the *cost*

$$Cost = p \times T_p$$

- *Cost-optimal parallel system solves a problem with a cost that matches the execution time of the fastest known sequential algorithm on a single processor.*

  - E.g.) Pairwise sum: $Cost = O(n \log n)$

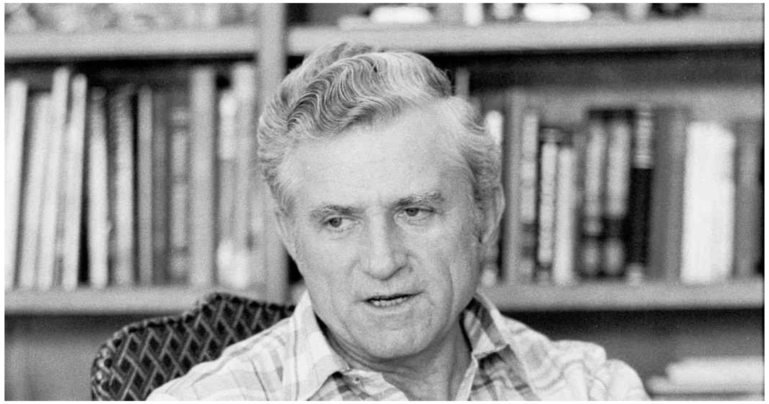
    Cost of the fastest sequential algorithm = O(n)

    Therefore, parallel pairwise sum is not cost-optimal.

Amdahl's Law

- Gene Amdahl
  - Chief architect of IBM's first mainframe series
  - found that there were stringent restrictions on how much of a speedup one could get for a given parallelized task.
  - These observations were called *Amdahl's Law* (1967)

- If F is the fraction of a calculation that is sequential, and (1-F) is the fraction that can be parallelized, then the maximum speed-up that can be achieved by using P processors is 1/(F+(1-F)/P).



Amdahl's Law

- f is the fraction of a program that is sequential
- $1\text{-}f$ is the fraction that can be parallelized
- Let T_1 be the execution time on 1 processor
- Let T_p be the execution time on p processors
- S_p is the *speedup*

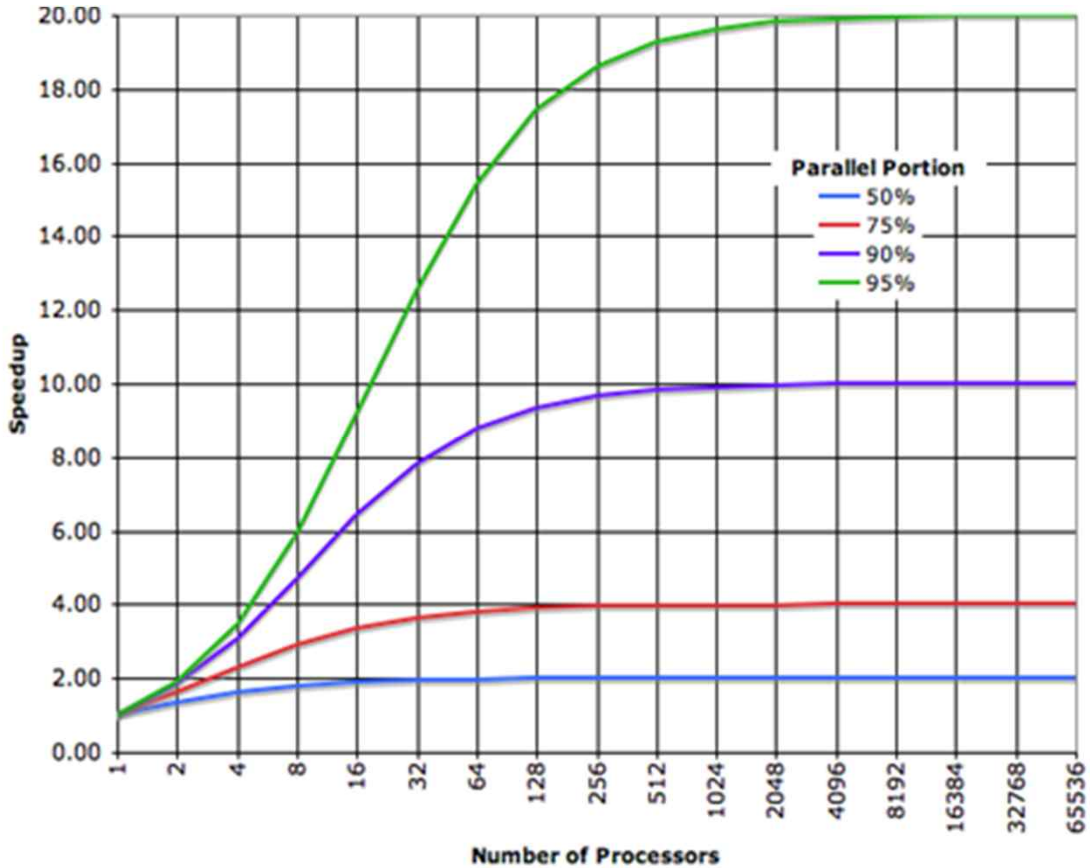
$$
\begin{aligned}
S_p &= T_1 / T_p \\
&= T_1 / (fT_1 + (1-f)T_1 / p)) \\
&= 1 / (f + (1-f)/p))
\end{aligned}
$$

