Discussion 09

Parallelism (DLP, TLP)

Aditya Balasubramanian

`aditbala [at] berkeley [dot] edu`

Announcements 📣

Agenda

- Amdahl’s Law
- Data-Level Parallelism
- Thread-Level Parallelism

Amdahl's Law

Amdahl's Law

- Amount of speedup that can be achieved is limited by the serial portion(s)

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

Data-Level Parallelism

Flynn’s Taxonomy

- SISD (single instruction, single datum)
  - Most CPUs
  - Everything 61C covered until now
- SIMD (single instruction, multiple data)
  - Vectorized instruction operates on (usually) consecutive data loaded into a vector
  - Usually uses Intel intrinsics
- MISD (multiple instructions, single datum)
  - Multiple instructions operate on a single piece of data (Not widely used)
- MIMD (multiple instructions, multiple data)
  - A lot of operations happening at once with complicated concurrency & reordering structures

Vectorization

- Vectors in computers are memory placed physically consecutively in large registers
- Operations are performed at the same time to all elements
- Most CPUs have extra-big registers
  - Intel AVX has 128-bit registers, which hold 4 * 32 bit integers
  - Also 256-bit registers, which hold 4 * 64 bit floats

Workflow for loop parallelization

- Only works if we’re accessing consecutive bytes of memory!!

1. Iterates over data in strides of 4
2. Load 4 elements into a vector
3. Do operations on the vector
4. Store vector back to memory if needed
5. If data isn’t a multiple of 4, finish the tail with naive implementation

Thread-Level Parallelism

Thread Level Parallelism

- Processes vs. Threads
  a. Processes — large programs
  b. Threads — small segments of programs
- Hardware vs. Software
  a. Hardware Thread — also called cores
  b. Software Thread — OS schedules tasks
- Parallelization Overhead
  a. Parallelization adds in a bit of extra time

Open MP

- Library for multithreading

```
# pragma omp parallel
{
    # every thread will run this code
}
# pragma omp parallel for
for (...; ...; ...){
    # each iteration is run by only 1 thread
}
```

Race Conditions

- Happens when multiple threads are changing public variables
  - We don't know which one gets executed first
- Biggest source of race conditions is on read-modify-write (RMW) operations
  - e.g. `sum += arr[i]`
  - `sum = sum + arr[i]`
  - read: `sum, arr[i]`
  - modify: `sum + arr[i]`
  - write: `sum = modified`
- wrap shared variable `#pragma omp critical` or use reduction `#pragma omp parallel for reduction(+:sum)`

Race Condition Example

```c
static int counter = 0;
// Assume there are exactly 2 threads
#pragma omp parallel for
for (int i = 0; i < 2; i++) {
        counter++;
}
```

```
thread 1: la t0, counter
thread 1: lw t1, 0(t0)      # Thread 1 reads counter = 0
thread 2: la t0, counter
thread 2: lw t1, 0(t0)      # Thread 2 reads counter = 0
thread 2: addi t1, t1, 1    # Thread 2 increments its copy of counter from 0 -> 1
thread 1: addi t1, t1, 1    # Thread 1 increments its copy of counter from 0 -> 1
thread 1: sw t1, 0(t0)      # Thread 1 stores back counter = 1
thread 2: sw t1, 0(t0)      # Thread 2 stores back counter = 1
```

Credits to Rosalie Fang

Thank you!