CISC 372: Parallel Computing

Performance

Stephen F. Siegel

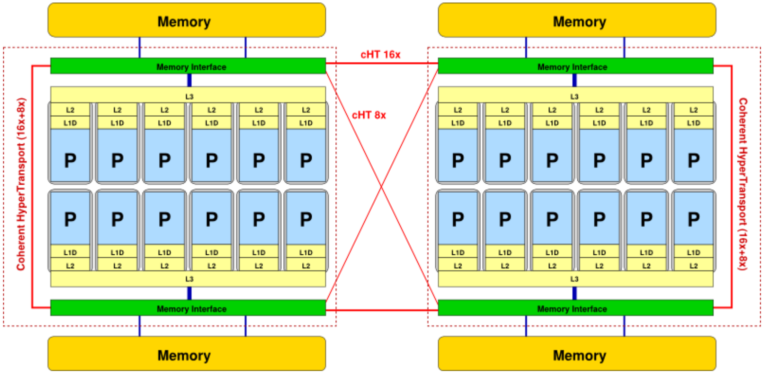
Department of Computer and Information Sciences
University of Delaware

Performance: definition

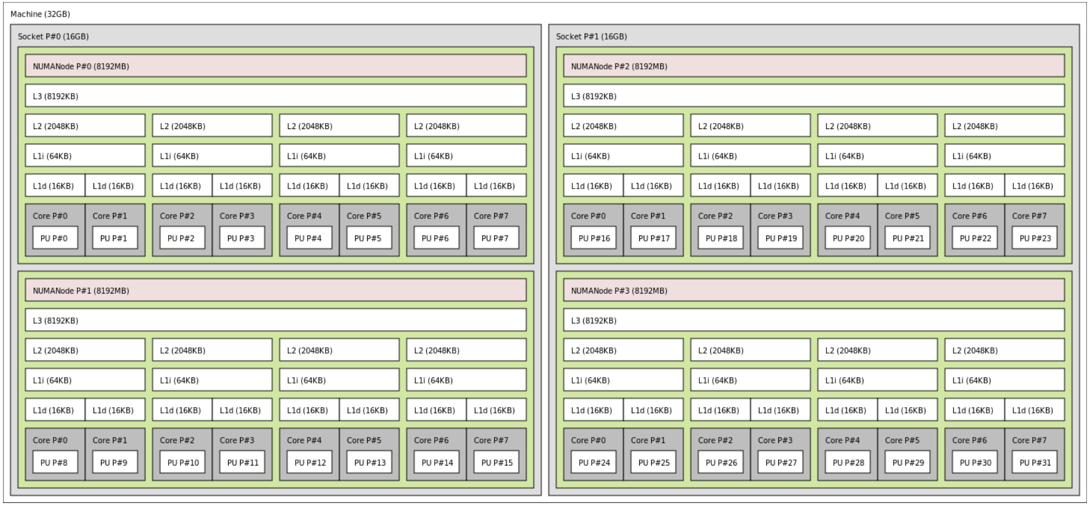
- how efficiently resources are used to solve a problem
- resources?
  - memory
  - energy
  - time

Factors that affect performance: effective use of memory hierarchy

- modern CPUs have a hierarchy of data caches between CPU and memory
  - L1 cache: closest to core, very fast connection to registers (typical size: 32 KB/core)
  - L2 cache: further than L1, bigger, slower (256 KB/core)
  - L3 cache: further than L2, bigger, slower (2 MB/core)
  - DRAM: very slow



Memory hierarchy: AMD Bulldozer server



`https://en.wikipedia.org/wiki/CPU_cache`

Memory hierarchy: example: matrix-vector multiplication

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 \\ a_{10}x_0 + a_{11}x_1 + a_{12}x_2 \\ a_{20}x_0 + a_{21}x_1 + a_{22}x_2 \\ a_{30}x_0 + a_{31}x_1 + a_{32}x_2 \end{bmatrix}$$

Layout of a in memory:

a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}	a_{20}	a_{21}	a_{22}	a_{30}	a_{31}	a_{32}

- see `colmaj.c`: a is $N \times N$ array of doubles, $N = 20,000$
  - consider accesses to `a`
  - `a[0][0]`, `a[1][0]`, `a[2][0]`, ...
  - these are separated by $20,000 *$ `sizeof(double)` bytes!
  - each access loads into cache an entire block (cache line) containing the requested location
- see `rowmaj.c`
  - functionally equivalent to `colmaj.c`
  - `a[0][0]`, `a[0][1]`, `a[0][2]`

Factors that affect performance: compiler optimizations

Compilers can transform programs in myriad ways to use resources more effectively...