- As $p \rightarrow \infty$

$$S_p = 1 / f$$

Amdahl's Law

- Speed up = 1/(F+(1-F)/P)
  - E.g.) 90% is parallel (i.e. 10% is sequential)
    → Speed-up on 5 processors = 1/(0.1+(1-0.1)/5) = about 3.6



Amdahl's Law and Scalability

- Scalability
  - Achieving performance proportional to the number of processors and the size of the problem

- What's the point of this?
  - Speedup is determined by sequential execution time, not by # of processors!!!
  - The performance is constrained by the slowest point.
  - Perfect efficiency is hard to achieve
    - Straggler Problem: Some processors will likely run out of work to do before others are finished
  - Amdahl was trying to argue in support of making single processor faster

Gustafson's Law (Scaled Speedup, 1988)

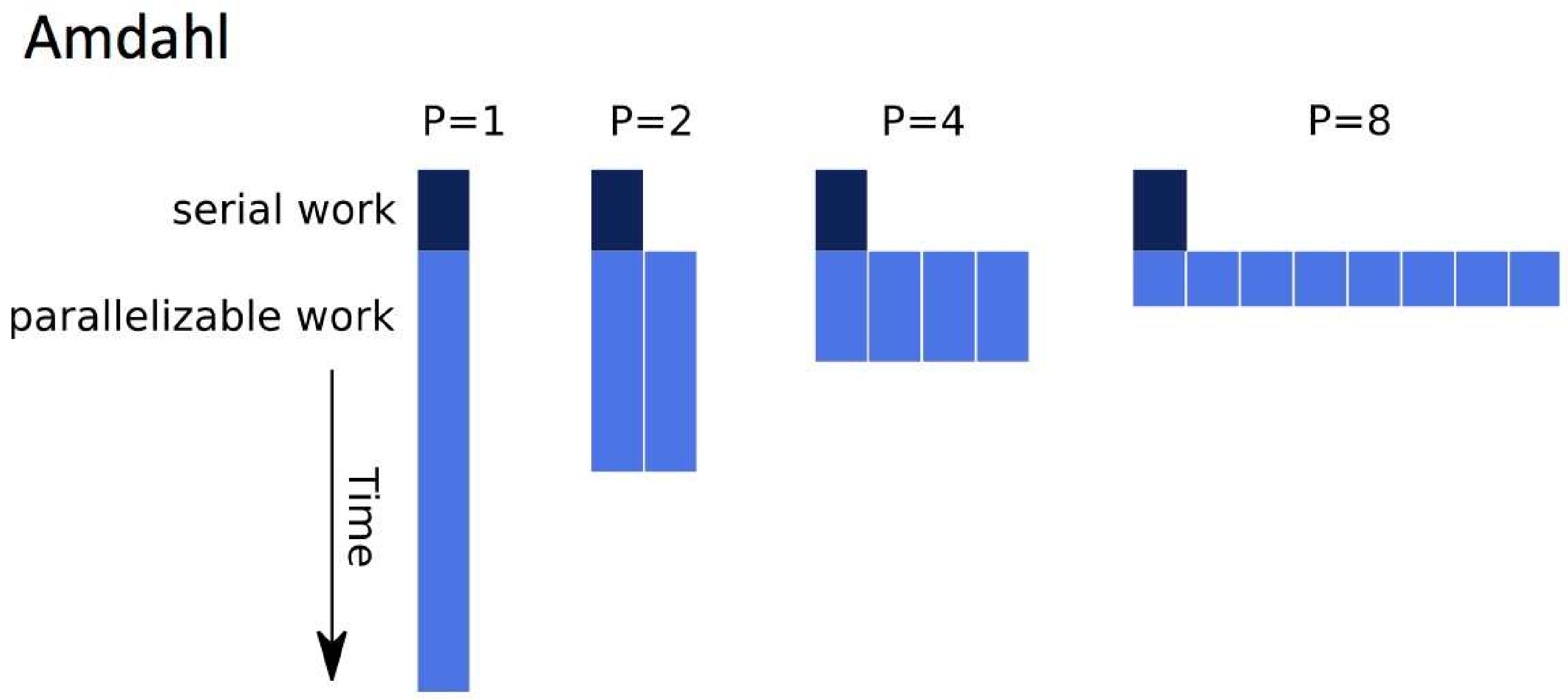
- Amdahl's Law is applied when the problem is fixed
- What about larger problems?
  - HPC Linpack
  - Constrain problem size by parallel time
- Assume parallel time is kept constant
  - $T_p = C = (f + (1-f)) * C$
  - f_{seq} is the fraction of T_p spent in sequential execution
  - f_{par} is the fraction of T_p spent in parallel execution
- What is the execution time on one processor?
  - Let C=1, then $T_s = f_{seq} + p(1 - f_{seq}) = 1 - f_{par} + p\ f_{par} = 1 + (p-1) f_{par}$
- What is the speedup in this case?
  - $S_p = T_s / T_p = T_s / 1 = f_{seq} + p(1 - f_{seq}) = 1 + (p-1) f_{par}$

Gustafson's Law and Scalability

- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
- When does Gustafson's Law apply?
  - When the problem size increases as the number of processors increases
  - Weak scaling ($S_p = 1 + (p-1)f_{par}$)
  - Speedup function includes the number of processors!!!
  - Can maintain or increase parallel efficiency as the problem scales

