Analytical Modeling of Parallel Systems
Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

Topic Overview

• Sources of Overhead in Parallel Programs
• Performance Metrics for Parallel Systems
• Effect of Granularity on Performance
• Scalability of Parallel Systems
• Asymptotic Analysis of Parallel Programs
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.
Analytical Modeling - Basics

• 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** of 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

• **Raw FLOPS (FLoating-point Operations Per Second)** - How good is FLOPS measure when it 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*.

- **Excess Computation**: This is computation *not performed by the serial version*. This might be because the *serial algorithm is difficult to parallelize*, 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 spend by all processors combined in non-useful work. This is called the total overhead.

• The total time collectively spent by all the processing elements $T_{all} = p T_P$ ($p$ is the number of processors).

• The overhead function ($T_o$) is therefore given by

$$T_o = p T_P - T_S$$ (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.

$$S = \frac{T_S}{T_P}$$
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.
Performance Metrics: Example

(a) Initial data distribution and the first communication step

(b) Second communication step

(c) Third communication step

(d) Fourth communication step

(e) Accumulation of the sum at processing element 0 after the final communication
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.
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 $150/40 = 3.75$.

• 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 \( T_s/p \) 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. The corresponding **serial formulation expands the entire tree**. It is clear that **the serial algorithm does more work than the parallel algorithm**.
Amdahl’s Law

- All programs contain parts that are naturally sequential ($\beta$) and the other fraction is naturally parallel ($1 - \beta$).

- Speedup of the algorithm is limited by the naturally sequential part of the algorithm. Amdahl’s Law defines theoretically possible speedup, ignoring overhead and communication costs.

- The serial part of the program can be computed in $\beta T_s$, and the parallel program in time $(1 - \beta) T_s/p$. Then $T_p = \beta T_s + (1 - \beta) T_s/p$.

- The speedup is then $S \leq \frac{T_s}{\beta T_s + (1 - \beta) T_s/p} = \frac{p}{\beta p + (1 - \beta)}$

- Example: $\beta = 0.1$, $p=100 \Rightarrow S \leq 9.1$
  $\beta = 0.1$, $p=1000 \Rightarrow S \leq 9.91$
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

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

• Efficiency is given by

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

\[ = \Theta \left( \frac{1}{\log n} \right) \]
Parallel Time, Speedup, and Efficiency Example

Consider the problem of **edge-detection in images**. The problem requires us to apply a **3 x 3 template** to each pixel. If each multiply-add operation takes time $t_c$, the serial time for an $n \times n$ image is given by $T_S = 9t_c n^2$.

![Example of edge detection: (a) an 8 x 8 image; (b) typical templates for detecting edges; and (c) partitioning of the image across four processors with shaded regions indicating image data that must be communicated from neighboring processors to processor 1.](image)
• One possible parallelization partitions the image equally into vertical segments, each with $\frac{n^2}{p}$ pixels.

• The boundary of each segment is $2n$ pixels. This is also the number of pixel values that will have to be communicated. This takes time $2(t_s + t_w n)$.

• Templates may now be applied to all $\frac{n^2}{p}$ pixels in time $9 t_c n^2 / p$. 
Parallel Time, Speedup, and Efficiency Example (continued)

- The total time for the algorithm is therefore given by:

\[ T_P = 9t_c \frac{n^2}{p} + 2(t_s + t_w n) \]

- The corresponding values of speedup and efficiency are given by:

\[ S = \frac{9t_c n^2}{9t_c \frac{n^2}{p} + 2(t_s + t_w n)} \]

and

\[ E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}. \]
Cost of a Parallel System

- **Cost** is the product of parallel runtime and the number of processing elements used \((p \times T_P)\).

- Cost reflects the sum of the **time** that each processing element **spends solving the problem**.

- A parallel system is said to be **cost-optimal** if the cost of solving a problem on a parallel computer is asymptotically identical to serial cost.

- Since \(E = T_S / p T_P\), for cost optimal systems, \(E = O(1)\).

- Cost is sometimes referred to as **work** or **processor-time product**.
Cost of a Parallel System: Example

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 $p \cdot T_P = n \log n$.
- 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 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.

• Since the number of processing elements decreases by a factor of \( \frac{n}{p} \), the computation at each processing element increases by a factor of \( \frac{n}{p} \).

• The communication cost should not increase 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 building granularity.
Building 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 \( \log p \) of the \( \log n \) steps of the original algorithm are simulated in \( (n / p) \log p \) steps on \( p \) processing elements.

• Subsequent \( \log n - \log p \) steps do not require any communication.
Building Granularity: Example (continued)

(a) Four processors simulating the first communication step of 16 processors
Building Granularity: Example (continued)

(b) Four processors simulating the second communication step of 16 processors
Building Granularity: Example (continued)

Substep 1

(c) Simulation of the third step in two substeps

Substep 2

(d) Simulation of the fourth step

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

• The cost is \( \Theta (n \log p) \), which is asymptotically higher than the \( \Theta (n) \) cost of adding \( n \) numbers sequentially. Therefore, the parallel system is not cost-optimal.
Can we build granularity in the example in a cost-optimal fashion?

- Each processing element locally adds its $n / p$ numbers in time $\Theta(n / p)$.
- The $p$ partial sums on $p$ processing elements can be added in time $\Theta(\log p)$.

