Parallelism

We have been discussing the abstractions and implementations that make up an individual computer system in considerable detail up to this point. Our model has been a largely sequential one, however. We think of the system as executing an instruction at a time. This is reinforced by the languages we typically use for programming (such as C, C++, Java, C#, Racket, and Python) which also have a sequential, one-step-at-a-time model. In reality, neither the hardware nor software is limited to this. Indeed, to take full advantage of almost any modern hardware, we must go beyond this model. We must write parallel software to take advantage of parallel hardware.

In the following, we will describe, at a high level, the well-established forms of hardware parallelism/concurrency that are available to most programmers, and some of the tools/approaches we can use to program such hardware. It is important to understand that while the development of parallel hardware is well established, the development of tools to help create parallel software is much less so. Additionally, in many cases the very algorithms that we might seek to implement need to be rethought in light of parallelism.

The focus of this document is on how to exploit the concurrency of hardware to get individual jobs done more quickly, or to allow us to tackle jobs that would otherwise be impossibly big. These two focuses are typical of the performance needs of parallel computing or high performance computing, for example to support science and engineering, data mining, deep learning, and similar needs. There are many other reasons to exploit hardware concurrency as well.

Parallel Hardware

An individual computer system is internally highly parallel, doing multiple things simultaneously without any programmer input. For example, at any one point in time, many instructions may be simultaneously "in flight" (in the process of being executed), while the memory system is at the same time prefetching more instructions and the data to feed them.

Some of this concurrency in the hardware is externally visible, and directly controllable by the programmer. This control must be used if he or she is to extract the highest possible performance out of the hardware. An important source of controllable concurrency in the hardware is the vector instructions that we briefly discussed when learning about floating point. We can also leverage the concurrency of the hardware through controlling the dependencies among the instructions and data we feed the hardware, and their reference patterns. Chapter 5 of your book gives more details on this point if you are interested.

In addition, any modern computer system, from a phone on up, has at least one multicore processor. For example, our class server has two processors, each of which has ten cores,

and each of those cores has two hardware threads[1], for a total of forty hardware threads. To fully leverage all that the machine is capable of, the programmer needs to provide it with a program that can do forty things at a time. And each of those forty can itself use vector instructions and other forms of internal parallelism.

Much larger scale concurrency exists when we think about collections of computer systems that are interconnected by fast networks. For example, Northwestern's Quest system for research has about 700 machines somewhat like our class server, and to take full advantage of it, the programmer needs to provide it with a program that can do about 16,000 things at a time, each one of which can be a vector instruction. As is commonly the case for hardware like it, the programmer needs to leverage four kinds of hardware parallelism to take full advantage of Quest**:** distributed memory, shared memory, vector, and accelerator. This holds true even for the largest scale machines currently built or envisioned, which require the programmer to provide them with a billion things to do at a time.

Shared Memory Machines

The most common sort of parallel hardware that a programmer will encounter is a shared memory machine, such as our class server or any common server/desktop/laptop/phone. A shared memory machine presents a single common memory (more specifically, a common address space) to all of the hardware threads. This makes it possible to write a program in which threads communicate simply by reading and writing common addresses. Technically, there is no actual single common memory. The caches continuously run a distributed algorithm called a coherence protocol that produces the appearance of a single common memory. This is the kind of machine you will program in this class.

Distributed Memory Machines

Achieving shared memory becomes increasingly difficult and expensive as the number of hardware threads is increased. An alternative model is to segment hardware threads into small groups, and then have each group be paired with its own private memory, a pair that is called a node. Between nodes, it is not possible to communicate using common addresses. Instead, communication between nodes is done using explicit messages sent over a network. More compactly, a distributed memory machine consists of a collection of shared memory machines (nodes) interconnected with a network. This is how a cluster, data center, supercomputer, etc work. The core difference between these is in the nature and performance of the network.

---

[1] By "hardware thread", we mean an consumer of a stream of instructions. On x64, %rip is the central element of a hardware thread – it points to the current instruction in the stream. The instruction is fetched, decoded, executed, and then the hardware goes on to the next one. By "software thread", we mean the producer of a stream of instructions. The model of a software thread can vary quite a lot (this is software after all), but an important one is that of the thread abstraction provided by the operating system kernel as described in the book. Up to this point in this course you have been programming a single software thread to feed a single hardware thread.

Accelerators: GPUs, Phis, FPGAs, …

Shared memory and distributed memory machines both rely on the performance of individual processors. Processors, like the x64 processors you encounter in this class, are the subject of massive investments to continously enhance their performance. The addition of vector instructions is one example. However, there are other ideas for either augmenting processors or replacing them altogether.

The most successful of these ideas currently is to use a "Graphics Processing Unit" (GPU) for general purpose computing. Modern GPUs used on graphics cards have special support for this, and, indeed, the highest end GPUs for general purpose computing do not even have any graphics interface to plug a monitor into. One way (not the only way) to think of a GPU is that it provides vector processing with extremely long vectors. GPUs appear to many people to be on a better "technology curve" than mainstream processors, meaning that their raw performance, performance/dollar, and energy-efficiency appear to be advancing faster than mainstream processors.

Multiple vendors make GPUs with NVIDIA probably making the most widely used ones. A common scheme is to have one or more GPUs available as expansion cards in a shared memory machine or node. The programmer can then use both the processors of the machine and the GPUs together. However, the ways in which the GPU and the processor are programmed are radically different, making this a challenge.