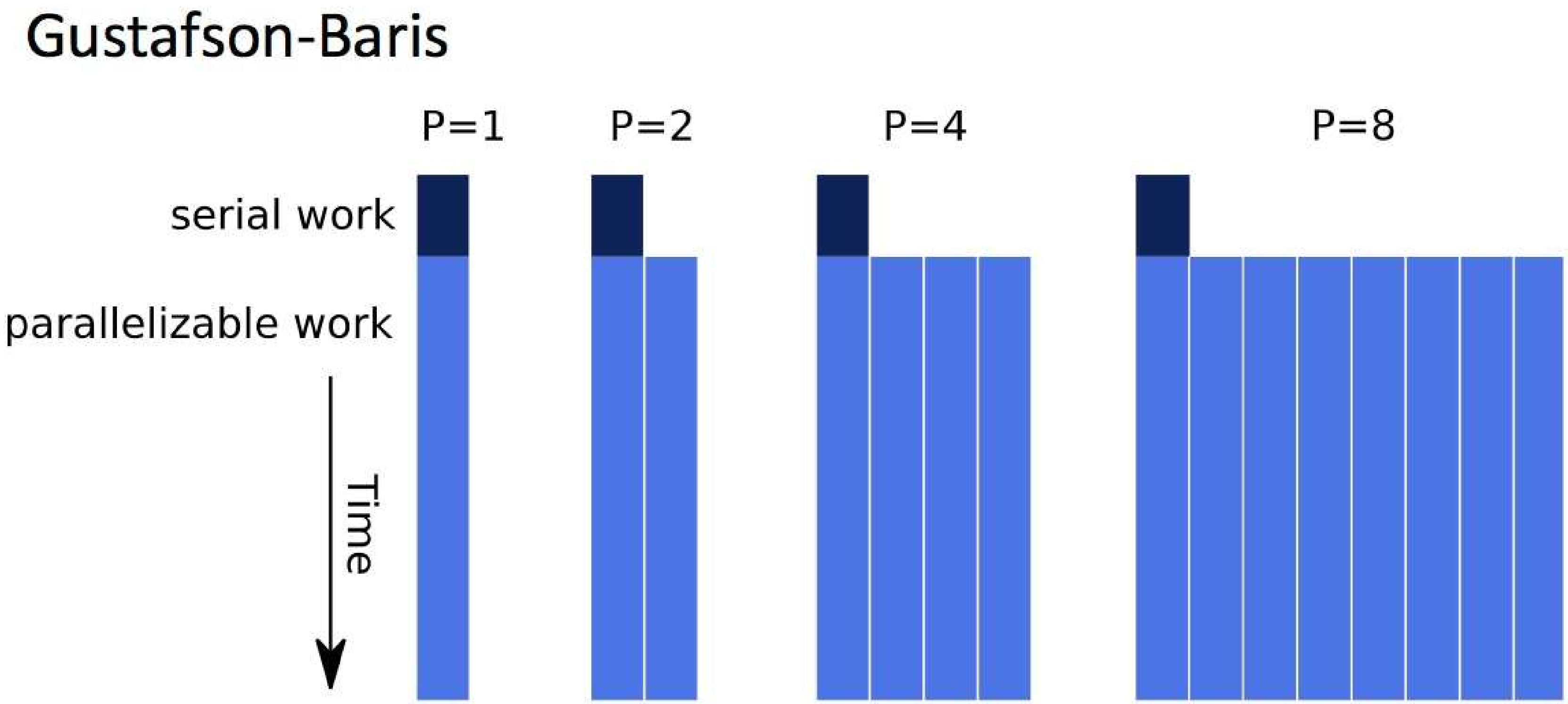
Weak vs Strong Scaling

- In the context of high performance computing, there are two common notions of scalability:
- The first is *strong scaling*, which is defined as how the solution time varies with the number of processors for a fixed *total* problem size.
- The second is *weak scaling*, which is defined as how the solution time varies with the number of processors for a fixed problem size *per processor*.



