



Reasoning about Performance

Alexandre David

1.2.05

adavid@cs.aau.dk



Basics

- A **sequential** algorithm is evaluated by its runtime in function of its input size.
 - $O(f(n))$, $\Omega(f(n))$, $\Theta(f(n))$.
- The asymptotic runtime is independent of the platform. Analysis “at a constant factor”.
- A **parallel** algorithm has more parameters.



Basics

- A parallel algorithm is evaluated by its runtime in function of
 - the input size,
 - the number of processors,
 - the communication parameters.
- Which performance measures?
- Compare to which (serial version) baseline?



Sources of Performance Loss

- In practice, $E < 1$.
 - Overhead (sync, communication, threads).
 - W_S – Amdahl's law.
 - Idling & contention.
 - Overhead function: $T_0 = pT_p - T_S$
- Ex. count3s:
 - false sharing (communication) was avoidable
 - at the cost of memory
 - incrementing global counter unavoidable



Example – Amdahl's Law

- Assume 20% not parallelizable.

$$T_2 = T_S/2 + 0.2T_S = 0.7T_S$$

$$S = 1/0.7$$

$$E_2 = S/2 = 0.71$$

$$T_{10} = T_S/10 + 0.2T_S = 0.3T_S$$

$$S = 1/0.3$$

$$E_2 = S/10 = 0.33$$



Overheads

- Communication overheads
 - explicit (send/recv) + implicit (threads, false sharing...)
- Synchronization
 - locks, barriers, semaphores
- Extra computations
 - typically reductions
- Memory
 - extra padding, local copies



Scalability

- As seen in Amdahl's law:
 - Speedup inherently limited by the problem size.
 - Increase p , lose speedup.
 - Increase the size, gain speedup – in general W_s grows slowly with the size.
- The question is:
 - How much do you need to increase the size in function of p to keep the same efficiency?
 - Measured by isoefficiency function = measure of scalability.



Scalability

- Suppose $T_S = cn^x$.
- We increase the problem size by m and we use p processors. *But we want to keep the same execution time.*
 - $c(mn)^x/p = cn^x$ – optimistic.
 $p = m^x$
 $m = p^{1/x}$.
 - $x=4$, $m=100$, we need 10^8 processors.
 - Example in the book is miss-leading: The problem comes from the complexity of the algorithm.



Contention

- Special because not seen directly in the code.
- Comes for behavior.
- Difficult to replicate.
- Sources:
 - excessive loads on memory, e.g., spin-lock
 - central locks – granularity problems
 - effects on the architecture – flood the bus



Idling

- Waiting for locks.
- Data dependencies.
 - See previous examples (sum, prefix).
 - May need algorithm rewrites.
- Load balancing.
- Memory bound computations – locality.
- Detecting termination!



Parallel Structure

- Identify dependencies

- task dependency graph
- data dependencies

flow dependency: read after write – *true dep.*

anti-dependency: write after read | → *memory*

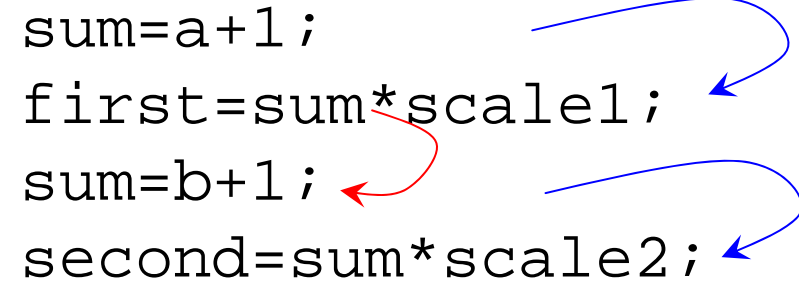
output dependency: write after write | → *reuse*



