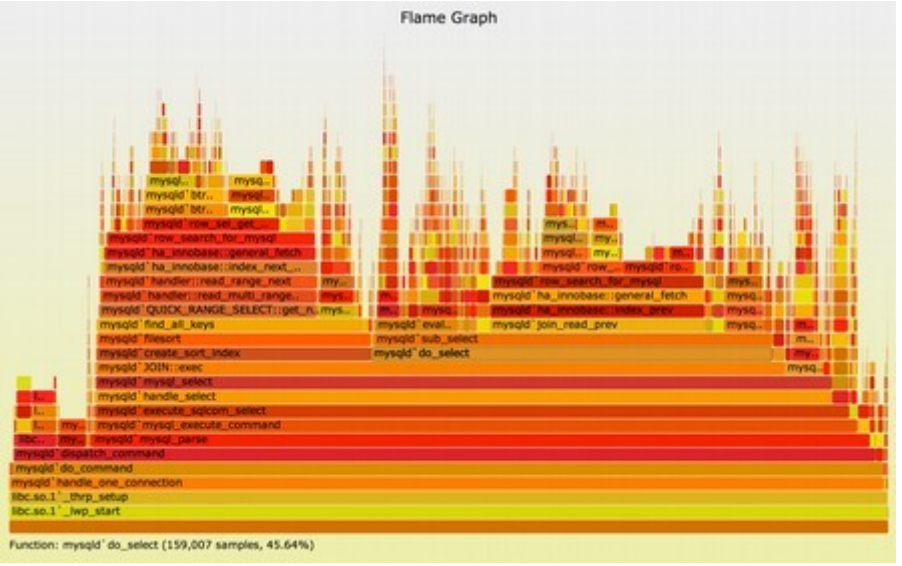
CS 470
Spring 2025

Mike Lam, Professor



Flame graph from brendangregg.com

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-thread/process 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 16 threads per node (or 32 if hyper-threading, but this is not always recommended)
  - Run multiple trials and use minimum time
    - Minimizes impact of OS interference or noise
    - Alternative: run a few “warmup” trials before “real” trials
    - Use `/shared/cs470/bin/hyperfine` on cluster for whole programs
  - Measure variance across trials
    - If your variance is high or if your slowest and fastest time are relatively far apart (as a percentage of the slower time), it's probably noise!

Best practices [Hoefler 2015]

- Rule 1: Report if the base case is a single parallel process or best serial execution, as well as the absolute execution performance
- Rule 2: Specify the reason for reporting subsets of applications or not using all system resources
- Rule 9: Document all varying factors as well as the complete experimental setup to facilitate reproducibility
- Rule 12: Plot as much information as needed to interpret the results – only connect measurements by lines if they indicate trends and the interpolation is valid

Torsten Hoefler and Roberto Belli. 2015. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15). Association for Computing Machinery, New York, NY, USA, Article 73, 1–12. DOI:https://doi.org/10.1145/2807591.2807644

Empirical analysis

T_s = serial time

T_P = parallel time

p = # of processes

S = speedup = $\frac{T_S}{T_P}$ *should increase as p grows*

E = efficiency = $\frac{S}{p} = \frac{T_S}{p\,T_P}$ *usually decreases as p grows*

r = serial % of original program

$$T_P = \underbrace{\frac{(1-r)\,T_S}{p}}_{\text{parallel portion}} + \underbrace{r\,T_S}_{\text{serial portion}}$$

