Lecture 10: Performance Metrics

Shantanu Dutt
ECE Dept.
UIC
Acknowledgement

- Adapted from Chapter 5 slides of the text, by A. Grama w/ a few changes, augmentations and corrections
Topic Overview

- Sources of Overhead in Parallel Programs
- Performance Metrics for Parallel Systems
- Effect of Granularity on Performance
- Scalability of Parallel Systems
- Minimum Execution Time and Minimum Cost-Optimal Execution Time
- Asymptotic Analysis of Parallel Programs
- Other Scalability Metrics
Analytical Modeling - Basics

- A sequential algorithm is evaluated by its runtime (in general, asymptotic runtime as a function of input size).
- The asymptotic runtime of a sequential program is identical on any serial platform.
- The parallel runtime of a program depends on the input size, the number of processors, and the communication parameters of the machine.
- An algorithm must therefore be analyzed in the context of the underlying platform.
- A parallel system is a combination of a parallel algorithm and an underlying platform.
A number of performance measures are intuitive.

Wall clock time - the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble. But how does this scale when the number of processors is changed or the program is ported to another machine altogether?

How much faster is the parallel version? This begs the obvious followup question - what's the baseline serial version with which we compare? Can we use a suboptimal serial program to make our parallel program look better?

Raw FLOP count - What good are FLOP counts when they don't solve a problem?
Sources of Overhead in Parallel Programs

- If I use two processors, shouldn't my program run twice as fast?

- No - a number of overheads, including wasted computation, communication, idling, and contention cause degradation in performance.

The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.
Sources of Overheads in Parallel Programs

• Interprocess interactions: Processors working on any non-trivial parallel problem will need to talk to each other.

• Idling: Processes may idle because of load imbalance, synchronization, or serial components [i.e., data/control dependencies—recall the task dependency graph]

• Excess Computation: This is computation not performed by the serial version. This might be because the serial algorithm is difficult to parallelize so that, say, all processors are not searching the same soln. space as in a sequential algorithm—specifically, some are searching bad soln. spaces, or that some computations are repeated across processors to minimize communication.
Performance Metrics for Parallel Systems: Execution Time

• Serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer.

• The parallel runtime is the time that elapses from the moment the first processor starts to the moment the last processor finishes execution.

• We denote the serial runtime by $T_s$ and the parallel runtime by $T_P$. 
Performance Metrics for Parallel Systems: Total Parallel Overhead

- Let $T_{all}$ be the total time collectively spent by all the processing elements.

- $T_S$ is the serial time.

- Observe that $T_{all} - T_S$ is then the total time spent by all processors combined in non-useful work. This is called the total overhead.

- The total time collectively spent by all the processing elements
  $T_{all} = p \cdot T_P$ \hspace{1cm} (p is the number of processors).

- The overhead function ($T_o$) is therefore given by
  \[ T_o = p \cdot T_P - T_S \] (1)
Performance Metrics for Parallel Systems: Speedup

• What is the benefit from parallelism?

• Speedup ($S$) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with $p$ identical processing elements.
Performance Metrics: Example

• Consider the problem of adding $n$ numbers by using $n$ processing elements.

• If $n$ is a power of two, we can perform this operation in $\log n$ steps by propagating partial sums up a logical binary tree of processors.
Computing the globalsum of 16 partial sums using 16 processing elements. $\Sigma^j_i$ denotes the sum of numbers with consecutive labels from $i$ to $j$. 
Performance Metrics: Example (continued)

• If an addition takes constant time, say, $t_c$ and communication of a single word takes time $t_s + t_w$, we have the parallel time $T_P = \Theta(\log n)$

• We know that $T_S = \Theta(n)$

• Speedup $S$ is given by $S = \Theta(n / \log n)$
Performance Metrics: Speedup

• For a given problem, there might be many serial algorithms available. These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees

• For the purpose of computing speedup, we always consider the best sequential program as the baseline

