Summary

Performance Modeling:
- Amdahl's Law
- Extrapolation from Observations
- Asymptotic Analysis

- Execution Time: Computation, Communication, Idle

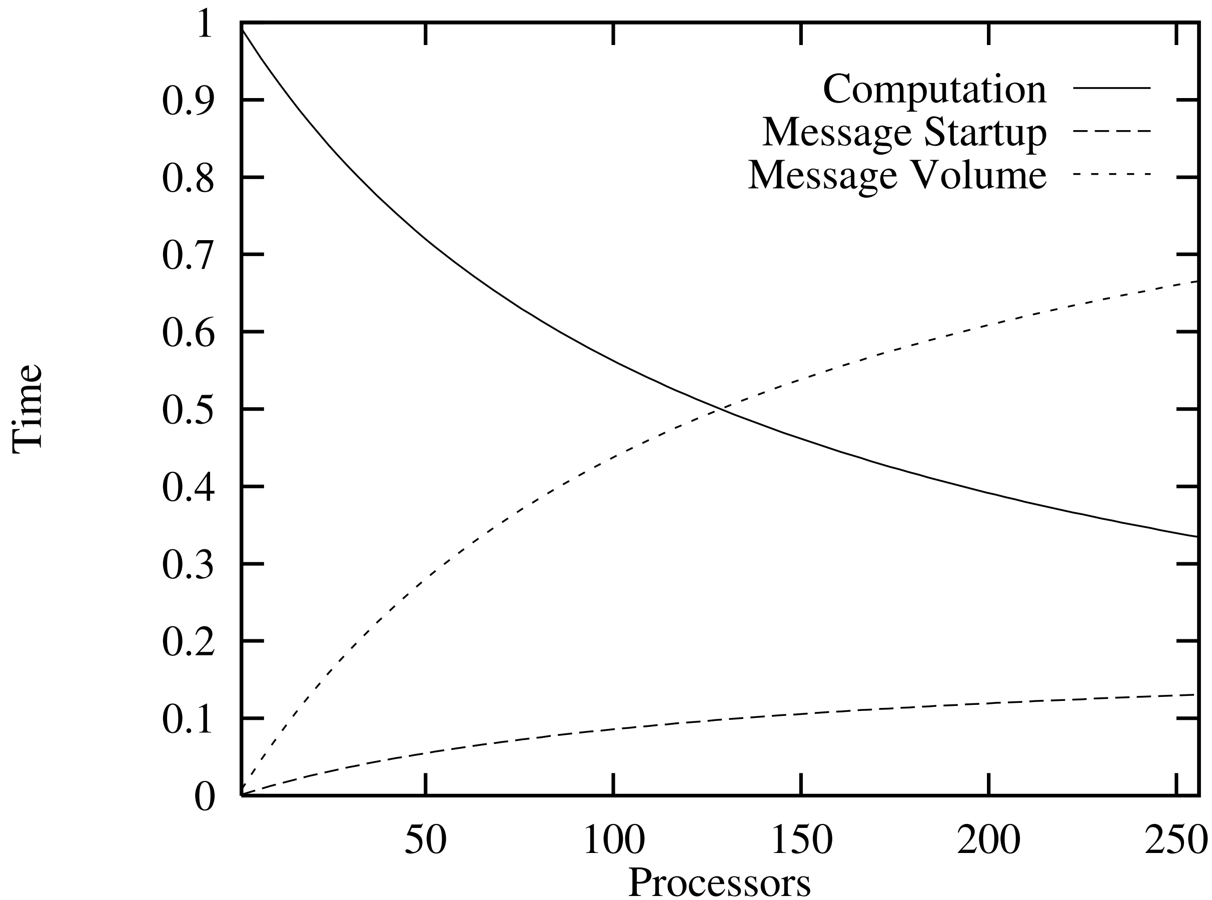
- Efficiency and Speedup

Scalability Analysis
- constant problem size
- scaled problem size

- Execution Profiles

Execution Profiles

detail execution times vs. P:



Scalability with Scaled Problem Size

Isoefficiency function: how computation (problem size=N) has to scale with P to keep efficiency constant.