Example

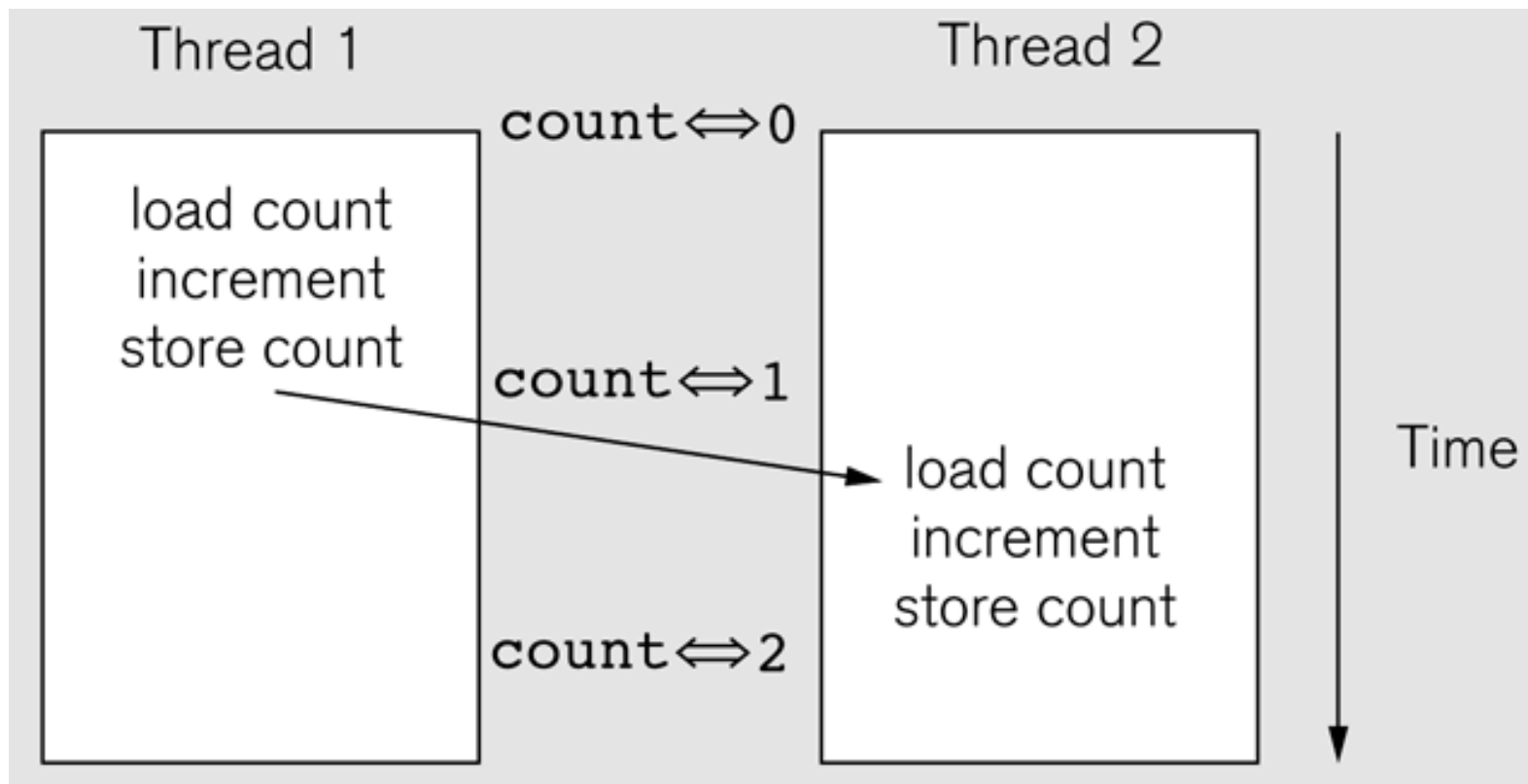
- Flow dependency.
 - Cannot be removed.
- Anti-dependency.
 - Can be removed by renaming/rewriting.

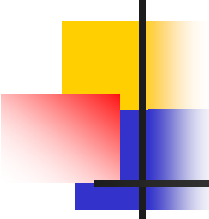
```
sum=a+1;
first=sum*scale1;
sum=b+1;
second=sum*scale2;
```