- function inlining; loop fission, loop fusion; loop interchange; loop unrolling; common subexpression elimination; constant folding, propagation ...

Tradeoffs: more optimization generally entails...

- longer compile time
- larger generated code size
- program gets harder to debug
- greater sensitivity to undefined behavior (but you shouldn't use any undefined behavior!)
- generated code might actually get slower

Most compilers present a few pre-packaged optimization levels:

- `-O0`: little optimization, the default; `-Og`: recommended for debugging
- `-O1`, `-O2`, `-O3`: increasingly more optimizations applied
- see `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

Measuring performance of a parallel program

- to measure performance of a parallel program, you need a baseline
- baseline: a "similar" sequential program
  - using same inputs and other parameters to the extent possible
- different notions of similar are possible
  - do you choose the best possible sequential algorithm that solves the problem?
  - or the parallel program with `-n 1` (one process)?
  - these are often very different!
  - you must always specify the baseline

Speedup

Let

$$\begin{aligned} T_{\text{seq}} &= \text{time to run sequential baseline} \\ T_{\text{par}} &= \text{time to run parallel program} \end{aligned}$$

Then

$$\text{Speedup} = \frac{T_{\text{seq}}}{T_{\text{par}}}$$

- higher speedup is better
- if seq took 10 seconds and par took 2 seconds, speedup is 5
  - “parallel program is 5x faster than sequential”
  - with those particular inputs and nprocs

Speedup as a function of `nprocs`

- hopefully: speedup will change (increase!) with `nprocs`
- ideal case: speedup = `nprocs`
  - double the number of procs, cut the execution time in half
- reality: rarely that good
  - communication time (sending messages)
  - synchronization time (procs have to sit around waiting, e.g., at a Barrier)
  - redundant work (two procs compute the same thing)
- after some point adding more processes no longer increases speedup
  - sort a list of 10^{12} elements
    - doubtful you can improve speedup when `nprocs` $> 10^{12}$
  - this is always the case for fixed problem size
- in best case, speedup may be approximatley linear over some range of nprocs, but never as nprocs $\to \infty$

Amdahl's Law

- typically some part of the code cannot be parallelized
- "inherently sequential"
- example: `diffuse1d` writes data to the screen
  - there is only one screen: no way to parallelize that part
- example
  - say in sequential program, 10% of time is spent doing "inherently sequential" work
  - the other 90% can be parallelized
  - even if a parallel program were PERFECT with unlimited resources, the best it could do is reduce the 90% to 0.
  - the 10% time would be unchanged
  - therefore the best possible parallel time is $(1/10) * T_{\text{seq}}$
  - best possible speedup is 10 `:-(`
- in general, if inherently sequential fraction of original program is r
  - then speedup $< 1/r$

Weak Scaling vs. Strong Scaling

- Amdahl assumes inputs are held constant as `nprocs` increases
  - "strong scaling"
- Amdahl shows that strong scaling is problematic
  - there is always some point after which adding more procs cannot help
- does this really reflect how people use parallel programs?
- Gustafson (1988) said **no**
  - most users do not have some fixed problem size and ask *how fast can I make it?*
  - instead, the more processors you give them, the bigger they will make the problem size
  - almost every problem in science and engineering benefits from increased resolution or scale
- so a more useful measure of performance increases problem size with `nprocs`
  - "weak scaling"
- parallelization is more effective with weak scaling than with strong scaling

