CSCE 626
Performance Analysis Methodology
Introduction

• This lecture discusses how to properly design an experimental setup, measure and analyze the performance of parallel algorithms you develop.

• The methodology analysis applies to both the MPI and OpenMP implementations.
Experimental Setup - Parameters

Choose the problem size.
• Understand the machine architecture, cache size, etc.
• Choose a problem size that is larger than cache size.
  • Ensure this holds for all processor counts studied.
  • Otherwise, may lead to super linear speedup.

Generate the input data aware of any input sensitivity.
• Does the complexity of the algorithm depend on the nature of the input?
  • For example, sorting algorithm often depend on the input data.
  • Varying data type may effect running time.
• Choose test cases that cover average case, best case and worst case performance.
Choose the number of processors.
• Based on available resources.
• Depends on the nature of the algorithm.
  • Generally convenient to use powers of 2 (i.e., 1, 2, 4, 8, 16 …).

Use an appropriate timing mechanism.
• Low overhead of timer (PAPI, system specific).
• Measure within clock resolution.
• Running times should be at least a second or two.
• Place timers around actual algorithm (e.g., don’t measure input initialization).
Experimental Setup

Repeat experiments to increase confidence in results.

- Variability in the system will often lead to variability in the measured running time.
  - Shared resource – other users affect your jobs.
  - Operating system processes are non-deterministic.
  - Allocation of nodes to a job changes communication costs.

- If possible, repeatedly invoke executable in same batch job.

- Confidence Interval.
  - Used to indicate the reliability of the estimate.
  - Qualified by a particular confidence level usually specifies as a percentage (e.g., 95%).
  - Especially useful when comparing the relative performance of two candidate implementations or algorithms.
Unclear from the first graph which algorithm performs better. Overlapping confidence intervals.

In second graph, Algorithm 1 clearly performs better. Disjoint confidence intervals.
Experimental Setup

Computing the Confidence Interval.
Usually computed for a confidence level of 95%.

*An observation lies between the computed interval around the mean with 0.95 probability.*

1. Repeat the experiment 32 times.
2. Compute the mean (m) and standard deviation (sd) of observations.
3. Compute $k = \alpha \times \frac{sd}{\sqrt{32}}$ ($\alpha = 1.96$ for confidence interval of 95%).

*Observation lies in interval $[m-k, m+k]$ with 95% probability.*

Can “tightly” interval by gathering more samples (i.e., runs).
Experimental Setup - Speedup

**Speedup**

Ratio of time taken by *the best sequential algorithm* to time taken by a parallel algorithm for a given value of P.

- Ideal speedup is linear (equal to number of processors).

- Super-linear speedup generally due to:
  - External Factors (e.g., cache effects where input fits in cache for larger P).
  - Algorithmic Effects (e.g., algorithm runs same function with different inputs until one succeeds so that the parallel one finds the solution faster).
Example Plot - Speedup

![Graph showing speedup vs. number of processors. The blue line represents the linear speedup, and the red line represents the experimental speedup. The graph indicates a sub-linear speedup with increasing numbers of processors.]
Experimental Setup - Scalability

**Strong scaling**

Analyzes how the solution time varies with the number of processors for a fixed problem size.

- Even with $P=1$, not same as speedup (base is $T_{P1}$ vs. $T_{seq}$). *Speedup at $P=1$ is typically $< 1$, whereas scalability is 1.*

- Scaling base need not be $P=1$. *For larger systems, may start at 128, 1K, 10K, etc.*

- Increasing $P$ decreases $N/P$, which can affect behavior. *Examples:*
  - Cache utilization may change (e.g., super-linear curves).
  - Ratio of communication to computation can change.
Example Plot - Strong Scaling

Note in this case, we start the scalability study at P=16.
Experimental Setup - Scalability

Weak Scaling

Analyzes how the solution time varies with the number of processors for a fixed problem size per processor.

- Often most viable option for larger ranges of P. Fixed problem sizes may prove to large for small processor counts (or too small for large P).

- Important to many users who employ parallelism to solve larger problems, not just the same problem faster.

- Ideal weak scaling curve is constant (flat).
Example Plot - Weak Scaling

![Graph showing weak scaling with scalability on the y-axis and processors on the x-axis. The graph compares constant and experimental lines.](image)
Finding Big-O constants (sequentially)

Recall the definition of Big-O:

\[ f(n) = O(g(n)) \text{ means } 0 \leq f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0 \]

How to experimentally determine c and \( n_0 \):

1. Plot ratio of experimental time to the theoretical complexity of the sequential algorithm (y-axis), varying the input size (x-axis).

2. The values of \( n_0 \) and c can be determined from the plot.
   - \( n_0 \) is the value of n (x-axis) where the plot becomes horizontal.
   - C is the value of ratio (y-axis) after it stabilizes (after \( n_0 \)).
Finding Big-O constant

At point where plot flattens:

\[ n_0 = 25 \times 10^6, \ c = .002 \]

What does it tell us about the proposed theoretical bound if:

- The plot continues upwards?
- The plot trends downward?
Finding Big-O constants (In Parallel)

Analysis in parallel is more complex. *Both input size and the number of processors may vary.*

For a fixed $P$, determine values of $n_0$ and $c$.
- Use same approach as in sequential case.
- Repeat for multiple values of $P$ (at least 3).
- Problem size per processor should be greater than $n_0$ for sequential algorithm.
Printing Output in Parallel

All MPI processors / OpenMP threads share common stdout. *Unless synchronized, order of output is non-deterministic.*

One simple approach for debugging:

```c++
for (int i = 0; i < num_procs; ++i)
{
    if (i == my_id)
    {
        std::cout << "Hello World from " << my_id << "\n";
        MPI_Barrier();
    }
}
```
Previous discussion assumed you measured the complete running time of the algorithm. 

*We may also want to measure and analyze, separately:*

- The steps/phases of the algorithm.
- The communication and computation.

Why?

Measuring individual components is useful to pinpoint inefficiencies and bottlenecks sub-steps of an algorithm.

*Prioritize optimization and refinement efforts.*
Summary

- Experimental setup
- Timing mechanism
- Computing Big-O constants
- Displaying results
- Measure & analyze components of algorithm to better understand performance.