Outline

- Move away from asymptotic analysis
- Account for real machine behavior
- Communication time
- Idle time/load imbalance
Performance Analysis

- General formulas for speedup & efficiency
- Amdahl’s Law
- Gustafson’s Law (scaled speedup)
- Karp-Flatt Metric
- Isoefficiency
  - Design of scalable algorithms

Speedup & Efficiency

\[
\text{Speedup } \psi = \frac{\text{Sequential execution time}}{\text{Parallel execution time}} \\
\text{Efficiency } \varepsilon = \frac{\text{Sequential execution time}}{\text{Processors} \times \text{Parallel execution time}} \\
\varepsilon = \frac{\text{Speedup}}{\text{Processors}}
\]
Execution Time Components

- Inherently sequential computations (c.f. depth of the computation graph) \( \sigma \)
- Perfectly parallelizable computations \( \phi \)
- Overhead (may also depend on the number of processors, \( p \)) \( \kappa \)

\[ \psi \leq \frac{\sigma + \phi}{\phi + \kappa} \]

Amdahl’s Law

- Ignore \( \kappa \) for now (only makes speedup worse: Amdahl is optimistic)
- Bounds on speedup
- Inherently sequential fraction \( f = \frac{\sigma}{\sigma + \phi} \)

\[ \psi \leq \frac{1}{f + \frac{1-f}{p}} \]
Example 1

- 95% of a program’s execution time is spent in a tight loop that can be parallelized with `#omp pragma parallel for`. What is the maximum speedup that can be achieved on a 16 core machine?
  \[
  \psi_{16} \leq \frac{1}{0.05 + \frac{0.95}{16}} = 9.14
  \]

- What is the max speedup possible on any machine?
  \[
  \psi_{\infty} \leq \frac{1}{0.05 + \frac{0.95}{\infty}} = 20
  \]

Limitations

- Why did we just not give up?

- Why did people (the supercomputing community) continue to write codes that run on very large number of processors?
Recap

- Inherently sequential computations (c.f. depth of the computation graph) \( \sigma(n) \)
- Perfectly parallelizable computations \( \phi(n) \)
- Overhead (may also depend on the number of processors, \( p \)) \( \kappa(n, p) \)

\[
\psi(n, p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n) + \kappa(n, p)}
\]

Amdahl’s Law

- Inherently sequential fraction:

\[
f(n) = \frac{\sigma(n)}{\sigma(n) + \phi(n)}
\]

\[
\psi(n, p) \leq \frac{1}{f(n) + \frac{1 - f(n)}{p}}
\]
Gustafson-Barsis’ Law

- Let \[ s(n, p) = \frac{\sigma(n)}{\sigma(n) + \phi(n)/p} \]
- Then we can show that

\[ \psi(n, p) \leq p - (p - 1)s \]

Example 2

- An application running on 10 processors spends 3% of its execution time doing serial work. What is its scaled speedup?

\[ \psi(n, p) \leq 10 - (10 - 1)0.03 = 10 - 0.27 = 9.73 \]

Except that 9 don’t do the serial fraction
Execution on 1 processor takes 10 times longer
Karp Flatt Metric

- Both Amdahl and Gustafson-Barsis ignore the overhead term, \( \kappa(n,p) \)
- Overestimate the (scaled) speedup
- Karp & Flatt proposed a more realistic (and empirical) metric
- Allows to account for decreasing speedup

Empirical Serial Fraction

- Start with Amdahl's law. \( \psi(n,p) \leq \frac{1}{\frac{f(n)}{p} + 1 - f(n)} \)
- multiply top & bottom by \( p \),
- collect the \( f(n,p) \) terms together and take them to lhs – solve for \( f(n,p) \), assuming you know \( \psi \)

\[
e(n,p) = \frac{\psi(n,p)}{p} - 1 = \frac{1}{\psi(n,p)} - 1
\]

\[
e(n,p) = \frac{\psi(n,p)}{p} - 1 = \frac{1}{\psi(n,p)} - 1
\]

\[
e(n,p) = \frac{p}{\psi(n,p)} - 1 = \frac{1}{\psi(n,p)} - 1
\]
Example 1

<table>
<thead>
<tr>
<th>p</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ψ</td>
<td>1.8</td>
<td>2.5</td>
<td>3.1</td>
<td>3.6</td>
<td>4.0</td>
<td>4.4</td>
<td>4.7</td>
</tr>
<tr>
<td>e</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

- Why is speedup only 4.7 on 8 processors?

