Performance Analysis
Performance analysis

• Why do we parallelize our programs?
  – So that they run faster!
Performance analysis

• How do we evaluate whether we've done a good job in parallelizing a program?
  – Asymptotic analysis (e.g., for distributed sum)
  – Empirical analysis
Empirical analysis issues

- How do you measure time-to-solution accurately?
  - CPU cycles, OS clock "ticks", wall time, etc.
- How do you compare across systems?
  - Differing CPUs, memories, OSes, etc.
- How do you compare against the original?
  - 1-core parallel version will likely be slower
- How do you assess scalability?
  - Does performance improve as you add cores?
  - How do you quantify the improvement?
  - Is there a limit to how far we can improve performance?
Best practices

- Measure wall time for specific code regions of interest
  - Ignore startup and I/O time if not relevant
  - Make sure you have a high-resolution timer!
    - `/usr/bin/time -v` for whole programs
    - `gettimeofday()` from `sys/time.h` for Pthreads
    - `omp_get_wtime()` for OpenMP
    - `MPI_Wtime()` for MPI
  - Use barriers if necessary to make sure all threads/processes have finished before you stop a timer
Best practices

- Control for variance
  - Do all experiments on the same machine or cluster
  - Maximum of one thread per core and one job per node
    - Our cluster can support 8 threads per node (or 16 if hyper-threading, but this is not always recommended)
  - Run multiple trials and use minimum time
    - Avoids OS interference or noise
    - Use /shared/bin/hyperfine on cluster
  - Track variance across trials to measure system noise
    - If your variance is high or if your slowest and fastest time are relatively far apart, it's probably noise!
Empirical analysis

\( T_s = \) serial time
\( T_p = \) parallel time
\( p = \) \# of processes

\[ S = \text{speedup} = \frac{T_s}{T_p} \]

should increase as \( p \) grows

\[ E = \text{efficiency} = \frac{S}{p} = \frac{T_s}{p T_p} \]

usually decreases as \( p \) grows

\[ r = \] serial % of original program

\[ T_p = \frac{(1-r)T_s}{p} + r T_s \]

\[ S = \text{speedup} = \frac{T_s}{(1-r)T_s + r T_s} \]

Amdahl's Law: \( S \leq \frac{1}{r} \) as \( p \) increases
Amdahl's Law

\[ S = \text{speedup} = \frac{T_s}{(1-r)T_s + rT_s} \]

\[ p = \# \text{ of processors} \]

\[ r = \text{serial \% of program} \]

Amdahl's Law:

\[ S \leq \frac{1}{r} \text{ as } p \text{ increases} \]

- \( r = 50\% \rightarrow \text{speedup limited to 2x} \)
- \( r = 25\% \rightarrow \text{speedup limited to 4x} \)
- \( r = 10\% \rightarrow \text{speedup limited to 10x} \)
- \( r = 5\% \rightarrow \text{speedup limited to 20x} \)

Speedup limited inversely proportionally by serial \%
• Generally, we don't care about any particular $T_P$
  – Or with how it compares to $T_S$ (except as a sanity check)
• More important: how $T_P$, $S$, and $E$ change as $p$ increases
  – And/or as the problem size increases
  – Similar to asymptotic analysis in CS 240
  – In general, a program is **scalable** if $E$ remains fixed as $p$ and the problem size increase at fixed rates
  – Most common: graph $T_P$ on y-axis vs. $p$ on logarithmic x-axis

![Graphs showing good and bad scalability](attachment://graphs.png)
Scaling

- **Strong scaling** means we can keep the efficiency fixed without increasing the problem size.
- **Weak scaling** means we can keep the efficiency fixed by increasing the problem size at the same rate as the process/thread count.
  - Rate of work (e.g., Mop/s) per core remains roughly fixed.

\[
E = \text{efficiency} = \frac{S}{p} = \frac{T_{S}}{pT_{P}} \quad \text{usually decreases as } p \text{ grows}
\]
 Scaling

- **Strong scaling**: as $p$ increases, $T_P$ decreases
  - Linear speedup: same rate of change (2x procs → half time)
  - Sublinear (most common) / superlinear (exceedingly rare) speedup
  - Be careful to interpret linear vs. logarithmic axes correctly

- **Weak scaling**: as $p$ increases AND the problem size increases proportionally, $T_P$ stays roughly the same
Cluster access

• Detailed instructions online: w3.cs.jmu.edu/lam2mo/cs470/cluster.html

• Connect to login node via SSH
  – Hostname: login.cluster.cs.jmu.edu
  – User/password: (your e-ID and password)

• Recommended conveniences
  – Set up public/private key access from stu
  – Set up .ssh/config entries
  – Install Spack for access to more software
• Things to play with:
  - "squeue" or "watch squeue" to see jobs
  - "srun <command>" to run an interactive job
    • Use "-n <p>" to launch p processes
    • Use "-N <n>" to request n nodes (defaults to p/8)
    • The given "<command>" will run in every process
  - "salloc <command>" to run an interactive MPI job
    • Must run "module load mpi" first
    • Use "-n <p>" to launch p MPI processes

srun hostname
srun -n 4 hostname
srun -n 16 hostname
srun -N 4 hostname
srun sleep 5
srun -N 2 sleep 5

module load mpi
salloc -n 1 mpirun /shared/mpi-pi/mpipi
salloc -n 2 mpirun /shared/mpi-pi/mpipi
salloc -n 4 mpirun /shared/mpi-pi/mpipi
salloc -n 8 mpirun /shared/mpi-pi/mpipi
salloc -n 16 mpirun /shared/mpi-pi/mpipi
(etc.)

What’s the max n?
Job management

- **SLURM** (Simple Linux Utility for Resource Management) is a piece of system software outside the OS (a.k.a. middleware) that handles job submission and scheduling on our cluster.

- An **interactive** job takes control of your terminal:
  - Run with `srun` or `salloc`.
  - You may interact with it (provide standard input, etc.).
  - You also have to wait for it to finish.
  - Similar to a foreground shell job.

- A **batch** job runs in the background without interaction:
  - Create a shell script and run it with `sbatch`.
  - Sends output to a file (named “slurm-JOBID.out” by default).
  - Use `squeue` to check to see if it has finished.
Batch jobs

- To run a batch job on the cluster, create a shell script and run it with `sbatch`

- Bash example:

```bash
#!/bin/bash
#
#SBATCH --job-name=hostname
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --nodes=1

<your commands go here>
```
Running experiments

- Common experimentation patterns in Bash:

  # run 5 times
  for i in $(seq 1 5); do
    <cmd>
  done

  # run common thread counts
  for t in 1 2 4 8 16; do
    OMP_NUM_THREADS=$t <cmd>
  done