F1: Performance and Scalability

Lars Karlsson

2009-03-27

Outline

- Complexity analysis
- Runtime, speedup, efficiency
- Amdahl's Law and scalability
- Cost and overhead
- Cost optimality
- Iso-efficiency function
- Case study: matrix vector product

Complexity Analysis (Upper Bounds)

- **Definition:** Given a function $g(x)$, $f(x) = O(g(x))$ if and only if for any constant $c > 0$ there exists an $x_0 > 0$ such that

  $$f(x) \leq cg(x)$$

  for all $x \geq x_0$.

Complexity Analysis (Lower Bounds)

- **Definition:** Given a function $g(x)$, $f(x) = \Omega(g(x))$ if and only if for any constant $c > 0$ there exists an $x_0 > 0$ such that
$$cg(x) \leq f(x)$$
for all $x \geq x_0$.

Complexity Analysis (Tight Bounds)

- **Definition:** Given a function $g(x)$, $f(x) = \Theta(g(x))$ if and only if for any constants $c1, c2 > 0$ (with $c_2 \geq c_1$) there exists an $x_0 > 0$ such that

$$c_1 g(x) \leq f(x) \leq c_2 g(x)$$

for all $x \geq x_0$.

Complexity Analysis: Properties

1. $x^a = O(x^b)$ iff $a \leq b$.
2. $\log_a(x) = \Theta(\log_b(x))$ for all a and b.
3. $a^x = O(b^x)$ iff $a \leq b$.
4. For any constant c, $c = O(1)$.
5. If $f = O(g)$ then $f + g = O(g)$.
6. If $f = \Theta(g)$ then $f + g = \Theta(g) = \Theta(f)$.
7. $f = O(g)$ iff $g = \Omega(f)$.
8. $f = \Theta(g)$ iff $f = \Omega(g)$ and $f = O(g)$.

Communication Cost Model

- The cost of sending an *m*-word message is modeled by

$$t_s + m t_w.$$

- t_s is the *startup cost*, and
- t_w is the *word transfer time* (or, *inverse bandwidth*).

Beware of the units on t_w. It is seconds per word (not byte, in general). A word is user-defined: if you model a numerical algorithm you might pick a word to be a `double` (8 bytes).

Parallel Runtime

- The *parallel runtime* or *parallel execution time* is the time that elapses from the start of the parallel computation to the moment the last process finishes execution.
- The parallel runtime is denoted by T_p and the sequential runtime (of the best sequential algorithm) is denoted by T_s.
- In particular, for $p = 7$ processes we use T_7 to denote the parallel runtime.
- Note that $T_1 \neq T_s$, but $T_s \leq T_1$ (why?).

Speedup

- The *speedup* S_p is defined as

  $$S_p = \frac{T_s}{T_p}.$$

  Often, $0 < S_p \leq p$.
- The *relative speedup* is defined as

  $$\tilde{S}_p = \frac{T_1}{T_p}.$$

  Note that $\tilde{S}_p \geq S_p$ (hence, an overestimate).
- With $S_p = p$ we have *linear speedup*.
- With $S_p > p$ we have *superlinear speedup*.
- With $S_p < 1$ we have *parallel slowdown*.

Efficiency

- The *efficiency* is defined as

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}.$$

  Often, $0 < E_p \leq 1$.

- Similarly, *relative efficiency* is defined as

$$\tilde{E}_p = \frac{\tilde{S}_p}{p} = \frac{T_1}{pT_p}.$$

  Note that $\tilde{E}_p \geq E_p$ (hence, an overestimate).

Remarks on speedup and efficiency

- Relative speedup and efficiency are easy to compute (no need to find the best sequential algorithm).
- Note that $\tilde{S}_p = p$ does not exclude the possibility of $S_p < 1$.
- The speedup S_p directly tells you something about T_p:

$$S_{p_1} > S_{p_0} \Rightarrow T_{p_1} < T_{p_0}.$$

- However, the efficiency does not tell you much about T_p for various p.
- But a constant efficiency translates into a linear speedup curve with slope ≤ 1:

$$E_p = \frac{S_p}{p} = C \Rightarrow S_p = Cp.$$

Example of T_p, S_p, and E_p

Assuming

$$T_p = \frac{100}{p} + p, \qquad T_s = 100.$$

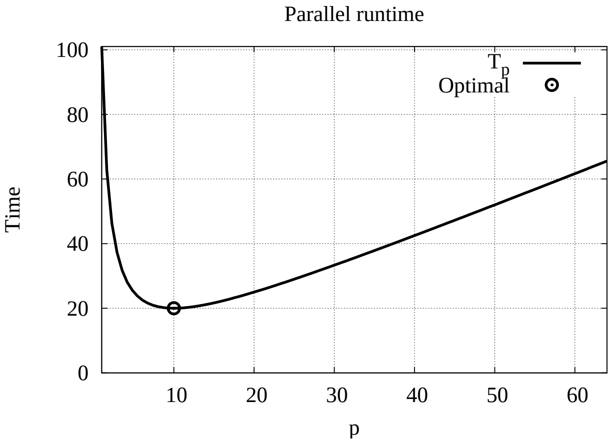
We get

$$S_p = \frac{T_s}{T_p} = \frac{100}{\frac{100}{p} + p} = \frac{1}{p^{-1} + \frac{p}{100}}$$

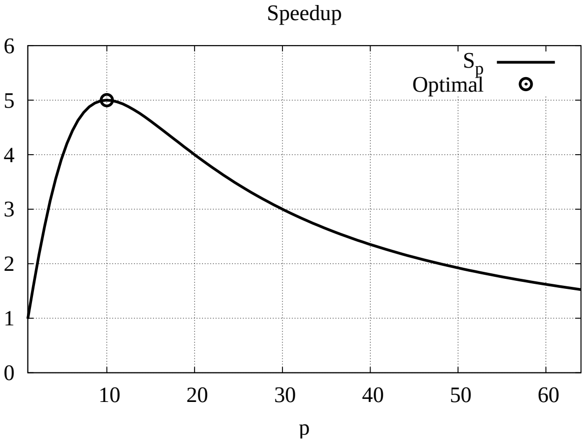
and

$$E_p = \frac{S_p}{p} = \frac{1}{p\left(p^{-1} + \frac{p}{100}\right)} = \frac{1}{1 + \frac{p^2}{100}}.$$

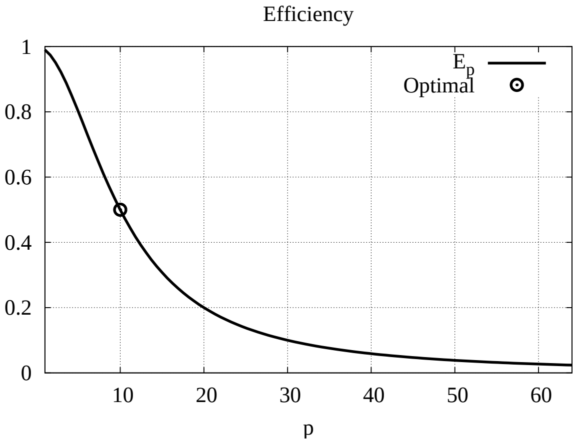
Example of T_p, S_p, and E_p



Example of T_p, S_p, and E_p



Example of T_p, S_p, and E_p



Amdahl's Law

- A famous upper bound on speedup is Amdahl's Law.
- Assume that the work W can be partitioned into sequential work W_s and (fully) parallel work $(W - W_s)$.
- Runtimes:

$$T_s = W, \quad T_p = W_s + \frac{W - W_s}{p}.$$

- Speedup:

$$S_p = \frac{T_s}{T_p} = \frac{W}{W_s + \frac{W-W_s}{p}} \leq \frac{W}{W_s}.$$

- Example: $W_s = 0.1W \Rightarrow S_p \leq 10$ no matter how many processors.

Strong Scalability

- Amdahl's Law applies to a scenario in which a fixed problem size is solved with increasing number of processors.
- A system which maintains a high speedup in such a scenario is said to be *strongly scalable*.
- A fundamental issue is that few applications care about strong scalability.
- **Discussion:** find applications for which strong scalability is a natural evaluation criteria.

(Parallel) Cost and the Overhead Function

- The (parallel) *cost* pT_p is the number of CPU seconds used by a parallel computation.
- (The monetary cost is often based on the parallel cost (e.g., in USD per CPU hour).)
- The *overhead function* T_o is defined as

$$T_o = pT_p - T_s.$$

- Note that $E_p \leq 1 \Rightarrow T_s \leq pT_p \Rightarrow T_o \geq 0$.
- We will see soon how T_o is related to scalability.

Problem Size

- We often use parameters of the problem to specify the size of the problem:
  - The integers m, n, k specify the dimensions of the matrices in the matrix update
    $$C \leftarrow C + AB.$$
    The number of floating point operations is $2mnk$.
  - The integer n specifies the length of a list to sort. The complexity of comparison-based sorting is $\Omega(n \log n)$.
- We need a *problem independent definition* of problem size.
- The problem size is the number of basic operations in the best sequential algorithm and is denoted by W.
- We can normalize hardware parameters so that for all intents and purposes, $W = T_s$.

