QUALITY OF SERVICE

MacOS Job Scheduling on the M1 Chip

MAY 4, 2022

DILLON BOATMAN
Arkansas State University

I. Introduction

In history, the term multiprocessor has been used to refer to computer systems that provided two or more physical processors; each processor in this system usually contained one single cored CPU. This term has since evolved to have several meanings, but in this work we will be using the term focus more on Asymmetric Multicore process scheduling, namely within the M1 Silicon Chip. Asymmetric Multicore processing is used to explain a system containing one CPU with multiple cores of varying specialty.

Since Apple's release of the M1 Silicon Chip in November 2020, there has been little research on the chip's performance. This work strives to provide practical evidence of how this system schedules it's work split between performance and efficiency cores and timing analysis compared to synchronized work.

II. Background

The M1 processor is ARM based, consisting of half performance cores, with the other half consisting of efficiency cores. These are also referred to as FireStorm and IceStorm cores. For the system that is being tested in this work, it consists of an even 4-4 split between the two respective types. This is the first deviation from the Intel Chips for MacOS, which seems to not be leaving their system anytime soon. Especially since they are now offering this in the iPad Pro, iMac Mini, MacBook Air, and both 14" and 16" MacBook Pro systems.

MacOS's job scheduling is referred to as a system called Quality of Service. Essentially, each task is given a priority number within the ranges of QoS. A lower priority will most generally be a background process that will undoubtedly end up on the

IceStorm cores, and a higher priority task, labeled User Interactive within the operating system, will land itself in the FireStorm cores.

The spread remains consistent for the IceStorm cores even when they are showing full capacity use. That is, if a task is given priority 9, it will be spread across the IceStorm cores even if they are at full capacity. While this may be the case for background processes, later in this work we will show that the opposite impact is held for higher priority processes.

III. M1 Specs

For this experiment I am using the latest MacBook Air. This laptop is equipped with Apple's new 8-core M1 Silicon Chip, as well as a maximum 16GB's of unified memory for this model. The operating system it is running is MacOS Big Sur, version 11.5.1. The g++ compiler version is 11.2.0, with the OpenMP library being used for multithreading.

IV. Experiment

To test this CPU, an incredibly CPU intensive task must be written. This will display how the CPU performs and how it spreads data across cores. With the help of Instructor Saldivar, a Monte Carlo simulation to predict the value of PI is written. The synchronized code was written by Mr. Saldivar, but the parallelization was written by me.

This task entails doing a large sum of iterations, with each iteration generating two random x and y coordinate plane values. With this, a multiplication operation, it will do a check to see if $x^2 + y^2$ is less than or equal to 1, it will increment the shared counter. See the code snippet below:

```
if (x*x + y*y <= 1) {
      ++count;
    }
```

The problem of each thread sharing is resolved by using the *reduction* functionality within the OpenMP library. This allows for each process to increment the counter, without having to explicitly create a critical section within the code that would render the performance to that of a synchronous program. Finally, the sum of the count will be multiplied by 4 and divided by the number of iterations. The entire function can be referenced below:

```
double approximatePI (int iterations, int numThreads) {
  int count = 0;
  std::uniform_real_distribution<double> dist(-1.0, 1.0);
  std::mt19937 gen(42); // create and seed random number generator
  #pragma omp parallel for reduction (+:count) num_threads(numThreads)
  for (int i=0; i < iterations; i++) {
    double x = dist(gen); // generate a random double in range [-1.0, +1.0]
    double y = dist(gen); // generate a random double in range [-1.0, +1.0]
    if (x*x + y*y <= 1) { ++count; }
  }
  return 4.0 * count / iterations;
}
```

With this code in mind, the experiment is conducted with the constant of 1,000,000,000 iterations. Alongside this, a varying pool of threads are tested: 1(synchronous constant), 2, 4, 8, and 10. With this CPU only containing 8 single-threaded cores, we will be able to analyze the performance when there are more threads than can be allocated on cores as well as the contrast to this.