Source: Introduction to Parallel Computing, University of Oregon, IPCC

Amdahl versus Gustafson

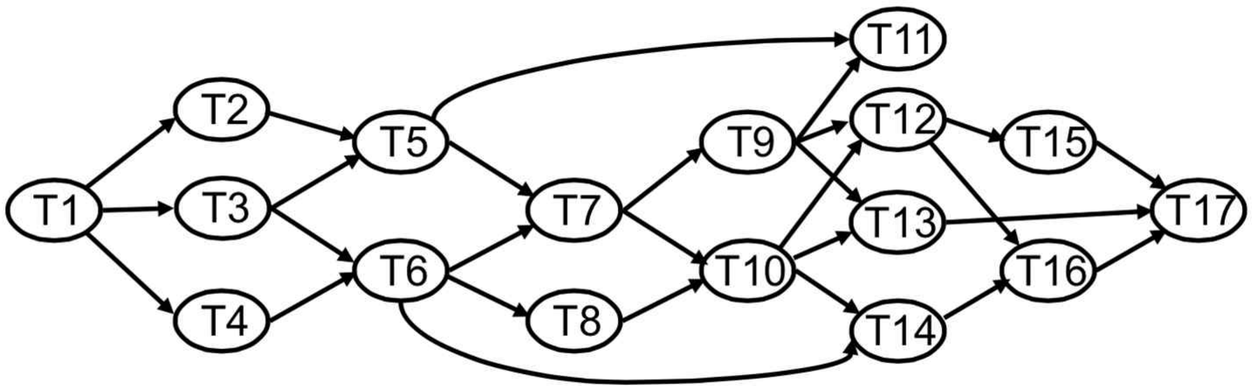


Source: Introduction to Parallel Computing, University of Oregon, IPCC

Work decomposition and Principles of Parallel Programming

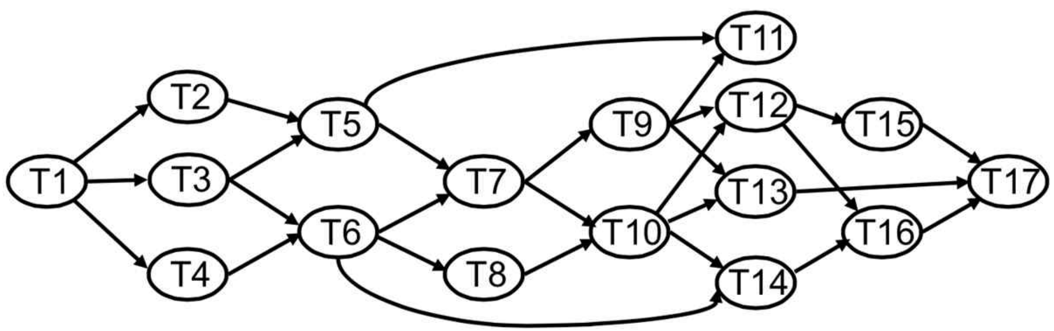
Work Decomposition

- The first step in developing a parallel algorithm
- Divide work into tasks that can run concurrently
- Many different ways of decomposition
- Tasks may be same, different, or even indeterminate sizes
- Tasks can be independent or dependent

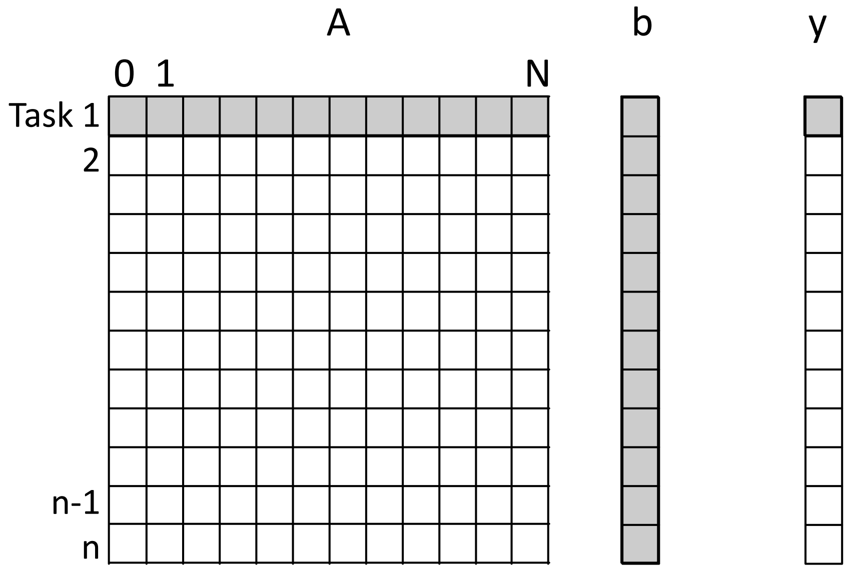


Task Dependency Graph

- In most cases, there are dependencies between different tasks
  - Certain tasks can only start once some other tasks have finished
    - Ex) producer-consumer relationship
- Task dependency can be drawn using directed acyclic graph (DAG)
  - Node: task
  - Weight of a node: size (load) of a task
  - Edge: control dependency between tasks