Weak vs. Strong Scaling

- strong scaling: baseline is constant as nprocs increases
    - always comparing against sequential run on fixed problem size
    - speedup is bounded
- strong scaling examples
    - fix list of length 10^6; compare sequential time to sort vs. parallel time to sort with p procs
    - fix `nx` $= 10^3$; compare sequential diffusion1d vs. parallel diffusion1d with p procs
        - note `nxl`, the amount of data per process, decreases as p increases
- weak scaling: baseline increases with nprocs
    - problem size of sequential program increases with nprocs
    - it is possible for speedup $\to \infty$ as nprocs $\to \infty$
- weak scaling examples
    - for $p > 0$, compare sequential time to sort list of length $10^6 p$ with parallel time using p procs
    - for $p > 0$, compare sequential diffusion1d with `nx` $= 10^3 p$ vs. parallel diffusion1d with p procs
        - note `nxl` $= 10^3$ is held constant as p increases

Efficiency

$$\text{efficiency} = \frac{\text{speedup}}{\text{nprocs}} = \frac{T_{\text{seq}}}{T_{\text{par}} * \text{nprocs}}$$

- efficiency is "speedup per process"
- Amdahl says that for strong scaling:
  - $\texttt{efficiency} \to 0$ as $\texttt{nprocs} \to \infty$
- for weak scaling, in best case it is possible:
  - $\texttt{efficiency} \to 1$ as $\texttt{nprocs} \to \infty$
- more common: something between 0 and 1

Automating performance experiments

- see `exp/sat_strong` in public course repo
- a strong scaling experiment of MPI SAT solver
- `sat_mpi.c` has been altered to
  - print to `stdout` only the number of processes and time
  - other things are sent to `stderr`

```
  if (rank == 0) {
    const double time = MPI_Wtime() - start_time;

    fprintf(stderr, "Number of solutions = %u.  Time = %lf.\n",
            nsolutions, time);
    fflush(stderr);
    printf("%d %lf\n", nprocs, time);
  }
```

Data file generated by SAT performance experiment

- the `Makefile` executes `sat_mpi.exec` with 1, 2, 4, 8, 16, 32 procs
- the results are accumulated in a file `sat_mpi.dat`:

```
1 42.693483
2 29.942159
4 16.342128
8 9.844605
16 5.327622
32 2.452447
```

nprocs	time	speedup	efficiency
1	43.75	1.00	1.00
2	30.46	1.44	0.72
4	16.47	2.66	0.66
8	8.68	5.04	0.63
16	4.89	8.95	0.56
32	2.52	17.36	0.54