As for metrics, a timing analysis is conducted for each pool of processes. As well as this, MacOS provides a graphical rendering of each CPU core displaying the spread of utilization on each core. The cores for this are split up and labeled by number and core type, either being Performance or Efficiency.

V. Results

For this section, the timing analysis will first be discussed. As was accounted for, the synchronous task was the most time intensive at 89.4746 seconds. With one more thread, the process execution time is nearly halved at 55.9997 seconds. Now for the most interesting portion of the result section, the fastest thread pool was 4 allocated threads at 40.3428s. As for the 8-core pool, which caps out the core usage that can be allocated, the speed was slower at 54.8352. A table of easier to read values is displayed below:

Threads	Iterations	π	Time (s)
1	1 Billion	3.14163	89.4746
2	1 Billion	3.13221	55.9997
4	1 Billion	3.09683	40.3428
8	1 Billion	3.11374	54.8352
10	1 Billion	3.11355	54.4361

These number would be synonymous to what we would expect following the discussion of how the M1 chip schedules processes. The split between Performance and Efficiency cores is drastic, showing a definite decrease in performance when the tasks must rely on the Efficiency cores. When the tasks can be solely ran on the Performance cores, the time is greatly improved.

The discussion now moves to the visual representation of the actual CPU usage during each task. First the images will be displayed and an analytic description will follow.

