COMPSCI 590S Systems for Data Science **Fall 2016**

Lecture 6: February 12, 2018

Lecturer: Emery Berger *Scribe(s): Amit Rawat*

Previous class

- Cache coherence protocols, e.g. MESI.
- Required to maintain data consistency in the memory hierarchy.

This class

- Scalability of cache coherence protocols.
- Cache details.

Scalability of cache coherence protocols

Cache coherence protocols are important to maintain data consistency in the memory subsystem and processor caches. Scalability is defined as the ability of a system to cope with larger workloads, or it potential to be enlarged in order to accomodate that growth. It has a direct impact on the execution time of a process. The performance of a scalable system increases proportional to the capacity added. We would like to minimize the overhead in maintaining the new hardware added.

Let us look at an example, assume a multi-core system with N processors. If we implement a very naive cache coherence sytem where a processor communicates with every processor to send cache invalidation message, we get a system where N^2 messages are sent. This is not ideal at all, and does not scale well.

A directory based cache coherence protocol can be used to scale up to 128 processors. A global directory is maintained in the main memory, where a diretory entry contains information about a cache line, and the state of this cache line in all caches. Caches can simply look up this directory on a per-need basis rather than sending messages over the bus.

Cache details

Amdahl's Law

It gives the theoretical speedup that can be gained by improving the resources (mainly processors) of a system. It is often used to figure out how much speedup can be achieved in parallel programs. A parallel program tries to run independent parts on different processors at the same time to achieve faster execution. However, the speedup is limited by how much of the program is parallel. The speedup is given by the

equation $S = \frac{1}{(1-p)+\frac{p}{n}}$, where S is the maximum possible speedup, p is the portion of the program that can be parallelized (usually given as a percent), and n is the number of processor cores or threads available.

An utopian world would have a compiler, which turns a sequential program into a parallel program automatically. However, writing parallel programs is hard for humans let alone a machine. Parallel programs can be written by either creating multiple threads (multi-processing) or creating multiple processes (multi-processing). The problem with multi-threading is that the address space is shared between threads, which means that one has to be really careful about threads not stomping on each others memory. The number of ordering or interleavings of instructions is T^I, where T is the number of threads, and I is the number of instructions. Emery says that threads will be seen as a system mistake in the future.

Multi-processing has no such issue, but the processes need to use OS provided communication channel between processes such as message passing.

Fork-join parallelism

Lets look at the program below:

```
int x[100000];
for (int k = 0; k < 100000; ++k) {
    x[i] = x[i] * 2;
}
```

Different threads can work on different portions of the array x at once, as there is no loop carried dependency in the above code snippet i.e. one iteration of the loop does not depend on the other. A possible execution may consist of four threads, where the threads process indices $[0, 25k)$, $[25k, 50k)$, $[50k, 75k)$, and $[75k, 100k)$. The join operation does nothing, as there is no need to consolidate results after the loop ends. Such a process may be required in complex programs.

If however, the above program was transformed to something like below

```
int x[100000];
for (int k = 0; k < 100000-1; ++k) {
    x[i+1] = x[i] * 2;
}
```

It would be difficult in its current form by an automatic compiler to divide the program in parallel. However, a "smart" compiler could figure out the array value just by looking at the transformations on the elements, that every element gets multiplied by twice its predecessor.

Polyhedral analysis is one such way which determines the shape of the structure formed by such operations, and run parts in parallel, but not every program may be amenable to it.

An easier way to parallelize is to constrain operations allowed, which means a custom DSL, or to parallelize manually, which requires experienced programmers.

There are libraries available to make writing parallel programs easier, as they abstract away many expensive operations, and are well tested. BLAS libraries by Intel is a well known library. FFTW is another.

Performance with threads is challenging, as there are concerns regarding synchronization, and contention at cache line/memory level. If P is the number of threads, and P is the number of processors, then three cases arise.

- $T << P$, we have insufficient number of threads,
- $T >> P$, too many threads leading to resource contention,
- $T == P$, still bad, as other processes running on the system.

Coarse-grained parallelism, $T \approxeq P$. If there are no dependencies between parts of system such that threads can run on different parts without synchronization, such a program is called emabarassingly parallel. If T_1 is the time taken to run the program with one processor (serial), and T_p is the time taken to run the program with p processors, the goal is to have $T_p = \frac{T_1}{p}$. Let T_∞ be the time taken to execute the program in there are ∞ processors available. The critical path is defined as the longest path from start to end, that needs to be executed in sequence.

$T_p \approxeq \frac{T_1}{p} + T_\infty$

If only 10% of a program is sequential, then the program can't be made faster than 10x. $T_\infty = 0.1$, then maximum 10x speedup. $T_\infty = 0.01$, then maximum 100x speedup. In real-world programs however, a speedup of more than 30x is hard ot achieve.

In T_∞ all lock and unlock times are added. Practically, add more cache for speedup, as cache is 10x faster than RAM.

Cache misses can be partitioned into the following categories:

- Compulsory misses - when the process has just started running, then none of the data is in the cache.
- Capacity miss - if the cache is already full.
- Conflict miss - two addresses share the same cache line, even if there is space available in the cache.