Alexander Powell
CS 524
10.13.2016
Midterm Research Paper

**Cache Performance on Many-Core and Many-Thread Machines**

The importance of an efficient cache managing system on today's computers cannot be overstated. The current philosophy is that we should construct our programs and applications to maximize the number of times something is fetched from a cache and minimize the number of times one must fetch from data memory. This is the case because the memory latency is so much slower than cache latency on most modern machines. There are two general directions that multithreaded programming is headed. One on hand we are moving towards machines with many cores, where a chip can hold dozens of cores. This design relies on efficient cache utilization to mask the memory access latency, and are known as many-core machines.

The alternative to this practice is to build machines that can process numerous threads at the same time. In the past, many-thread machines were more suited towards GPUs for processing graphics and other multimedia applications and generally do not use caches at all, but instead rely on thread level parallelism. There are pros and cons to both approach and some perform better in different environments than others. The interesting thing to note between the two strategies is that while both extremes might yield a high performance, a valley forms between the two extremes. Figure 1. below displays this behavior.

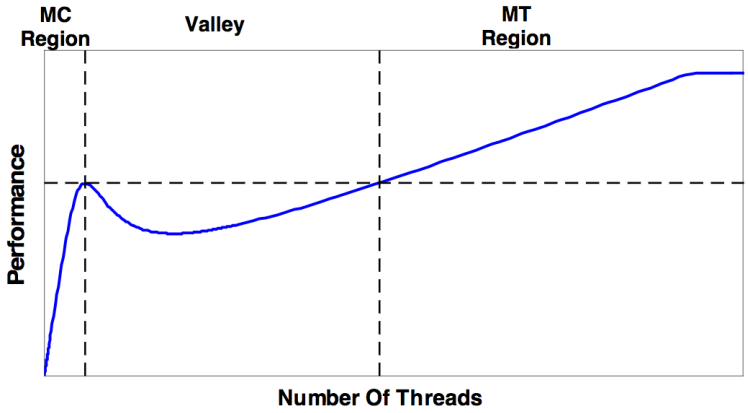


**Figure1. Many-core vs many-thread machines**

You can see above that on the left side of the graph the performance improves at a rapid rate for each additional thread that is added. This is denoted as the many-core (MC) region. However, at a certain point this performance slows down, and actually begins to drop by some amount before slowly increasing again. Eventually the performance meets and exceeds what it reached initially and keeps increasing for some time until it eventually plateaus. The many-thread (MT) region begins at the point where the performance reaches its last highest point.

The significant thing to take away from this graph is that between the many-core and many-thread regions, there's actually a valley, where program performance drops. This is because the cache becomes too small for covering the growing stream of access requests and memory latency is no longer adequately masked by the cache. Guz et.al. propose a model to explain this "valley" effect based on a number of different parameters, like number of processing elements, cache size, cycles per instruction, memory latency, cache latency, frequency and cycle time. By using their model and slightly modified parameters, we can

represent a similar, albeit less smooth, figure displaying the valley behavior previously mentioned.

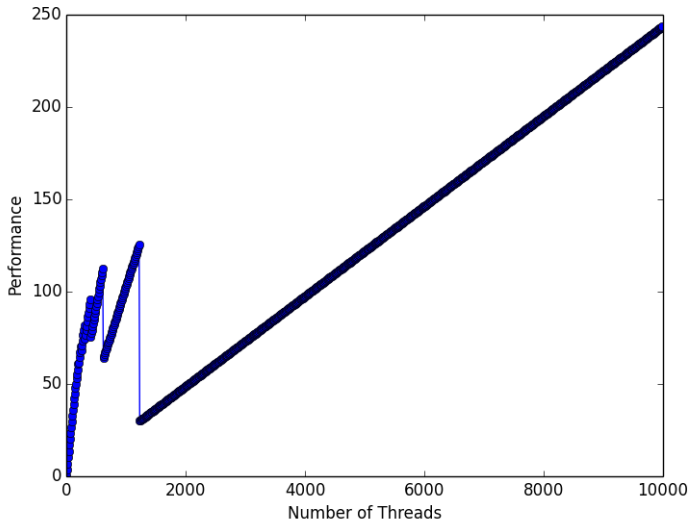


**Figure 2. Shows the relationship between the number of threads and the performance in GOPS. While not as smooth as Figure 1., we can clearly see a relatively steady increase in performance until around 1000 threads, and then a sharp drop. The trajectory stays constantly increasing afterwards.**

For the sake of simplicity, this model assumes that threads can't share any data. Since this is only a model of cache behavior, the parameters can clearly be altered to represent different scenarios. For example, the more computation instructions per memory instructions are given, the steeper the performance climbs. This is due to the fact that as more instructions become available, fewer threads are needed to fill the stall time. This results in the "plateau" being reached sooner and as computation instructions per memory instructions approaches infinity, the valley disappears entirely. Another factor that affects the shape of the plot is memory latency. It's no surprise that as the latency for memory accesses grows larger and larger the effect of the valley will be felt more and more. This behavior is significant because

the memory latency is only expected to grow in the coming years. Additionally, as cache size increases the plot gets shifted to the right. That is, a larger cache will inflate the many-core area.

However, it's not always easy to predict how a given application will perform on a given architecture. In order to update our model, we need to account for both the architecture as well as the application characteristics. Many of the parameters previously mentioned still play a factor in this new model, but now we must also take into account things like off-chip bandwidth and operand size.

Figure 3. below shows how the performance of a program eventually evens out and plateaus. The point that this occurs is known as the bandwidth saturation point.

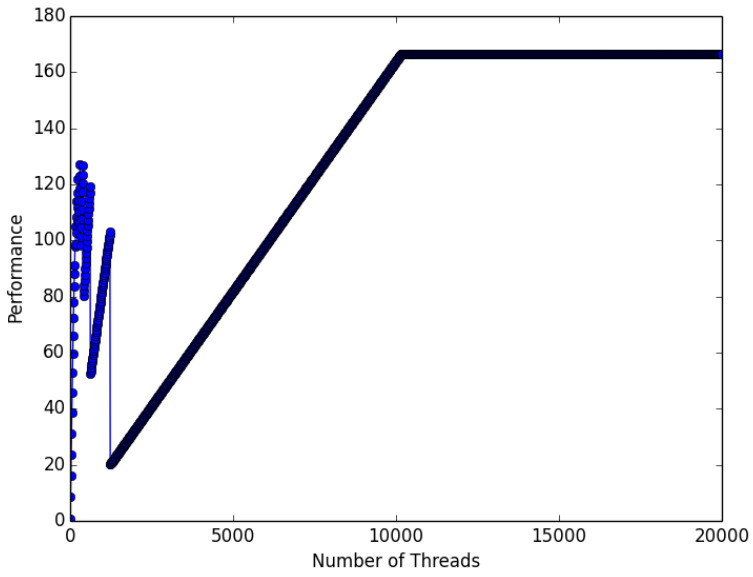


**Figure 3. We can see that performance increases with compute intensity, until the bandwidth saturation point is hit.**

It's clear that workloads with higher data locality better exploit the cache and therefore the "plateauing" effect is delayed. The shape of the graph is dependent on how fast the hit rate degrades as a function of the number of threads. In general, it can be said that a high compute/memory ration decreases the need for caching and multi-threaded scaling.

In conclusion, there are two main philosophies of multithreaded programming. One side promotes a many-core architecture while the other promotes many-thread. While both of these can offer optimized performance, it's important to understand the relationship with the cache that can lead to a performance "valley" somewhere between the two extremes. That is, the number of threads is not necessarily directly correlated with the performance of a system.