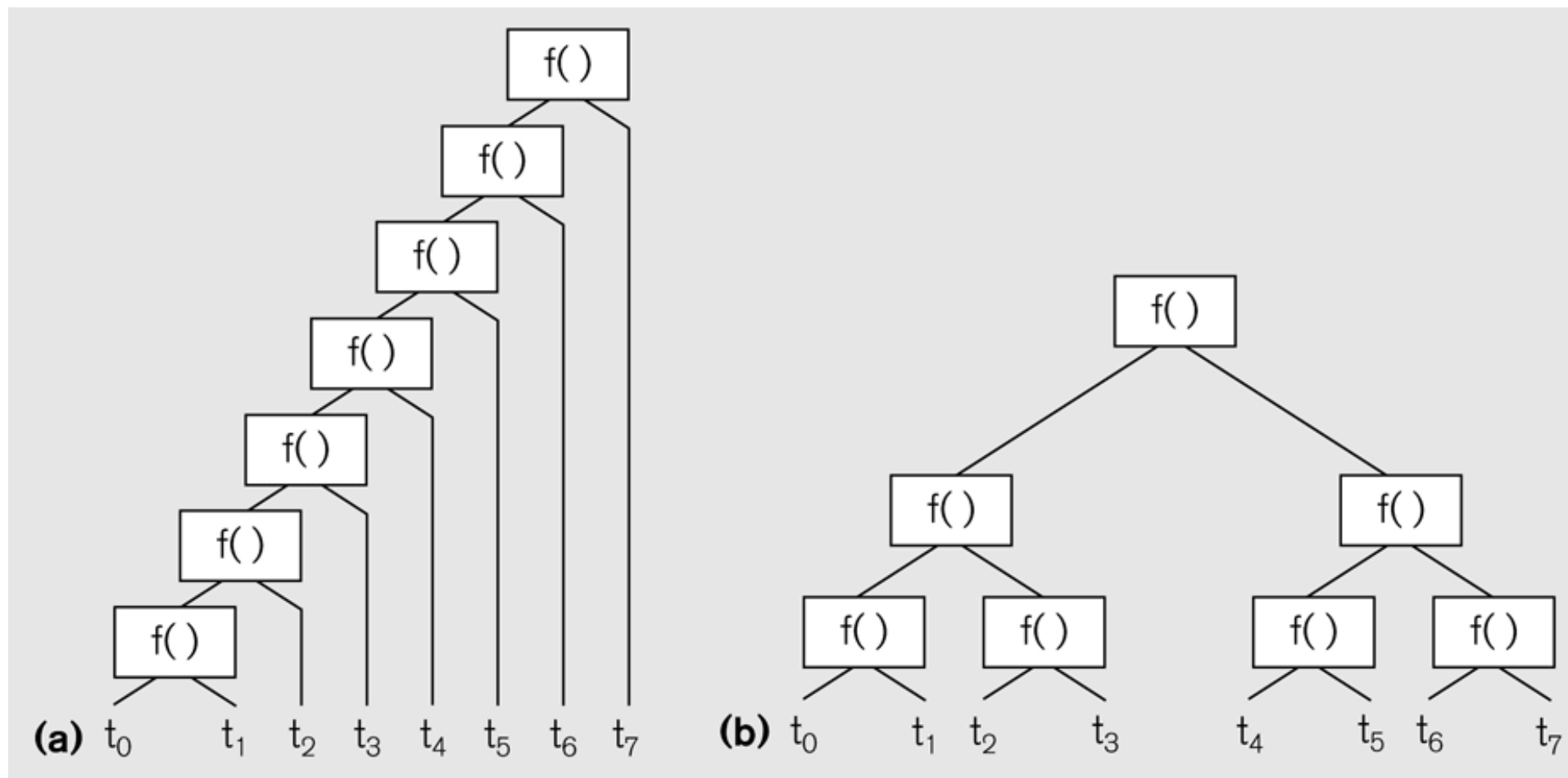
```
first=(a+1)*scale1;
second=(b+1)*scale2;
```

Example





Example





Granularity

- Size of
 - tasks
 - data associated to threads
 - → determines frequency of interactions.
- Fine grain: small independent computations/small data sets → frequent interactions.
 - Multi-core – hardware support, low latency.
- Coarse grain: large independent computations/large data sets → infrequent interactions.
 - MPI – high latency.
- Opposite to load balancing goals.



Locality

- Temporal locality.
- Spatial locality.
- Use cache/paging efficiently.
 - Even more important for parallel programs.
- Locality effect: Reduce dependencies.