$$S = \text{speedup} = \frac{T_S}{\frac{(1-r)\,T_S}{p} + r\,T_S}$$

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

Amdahl's Law

p = # of processors

r = serial % of program

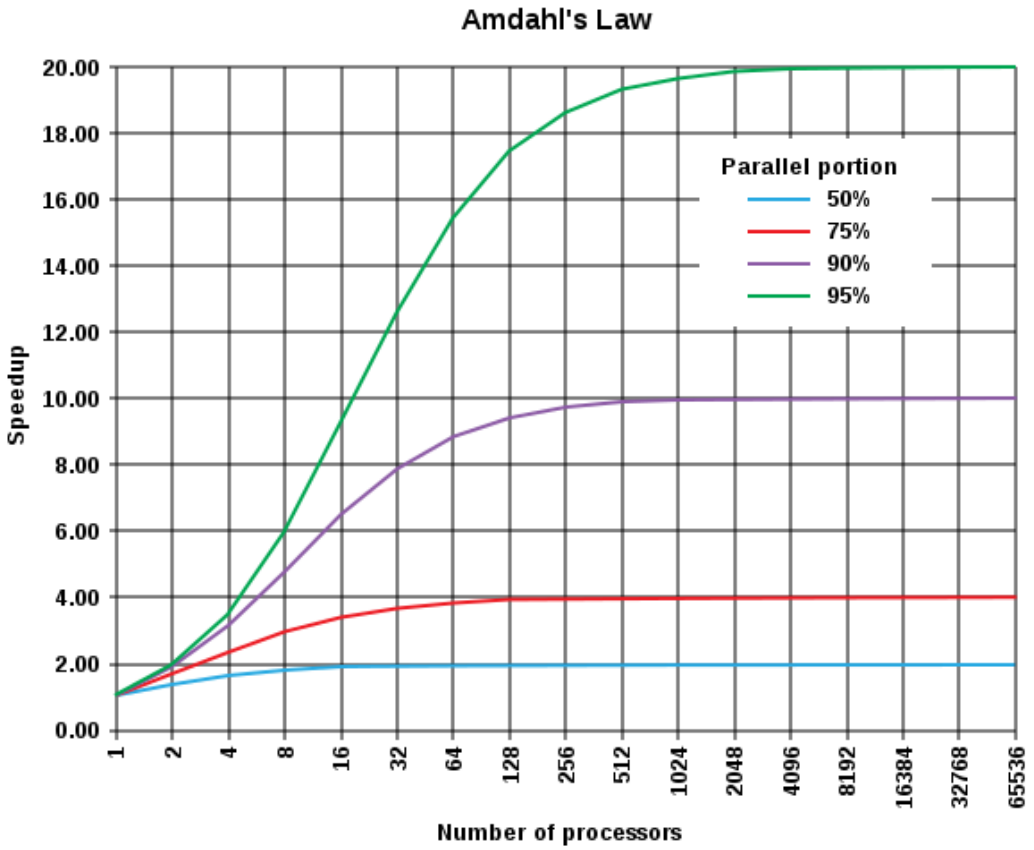
$$S = \text{speedup} = \frac{T_s}{\frac{(1-r)T_s}{p} + r\,T_s}$$

Amdahl's Law:

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

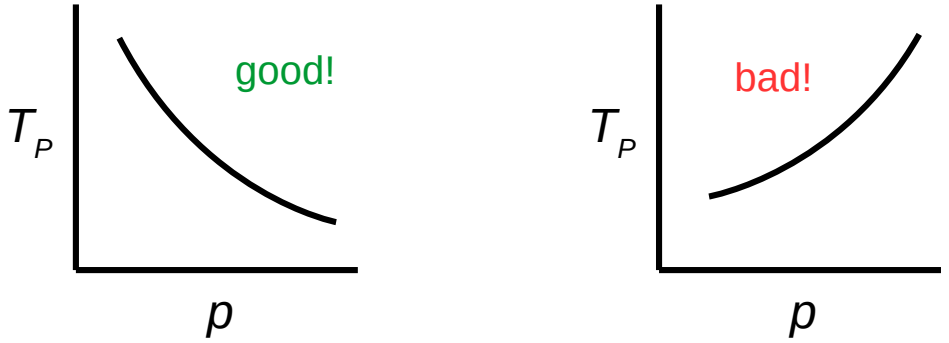
r = 50% → speedup limited to 2x
r = 25% → speedup limited to 4x
r = 10% → speedup limited to 10x
r = 5% → speedup limited to 20x