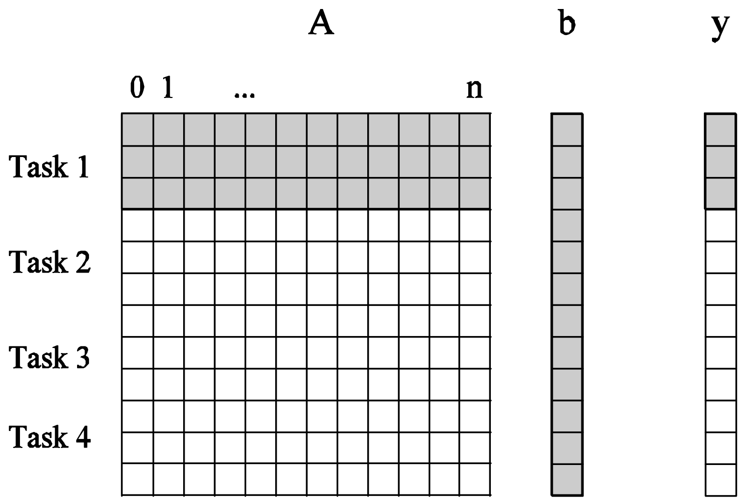
Example: Dense Matrix-Vector Multiplication



- Computing each element of output vector y is independent
- Easy to decompose → one task per element in y
- Observations
  - Tasks share b
  - No control dependency between tasks
  - Task size is uniform

Granularity of Task Decompositions

- Granularity: task size
  - The number of tasks determines its granularity
- Fine-grained decomposition
  - A large number of tasks
- Coarse-grained decomposition
  - A small number of tasks



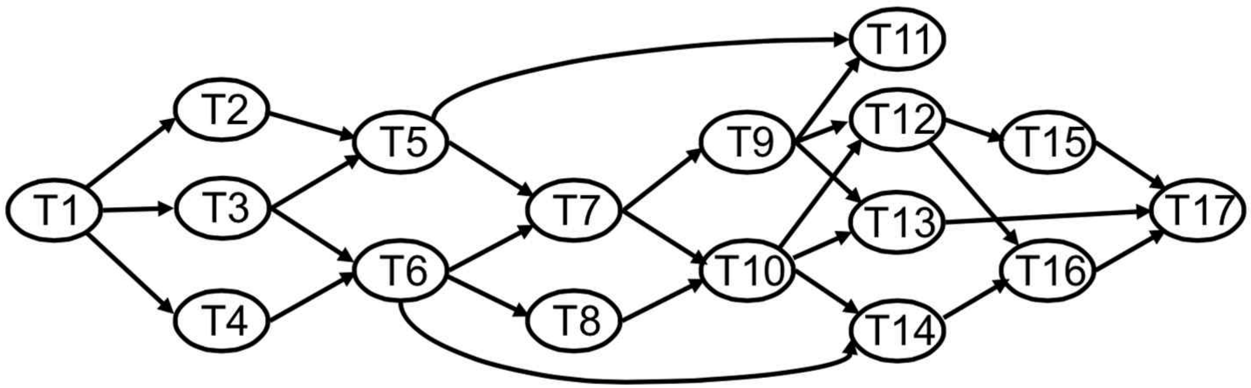
  - Fine-grained decomposition: task per element in y
  - Coarse-grained decomposition: task per 3 elements in y

Degree of Concurrency

- Definition: number of tasks that can run in parallel
- May change over program execution
- Two metrics
  - Maximum degree of concurrency
    - Largest number of concurrent tasks at any point of computation
  - Average degree of concurrency
    - Average number of concurrent tasks that can be executed concurrently
- Finer task granularity → larger degree of concurrency
- Coarser task granularity → smaller degree of concurrency

