HW 5 Modeling, Multicore, Multithreading

Joshua Carlson

February 19, 2021

1a. In a symetric multicore chip with 16 BCE's, the percentage of code that must be parallelizable to achieve a speedup greater than 10 is calculated as follows:

$$S = \frac{1}{\frac{1-f}{perf(r)} + \frac{fr}{perf(r)n}}$$

f = the fraction of parallelizable code.
n = total recources (BCE).
r = resources per core.
$perf(r) = \sqrt{r}$.
Plugging in these values gives:

$$S = \frac{1}{1 - f + \frac{f}{n}}$$

Solving for f gives:

$$\begin{aligned} f &= \frac{1 - \frac{1}{S}}{1 - \frac{1}{n}} \\ &= \frac{1 - \frac{1}{10}}{1 - \frac{1}{16}} \\ &= \frac{9(16)}{10(15)} \\ &= 0.96 \\ &= 96\% \end{aligned}$$

96 percent of the code must be parallelizable in order to achieve a 10 times speedup.

1b. The paper *Amdahl's Law in the Multicore Era* argues that having a single powerful core along with multiple 1-BCE cores can give greater speedups because the more powerful core increases sequential performance *and* parallel performance. In other words, sequential code runs faster on the powerful core, and during parallel code, the powerful core contributes by running one of the threads.

2a. In the "roofline" model, the best performance is under the flat part of the roof.

2b. My processor is a Ryzen 3 3200u. Its maximum floating point computation speed is 6.661 GFLOPS. This figure was found at www.cpubenchmarks.net.

3a. When running the sequential matrix multiplication code on a matrix of size 1000, the mean fraction of time spent in the matrix multiplication code was 100.07%, capped to 100%. Mean total running time was 5.65 seconds. My system is dual core with 4 logical processors. Using Amdahl's Law, the estimated speedup of parallelizing on 4 threads is:

$$\begin{aligned} S &= \frac{1}{(1 - 1.00) + \frac{1.00}{4}} \\ &= 4 \end{aligned}$$

Estimated runtime is 1.41 seconds.

3b. I implemented matrix multiplication using pthreads, including cache optimizations that can be turned on and off with compiler flags. The first optimiztion was to accumulate the matrix multiplication of a row in a local variable. The second was to transpose the second matrix before multiplication. The transposition allowed the matrix to be accessed in column-major order. I then measured the performance using `time`, and `perf stat -e cache-misses:u`. I found that both optimizations decreased cache misses and runtime, but the time savings did not corelate to the number of cache misses. For example, accumulating to local decreased cache misses by 20 percent and runtime by 35 percent. Transposing the matrix decreased misses by an *order of magnitude*, but only decreased runtime by 9 percent.

I think that maybe the performance tool is tracking low level cache misses: even though we miss on L1, we still get major performance gains because we are not going all the way to the last level. This would also mean that L2 is still significantly faster than LLC, while not being significantly slower than L1.

Size = 1000, Threads = 4		
Optimizations	Number of Cache Misses	Total Runtime (sec)
None	3.1×10^7	2.24
Accumulate to local	2.5×10^7	1.47
Accumulate to local + Transpose	2.2×10^6	1.34

3c. Multithreading alone did not get me to the time predicted by Amdahl's Law. However, once I optimized my code to reduce cache misses, I easily reached and exceeded that time.

3d. See excel spreadsheet.