CIS 431/531
Intro to Parallel Computing

Lecture 2

What is Performance?

Instructor

Instructor Contact Information

Instructor: Jee Whan Choi
Office: 328 Deschutes
Lecture: MWF 12:00 - 12:50
Office Hours: MWF 13:00 - 14:00, or by appointment

Contact: jeec [at] uoregon.edu

Course website: https://jeewhanchoi.github.io/uocis531

Grading

	Undergraduate	Graduate
Quiz	10%	5%
Homeworks	40%	35%
Project	40%	40%
Research Paper	0%	10%
Presentation	10%	10%

Workload

Quiz - randomly, on the topics from previous lectures

Homeworks - hands-on experience on the covered topics

Group Project - Term-long project on a topic of your choosing

Research Paper - short survey on a topic of your choosing

- graduate students only, but undergraduates are welcome to do one as well (extra credit)

Presentation - part of the group project (but another 10% because of its important)

NO EXAMS!

Group Project

Deliverables

1. **Proposal** (*Due October 13th*) A written proposal describing the area and problems to be explored, the motivation for choosing this area, possible directions of investigation, and expected results. This should be 4-5 pages (not counting references, if any).
2. **Final Presentation** (*November 27th & November 29th & December 1st*) A final presentation and demonstration of your accomplishments is required. It should be approximately 15 minutes, with every member participating in the presentation.
3. **Source Code** (*Due December 4th*) The source code for your project should be version controlled using *git* or *BitBucket*, and your work should be committed frequently to track your progress.
4. **Report** (*Due December 6th*) A final report summarizing your attempts, accomplishments, and what was learned from the project. The report should be approximately 10 pages (not counting references), and should include the following sections - abstract, introduction, methodology, result, and conclusion.

Group Project

All coding must be done in **C/C++** (unless discussed with instructor beforehand)

Code must work on the **Talapas** supercomputer (you'll receive an email on how to set up an account soon)

See Canvas for sample final reports from previous terms

Survey

The survey paper is a short research paper-like report on a topic of your choice in HPC. The purpose of the survey paper is to evaluate your ability to identify a relevant topic, find & understand the related literature, and deliver a coherent summary of it.

See Canvas for more details

Deliverables

1. **Survey Paper** (*Due November 22nd, 11:59 PM PT*)

Again, if you have trouble identifying a topic, please come talk to me.

Misc.

- All coding **must be** version controlled using **git** & **BitBucket**.
  - Commit your code frequently!
- All reports must be written using **Latex**.
  - Use templates - it will make things easy
  - ACM conference format (double-column) is recommended
- See Canvas for details on grading policy, late submission, AEC, etc.

Logistics

Due end of this week

Invite me to your Bitbucket account: `jeec@uoregon.edu`

Find a partner for your group project

Create two Bitbucket Repos -

1) Personal repo (1 per student) - `uoregon-cis431531-f23`
   - survey/
   - homework/01
   - homework/02
   - Etc.
2) Group repo (1 per group) - `uoregon-cis431531-f23-group`
   - proposal/
   - code/
   - report/

Come up with a cool group name! We'll have a vote to pick the best one!

COVID

Please go over the COVID related information on the website

More details and examples on the website regarding the project and survey paper

Questions?

How do we measure performance?

P = W / T

- "**Work**" should be defined by the problem you are trying to solve - a popular metric for work is floating point operations (**FLOP**).
- Performance measures how **much work is done in a unit time** (i.e., FLOP/s)
- Other common metrics are
  - **keys/second** for **sorting**, or
  - **traversed edges per second** (TEPS) for **graphs**.
- It should **not** include **overhead** that does not contribute to calculating the **result**
  - For example, if you compress/uncompress data to minimize data movement for a sorting algorithm, should this be counted as work?

Is FLOP/s a Good Metric?

It is certainly the **most popular** metric in HPC

However, it can be tricky to use - for example

Code 1

```
do i = 1, 1,000,000
    A[i] = s * (B[i] + C[i]) + D[i]
enddo
```

Code 2

```
do i = 1, 1,000,000
    A[i] = s * B[i] + s * C[i] + D[i]
enddo
```

What is the correct number of flops (in the inner loop)?

Is FLOP/s a Good Metric?

