
Cassiopeia
Analytical Modeling of Parallel Programs
Alexandre David
AALBORG UNIVERSITY
Introduction to Parallel Computing

Topic overview

- Sources of overhead in parallel programs.
- Performance metrics for parallel systems.
- Effect of granularity on performance.
- Scalability of parallel systems.

Analytical modeling – basics

- A sequential algorithm is evaluated by its runtime in function of its input size.
  - O(f(n)), Ω(f(n)), Θ(f(n)).
- The asymptotic runtime is independent of the platform. Analysis “at a constant factor”.
- A parallel algorithm has more parameters.
  - Which ones?

Reminder O-notation, Ω-notation, Θ-notation.

Analytical modeling – 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?

Note: The underlying RAM model may play a role, keep in mind that they are equivalent and the more powerful models can be emulated by the weaker ones in polynomial time.

Parallel system = parallel algorithm + underlying platform, which we analyze.

Performance measures: time obvious, but how does it scale?

Cassiopeia

Sources of overhead in parallel programs

- Overheads: wasted computation, communication, idling, contention.
  - Inter-process interaction.
  - Load imbalance.
  - Dependencies.

Naïve question

Shouldn't my program run twice faster if I use two processors?

Different sources of overhead: We have already seen them. Wasted computation = excess computation (speculative execution for example, or duplicate work).

Performance metrics for parallel systems

- Execution time = time elapsed between
  - beginning and end of execution on a sequential computer.
  - beginning of first processor and end of the last processor on a parallel computer. T_P.

Intuitive for sequential programs but be careful for parallel programs.

Execution time denoted T_P.

Performance metrics for parallel systems

- Total parallel overhead.
  - Total time collectively spent by all processing elements = pT_P.
  - Time spent doing useful work (serial time) = T_S.
  - Overhead function: $T_O = pT_P - T_S$.
    General function, contains all kinds of overheads.

Quantitative way of measuring overheads, this metric contains all kinds of overheads.

Performance Metrics for Parallel Systems

- What is the benefit of parallelism?
  - Speedup of course… let's define it.
- Speedup $S = T_S/T_P$.
- Example: Compute the sum of n elements.
  - Serial algorithm $\Theta(n)$.
  - Parallel algorithm $\Theta(\log n)$.
  - Speedup = $\Theta(n/\log n)$.
- Baseline (T_S) is for the best sequential algorithm available.

And by the way speedup is one benefit, you can find others like simpler hardware architectures (several simple CPUs better than one big complex) and heat issues.

Adding 2 elements and communication time are constants.

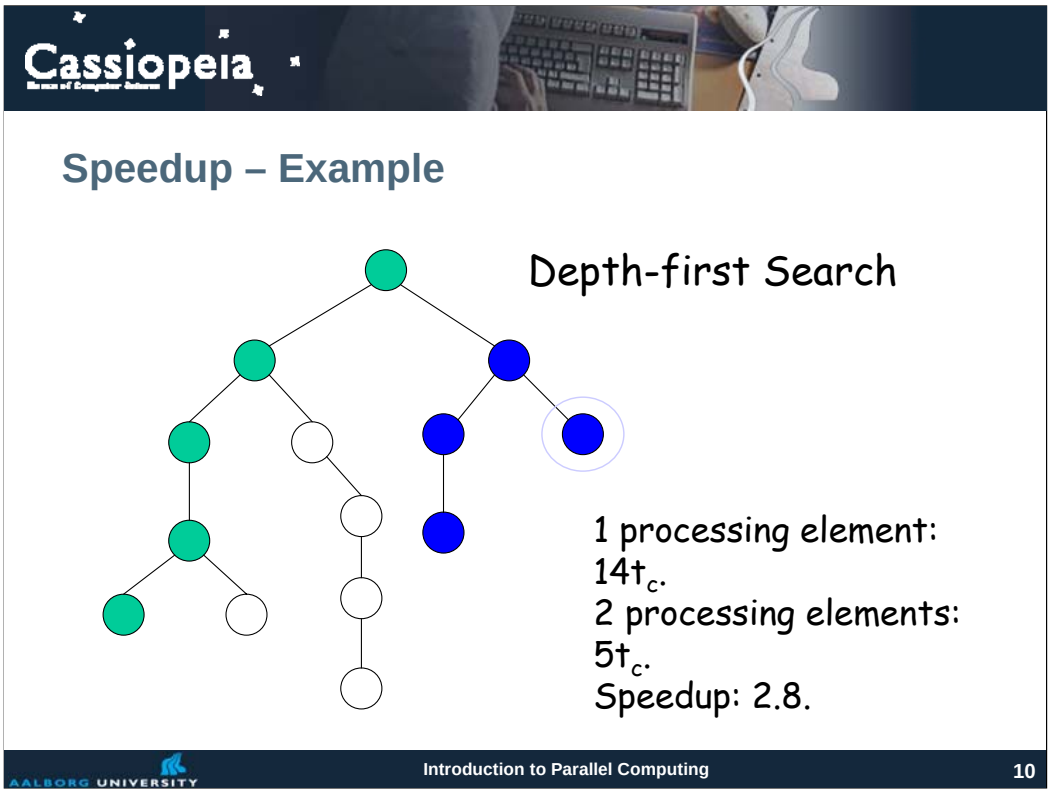
Question: Compare to what? Which T_S to take? All the sequential algorithm are not equally parallelizable and do not perform the same.