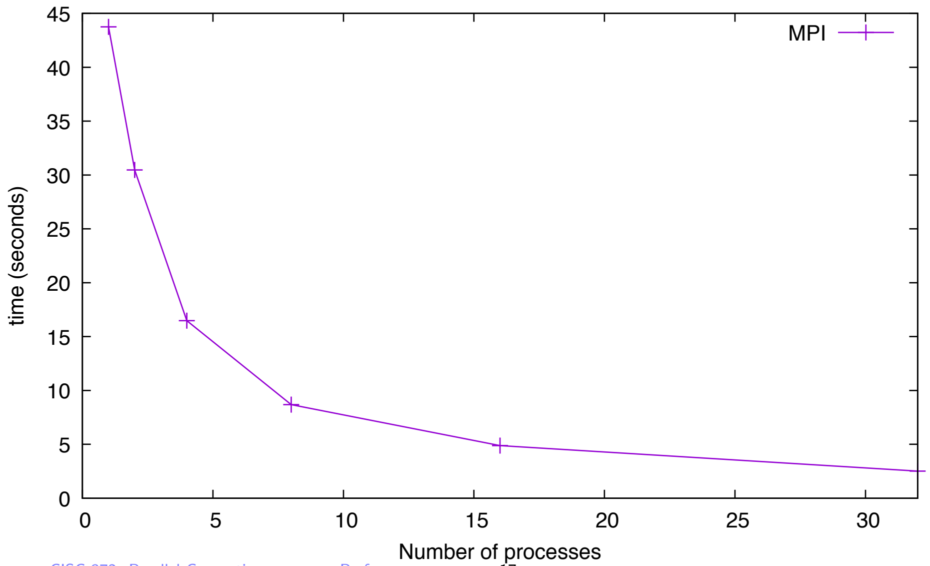
Graphing data with gnuplot

- free, open-source command-line tool for creating graphs
- http://www.gnuplot.info
- command: `gnuplot sat_mpi.gnu`

```
set terminal pdf
set output "sat_mpi.pdf"
set xlabel center "Number of processes"
set ylabel center "time (seconds)"
set xr [0:32]
set yr [0:45]
plot "sat_mpi.dat" using 1:2 title 'MPI' with linespoints
```

- meaning of `using 1:2`
  - use column 1 of the data file for the x-coordinates
  - use column 2 of the data file for the y-coordinates

PDF file resulting from SAT scaling experiment



Makefile for SAT performance experiment

```
ROOT = ../../
include $(ROOT)/common.mk
NAME = sat_mpi
all: $(NAME).exec
$(NAME).exec: $(NAME).c  Makefile
        $(MPICCC) -o $@ $<
$(NAME).dat: $(NAME).exec
        $(MPIRUN) -n 1 ./$(NAME).exec > $(NAME).dat
        $(MPIRUN) -n 2 ./$(NAME).exec >> $(NAME).dat
        $(MPIRUN) -n 4 ./$(NAME).exec >> $(NAME).dat
        $(MPIRUN) -n 8 ./$(NAME).exec >> $(NAME).dat
        $(MPIRUN) -n 16 ./$(NAME).exec >> $(NAME).dat
        $(MPIRUN) -n 32 ./$(NAME).exec >> $(NAME).dat
graphs:
        gnuplot $(NAME).gnu
.PHONY: all graphs
```

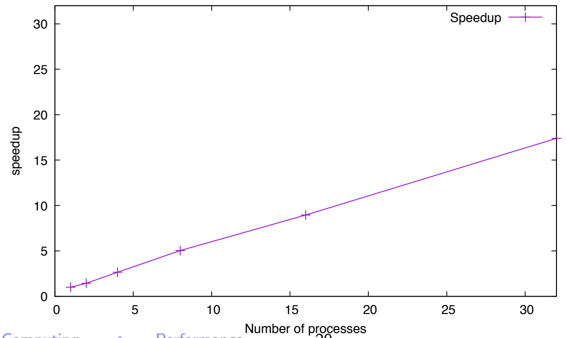
Using `gnuplot`

Much more is possible. . .

- graph speedup
- graph efficiency
- graph multiple plots in one picture
  - e.g.: sequential vs. MPI vs. OpenMP

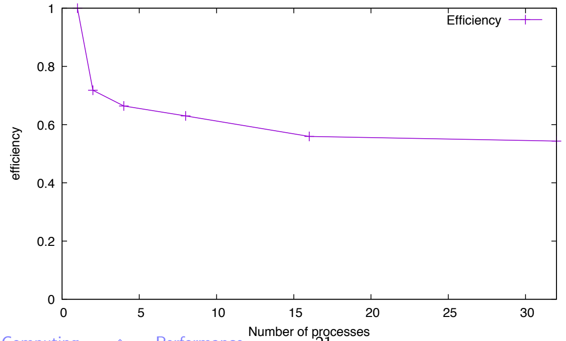
Graphing speedup with gnuplot

```
set output "sat_speedup.pdf"
set xlabel center "Number of processes"
set ylabel center "speedup"
set xr [0:32]
set yr [0:32]
first(x) = ($0 > 0 ? base : base = x)
plot "sat_mpi.dat" using 1:(first($2), base/$2) title 'Speedup' with linespoints
```



Graphing efficiency with gnuplot

```
set output "sat_efficiency.pdf"
set xlabel center "Number of processes"
set ylabel center "efficiency"
set xr [0:32]
set yr [0:1]
first(x) = ($0 > 0 ? base : base = x)
plot "sat_mpi.dat" using 1:(first($2), base/($2*$1)) title 'Efficiency' with linespoints
```


Efficiency
efficiency
Number of processes