• For the purpose of determining how effective our parallelization technique is, we can determine a pseudo-speedup w.r.t. the sequential version of the parallel algorithm
Performance Metrics: Speedup Example

- Consider the problem of parallel bubble sort.
- The serial time for bubblesort is 150 seconds.
- The parallel time for odd-even sort (efficient parallelization of bubble sort) is 40 seconds.
- The speedup would appear to be $150/40 = 3.75$. This is actually a pseudo-speedup.
- But is this really a fair assessment of the system?
- What if serial quicksort only took 30 seconds? In this case, the speedup is $30/40 = 0.75$. This is a more realistic assessment of the system.
Performance Metrics: Speedup Bounds

- Speedup can be as low as 0 (the parallel program never terminates).
- Speedup, in theory, should be upper bounded by $p$ - after all, we can only expect a $p$-fold speedup if we use times as many resources.
- A speedup greater than $p$ is possible only if each processing element spends less than time $Ts/p$ solving the problem.
- In this case, a single processor could be timeslided to achieve a faster serial program, which contradicts our assumption of fastest serial program as basis for speedup.
Performance Metrics: Superlinear Speedups

One reason for superlinearity is that the parallel version does less work than corresponding serial algorithm.

Searching an unstructured tree for a node with a given label, `S', on two processing elements using depth-first traversal. The two-processor version with processor 0 searching the left subtree and processor 1 searching the right subtree expands only the shaded nodes before the solution is found at the rightmost leaf node of the tree. The corresponding serial formulation expands the entire tree. It is clear that the serial algorithm does more work than the parallel algorithm.
Performance Metrics: Superlinear Speedups

For a large enough problem size \( n \), a single processor will have a large number of cache misses and probably more than a few page faults. These contribute to making \( T(n) \) large.

With \( P \) processors, and a reasonable input data distribution, each processor has access to the amount of data on a single serial processor. Hence it has very few cache misses and almost no page faults. Ignoring communication time (this is not unrealistic, since communication could be sparse and/or overlapped with computation), the
Performance Metrics: Superlinear Speedups

This means that each processor takes time to solve its portion of the problem in time less than \( T_p/N \) thus leading to superlinear speedup.

\[ T_p < T_p/N \]

\[ \Rightarrow T_p = T_p/N > p \]

However, if memory is "flat" (i.e., there is only main memory), and the data fits into the main memory of a single processor, then this superlinear speedup effect will not be observed.
Example: A processor with 64KB of cache yields an 80% hit ratio. If two processors are used, since the problem size/processor is smaller, the hit ratio goes up to 90%. Of the remaining 10% access, 8% come from local memory and 2% from remote memory.
Performance Metrics: Efficiency

• Efficiency is a measure of the fraction of time for which a processing element is usefully employed.

• Mathematically, it is given by

\[ E = \frac{S}{P}. \]  

• Following the bounds on speedup, efficiency can be as low as 0 and as high as 1.
Performance Metrics: Efficiency Example

- The speedup of adding numbers on processors is given by
- Efficiency is given by:

\[ S = \frac{n}{\log n} \]

\[ E = \Theta \left( \frac{n}{\log n} \right) \]

\[ = \Theta \left( \frac{1}{\log n} \right) \]
A parallel system is *cost-optimal* if the total parallel work $= \Theta(\text{sequential work})$

Consider the problem of adding numbers on processors.

We have, $T_P = \log n$ (for $p = n$).

The cost of this system is given by $p \, T_P = n \log n$.

Since the serial runtime of this operation is $\Theta(n)$, the algorithm is not cost optimal.
Effect of Granularity on Performance

- Often, using fewer processors improves cost-based performance of parallel systems.

- Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called *scaling down* a parallel system.

- A naive way of scaling down is to think of each processor in the original case as a virtual processor and to assign virtual processors equally to scaled down processors.

- Let $n$ be the original # of processors (proportionate to the max # of parallel tasks), and $p$ the scaled down number.