Intel has been developing Phi, an alternative to GPUs that augments x64 processors with some of the properties of GPUs while retaining x64's familiar programmability. At first glance, a Phi looks much like an ordinary shared memory x64 machine, just with a very large number of hardware threads. The programmer can easily port existing shared memory programs to it, and then incrementally take advantage of the Phi's special properties.

An emerging accelerator model, one which has been under discussion for about 30 years, and which is now seeing substantial commercial investment by Intel and others, is the use of Field Programmable Gate Arrays (FPGAs) for computing. Unlike traditional processors, GPUs, or Phis, which have fixed hardware, an FPGA is *malleable hardware*. The programmer in effect builds custom hardware instead of a program.

It may appear that some of these forms of hardware concurrency are too esoteric to be of concern to most programmers, but it is important to understand that Moore's Law and other forces push esoteric hardware toward the commonplace, and they also push the scale of hardware concurrency up exponentially with time. This quickly makes exploiting hardware concurrency everyone's problem.

Parallel Software

Despite several decades of research, we are not yet at the point where there is a parallel programming language that has received widespread adoption. Most development of parallel software is done today using sequential languages (like C/C++) augmented with libraries and small extensions.

Parallel Algorithms

The design of parallel algorithms is a bit different from their sequential counterparts. Except for simple situations, parallel algorithms are not simple extensions of sequential algorithms. Their design is strongly concerned with worst-case asymptotic performance. However, unlike sequential algorithms, there are two ”big O” values of concern. One is “work complexity” – the amount of work that is done. The other is “depth complexity” – the longest path in the computation, or how long the algorithm takes given an infinite number of processors. Consider an algorithm like quick sort, which you may know as being of complexity O(n log n), meaning it requires doing O(n log n) work and will require O(n log n) time to do that work. In contrast, a parallel quick sort is of work complexity O(n log n), but it has depth complexity of only O(log n)! This means it is possible to sort n things in O(log n) time, which may seem surprising. The general point here is that the fundamental limits of algorithms and problems change with the introduction of parallelism because the abstract machine is no longer limited in a fundamental way.

Automatic Parallelization

Over the past 40+ years, massive amounts of research dollars have been spent with little success in pursuit of automatic parallelization – producing parallel programs by compile-time analysis of programs written in ordinary sequential languages like C/C++. There is one major success story here, however: automatic vectorization, which is particular to vector machines. While vector machine technology was originally confined to very expensive supercomputers, it is now everywhere, for example in the vector instructions of an x64. As part of their optimizations, compilers such as Intel’s icc are capable of applying automatic vectorization to the inner loops of your C/C++ code to take advantage of the vector instruction set of the processor and therefore run significantly faster.

Shared Memory Programming

Shared memory programming targets shared memory hardware. The overall idea is that that it is a generalization of the sequential programming model that programmers are already familiar with, and so, it is argued, the easiest to adopt. In its most common form, the core programming concept is that of having multiple software threads, as many as the programmer wants. The operating system kernel ultimately manages these software threads and dynamically maps them to hardware threads.

Pthreads (POSIX threads) is a standard for thread-based programming that is available on most platforms. This is the standard you’ll use in this class for the last lab and is described in Chapter 12. In pthreads, the core generator of parallelism is thread creation (the pthread_create function), and thread creation can be thought of as a form of function call in which the caller does not wait for the callee to finish. Instead, the caller and callee run simultaneously, both accessing the same common memory provided by the hardware.

While this may seem like a simple enough model, the problem is that a very wide range of bugs are now possible unless the programmer coordinates threads (for example, the

caller and callee above) correctly. Getting thread coordination right is a very tough job in most cases for most programmers. Worse still, incorrect or missing thread coordintaion can result in entirely new forms of bugs (race conditions, deadlocks, livelocks, …), most of which are intermittent (they don't show up every time), and may even change their behavior when being debugged. These potential bugs stem from the fact that all state of the program is shared, as it is all in the single, common memory by design.

By restricting the forms of structures and algorithms that can be implemented using threads, it is possible to both simplify shared memory programming and to avoid some of its potential bugs. OpenMP is a widely-used tool for this. OpenMP is a small set of extensions to C/C++ that allow a programmer to compactly describe parallelism. For example, the programmer can tag an outer loop as being parallelizable. This allows the compiler to make assumptions about the properties of the loop that enable it to generate code that runs the loop's interations as separate threads. The programmer only needs to get the tag right, not all the thread coordination code.

Distributed Memory Programming

Distributed memory programming targets distributed memory hardware, but can also be used to program shared memory hardware. In contrast to shared memory programming, where all state is shared among threads, in distributed memory programming no state is shared among threads. This eliminates most of the need to coordinate threads, but it also means the threads cannot communicate with each other implicitly through global variables. Instead, all communication is made explicit. This makes for more work for the programmer, but it is necessary to leverage a distributed memory machine. Also, because all communication is explicit, it is less likely that the kinds of nasty bugs noted earlier can occur, although it is not impossible.

The core functionality in distributed memory programming is the ability to send a message to another thread, and to receive a message from another thread. There are dozens of hardware technologies for networking. MPI is commonly used to hide the details of these technologies from the parallel programmer. Beyond basic message passing (send/receive), MPI also allows the programmer to express higher level communication, called collective communication, among a set of nodes.

Accelerator Programming

GPUs are today generally programmed in CUDA (NVIDIA) or OpenCL/OpenACC (most vendors). Phis are programmed using shared memory programming techniques. FPGA programming is essentially hardware design and thus outside the scope of this class.