- What about other types of operations (e.g., trigonometric, square root, divide)?
- It is not always clear how many flops a given piece of code will use
  - Some architectures might have specific instructions for those operations, while others may use a software implementations to calculate them
  - For example, with certain generation of GPUs, double-precision division was implemented using single-precision division, followed by Newton’s method to improve its precision
  - Some algorithms do not use only floating point operations (e.g., integer or bit manipulation operations)

Good strategy for selection

Most common way to choose a metric is to look at what others have done in the past for the application area

Assuming it’s not a terrible metric,

- Use the same metric
- Compare against prior studies using the same metric

Other Metrics

- Time (simple)
- Percentage (%) of Peak FLOP/s - what ratio of the theoretical peak performance of the system are you achieving?
- Percentage (%) of Bandwidth (memory)
  - some algorithms are limited by data transfer - expressing the performance in terms of data transfer may make more sense.
- Speedup (i.e., how much faster) over prior state-of-the-art.
- Number of iterations or updates (e.g., iterative solvers)
- Clock cycles
  - Invariant to the changes in clock frequency.

Speedup

Let’s say you’ve improved the performance of your code - how do you measure your “success?”

$$
\begin{aligned}
\text{Speedup } S &= P_{new} / P_{original} \\
&= (W / T_{new}) / (W / T_{original}) \\
&= T_{original} / T_{new}
\end{aligned}
$$

You can also have $T_{new} > T_{original}$, in which case you have a slow-down

Speedup (on Parallel Systems)

Sequential execution time: T_1 (or T_{seq})

Parallel execution time on P processing units: T_p (or T_{par})

Speedup on this parallel system = T_1 / T_p

Scalability

- Ability of a parallel algorithm to achieve gains proportional to the number of processors p
- **Perfect** scaling: $T_1 / T_p = p$
    - There are exceptions - superlinear scaling is possible
- Near perfect scaling can typically be seen in **embarrassingly parallel** applications/problems

Speedup (on Parallel Systems)

Sequential execution time: T_1 (or T_{seq})

Parallel execution time on P processing units: T_p (or T_{par})

Speedup on this parallel system: $S_p = T_1 / T_p$

Scalability

- Ability of a parallel algorithm to achieve gains proportional to the number of processors p
- **Perfect** scaling: $T_1 / T_p = S_p = p$
- Near perfect scaling can typically be seen in **embarrassingly parallel** applications/problems

Parallel efficiency: $E_p = S_p / p$

$0 < E_p = 1.0$ (perfect scaling) $< n$ (superlinear scaling)

Speedup Bounds (on Parallel Systems)

Amdahl’s Law

- Assume a program takes 10 hours to complete on a single processor.
- 10% of this program (i.e., 1 hour) **cannot** be parallelized
- 90% of this program (i.e., 9 hour) is **embarrassingly parallel**
- What is the maximum possible speedup with 20 processors?

Speedup Bounds

Amdahl's Law

- Assume a program takes 10 hours to complete on a single core.
- 10% of this program (i.e., 1 hour) **cannot** be parallelized.
- 90% of this program (i.e., 9 hour) is **embarrassingly parallel**
- What is the maximum possible speedup with 20 cores?

- 1 hour is "untouchable."
- 9 hours with 20 cores -> 9 / 20 is the new exec. time -> 0.45 hours
- New execution time = 1 + 0.45 = 1.45 hours.
- Speedup = 10 / 1.45 = 6.9x

What about when you have an **infinite** number of cores?

What about when 50% of the program cannot be parallelized?

Speedup Bounds

Amdahl's Law

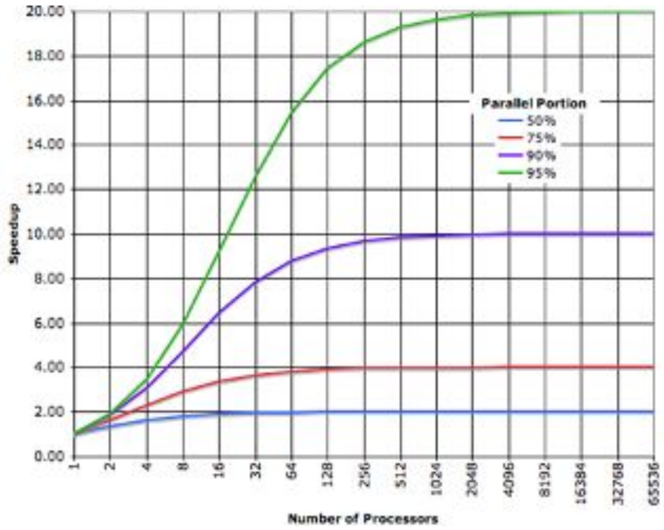
Formally,

- Let f be the fraction of a program that is sequential (i.e., can *never* be parallelized)
- $1 - f$ is the fraction that *can* be parallelized
- T_1 and T_p are execution times on 1 and p processors, respectively

$$S_p = T_1 / T_p$$
$$= T_1 / (f T_1 + (1-f) T_1 / p)$$
$$= 1 / (f + (1-f) / p)$$