For **example**, 1D dec. fin. dif. alg: $\eta = \dfrac{t_c N^2 Z}{\left(t_c N^2 Z + t_s 2P + t_w 4NZP\right)}$,

if N=P, then η is going to hold almost constant: $\eta = \dfrac{t_c Z}{\left(t_c Z + \dfrac{2t_s}{P} + t_w 4Z\right)}$

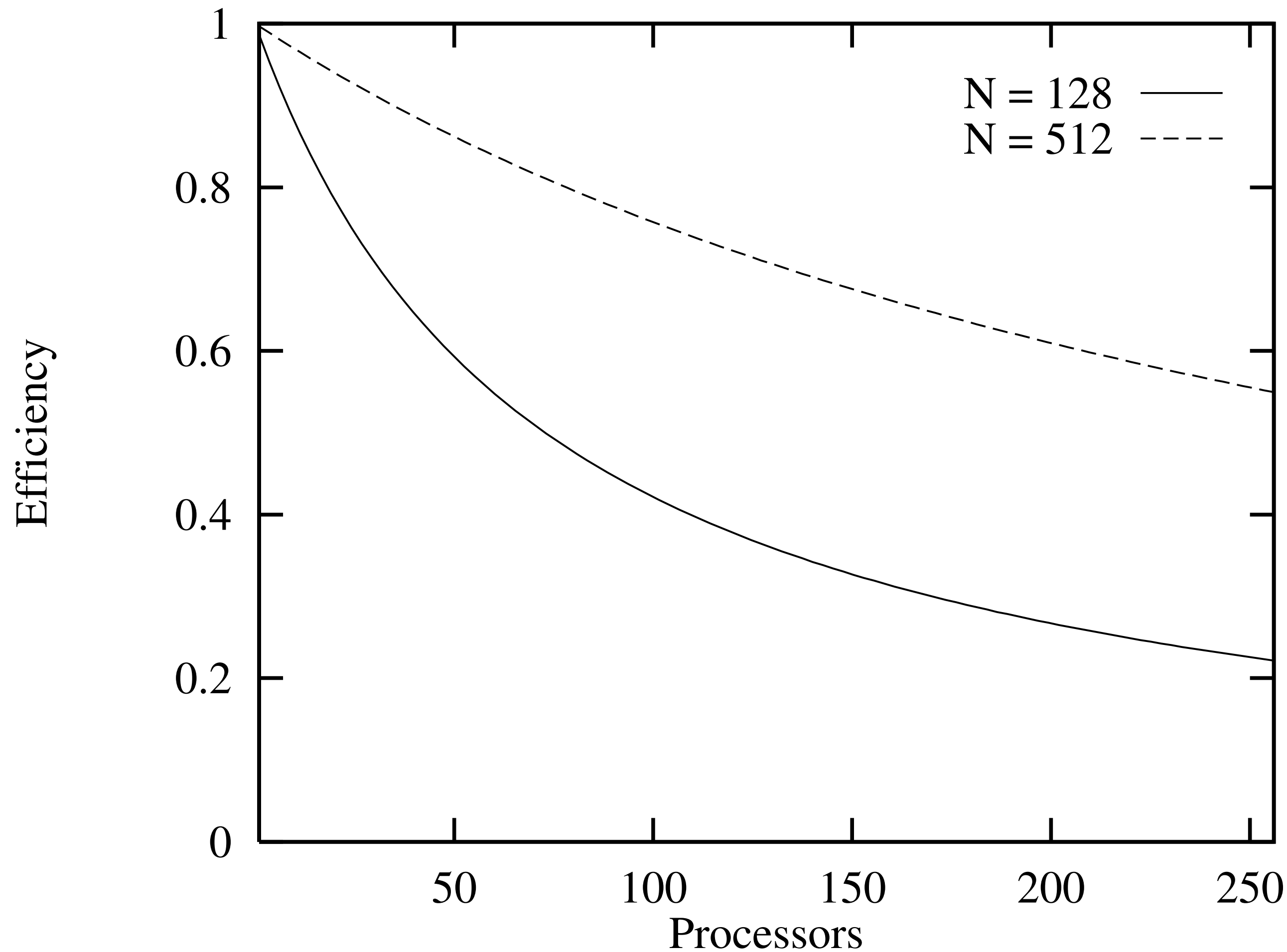
Another example:

2D decomp. fin. dif. alg: $\eta = \dfrac{t_c N^2 Z}{\left(t_c N^2 Z + t_s 4P + t_w 8NZ\sqrt{P}\right)}$;

$N = \sqrt{P}$ isoefficiency function.

Scalability Analysis

plot total time and efficiency vs. P, with N held constant.



Absolute Efficiency and Speedup

Same formulae as above, except baseline for one processor is the best known sequential algorithm

Example:

Efficiency of previous example program: $\eta = \dfrac{t_c N^2 Z}{t_c N^2 Z + t_s 2P + t_w 4NZP}$

Since sequential algorithm is the best know, this is an absolute efficiency measure.

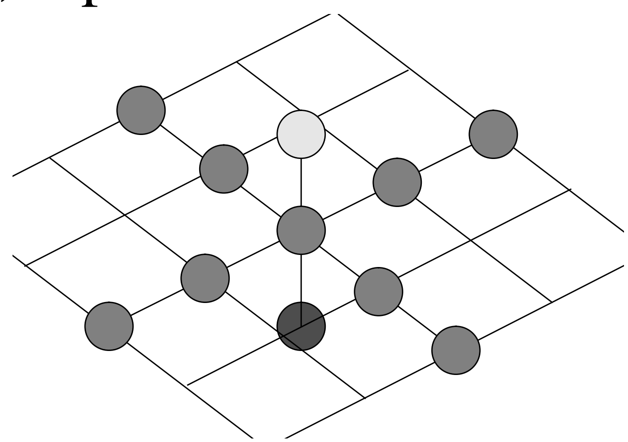
Efficiency and Speedup

- Execution time is misleading because it varies with problem size
- ***Relative Speedup*** measures how much faster the parallel program runs than the serial implementation: $S_{relative} = \frac{T_1}{T_p}$
- **Ideal speedup**: P (very hard to achieve)
- **Superlinear speedup**: sometimes achievable, because by throwing multiple processors at the problem we also throw other resources which are not modelled (like cache and IO ports), thereby reducing computation time beyond expectations

- ***Relative Efficiency****:* $\eta = \frac{T_1}{PT_p} = \frac{S_{relative}}{P}$
- Ideal speedup means 100% efficiency