- Since the number of processing elements decreases by a factor of $n/p$, the computation at each processing element increases by a factor of $n/p$.

- The communication cost should also decrease (assuming uniform commun. traffic among the $n$ processors) by this factor since some of the virtual processors assigned to a physical processors might talk to each other. This is the basic reason for the improvement from increasing granularity.
Increasing Granularity: Example

- Consider the problem of adding $n$ numbers on $p$ processing elements such that $p < n$ and both $n$ and $p$ are powers of 2.

- Use the parallel algorithm for $n$ processors, except, in this case, we think of them as virtual processors.

- Each of the $p$ processors is now assigned $n / p$ virtual processors.

- The first $\log n - \log p$ steps can be thought of occurring among the $n/p$ virtual processors assigned to each physical processor, and thus do not require any communication.

- Subsequent $\log p$ steps of the original algorithm are simulated between the $p$ processing elements, and thus requires communication.
Increasing Granularity: Example (continued)

• The overall parallel execution time of this parallel system is $\Theta\left(\frac{n}{p} + \log p\right)$.

• The cost [i.e. total parallel work] is $\Theta\left(n + p\log p\right)$, which is asymptotically higher than the $\Theta\left(n\right)$ cost of adding $n$ numbers sequentially only if $p\log p > n$. Otherwise (which will the true in most “reasonable” cases), the parallel cost is asymptotically $\Theta(n)$, and is thus asymptotically cost optimal.
Increasing Granularity: Example (continued)

- More Details:
- The $p$ partial sums on $p$ processing elements can be added in time $\Theta(n/p)$.

A cost-optimal way of computing the sum of 16 numbers using four processing elements.
Increasing Granularity: Example (continued)

- More Details (contd.):
- The parallel runtime of this algorithm is
  \[ T_P = \Theta(n/p + \log p), \]
- The cost is \( \Theta(n + p \log p) \)
- This is cost-optimal, so long as \( n = \Omega(p \log p) \)!
Scalability of Parallel Systems

- How do we extrapolate performance from small problems and small systems to larger problems on larger configurations?
- Consider three parallel algorithms for computing an $n$-point Fast Fourier Transform (FFT) on 64 processing elements.

A comparison of the speedups obtained by the binary-exchange, 2-D transpose and 3-D transpose algorithms on 64 processing elements with $t_c = 2$, $t_w = 4$, $t_s = 25$, and $t_h = 2$.

- Clearly, it is difficult to infer scaling characteristics from observations on small datasets on small machines.
- Need analytical techniques
Scaling Characteristics of Parallel Programs

• The efficiency of a parallel program can be written as:

\[ E = \frac{S}{p} = \frac{T_s}{pT_p} \]

or

\[ E = \frac{1}{1 + \frac{T_o}{T_s}} \quad (pT_p = T_s + T_o) \]  \hspace{1cm} (4)

• \( T_s \) is the sequential time (note this is different from \( t_s \), the message setup time per hop)

• The total overhead function \( T_o \) is an increasing function of \( p \) (each processor has at least 1 unit of overhead—sending/recv’ing msgs, idling, etc.).
Scaling Characteristics of Parallel Programs

• For a given problem size (i.e., the value of $T_s$ remains constant), as we increase the number of processing elements, $T_o$ increases.

• The overall efficiency of the parallel program goes down. This is the case for all parallel programs.
Scaling Characteristics of Parallel Programs: Example

• Consider the problem of adding numbers on processing elements.

• We have seen that:

\[ T_P = \frac{n}{p} + 2 \log p \]  \hspace{1cm} (5)

\[ S = \frac{n}{p + 2 \log p} \]  \hspace{1cm} (6)

\[ E = \frac{1}{1 + \frac{2p \log p}{n}} \]  \hspace{1cm} (7)
Plotting the speedup for various input sizes gives us:

Speedup versus the number of processing elements for adding a list of numbers.