- As $p \to \infty$

$$S_p \to 1 / f$$



Scalability

There are two methods of evaluating scalability

Strong scaling

- Problem size is fixed, increase the # of processors
- Difficult - Amdahl’s Law demonstrates that there is an upper-bound on speedup
- It is more difficult to achieve good performance with small workloads

Weak scaling

- Problem size ***increases*** with the # of processors
- Easier - emulates an embarrassingly parallel problem (but not always)

What can we expect for execution time when we have “good” strong scaling vs. “good” weak scaling?

Scaled Speedup

Gustafson's Law (or Gustafson-Barsis' Law)

- Given p processors and serial fraction f,
- Speedup = $p + (1 - p) f$
- 1 ($f = 1$) <= Speedup <= p ($f = 0$)

Gustafson's Law assumes **weak scaling** (i.e., work increases with P)

- If W is work and f is the fraction that is serial
- $W = fW + (1-f)W$
- If we want to increase the number of processors to p, then
- $W_{new} = fW + p(1-f)W$ -> only increase portion that can be parallelized
- Assuming both W and W_{new} can be done in time T, then
- Speedup = $Perf_p / Perf_1$

$$= (W_{new}/T) / (W/T)$$

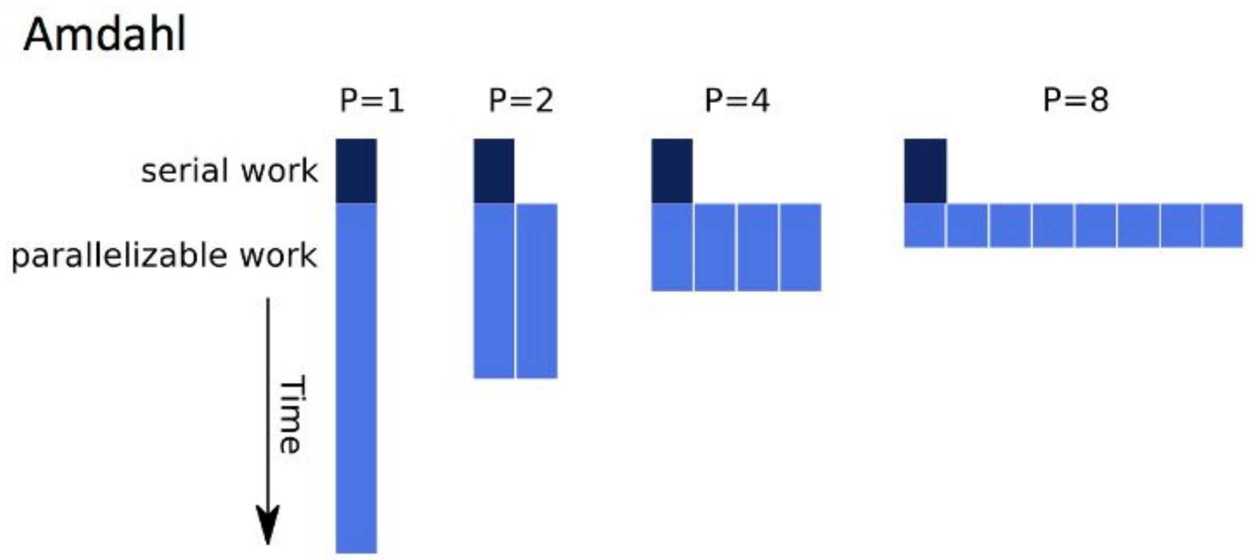
$$= W_{new} / W$$

$$= (fW + p(1-f)W) / W$$

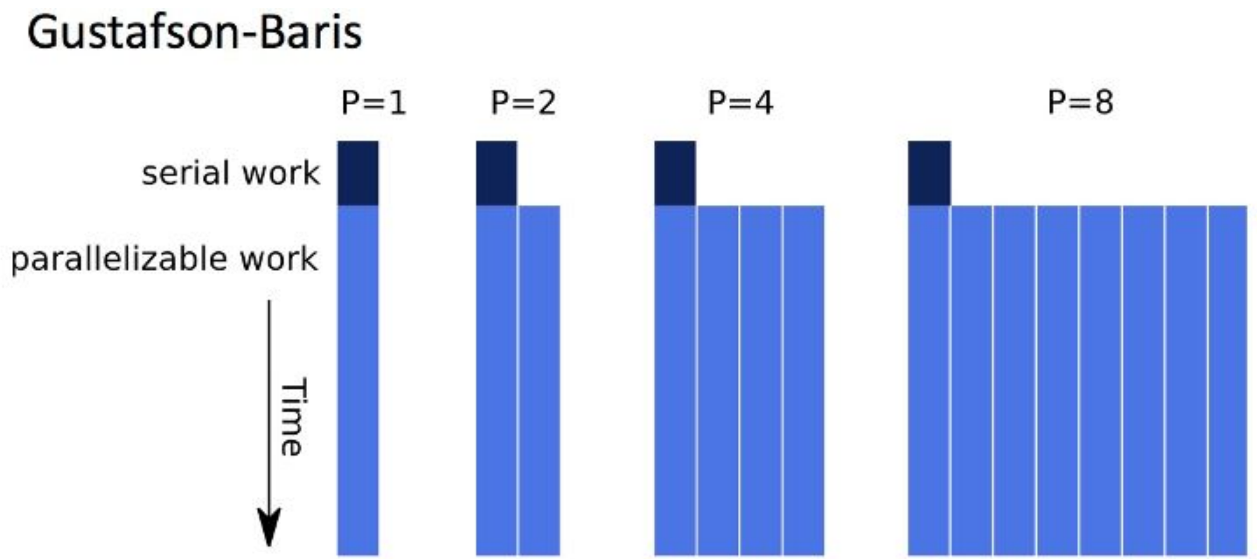
$$= f + p(1-f)$$

$$= p + (1-p)f$$

Amdahl's vs. Gustafson



Amdahl’s vs. Gustafson



Questions?

Work

Performance

Scalability

Upper bound on Speedup for

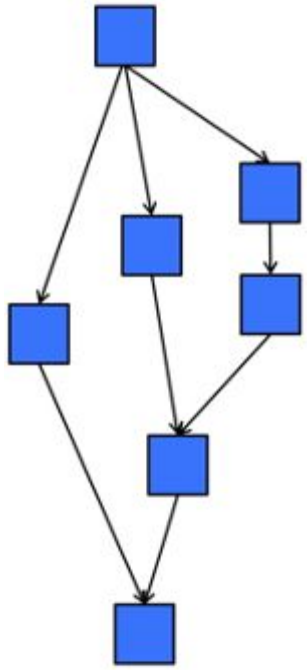
- Strong scaling
- Weak scaling

Workload-centric view of performance

DAG & Work-Span

Directed Acyclic Graph (DAG) model of computation

- Consider an application as a collection of **tasks**
- Tasks are connected as a graph where a link denotes a **dependency**



DAG & Work-Span

Work-Span Model for a DAG

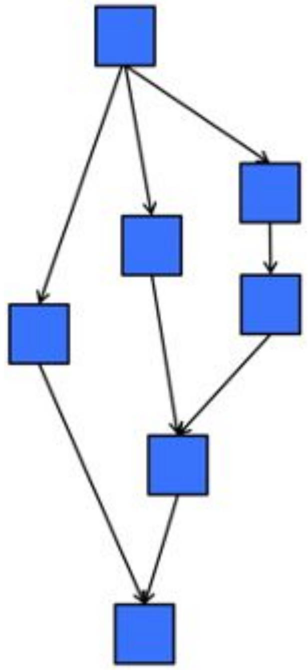
- A processor can work on a single task

T_1 = # of tasks

T_p = time to execute with p processors

T_∞ = span = number of tasks along the **critical path**

- Critical path = sequence of tasks along the DAG that takes the **longest** to execute
- What is the span in this example?