Cost Optimality

- A system is said to be *cost optimal* if

$$pT_p = \Theta(W).$$

- In other words, the cost of the parallel computation grows no faster than the best known sequential algorithm.
- Intuitively, this allows us to compare the complexities of the parallel and sequential algorithms.
- A **non**-cost optimal system can easily exhibit very low efficiency since pT_p (the denominator) grows faster than T_s (the nominator).

Implications of **Non**-Cost Optimality

- Assume a sorting algorithm which sorts a list of n elements on n processors in time $(\log_2 n)^2$.
- The sequential runtime is $T_s = n \log_2 n$.
- The system is not cost optimal:

  $$pT_p = nT_n = n(\log_2 n)^2 \neq \Theta(n \log_2 n).$$

  But the factor is only $\log_2 n$...
- Now assume we execute this algorithm on $p \ll n$. The parallel runtime is

  $$T_p = \frac{n(\log_2 n)^2}{p}.$$

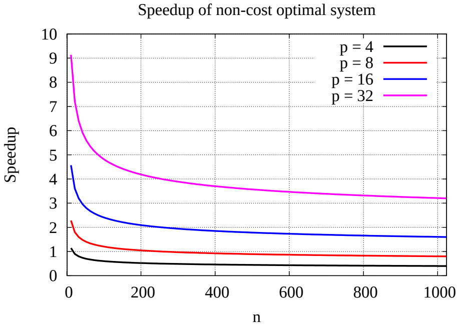
Implications of **Non**-Cost Optimality

- Speedup:

$$S_p = \frac{T_s}{T_p} = \frac{p}{\log_2 n}.$$

- Efficiency:

$$E_p = \frac{S_p}{p} = \frac{1}{\log_2 n}.$$



Conditional Cost Optimality

- Some systems are cost optimal given a condition on the problem size and the number of processors.
- Consider a system with

$$T_s = \Theta(n) \quad T_p = \Theta\left(\frac{n}{p} + \log p\right).$$

- Its cost is

$$pT_p = \Theta\left(n + p\log p\right).$$

- The system is cost optimal only if

$$n = \Omega(p \log p).$$

Scalability

- Recall:

  $$E_p = \frac{1}{1 + \frac{p^2}{100}}.$$

  The red term controls the decrease in efficiency.
- Generally:

  $$E_p = \frac{T_s}{pT_p} = \frac{T_s}{W + T_o} = \frac{1}{1 + \frac{T_o}{W}}.$$

- **Question:** can we keep E_p constant by manipulating the problem parameters (i.e., the problem size) while we increase p?
- **Answer:** yes, sometimes. We define a *scalable system* as one for which this is possible. Otherwise, the system is *non-scalable*.

Iso-efficiency Function

$$E_p = \frac{1}{1 + \frac{T_o}{W}}.$$

We take a closer look at T_o and W:

- The problem size W can be increased arbitrarily.
  $\Rightarrow$ Increasing the problem size increases the efficiency.
- The overhead T_o is a function of the problem size and the number of processes p.
  $\Rightarrow$ Increasing the number of processes decreases the efficiency.

Iso-efficiency Function

- Assuming the system is scalable:

  $$W = \frac{E}{1-E} T_o$$

  for some constant efficiency E.
- If we can obtain W as a function of p from the equation above, we get more knowledge of how scalable a system is.
- This function is known as the *iso-efficiency function*.
- It tells us how much the problem size must be increased to maintain efficiency.
- A slow growing iso-efficiency function is good news while a fast growing function is bad news.

Iso-efficiency Function: Example

- Assume that the overhead function is

$$T_o = 2p \log p.$$

- We get

$$W = \frac{E}{1-E} T_o = \frac{E}{1-E} 2p \log p = \Theta(p \log p).$$

- If we on p_0 processors need problem size W_0 to get a certain efficiency, we expect that on $p_1 > p_0$ processors we need problem size

$$W_1 = \frac{p_1 \log p_1}{p_0 \log p_0} W_0.$$

to attain the same efficiency.