Speedup tends to saturate and efficiency drops as a consequence of Amdahl's law.
Digression: Serial Fraction $f$ and Amdahl’s Law

- The serial fraction $f$ of a parallel program is defined as:
  \[ f = \frac{T_{ser}}{W}. \]
  $T_{ser}$ is the amount of time needed to be spent serially irrespective of the # of procs. $W = T_s = \text{total amount of work}$.

  \[ T_P \geq f \times W + \frac{W - f \times W}{p} \]

  Note: $T_s = \Theta(W)$

- The rightmost term can be made arbitrarily small by throwing as many processors as the # of operations. Therefore, we have:
  \[ T_p \geq f \times W \Rightarrow S_p \leq \frac{W}{f \times W} = \frac{1}{f} \]

- Thus the speedup is limited by the inverse of the serial fraction. This is Amdahl’s Law.

- What is the serial fraction for $n \times n$ matrix multiplication?

- For parallel sum: $f = \frac{(\log n)}{n}$

- $f = (\log n)/n^3$
Scaling Characteristics of Parallel Programs

- Note that a cost-optimal parallel system has an efficiency of $\Theta(1)$.

- Scalability and cost-optimality are therefore related.

- A scalable parallel system can always be made cost-optimal if the number of processing elements and the size of the computation are chosen appropriately.
Isoefficiency Metric of Scalability

- For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down for all systems.

- For some systems, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.
Variation of efficiency: (a) as the number of processing elements is increased for a given problem size; and (b) as the problem size is increased for a given number of processing elements. The phenomenon illustrated in graph (b) is not common to all parallel systems.
Isoefficiency Metric of Scalability

• What is the rate at which the problem size must increase with respect to the number of processing elements $P$ to keep the efficiency fixed as $P$ increases?

• This rate determines the scalability of the system. The slower this rate, the better.

• Before we formalize this rate, we define the problem size $W$ as the asymptotic number of operations associated with the best serial algorithm to solve the problem.
Isoefficiency Metric of Scalability

- We can write parallel runtime as:
  \[ T_P = \frac{W + T_o(W, p)}{p} \]  

- The resulting expression for speedup is:
  \[ S = \frac{W}{T_P} = \frac{Wp}{W + T_o(W, p)}. \]  

- Finally, we write the expression for efficiency as:
  \[ E = \frac{S}{p} = \frac{W}{W + T_o(W, p)} = \frac{1}{1 + \frac{T_o(W, p)}{W}}. \]

This comes from: \( pT_P = T_s + T_o = W + T_o \Rightarrow T_p = \frac{(W+T_o)}{p} \)

This is the same as saying \( E = \frac{1}{1 + \frac{T_o}{T_s}} \) that had been derived earlier.
Isoefficiency Metric of Scalability

- For scalable parallel systems, efficiency can be maintained at a fixed value (between 0 and 1) if the ratio $T_o / W$ is maintained at a constant value.
- For a desired value $E$ of efficiency,
  \[ E = \frac{1}{1 + T_o(W, p)/W}, \]
  \[ \frac{T_o(W, p)}{W} = \frac{1 - E}{E}, \]
  \[ W = \frac{E}{1 - E} T_o(W, p). \] (11)

- If $K = E / (1 - E)$ is a constant depending on the efficiency to be maintained, since $T_o$ is a function of $W$ and $p$, we have
  \[ W = KT_o(W, p). \] (12)
Isoefficiency Metric of Scalability

- The problem size $W$ can usually be obtained as a function of $p$ by algebraic manipulations to keep efficiency constant.

- This function is called the isoefficiency function.

- This function determines the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processing elements.
Isoefficiency Metric: Example

- The overhead function for the problem of adding \( n \) numbers on \( p \) processing elements is approximately \( 2p \log p \).

- Substituting \( T_o \) by \( 2p \log p \), we get

\[
W = K 2p \log p. \tag{13}
\]

Thus, the asymptotic isoefficiency function for this parallel system is \( \Theta(p \log p) \).