Trade-offs

- Identify the 10% code taking 90% time.
 - Not enough for parallel programs but still useful.
- Computation vs. Communication
 - Overlap com. & comp. – trylocks, async I/O.
 - Redundant comp. – recompute (cheaper than communication).
 - Reduces dependencies & sync. costs.
- Memory vs. Parallelism
 - private memory – copy – improve locality
 - padding – avoid false sharing



Trade-offs

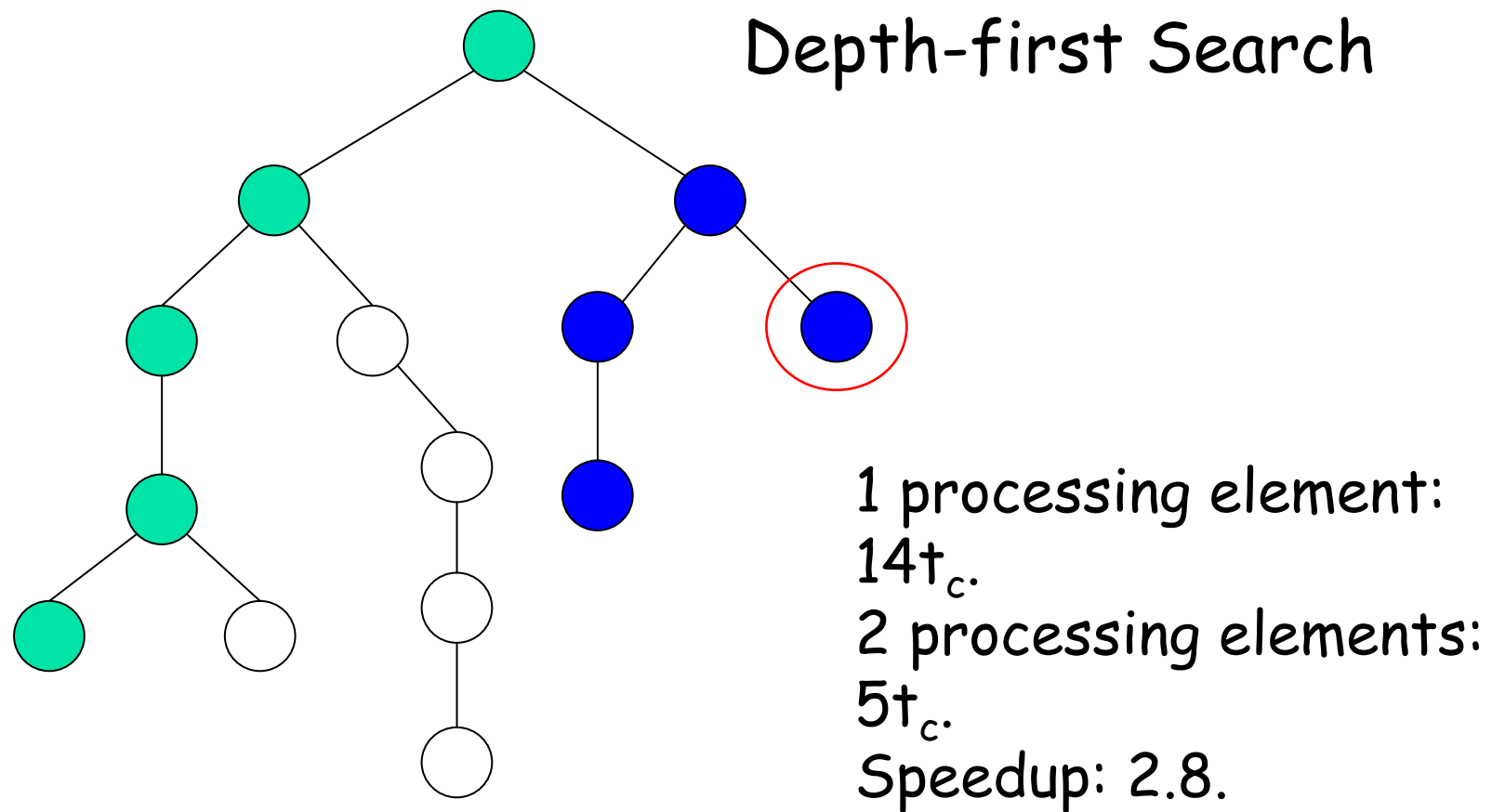
- High parallelism vs. overhead
 - need to combine/reduce results eventually
 - load balance – communication
 - granularity – batching is useful for communication but may result in idling