Example 2

<table>
<thead>
<tr>
<th>p</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ψ</td>
<td>1.9</td>
<td>2.6</td>
<td>3.2</td>
<td>3.7</td>
<td>4.1</td>
<td>4.5</td>
<td>4.7</td>
</tr>
<tr>
<td>e</td>
<td>0.07</td>
<td>0.075</td>
<td>0.08</td>
<td>0.085</td>
<td>0.09</td>
<td>0.095</td>
<td>0.1</td>
</tr>
</tbody>
</table>

- Why is speedup only 4.7 on 8 processors?
- e is steadily increasing. Overhead is the culprit
Isoefficiency Metric

- Main goal is to quantify the relative scaling of problem size and number of processors
- And account for the overhead term quantitatively
  - To maintain “good” performance
  - What is good?
  - Maintain constant efficiency = linear speedup

Isoefficiency Analysis

- Start with speedup formula
- Identify the total overhead
  - Non-essential work done by the parallel program
- Do algebra so that efficiency = constant
- Determine the relationship between the sequential execution time (work) and overhead
Problem size

- Isoefficiency analysis studies
  - How problem size should increase
  - As \( p \) is increased
  - To keep efficiency constant
- What is problem size?
  - Not a parameter like \( n \) as in most analyses
  - But rather the work of the best sequential algorithm, \( W = T(n,1) = \sigma(n) + \phi(n) \)

General Approach

- Express overhead as function of \( n \) and \( p \).
- Isoefficiency relation:
  \[ W(n) = KT_o(n,p) \]
- Massage this to remove \( n \) from the rhs
  \[ W(n) = f(p) \]
- Function \( f(p) \) is isoefficiency function
Scalability

- The smallest growing isoefficiency function is the most scalable.
- Factors that impose a lower bound on $f(p)$
  - Communication costs, and load imbalance
  - Memory bounds
  - Degree of parallelism in the application itself

Isoefficiency relation

- Remember, $T(n, p) = \sigma(n) + \frac{\phi(n)}{p} + \kappa(n, p)$
- Overhead = useless work:
  - Multiply $T(n, p)$ by $p$, remove useful work:
    $T_o(n, p) = (p - 1)\sigma(n) + p\kappa(n, p)$
  - Modify speedup equation to use $T_o$ rather than $\kappa$
    $\psi(n, p) = \frac{p}{1 + \frac{T_o(n, p)}{W}}$
- For efficiency = constant
  $T(n, 1) = W = KT_o(n, p) = KT_o(W, p)$
Example 1: Reduction

- **Sequential**  \[ T(n,1) = W = n \]
- **Parallel**  \[ T(n,p) = (n/p) + \log(p) \]
- **Overhead**  \[ T_o(W,p) = p \log(p) \]
- **Isoefficiency function:**  \[ p \log(p) \]
- **How should work increase as \( p \) increases?**

Example 2

- **Complicated overhead function**
- **Overhead**  \[ T_o(W,p) = p^{3/2} + pW^{3/4} \]
- **Separate the different parts, analyze each one and take the worst case**
  - **First function:**  \[ W = \Theta(p^{3/2}) \]
  - **Second**  \[ W = Kw^{3/4} \text{ i.e., } W = \Theta(p^4) \]
- **Second one dominates. The work must grow as the 4th power of the number of processors, to maintain linear speedup.**
Work optimality & Brent

- Work optimal:

\[ pT(n, p) = \Theta(W) \]

- This yields

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

- Isoefficiency implies work optimality

Memory Constraints

- We assumed that as \( p \), and hence, \( W(n) \) scales, memory required for the larger problem is adequate.
- This may not hold – on supercomputers (and memory constrained architectures, like GPUs)
- Define \( M(x) \) as the memory required for a problem size that does \( x \) amount of work. e.g., for matrix multiplication \( M(x) = x^{2/3} \)
- Scalability function: per node memory growth in order to retain iso-efficiency: \( g(n) = M(f(p))/p \)
Parallelism Constraints

- Even for an ideal parallel system with no overhead, and no serial part, the parallel part may not be “perfectly parallel”
- Inherent dependences between computations (degree of concurrency) expressed as a function, \( C(W) \) of the work

Matrix Multiplication

- CUDA style – practical benefits for you