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

- The parallel runtime of this algorithm is

\[ T_P = \Theta\left(\frac{n}{p} + \log p\right), \]  

(3)

- 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.
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

\[ E = \frac{1}{1 + \frac{T_o}{T_S}}. \]  

(4)

• The total overhead function \( T_o \) is an increasing function of \( p \).

• 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)

\[ S = \frac{n}{\frac{n}{p} + 2 \log p} \]  \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
Scaling Characteristics of Parallel Programs

- Total overhead function $T_o$ is a function of both problem size ($n \to T_s$) and the number of processing elements $p$.

- In many cases, $T_o$ grows sublinearly with respect to $T_s$.

- In such cases, the efficiency increases if the problem size is increased keeping the number of processing elements constant.

- For such systems, we can simultaneously increase the problem size and number of processors to keep efficiency constant.

- We call such systems scalable parallel systems.
Scaling Characteristics of Parallel Programs

- Recall that \textit{cost-optimal parallel systems have an efficiency of }\Theta(1).

- \textbf{Scalability} and \textit{cost-optimality} are therefore related.

- A \textit{scalable parallel system can always be made cost-optimal} if the number of processing elements and the size of the computation are chosen appropriately.

- For a \textit{given problem size}, as we increase the number of \textit{processing elements}, the overall efficiency of the parallel system goes down for all systems.
Isoefficiency Metric of Scalability

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 to keep the efficiency fixed?

• 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:

\[ T_P = \frac{W + T_o(W, p)}{p} \]  

(8)

• The resulting expression for speedup is

\[ S = \frac{W}{T_P} \]

\[ = \frac{Wp}{W + T_o(W, p)}. \]  

(9)

• Finally, we write the expression for efficiency as

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

\[ = \frac{W}{W + T_o(W, p)} \]

\[ = \frac{1}{1 + T_o(W, p)/W}. \]
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,

\[
E = \frac{1}{1 + \frac{T_o(W, p)}{W}},
\]

\[
\frac{T_o(W, p)}{W} = \frac{1 - E}{E},
\]

\[
W = \frac{E}{1 - E} T_o(W, p).
\]  \hspace{1cm} (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

\[
W = KT_o(W, p).
\]  \hspace{1cm} (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.

- If $W$ needs to grow only linearly with respect to $p$, then the parallel system is highly scalable. On the other hand, if $W$ might need to grow, e.g., as an exponential function of $p$ to keep the efficiency from dropping as $p$ increases. Such parallel systems are poorly scalable.
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

$$W = K2p \log p.$$  \hspace{1cm} (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.
Isoefficiency Metric: Example

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

- Using only the first term of \( T_o \) in Equation 12, we get

\[
W = Kp^{3/2}. \quad (14)
\]

- Using only the second term, Equation 12 yields the following relation between \( W \) and \( p \):

\[
\begin{align*}
W &= Kp^{3/4}W^{3/4} \\
W^{1/4} &= Kp^{3/4} \\
W &= K^4p^3
\end{align*}
\quad (15)
\]

- The larger of these two asymptotic rates determines the isoefficiency. This is given by \( \Theta(p^3) \)
Minimum Execution Time

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

- We can determine the minimum parallel runtime \( T^\text{min}_P \) 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 \( p_0 \) is the value of \( p \) as determined by this equation, \( T_P(p_0) \) is the minimum parallel time.
Minimum Execution Time: Example

Consider the minimum execution time for adding \( n \) numbers.

\[
T_P = \frac{n}{p} + 2 \log p. \tag{20}
\]

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

\[
T_P^{\text{min}} = 2 \log n. \tag{21}
\]

(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.
Consider the problem of sorting a list of \( n \) numbers. The fastest serial programs for this problem run in time \( \Theta(n \log n) \). Consider four parallel algorithms, A1, A2, A3, and A4 as follows:

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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>( n^2 )</td>
<td>( \log n )</td>
<td>( n )</td>
<td>( \sqrt{n} )</td>
</tr>
<tr>
<td>( T_P )</td>
<td>1</td>
<td>( n )</td>
<td>( \sqrt{n} )</td>
<td>( \sqrt{n \log n} )</td>
</tr>
<tr>
<td>( S )</td>
<td>( n \log n )</td>
<td>( \log n )</td>
<td>( \sqrt{n \log n} )</td>
<td>( \sqrt{n} )</td>
</tr>
<tr>
<td>( E )</td>
<td>( \frac{\log n}{n} )</td>
<td>1</td>
<td>( \frac{\log n}{\sqrt{n}} )</td>
<td>1</td>
</tr>
<tr>
<td>( pT_P )</td>
<td>( n^2 )</td>
<td>( n \log n )</td>
<td>( n^{1.5} )</td>
<td>( n \log n )</td>
</tr>
</tbody>
</table>
Asymptotic Analysis of Parallel Programs

- If the metric is **speed**, algorithm A1 is the best, followed by A3, A4, and A2 (in order of increasing $T_P$).

- In terms of **efficiency**, A2 and A4 are the best, followed by A3 and A1.

- In terms of **cost**, algorithms A2 and A4 are cost optimal, A1 and A3 are not.

- It is important to identify the objectives of analysis and to **use appropriate metrics**!