EDA 284

Lab 2 – Q1 2018

The purpose of this lab exercise is to acquaint you performance monitoring and understanding parallel program performance. For this you will be using the same code as in the previous lab.
By now you should have available three versions of the life game program.
1. The serial version provided to you
2. The parallel version using threads
3. The OpenMP version you made for the lab.

For this lab we are goind to use **perf** to get some insight into the runtime characteristics of each version of your programs.
For a deeper tutorial on perf look ta (https://perf.wiki.kernel.org/index.php/Tutorial).

Go through the steps in this lab one by one and answer all questions in your lab report.

Part A.

Perf can be used via the command line to diplay statistics gathered through a program execution on the CPU. It supports a series of events, however for this lab we are going to use the following:

instructions
L1-dcache-loads
L1-dcache-load-misses
LLC-loads
LLC-load-misses

These are in order:
The number of executed instructions
the L1 data cache loads
the L1 data cache load misses
The last level cache loads
The last level cache load misses

An example of using perf to monitor one of these is:

perf stat -e cycles,instructions ./life_par

And the output is:

Performance counter stats for './life_par

329 431 340 130 cycles
914 175 650 346 instructions # 2,78 insns per cycle

22,792518047 seconds time elapsed

You can monitor multiple events with perf, however it is best to monitor up to three events at a time to get accurate readings. For our examples you will use two different measurement commands for

each executable:

1. perf stat -e instructions,L1-dcache-loads,L1-dcache-load-misses ./exec <params>
2. perf stat -e LLC-loads,LLC-load-misses ./exec <params>

1. Gather the above statistics for all three versions of the program using the same parameters that were used in lab 1.

2. Plot the statistics for each counter in a graph for every version and note your observations.

3. Try to correlate the different statistics with the performance (runtime) also provided by perf and see if you can explain the difference in performance based on the instruction count and cache misses.

4. For your lab report provide the graphs and tables with all your measurements and a short description of your observations.

5. Optional: How fast can you make each version, try changing the ordering of the loops, make the outer loop the inner loop in the sum_array function and see what happens. See what happens for OpenMP and Pthreads and also for the serial version.

Part B.

In this part we will take a closer look at false sharing. Take a look at the file false_sharing.c and notice that there is no data sharing between threads. All the threads do is update a value in a struct.
Compile and run the example as is and note the execution time.
g++ -std=c++0x -pthreads -o false_sharing false_sharing.c

Then uncomment the line

// #define PADDING_ENABLED

To “pad” the struct so that each threads data resign in a different cache line.
Compile and run again, what is the execution time now?

Use perf to see what happens to the L1 data cache misses, is the number of instructions different?

Run the false sharing program for different number of threads (1,2,4) and see what happens when padding is enabled/disabled.

Play around with the line:

uint8_t padding[64 – sizeof(uint64_t)];

This line of code fills the struct with as much data as needed to have a total size of 64 Bytes (The size of a cache line in our setup).

what happens if you change 64 to 32 while padding is enabled?

Report your measurements in your lab report and shortly describe your conclusions.