DAG & Work-Span

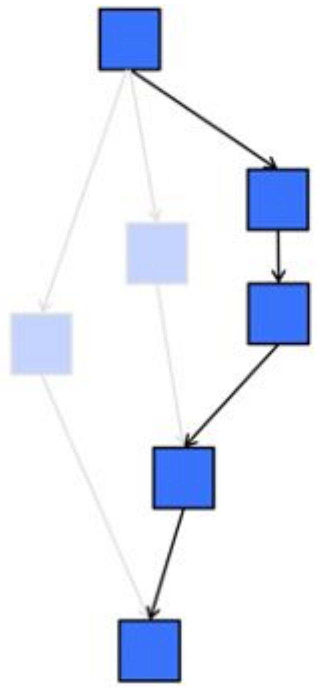
Work-Span Model for a DAG

T_1 = # of tasks

T_p = time to execute with p processors

T_∞ = span = number of tasks along the **critical path**

- Critical path = sequence of tasks along the DAG that takes the **longest** to execute
- What is the span in this example?
- $T_\infty = 5$
- $T_1 = 7$



Bounds on Greedy Scheduling

What would be the time to process a DAG given p processors using a **greedy** algorithm?

Suppose we have only p processors

Lower bound

$\max(T_1/p, T_\infty) <= T_p$

T_∞ is the lowest possible execution time

T_1/p assumes everything can be processed in parallel

Bounds on Greedy Scheduling

What would be the time to process a DAG given p processors using a greedy algorithm?

- Suppose we have only p processors
- **Lower** bound
  - $\max(T_1/p, T_\infty) <= T_p$
  - T_∞ is the best possible execution time
  - T_1/p assumes everything can be processed in parallel
- **Upper** bound
  - Derived using Brent's Lemma

Bounds on Greedy Scheduling

Brent's Lemma

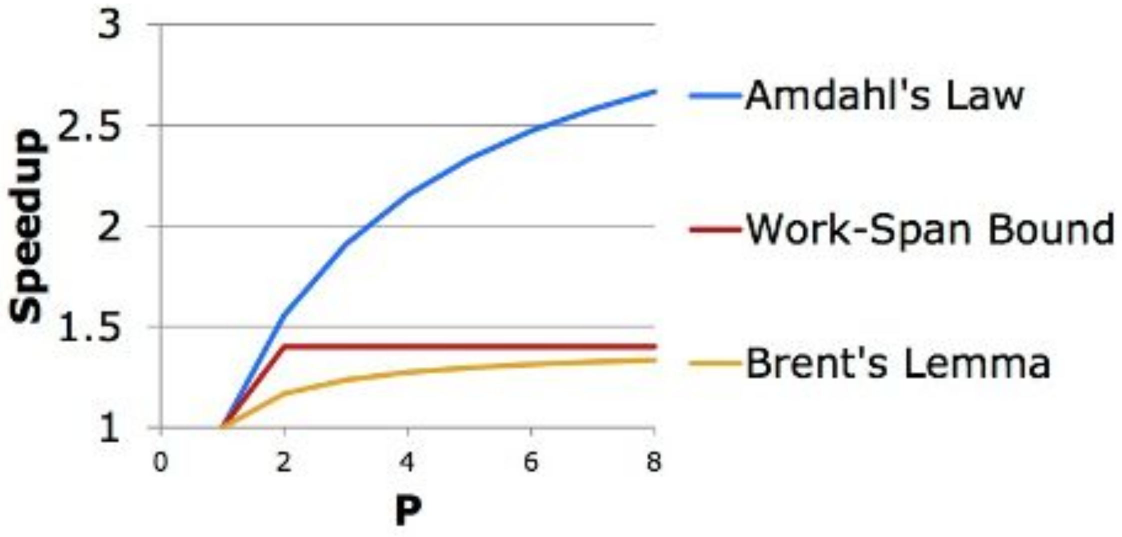
- $T_p <= T_\infty + (T_1 - T_\infty) / p$
- Does this make sense?

Bounds on Greedy Scheduling

Brent's Lemma

- $T_p <= T_\infty + (T_1 - T_\infty) / p$
- Critical path (T_∞) + Off-critical path

Amdahl vs. Brent



Questions?

Work-span model captures the dependency between tasks in a program

- As opposed to looking at the workload in Amdahl and Gustafson analysis

You can also derive a lower and upper bound using the work-span model

- Less optimistic than Amdahl and Gustafson

Scalability

As we saw, weak scaling is easier to obtain than strong scaling.