Example: 1D decomp. Finite Difference

- parallel finite difference algorithm
- grid size N * N * Z, 9-point stencil



each task responsible for grid size N * (N/P) * Z

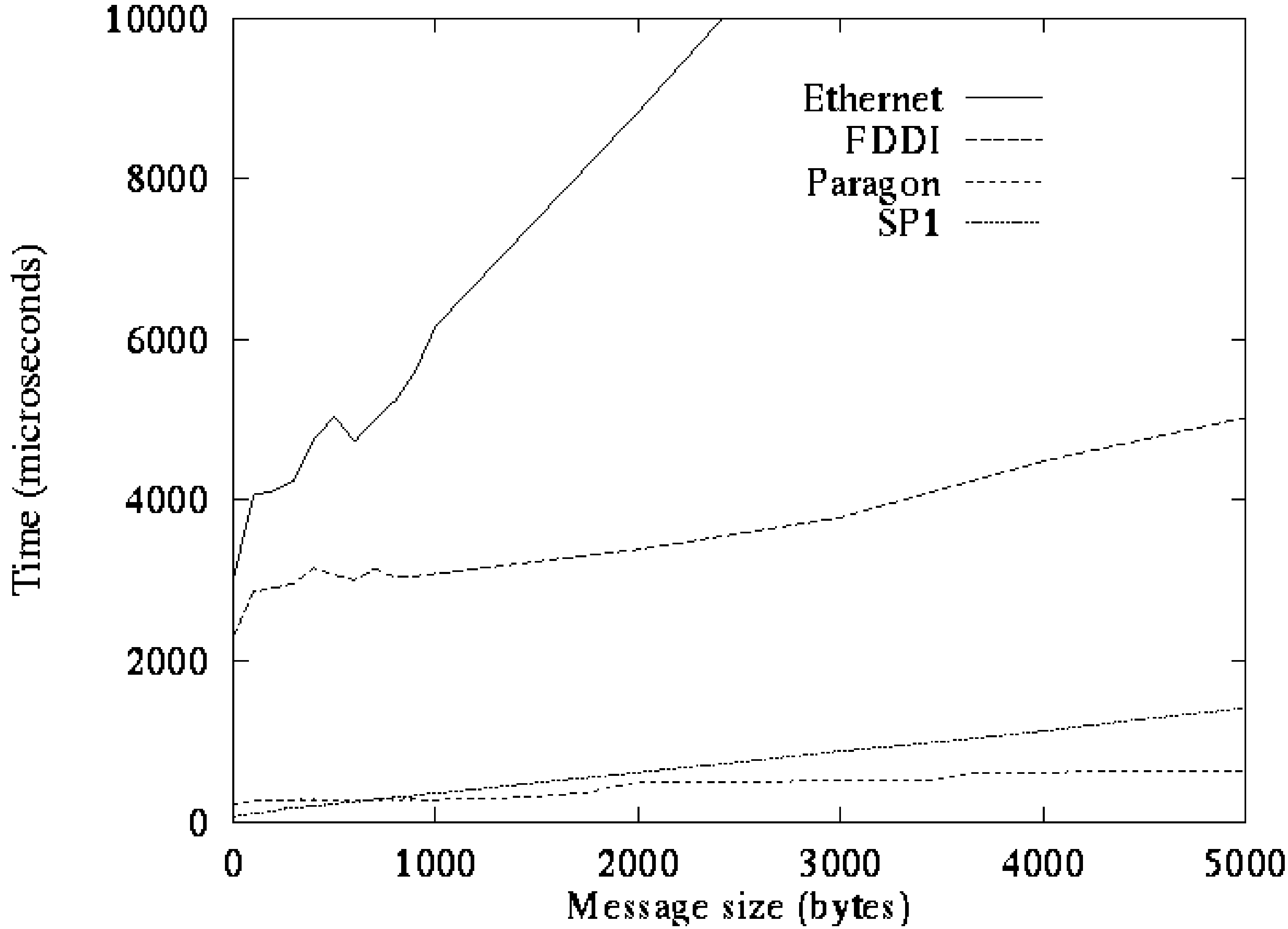
- $T_{comp}=t_c N^2 Z$
- $T_{comm}=2P(t_s+t_w 2NZ)$
- $T_{idle}=0$

So, we end up with $T = \dfrac{T_{comp} + T_{comm}}{P} = \dfrac{N^2 Z}{P} t_c + 2t_s + 4NZt_w$

Idle Time

- Time spent waiting (synchronization costs, various overheads)

Can be minimized by overlapping computation with communication (multithreading!)