Speedup

- Theoretically, speedup can never exceed *p*. If > *p*, then you found a better sequential algorithm… Best: $T_P=T_S/p$.
- In practice, super-linear speedup is observed. How?
  - Serial algorithm does more work?
  - Effects from caches.
  - Exploratory decompositions.

Serial algorithm may do more work compared to its parallel counterpart due to features in parallel hardware.

Caches: aggregate amount of caches is larger, so “more data can fit in the cache”, if the data is partitioned appropriately.



The works performed by the serial and the parallel algorithms are different. If we simulate 2 processes on the same processing element then we get a better serial algorithm for this **instance** of the problem but we cannot generalize it to all instances. Here the work done by the different algorithms depends on the input, i.e., the location of the solution in the search tree.

Performance Metrics

- Efficiency ***E=S/p***.
  - Measure time spent in doing useful work.
  - Previous sum example: $E = \Theta(1/\log n)$.
- Cost $C=pT_P$.
  - A.k.a. work or processor-time product.
  - Note: $E=T_S/C$.
  - Cost optimal if E is a constant.
- Example: Compute the sum of n elements.
  - Efficiency = $\Theta(1/\log n)$.

Speedup/number of processing elements. Ideally it is 1 with S = p.

Comment for 1/logn: efficiency (and speedup too) goes down with n. If the problem size increases you win less by using more processors.

Check yourself edge detection example in the book.

Cost = parallel runtime * number of processing elements = total time spent for all processing elements.

C is a constant = T_S and T_P have the same asymptotic growth function (at a constant factor).

Related to previous lecture on Brent's scheduling principle.

Effect of Granularity on Performance

- Scaling down: To use fewer processing elements than the maximum possible.
- Naïve way to scale down:
  - Assign the work of *n/p* processing element to every processing element.
    - Computation increases by *n/p*.
    - Communication growth ≤ *n/p*.
  - If a parallel system with *n* processing elements is cost optimal, then it is still cost optimal with *p*.

If it is not cost optimal, it may still not be cost optimal after the granularity increase.

Communication growth bounded if the mapping is appropriate.

Recall Brent's scheduling algorithm: Re-schedule tasks on processes. It doesn't do miracles, it's only a re-scheduling algorithm.

Reason for improvement in increasing the granularity (coarse grained vs. fine grained): Decrease of global communication (instead of growing with *n*, it should grow with *n/p*) because tasks mapped on the same process communicate together without overhead.

Amdahl's law

If a problem of size W has a serial component W_s then $S \leq W/W_s$ for any p.

$S=W/T_P=W/(W_S+(W-W_S)/p)$

$(W-W_S)/p \rightarrow 0$ when p increases. No matter how large p is, we have the bound $S \leq W/W_S$.

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*.

Scalability: ability to use efficiently increasing processing power.