However, how can we (theoretically) determine how much W has to be increased to maintain efficiency for p processors?

Isoefficiency

How much increase in problem size is required to retain the same level of efficiency on a larger system?

Remember that, in perfect scaling $T_1/T_p = p \rightarrow T_1 = pT_p$

However, using a parallel system usually incurs an overhead (e.g., communication between processors, contention)

Let's assume

T_o = overhead = T(W, p)

pT_p = **total** time spent by **all** p processors

$pT_p = T_1 + T_o$

$T_p = (T_1 + T_o) / p$

$S = T_1/T_p = pT_1 / (T_1 + T_o)$

$E = S / p = T_1 / (T_1 + T_o) = 1 / (1 + T_o/T_1)$

Isoefficiency

$E = 1 / (1 + T_o/T_1)$

Let's say t_c = time to do 1 operation (unit of work)

$T_1 = Wt_c$

$E = 1 / (1 + T_o/Wt_c)$

You can see that if W remains constant while p increases (i.e., T_o increases), then the efficiency decreases.

On the other hand, if W increases while p remains constant, efficiency increases (assuming T_o grows more slowly than W)

We can maintain the same efficiency by increasing p, provided W also increases.

How much we increase W with p depends on the system.

Isoefficiency

$E = 1 / (1 + T_o/Wt_c)$

We can keep efficiency at a desired rate if we keep T_o/W constant

$T_o/W = t_c((1 - E) / E)$

$W = 1/t_c\ (E / (1 - E))\ T_o$

If $K = 1/t_c\ (E / (1 - E))$ is a constant that depends on the efficiency, then

$W = KT_o$

We can use this equation to obtain W as a function of p (i.e., of T_o)

This is the Isoefficiency function

- Small isoefficiency function implies that small increase in problem size is sufficient to use an increasing number of processors efficiently

Questions?

We can determine how much W has to be increased to maintain efficiency using the Isoefficiency equation

Scalability

- What impacts scalability?
  - Scalability in parallel architectures is impacted by
    - Number of processors
    - Memory architecture
    - Interconnect network
    - Other hardware bottlenecks
  - Scalability in algorithms is impacted by
    - Problem size
    - Algorithm
      - How much computation
      - How much memory access