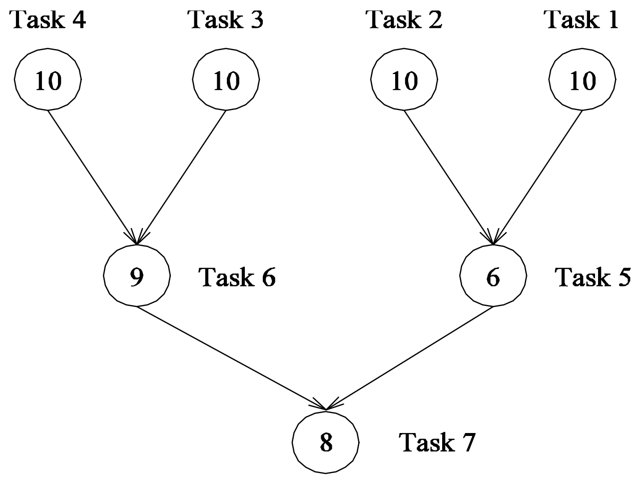
Critical Path Length

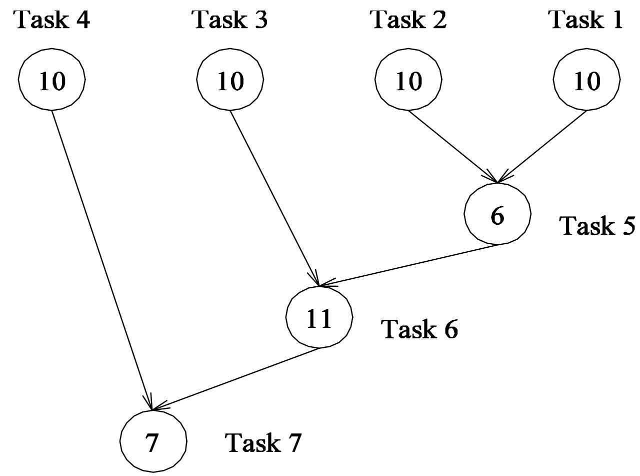
- Edges in task dependency graph serializes tasks
- Critical path: the longest weighted path in task dependency graph
- Critical path length bounds parallel execution time



Critical Path Length

- Example: task dependency graphs



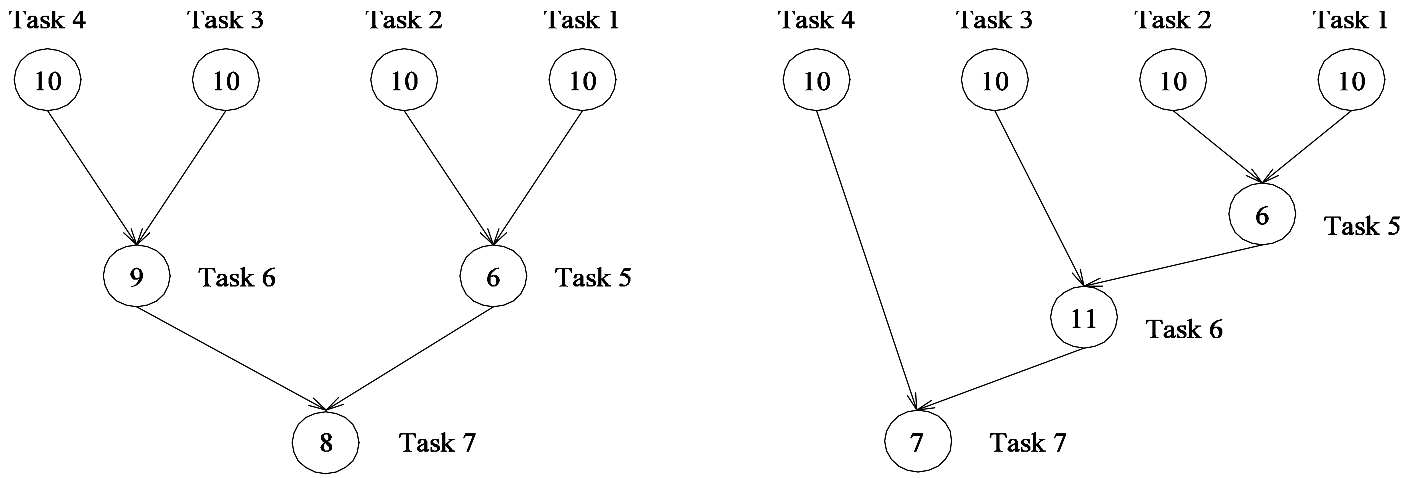


Number in each node is load of each task

- Critical path?
- How many processors are needed?
- Maximum degree of concurrency?
- Average degree of concurrency?

