University of California, Irvine
CS243 High Performance Architecture and Compiler Additional Questions

Jiacheng Feng
10601635

1. why do we need parallelism

The two primary reasons for using parallel are saving time and solving larger problems. In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel can be built from cheap, commodity components. On the other hand many problems are so large and complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory. Meanwhile there are still many reasons one in which is providing concurrency. A single compute resource can only do one thing at a time however computing resources can be doing many things simultaneously. For example the access grid provides a global collaboration network where people from around the world can meet and conduct work virtually.

In consideration of types of parallelism, there are instruction level parallelism and task level parallelism. A computer program, is in essence, a stream of instructions executed by a processor. These instructions can be reordered and combined into groups which are then executed in parallel without changing the result of the program. This is called the instruction level parallelism. Modern processors have multi-stage instruction pipelines that many instructions in different stages can be executed in one processor simultaneously. This strategy can improve the speed many times comparing with the formal way. On the other hand task parallelism is the characteristic of parallel program that entirely different calculations can be performed on either the same or different sets of data. Nowadays, there are many processors within one computer, many cores within one processor and a lot of available computer resources. Thus if we can use parallelism properly, we will save lot of time and solve larger problems. We can take advantage of non-local resources as well and overcome memory constraints.

2.The benefits and limitations of multicores shared memory

Many modern computer systems and most multicore chips (chip multiprocessors) support shared memory in hardware. In a shared memory system, each of the processor cores may read and write to a single shared address space. These designs seek various goodness properties, such as high performance, low power, and low cost. A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location. At the same time there is lower communication overhead because multicores can access the same memory address.

However, it is not valuable to provide these goodness properties without first providing correctness which is the most limitation of multicores shared memory about consistency and coherence. Consistency models define correct shared memory behavior in terms of loads and stores (memory reads and writes). The sequential consistency is intuitive for programmers and guarantees that all the memory operations executed across different threads appear in one total order. However many compiler and hardware optimizations can break the sequential consistency semantics. Data-race-free-0 memory model can guarantee sequential consistency for race-free programs and allows the most of optimizations. However, DRF0-based memory models provide weak or no guarantees for programs with data races. For instance, C++0x gives no semantics to racy programs. As a result, compiler and hardware optimizations can introduce arbitrary behavior for a racy program, potentially compromising the program's correctness and security properties. As this situation is unacceptable for type-safe languages, the Java memory model (JMM) provides a weak semantics for racy programs that precludes a class of adversarial behaviors. This semantics is subtle and complex and therefore makes understanding and debugging multithreaded program executions extremely difficult. It also makes it difficult to develop compiler and hardware optimizations.