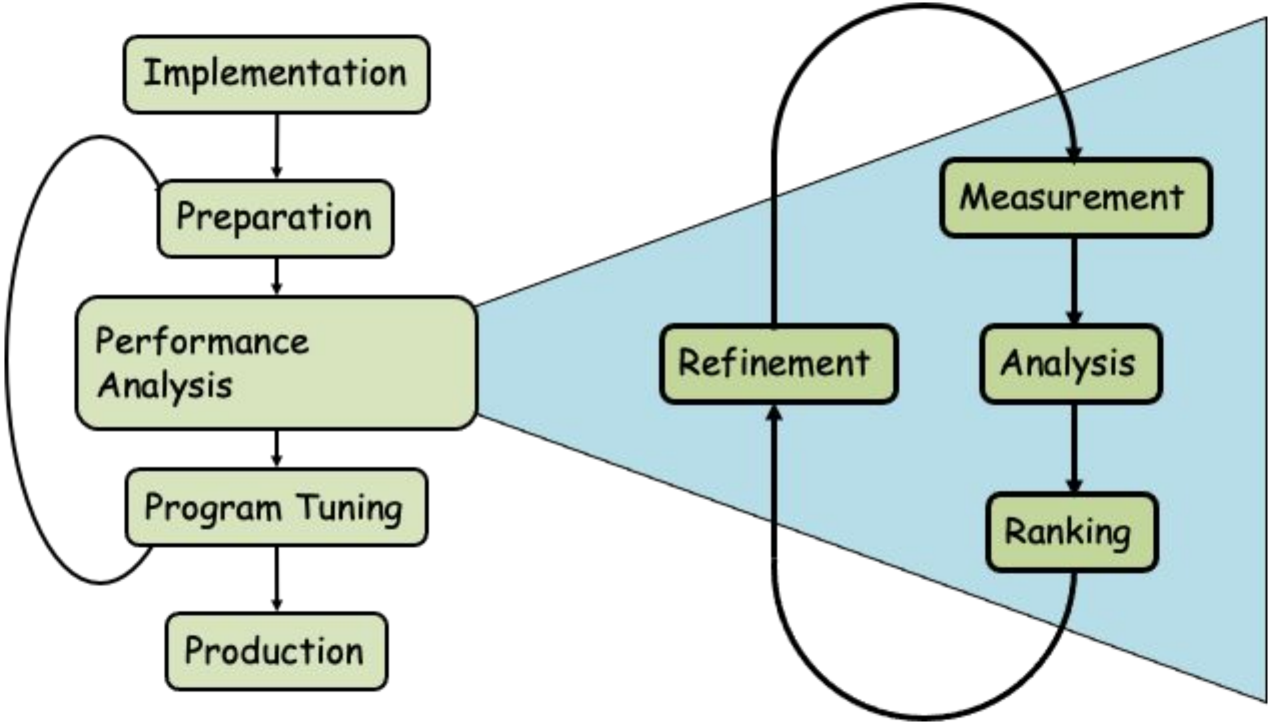
Scalability

Why aren't parallel applications scalable?

- Sequential performance
- Critical path
- Bottlenecks (e.g., one processor holding things up)
- Algorithmic overhead
- Communication overhead (more processors mean more communication)
- Load imbalance (some processors are given more work)
- Combinations of above

Performance Tuning

Performance tuning process



Performance Optimization

Parallel/distributed systems are complex

Four layers

- Application (algorithm, data structure)
- Parallel programming interface (compilers, libraries, synchronization)
- OS (process and memory management, I/O)
- Hardware (CPU, memory, network)

Mapping/interaction between different layers

Factors which determine a program's performance are complex, interrelated, and oftentimes hidden

Performance analysis tools can aid in optimizing performance quickly

- They help understand what, where, and how time is spent
- Hotspots - area of the code that uses a disproportionate amount of time
- Botteleneck - area of code that uses resources inefficiently (or there isn't enough of the resource), causing (unnecessary) delays

Measuring Time

- Measuring time can be tricky
  - Some clocks are more accurate than others
  - Clocks are themselves software, not immune to overheads
  - Parallel programs - make sure you have barriers to capture the correct completion point
- POSIX gettimeofday() function is a good tool for timing
  - Available on all POSIX-compliant systems
- You can also write your own timing function by reading certain registers that keep track of clock cycles

gettimeofday

```c
#include <sys/time.h>

void get_walltime_(double* wcTime) {
  struct timeval tp;
  gettimeofday(&tp, NULL);
  *wcTime = (double)(tp.tv_sec +
                     tp.tv_usec/1000000.0);
}
```

Custom Timer

```
uint32_t hi, lo;
__asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
return ( (uint64_t)lo)|( ((uint64_t)hi)<<32 );
```

- rdtsc - read time-stamp counter
- Processor monotonically increments this counter every cycle
- Have to translate the cycles to actual time (not shown here)

System Benchmarking

Evaluation of a system for specific application(s).

- Real(ish) applications
  - LINPACK
    - Used for Top500 (topp500.org)
    - Solves dense systems of linear equations
    - Generally considered **not** a good benchmark - only tests compute performance
    - Preferred due to its simplicity
    - Uses Basic Linear Algebra Subprogram (BLAS) library
  - HPCG (High Performance Conjugate Gradient)
    - Solves sparse systems
    - More realistic for solving real problems
  - Suites
    - HPC Challenge Benchmarks
    - UEABS (Unified European Applications Benchmark Suite)

System Benchmarking

- Synthetic microbenchmarks
  - Typically tests some specific feature of an architecture (i.e., low-level benchmarking)
  - Energy roofline model microbenchmarks (https://github.com/jeewhanchoi/a-roofline-model-of-energy-ubenchmarks)
  - Stream benchmark (bandwidth) - Dr. John McCalpin (aka Dr. Bandwidth)
  - Triad benchmark (multiply-add on 3 vectors)
- Often written in assembly
  - Better control over the architecture.
  - Typically simple enough to be written in assembly.
- Care must be taken to make sure the compiler does not optimize instructions away

System Benchmarking

Eliminating variation

- Benchmarking should be done in controlled environment (i.e., do not use your laptop)
- Use many iterations and provide statistical information (i.e., avg, min, max, outliers, etc.). Box and Whisker charts are useful for this.
- If testing data movement, make sure your data sizes are large enough that they **do not fit** in cache - otherwise, your loop will allow the data to be cached for subsequent iterations and lower the overall execution time significantly.

Questions?