Critical Path Length

- Example: task dependency graphs

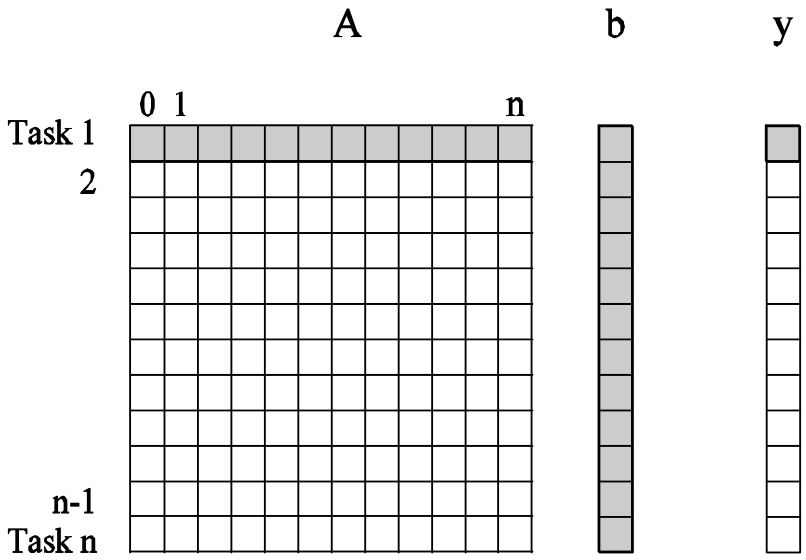


Number in each node is load of each task

- Critical path? 27 vs 34
- How many processors are needed? 4 vs 2 (or 3)
- Maximum degree of concurrency? 4 vs 4
- Average degree of concurrency? 63/27 vs 64/38 (or 64/34)

Critical Path Length

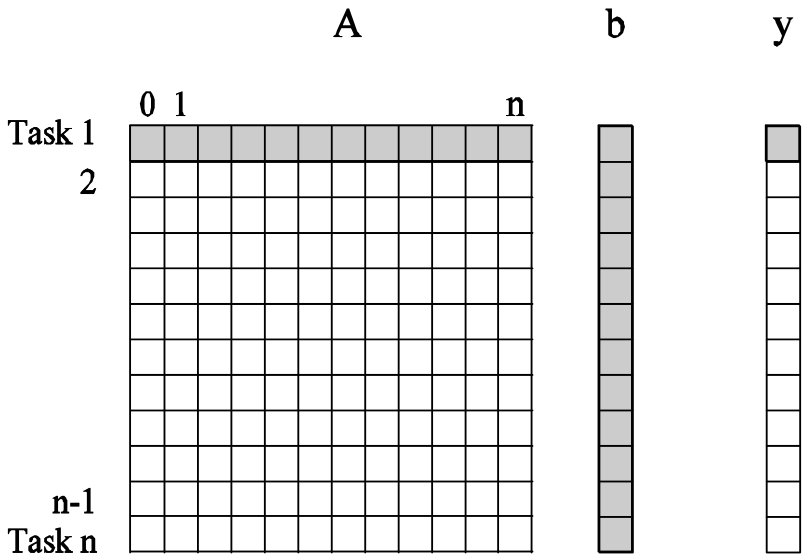
- Example: dense matrix-vector multiplication



- N tasks vs N^2 tasks
- Critical path?
- What is the shortest parallel execution time?
- How many processors are needed?
- Maximum degree of concurrency?
- Average degree of concurrency?

Critical Path Length

- Example: dense matrix-vector multiplication



- N tasks vs N^2 tasks
- Critical path? N vs $\log_2 N$
- What is the shortest parallel execution time? $\log_2 N$
- How many processors are needed? N vs N^2
- Maximum degree of concurrency? N vs N^2
- Average degree of concurrency? N vs $(2N^2)/\log_2 N$