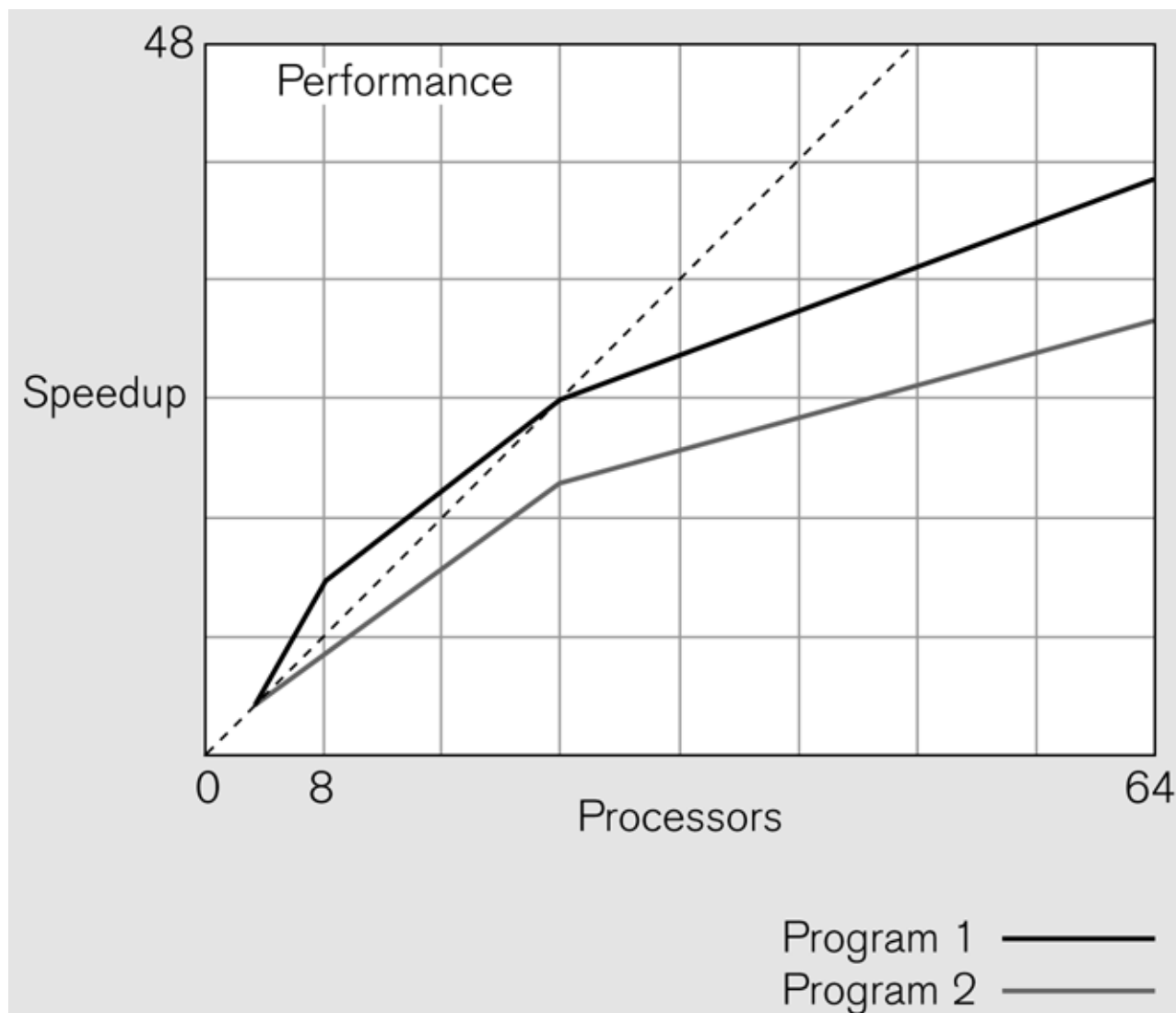
Measuring Performance

- Execution time, FLOPS – limited.
- Speedup, efficiency – useful.
 - Superlinear speedup: either performance anomalies (different work) or increased locality (caches) that overcomes overheads. Theoretically $S \leq p$.
 - Speedup may vary over different machines.
 - True vs. relative speedup.
 - Cold starts – cache/page issues.
 - I/O activities.