- If the number of processing elements is increased from \( p \) to \( p' \), the problem size (in this case, \( n \)) must be increased by a factor of \( (p' \log p') / (p \log p) \) to get the same efficiency as on \( p \) processing elements.

- So going from, say, 4 to 8 processors, the problem size needs to increase by a factor of \( (8 \log 8)/(4 \log 4) = 3 \), i.e., more than a linear factor of 2, to maintain the same efficiency as seen for \( p=4 \).
Isoefficiency Metric: Example

Consider a more complex example where \( T_0 = p^{3/2} + p^{3/4}W^{3/4} \)

- Using only the first term of \( T_0 \) in Equation 12, we get
  \[
  W = Kp^{3/2}.
  \]  
  (14)

- Using only the second term, Equation 12 yields the following relation between \( W \) and \( p \):
  \[
  W = Kp^{3/4}W^{3/4}
  
  W^{1/4} = Kp^{3/4}
  
  W = K^4p^3
  \]  
  (15)

- The larger of these two asymptotic rates determines the isoeficiency. This is given by \( \Theta(p^3) \)
Cost-Optimality and the Isoefficiency Function

• A parallel system is cost-optimal if and only if

\[ pTP = \Theta(W). \]  

(16)

• From this, we have:

\[ W + T_o(W, p) = \Theta(W) \]  

(17)

\[ T_o(W, p) = O(W) \]  

(18)

\[ W = \Omega(T_o(W, p)) \]

• If we have an isoefficiency function \( f(p) \), then it follows that the relation \( W = \Omega(f(p)) \) must be satisfied to ensure the cost-optimality of a parallel system as it is scaled up.

• Thus a cost-optimal and constant-efficiency systems intersect at:

\[ W = \Theta(f(p)) \], i.e., \( W = \Theta(T_o) \iff T_o = \Theta(T_s) \)

• Note that a system is scalable only if \( f(p) \) is not too large \((< p^2)\)
Lower Bound on the Isoefficiency Function

• For a problem consisting of \( W \) units of work, no more than \( W \) processing elements can be used cost-optimally.

• The problem size must increase at least as fast as \( \Theta(p) \) to maintain fixed efficiency; hence, \( \Omega(p) \) is the asymptotic lower bound on the iso-efficiency function.

• The reasons for both of the above is \( T_0 = \Omega(p) \) (each processor will incur at least 1 unit of overhead, e.g., just from initial data distribution, or having to send/receive at least 1 message, since the processors will not be working in complete isolation).
Minimum Execution Time and Minimum Cost-Optimal Execution Time

Often, we are interested in the minimum time to solution.

- We can determine the minimum parallel runtime $T_p^{\text{min}}$ for a given $W$ by differentiating the expression for $T_p$ w.r.t. $p$ and equating it to zero.

\[
\frac{d}{dp} T_p = 0
\]

(19)

- If $p_o$ is the value of $p$ as determined by this equation, $T_p(p_o)$ is the minimum parallel time.
Minimum Execution Time: Example

Consider the minimum execution time for adding $n$ numbers.

$$T_P = \frac{n}{p} + 2 \log p. \quad (20)$$

Setting the derivative w.r.t. $p$ to zero, we have $p = n/2$. The corresponding runtime is

$$T_P^{\text{min}} = 2 \log n. \quad (21)$$

(One may verify that this is indeed a min by verifying that the second derivative is positive).

Note that at this point, the formulation is not cost-optimal.
Minimum Cost-Optimal Parallel Time

• Let $T_P^{cost\_opt}$ be the minimum cost-optimal parallel time.

• If the isoeficiency function of a parallel system is $\Theta(f(p))$, then a problem of size $W$ can be solved cost-optimally if and only if $W = \Omega(f(p))$.

• In other words, for cost optimality, $p = O(f^{-1}(W))$.