Iso-efficiency and Complicated Overhead Functions

- For more complicated overhead functions it can be impossible to express W in terms of p.
- Example:

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

- Using only the **first** term of T_o:

$$W = Kp^{3/2} = \Theta(p^{3/2}).$$

- Using only the **second** term of T_o:

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

- Recall that we want to find a function for the numerator which grows fast enough to balance the denominator.
- Hence, from a term-by-term analysis we take the **maximum**:

$$W = \Theta(\max\{p^{2/3}, p^3\}) = \Theta(p^3).$$

Lower Bound on the Iso-efficiency Function

- What is the smallest possible iso-efficiency function (i.e., the most ideally scalable system)?
- For **any** system, no more than W processors can be used; the remaining will be idle.
- We can express this as

$$W = \Omega(p).$$

- Hence, the problem size must grow at least linearly with the number of processors.

Degree of Concurrency

- The maximum number of processors that can be active at any one time on a problem of size W is the *maximum degree of concurrency* and is denoted by $C(W)$.
- Using more than $C(W)$ processors is pointless since $p - C(W)$ processors will be idle.
- In general, the degree of concurrency can be the limiting factor when determining the iso-efficiency function.
- Generally,

$$p = O(C(W))$$
$$C(W) = \Omega(p)$$

- Example:

$$C(W) = \sqrt{W} = \Omega(p) \Rightarrow W = \Omega(p^2)$$

which sets a lower bound on the iso-efficiency function to p^2.

Analysis of Matrix-Vector Product

- An $n \times n$ matrix times an $n \times 1$ vector takes time
$$T_s = t_c n^2.$$
- The parallel runtime is (without going in to detail):
$$T_p = t_c \frac{n^2}{p} + t_s \log_2 p + t_w n.$$
- Overhead:
$$T_o = pT_p - T_s = t_s p \log_2 p + t_w pn.$$
- Iso-efficiency:
$$W = Kt_s p \log_2 p \Rightarrow W = \Theta(p \log_2 p)$$
$$W = Kt_w pn \Rightarrow t_c n = Kt_w p \Rightarrow W = \frac{K^2 t_w^2}{t_c} p^2 = \Theta(p^2)$$

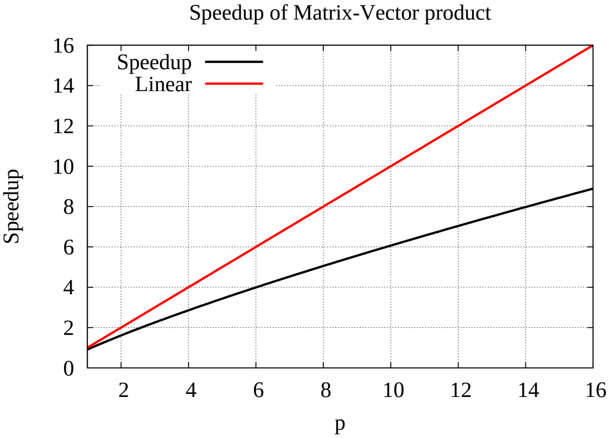
Analysis of Matrix-Vector Product

Memory constrained scalability.

- The memory available grows linearly with the number of processors:
$$m = \Theta(p).$$
- The memory required is
$$m = \Theta(n^2).$$
- Hence, for some constant c,
$$n^2 = cp.$$
- Plug this into S_p:
$$S_p = \frac{t_c cp}{t_c c + t_s \log_2 p + t_w \sqrt{cp}} = O(\sqrt{p}).$$

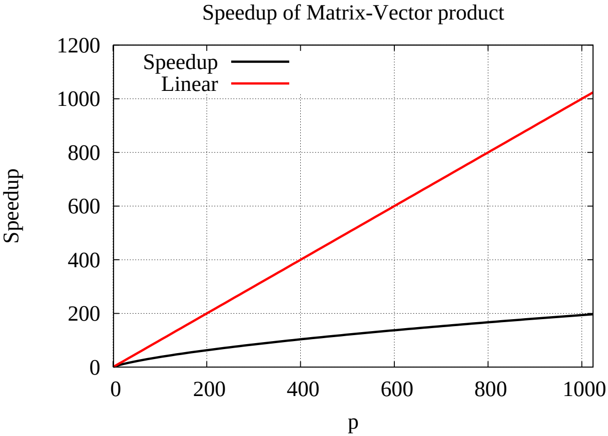
Analysis of Matrix-Vector Product

Memory constrained scalability (small scale).



Analysis of Matrix-Vector Product

Memory constrained scalability (large scale).



Analysis of Matrix-Vector Product: Conclusions

- The asymptotic iso-efficiency function for (this version of) matrix vector product is

$$W = \Theta(p^2).$$

- The algorithm is scalable.
- However, iso-efficiency scaling requires too much memory. The memory constrained speedup is only

$$S_p = O(\sqrt{p}).$$