Speedup – Example



Example

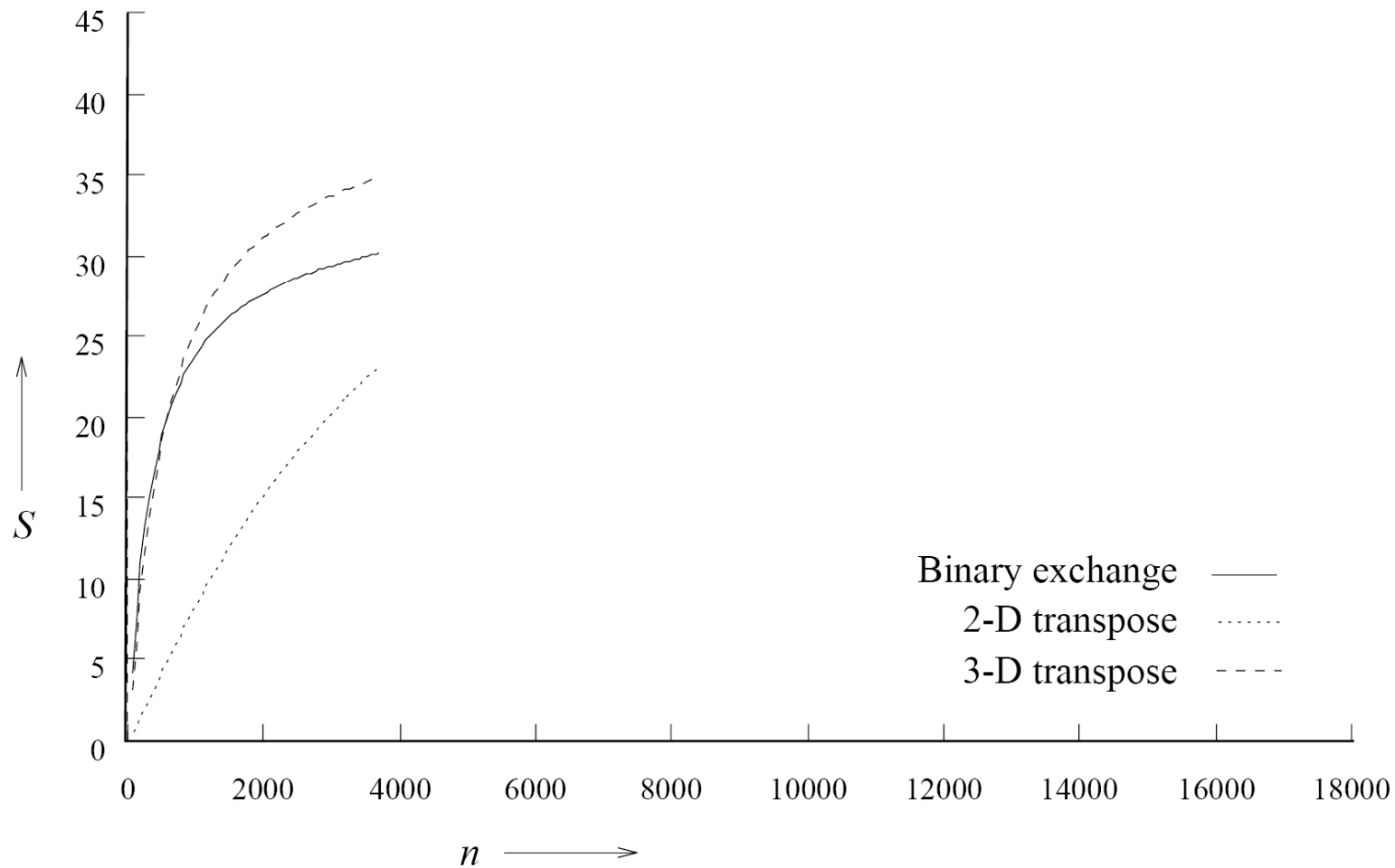




Scalability of Parallel Systems

- In practice: Develop and test on small systems with small problems.
- Problem: What happens for the real large problems on large systems?
 - Difficult to extrapolate results.