Ethernet
FDDI
Paragon
SP1
Time (microseconds)
10000
8000
6000
4000
2000
0
1000
2000
3000
4000
5000
Message size (bytes)

Communication Time

- Time spent sending messages or receiving messages.
    - inter-processor communication
    - intra-processor communication (comm between threads/ tasks)
- inter- and intraprocessor comm speeds are comparable in many systems

Idealized communication time per message sent: $T_{msg} = t_s + t_w L$

- t_s is the time spend to initialize communication ("startup time")
- t_w is time spent per information unit ("word time")

Computation Time

- The “useful load” on the system. Actually spent crunching numbers.

Measuring computation time:

- running a sequential program implementing the same algorithm (no idling or communication) (hard to do)
- let the operating system measure it for us (if so equipped)
- “instrument” the program to measure computation time (distorts results)

- Computation time depends on problem size, number of processors/tasks.
- heterogeneous networks - nonuniform times.
- many other factors (like cache size) affect computation time also.

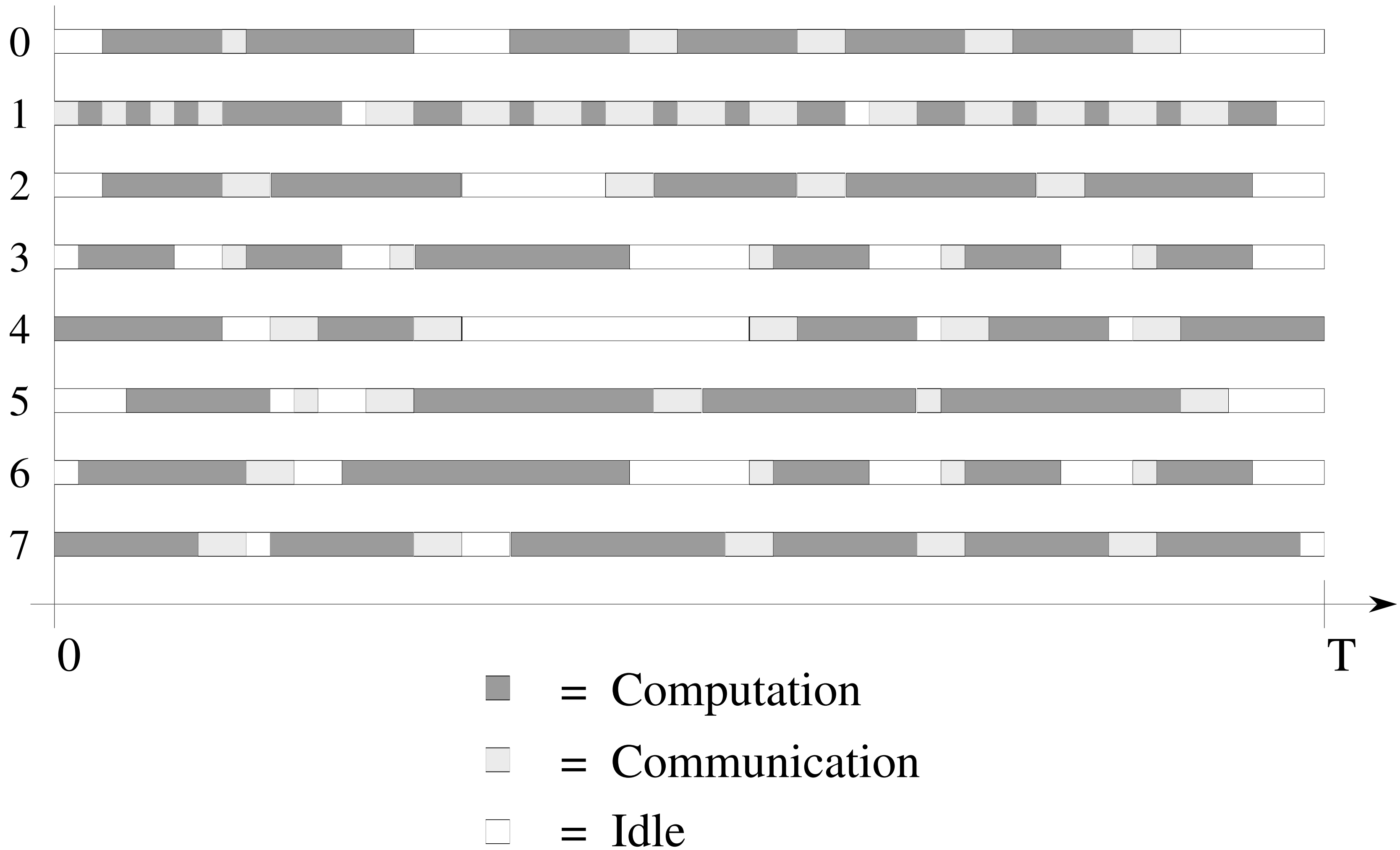
Composition of Execution Time

$$T = \sum_j \left(T^j_{comp} + T^j_{comm} + T^j_{idle} \right)$$

or

$$T = \frac{1}{p}\left(T_{comp} + T_{comm} + T_{idle} \right)$$

$$= \frac{1}{p}\left(\sum_j T^j_{comp} + \sum_j T^j_{comm} + \sum_j T^j_{idle} \right)$$


0
1
2
3
4
5
6
7
0
T
= Computation
= Communication
= Idle

A Model of Execution Time

T=f(N,P,U,...)

N = problem size

P= number of processors

U= number of tasks

other characteristics of hardware (e.g. CPU clock speed?) and software (e.g. efficiency of MPI implementation)

- Components of execution time:

  Computation time T_{comp}

  Communication time T_{comm}

  Idle time T_{idle}

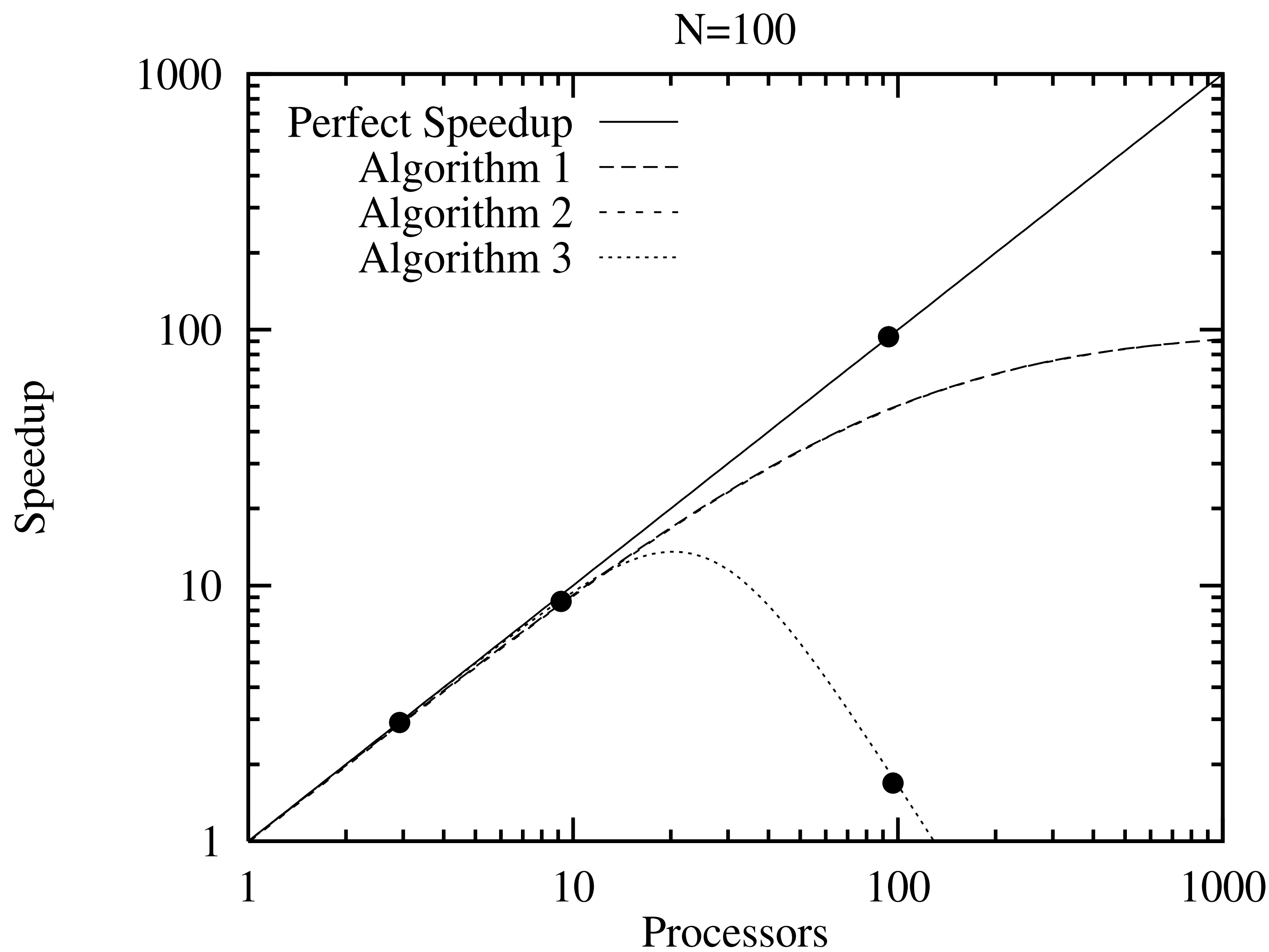
Asymptotic Analysis

Example: “the algorithm requires O(N log N) time on O(N) processors.”

- Information on execution time when N is relatively large.

Pitfalls: Asymptotic behavior doesn’t incorporate multiplicative constants, and can be quite misleading.
- Also, implementing the “ideal” algorithm can be complicated.

Example:
- Good parallel algorithm (quicksort) : O(N log N), on one processor. actual sorting time = 10 N log N
- “Best” parallel sorting algorithm: AKS sorter, O(log N) on N processors, actual sorting time: 6000 log N



N=100
1000
100
10
1
Perfect Speedup
Algorithm 1
Algorithm 2
Algorithm 3
Speedup
1
10
100
1000
Processors

Extrapolation from Observations

(in the book, page 85.): observations of the form

“Algorithm Z on parallel computer X with problem size N achieves a speedup of 10.8 with P=12 processors.”

Advantage: a small number of observations can give an idea as to how “good” the algorithm is.
Pitfalls: that idea may turn out to be misleading when conditions (X,N,P) change. Does require implementation of algorithm.

Amdahl's Law

Formulation 1: If the sequential component of an algorithm accounts 1/s of the execution time, then the maximum possible speedup on a parallel computer is s.

Formulation 2: Speedup on a parallel computer $speedup(N) = \dfrac{s+p}{s+\frac{p}{N}}$

Advantages: can be used to gauge effect of particular changes in the algorithm; can be used to find bottlenecks in the algorithm.

Pitfalls: in real life, when the algorithm has an inherently sequential part (i.e. bottleneck), there exists a better (and completely different) solution. Thus, Amdahl's law will *not* have the last word.

Defining Performance

In a theoretical sense, it is terribly hard to define:

- depends very much on the problem to be solved.

Generally, the degree of satisfying the requirements defines performance.

But we model only **execution time** and **parallel scalability**.

Modeling Performance (Chapter 3)

- How can you tell your algorithm is better than mine?
    - (run & compare the programs)
        - too expensive sometimes - e.g. I haven't written the program yet.

Performance Models:

- compare "efficiency" (tbd) of different algorithms
- evaluate "scalability"
- find algorithmic bottlenecks.