• For cost-optimal systems, $T_P = \Theta(W/p)$, therefore,

\[
T_P^{cost\_opt} = \Omega \left( \frac{W}{f^{-1}(W)} \right). \quad (22)
\]
Minimum Cost-Optimal Parallel Time: Example

Consider the problem of adding $n$ numbers.

- The isoefficiency function $f(p)$ of this parallel system is $\Theta(p \log p)$.
- For cost-optimality $W = n = f(p) \Rightarrow n = p \log p$
- From this, we have $p \approx \frac{n}{\log n}$ (\(\log n = \log p - \log \log p \sim \log p\))
- At this processor count, the parallel runtime is: \(\text{Note: } T_P = \frac{n}{p} + \log p\)

\[
T_P^{\text{cost-opt}} = \log n + \log \left(\frac{n}{\log n}\right) \quad (23)
\]

\[
= 2 \log n - \log \log n.
\]

- Note that both $T_P^{\text{min}}$ and $T_P^{\text{cost-opt}}$ for adding $n$ numbers are $\Theta(\log n)$. This may not always be the case.
Other Scalability Metrics

• A number of other metrics have been proposed, dictated by specific needs of applications.

• For real-time applications, the objective is to scale up a system to accomplish a task in a specified time bound.

• In memory constrained environments, metrics operate at the limit of memory and estimate performance under this problem growth rate.
Other Scalability Metrics: Scaled Speedup

- Various types of scaled (scaling of problem size) speedup:
  - Speedup obtained when the problem size is increased linearly with the number of processing elements (*processor scaling*).
  - If the aggregate memory grows linearly in $p$, scaled speedup increases problem size to fill memory (*memory scaling*).
  - The size of the problem is increased subject to an upper-bound on parallel execution time (*time scaling*).

- If scaled speedup is close to linear, the system is considered scalable.

- If the isoefficiency is near linear, processor-scaled speedup curve is close to linear as well.
Scaled Speedup: Example

- The serial runtime of multiplying a matrix of dimension \( n \times n \) with a vector is \( t_c n^2 \).

- For a given parallel algorithm, say:

\[
S = \frac{t_c n^2}{\frac{t_c n^2}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}}
\]

- Total memory requirement of this algorithm is \( \Theta(n^2) \).
Consider the case of memory-constrained scaling.

- We have \( m = \Theta(n^2) = \Theta(p) \) or \( n^2 = c \cdot p \), for some constant \( c \)

- Memory constrained scaled speedup is given by

\[
S' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + t_w \sqrt{c \times p}}
\]

- \( S' = O(\sqrt{p}) \)

or

- This is not a particularly scalable system
Consider the case of time-constrained scaling.

- We have $T_P = O(n^2/p)$.
- Since this is constrained to be constant, $n^2 = O(p)$.
- Note that in this case, time-constrained speedup is identical to memory constrained speedup.
- This is not surprising, since the memory and time complexity of the operation are identical.
Scaled Speedup: Example

- The serial runtime of multiplying two matrices of dimension $n \times n$ is $t_c n^3$.

- The parallel runtime of a given algorithm is:
  \[
  T_P = t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}
  \]

- The speedup $S$ is given by:
  \[
  S = \frac{t_c n^3}{t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}}
  \]  
  (25)
Consider memory-constrained scaled speedup.

- We have memory complexity $m = \Theta(n^2) = \Theta(p)$, or $n^2 = c \times p$.

- At this growth rate, scaled speedup $S'$ is given by:

$$S' = \frac{t_c(c \times p)^{1.5}}{t_c \frac{(c \times p)^{1.5}}{p} + t_s \log p + 2t_w \frac{c \times p}{\sqrt{p}}} = O(p)$$

- Note that this is scalable.
Consider time-constrained scaled speedup.

- We have $T_P = O(1) = O(n^3/p)$, or $n^3 = c \times p$.

- Time-constrained speedup $S''$ is given by:

$$S'' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + 2t_w \frac{(c \times p)^{2/3}}{\sqrt{p}}} = O(p^{5/6})$$

- Memory constrained scaling yields better performance.