Problem with Extrapolation





Rewriting

- Rewrite efficiency (E):

$$\left\{ \begin{array}{l} E = \frac{S}{p} = \frac{T_S}{pT_p} \\ pT_p = T_0 + T_S \end{array} \right. \Rightarrow E = \frac{1}{1 + \frac{T_0}{T_S}} \quad T_S=W$$

- What does it tell us?

Note: $T_0=f(p)$ increasing.



Scalable Parallel System

- Can maintain its efficiency constant when increasing the number of **processors** and the **size** of the problem.
- In many cases $T_0 = f(T_S, p)$ and grows sub-linearly with T_S . It **can be** possible to increase p and T_S and keep E constant.
- Scalability measures the ability to increase speedup in function of p .



Cost-Optimality

- Cost optimal parallel systems have efficiency $\Theta(1)$.
- Scalability and cost-optimality are linked.
- Adding number example: becomes cost-optimal when $n = \Omega(p \log p)$.



Scalable System

- Efficiency can be kept constant when
 - the number of processors increases and
 - the problem size increases.
- At which **rate** the problem size should increase with the number of processors?
 - The rate determines the **degree of scalability**.
- In complexity, problem size = size of the input. Here = number of basic operations to solve the problem. Noted $W (\sim T_S)$.

Isoefficiency Function

- For scalable systems efficiency can be kept constant if T_0/W is kept constant.

For a target E

$$E = \frac{1}{1 + T_0(W,p)/W},$$

Keep this constant

$$\frac{T_0(W,p)}{W} = \frac{1 - E}{E},$$

Isoefficiency function

$$W = \frac{E}{1 - E} T_0(W,p).$$

How to increase work in function of p to keep the **same** efficiency (*iso*)?

$$W = K T_0(W,p)$$



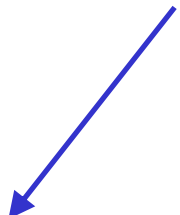
Example

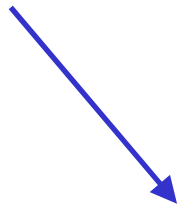
- Adding number: We saw that $T_0 = 2p \log p$.
- We get $W = K 2p \log p$.
- If we increase p to p' , the problem size must be increased by $(p' \log p') / (p \log p)$ to keep the same efficiency.
 - Increase p by p'/p .
 - Increase n by $(p' \log p') / (p \log p)$.



Example

$$T_o = p^{3/2} + p^{3/4}W^{3/4}$$


$$W = Kp^{3/2}.$$


$$\begin{aligned}W &= Kp^{3/4}W^{3/4} \\ W^{1/4} &= Kp^{3/4} \\ W &= K^4p^3\end{aligned}$$

Isoefficiency = $\Theta(p^3)$.



Why?

- After isoefficiency analysis, we can test our parallel program with few processors and then **predict** what will happen for larger systems.



Link to Cost-Optimality

A parallel system is cost-optimal iff
 $pT_p = \Theta(W)$.

$$W + T_o(W, p) = \Theta(W)$$


$$T_o(W, p) = O(W)$$

$$W = \Omega(T_o(W, p))$$

A parallel system is cost-optimal iff
its overhead (T_o) does not exceed
(asymptotically) the problem size.



Minimum Execution Time

- If T_p  in function of p , we want its minimum. Find p_0 s.t. $dT_p/dp=0$.
- Adding n numbers: $T_p=n/p+2 \log p$.
 - $p_0=n/2$.
 - $T_p^{min}=2 \log n$.
- Fastest but not necessary cost-optimal.

Asymptotic Analysis of Parallel Programs

Table 5.2 Comparison of four different algorithms for sorting a given list of numbers. The table shows number of processing elements, parallel runtime, speedup, efficiency and the pT_P product.

Best?

Algorithm	A1	A2	A3	A4
p	n^2	$\log n$	n	\sqrt{n}
T_P	1	n	\sqrt{n}	$\sqrt{n} \log n$
S	$n \log n$	$\log n$	$\sqrt{n} \log n$	\sqrt{n}
E	$\frac{\log n}{n}$	1	$\frac{\log n}{\sqrt{n}}$	1
pT_P	n^2	$n \log n$	$n^{1.5}$	$n \log n$



Other Scalability Metrics

- Scaled speedup: speedup when problem size increases linearly in function of p .
 - Motivation: constraints such as memory linear in function of p .
 - Time and memory constrained.