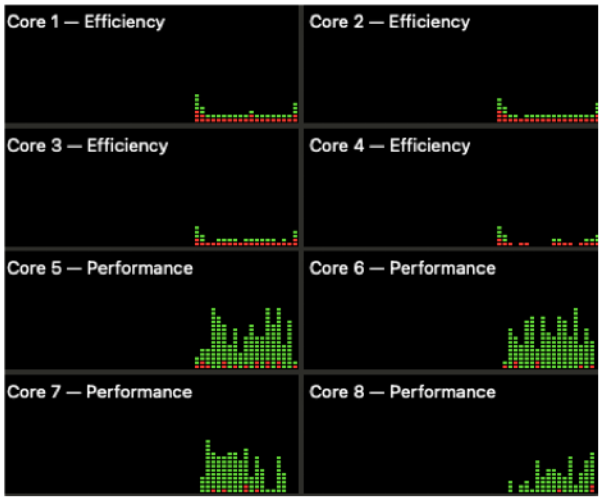

Figure: 1 thread

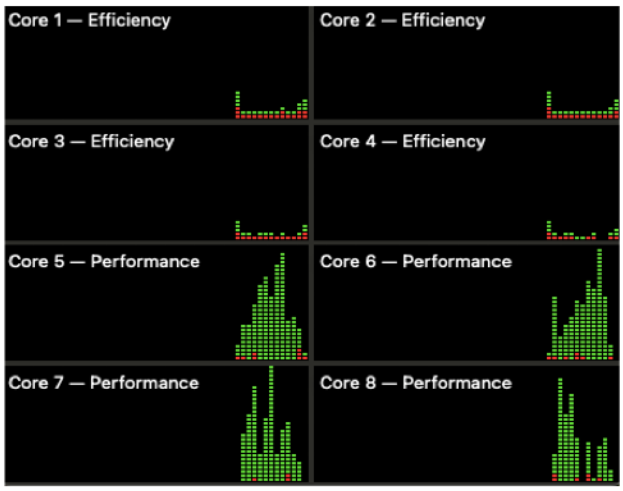

Figure: 2 Threads

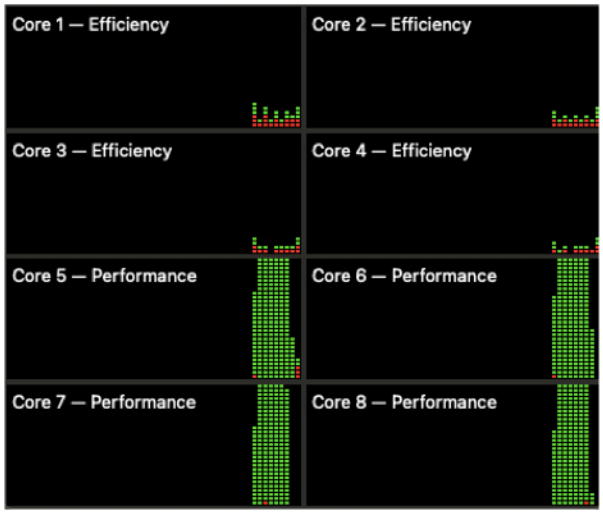

Figure: 4 Threads

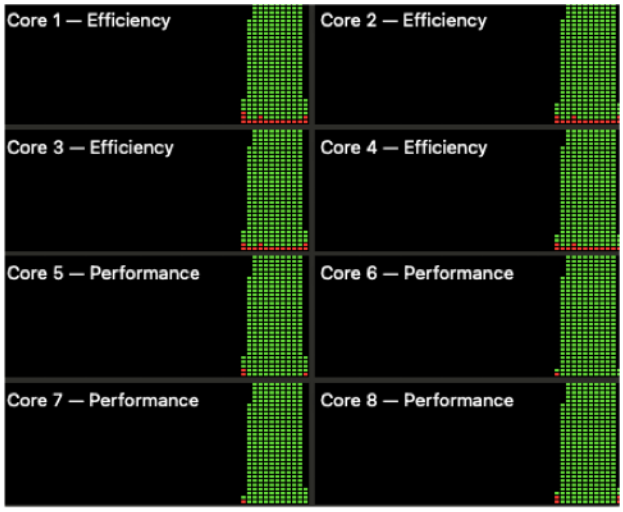

Figure: 8 Threads

For brevity's sake, the figure of 10 threads was left out for space because it was incredibly similar to the image of 8 Threads. Starting with the single threaded constant, you can see that the process seems to be spread across all 4 performance cores, but mainly the first 3 cores. This seems to slowly ramp up with 2 threads, utilizing the first 3 and starting to creep into the 4th thread.

As for the 4 threaded pool, it seems to follow a uniform distribution of utilization for each of the 4 performance cores. Lastly, as expected, the 8 threaded pool shows a uniform utilization all 8 cores.

This processor shows to maintain the reliance on performance cores when doing computationally expensive tasks. It also shows to spread the task among cores, even when there is less threads than cores used. To reference this, view the 2 threaded pool. Even though there are only 2 threads allocated, it seems to show a spread of utilization across all 4 performance cores. It also shows to utilize the efficiency cores only when entirely necessary. Referencing the 8 threaded pool, it shows to uniformly use all 8 cores at full capacity. It is interesting to note that the usage of 8 full cores is noticeably slower than when solely using 4 performance cores.

VI. Conclusion

MacOS's decision to transition to the M1 ARM processor was a shock the computing community, with this being the latest change from Intel Xeon Processor. With that being said, they scheduling on this system shows an incredible smoothness pulls toward the position of keeping this processor in production.

The split between IceStorm and FireStorm cores shows an effectiveness toward non-blocking background operations, whilst keeping the higher performance cores idle for incoming high priority tasks. The division of processing core capability is definitely evident, with the high performance tasks showing a negative impact in speed when ran on the slower efficiency cores.

This work strives to provide practical evidence of the division of work across two variations of core on the M1 MacOS chip. While this work only focuses on testing the scheduling of higher priority processes, work still needs to be written on the effectiveness of lower priority task scheduling.

VII. Works Cited

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts*. Wiley.
- Hoakley, 13, B. on J., Bob, 13, hoakley on J., 13, D. on J., Dave, 14, T. on J., Tony, 14, hoakley on J., & 18, T. on J. (2022, January 12). *Scheduling of processes on M1 series chips: First draft*. The Eclectic Light Company. Retrieved May 3, 2022, from https://eclecticlight.co/2022/01/13/scheduling-of-processes-on-m1-series-chips-first-draft/
- Hoakley, et al. "How M1 Macs Feel Faster than Intel Models: It's about QoS." *The Eclectic Light Company*, 17 May 2021, https://eclecticlight.co/2021/05/17/how-m1-macs-feel-faster-than-intel-models-its-about-qos/.
- Christopher Saldivar's Starter Monte Carlo PI Prediction cod