Speedup limited inversely proportionally by serial %



Scaling

- 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



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

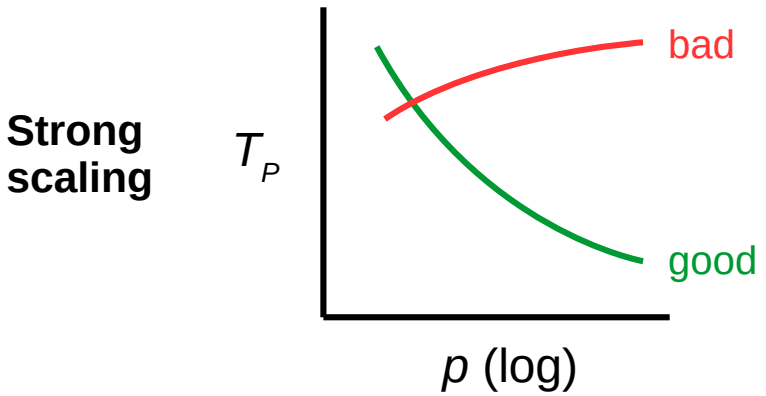
$$E = \text{efficiency} = \frac{S}{p} = \frac{T_S}{p\,T_P}$$

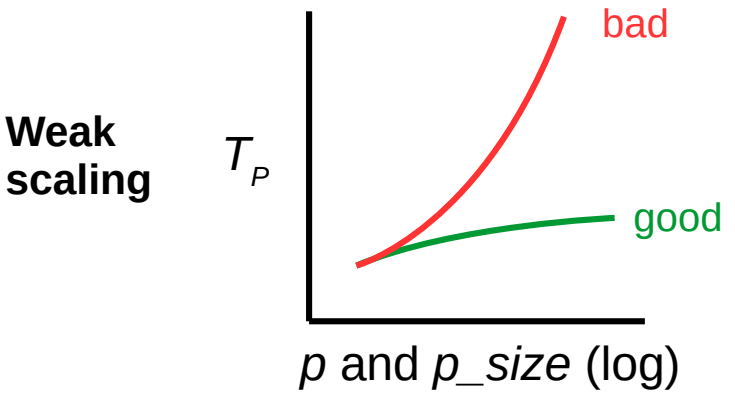
usually decreases as p grows

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





Scaling

- What do the following results exhibit?
  - A) No scaling
  - B) Strong scaling only
  - C) Weak scaling only
  - D) Both strong and weak scaling

# Processors	Problem Size	Wall Time
1	100x100	52s
2	100x100	28s
4	100x100	15s

Scaling

- What do the following results exhibit?
  - A) No scaling
  - B) Strong scaling only
  - C) Weak scaling only
  - D) Both strong and weak scaling

# Processors	Problem Size	Wall Time
1	100x100	116s
2	100x100	87s
4	100x100	93s

Scaling

- What do the following results exhibit?
  - A) No scaling
  - B) Strong scaling only
  - C) Weak scaling only
  - D) Both strong and weak scaling

# Processors	Problem Size	Wall Time
1	100x100	77s
2	150x150	85s
4	200x200	81s

Job management

- Slurm is 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:

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

Running experiments

- For MPI, use a templated run script to launch multiple jobs with different numbers of MPI tasks

run.sh

```
#!/bin/bash
#SBATCH –job-name=<cmd>-MPI_NUM_TASKS
#SBATCH --output=<cmd>-MPI_NUM_TASKS.txt
#SBATCH --ntasks=MPI_NUM_TASKS

module load mpi
srun -n MPI_NUM_TASKS <cmd>
```

launch.sh

```
#!/bin/bash
for n in 1 8 16 32 64 128; do
    sed -e "s/MPI_NUM_TASKS/$n/g" run.sh | sbatch
done
```

view.sh

```
#!/bin/bash
for n in 1 8 16 32 64 128; do
    echo "== $n processes =="
    cat <cmd>-$n.txt
    echo
done
```

Note re: sbatch and zsh

- If you use zsh instead of bash and want to write batch scripts, you may also need this line before "`module load mpi`":
  - `source /usr/share/Modules/init/zsh`