White Paper

How to sound like a Parallel Programming Expert
Part 2: parallel hardware

Tim Mattson, Principal Engineer, CTG, Intel Corp

Introduction and Historical overview

Parallel computing is old. The oldest parallel computers date back to the late 1950's (A.L Leiner et. al. Proc. EJCC, 71-75, 1958). Throughout the 60's and 70's, top-end computers utilized multiple processing elements and parallel software to achieve greater speed. But these were specialized machines for only the most critical (and best-funded) applications.

In the mid 1980's, a new era in parallel computing was launched with the Caltech Concurrent Computation project. Professors Charles Seitz and Geoffrey Fox led the project and built a supercomputer for scientific applications from 64 Intel® 8086/8087 processors placed at the corners of a higher dimension generalization of cube called a hypercube. This launched what came to be known as the "attack of the killer micros" as large numbers of commercial off-the-shelf (COTS) microprocessors invaded a supercomputing world dominated by "big iron" custom vector machines.

These massively parallel processors (MPPs) came to dominate the top end of computing culminating with the Intel ASCI option Red computer in 1997; the first computer to run a benchmark (MPLinpack) at over one TFLOP (one trillion double precision adds/multiplies per second). Since then, MPPs have continued to grow in size and power with PFLOP (one thousand trillion floating point operations per second) computers currently at the top of the heap.

MPP computers used commercial off-the-shelf microprocessors, but these were connected with custom networks and advanced high density packaging. In the late 80's, a new type of parallel supercomputing emerged called clusters. A cluster is a parallel computer built from large numbers of off-the-shelf computers connected by an off-the-shelf network. We started in the late 80's with Unix* workstations but in the mid-90's as the floating-point performance on PCs became competitive with that found on Unix workstations, PC-based clusters running Linux* took over. Today, clusters are the workhorse of scientific computing and are the dominant architecture in the data centers that power the modern information age.

The weakness for MPP computers (and clusters too) is the software. The burden of making software run in parallel was placed solely on the programmer's shoulders. These machines, sometimes with thousands of processors each with their own distinct memories (so-called, distributed memory MPPs), required the programmer to explicitly break their problems down into thousands or even millions of distinct tasks operating on local data and with all sharing between tasks occurring through explicit messages. Scheduling, communication, data decomposition … everything was done by the programmer with little or no automation.

A second track in parallel computing was shared memory multiprocessor computers. These emerged for supercomputing applications in the late 70's and for business computing in the late 80's. In the mid 90's Intel put multiple CPUs on a single board tightly integrated with the chipset to take multiprocessor computing into the mainstream. These machines were in many ways easier to program. The programmer, with help from the compiler, found concurrency to exploit in their software and expressed this in terms of multiple threads in a single application. These threads share memory so the programmer doesn't have to decompose the data in their application. The operating system took care of the scheduling decisions. It was in many ways a much easier environment to work in. And since the operation systems were multi-threaded, computers could benefit from multi-tasking; i.e. running multiple single threaded applications at one time; i.e. delivering benefits of parallel hardware "for free".

The modern era of parallel computing

Throughout the classic era of parallel supercomputing, Moore's law[i] drove a steady increase in single processor performance. This let most programmers get away with ignoring parallel programming. But this all began to change as the new millennium unfolded. Moore's law is progressing nicely and will do so for the foreseeable future. But to stay within a fixed-power envelope, we must use all those transistors from Moore's law to create multiple cores on a single chip. Parallel computing, once the exclusive purvey of supercomputing, is now a critical technology in **every** segment of computing.

And the core counts, the processing units on chips will be large; 4 and 8 cores today but at some point next decade, we could easily have hundreds of cores. Sounds silly? Note that in 2007,

Intel demonstrated an 80core microprocessor that ran a benchmark at over one TFLOP. It was a single precision TFLOP and ran a simpler benchmark (a stencil computation, not the full MPLINPACK benchmark), but still … in 10 years Intel took TFLOP-scale computing from a 9000+ CPU computer consuming a megawatt of electricity to a single 80-core CPU consuming around 100 watts. That is pretty exciting and amazing!!!

The many-core future is bright and is based on a solid foundation of parallel computing stretching back over several decades. There are many challenges to address on the hardware front, but for the most part, we have a good idea of how to address them.

The biggest challenge to putting power of many core chips into the hands of the computing consumer is the software. Software evolves slowly over years if not decades. Transitioning the software infrastructure into a many-core world will be very difficult and in all honesty, we aren't sure how to pull this off. We know we can make the applications we care about scale using message passing techniques. But these have proven very difficult for general purpose programmers to handle. But we won't worry about software until part 4 of this series. For now, let's make sure we are all on the same page with respect to the hardware.

Definitions

If you want to talk like a parallel programming expert, you need to learn some of the specialized jargon used in describing parallel hardware. Fortunately, there aren't very many terms to learn.

At the top level, we can think of parallel hardware in terms of streams of instructions and streams of data. How these relate to each other gives us a top level to think about parallel hardware. Other cases exist, but the two cases we really care about in parallel systems are:

- Single instruction stream operating on multiple streams of data or SIMD
- Multiple streams of instructions operating on multiple streams of data or MIMD

For a SIMD computer, the parallelism is in the data. You have one program counter that moves through a stream of instructions that operates on multiple elements of data. The vector units on a modern microprocessor are a common example of a modern SIMD computer.

For MIMD computers, each processing element has its own program counter and hence runs an *independent* stream of instructions. Most MPP computers and clusters are examples of MIMD computers.

Streams of data and instructions provide the top level of classification for a parallel computer. The next level comes from the memory hierarchy. For multiprocessor computers, the processors share a single address space (hence why these are sometimes called shared-memory multiprocessor computers). There are two major categories of shared-memory multiprocessor computers:

1. **Symmetric Multiprocessor or SMP computers**: the cost of accessing an address in memory is the same for each processor. Furthermore, the processors are all equal in the eyes of the operation system.

2. **Non-uniform Memory Architecture or NUMA computers**: the cost of accessing a given address in memory varies from one processor to another.

Of course, given the way caches work in a system, almost every multiprocessor computer is to some extent a NUMA machine. But when you have multiple processors connected to a single memory subsystem, the term SMP is typically used to describe the architecture.

The other alternative for the memory architecture of a parallel computer is to give each processing element its own local address space. This is a called a **distributed-memory architecture**. Most MPP supercomputers and virtually all clusters are distributed memory systems.

Putting multiple cores on a single chip

With these basic definitions we can talk more precisely about processors that contain multiple cores. An individual core is a distinct processing element and is basically the same as a CPU in an older single-core PC. In many cases a core includes a vector unit (A SIMD parallel computing unit).

When we put multiple cores on a single die and connect them to a single coherent address space, the chip is an SMP MIMD parallel computer. We call this a multi-core chip. Scaling a single address space in an SMP system is difficult as the number of cores increases beyond a modest number (around 16). If the cores are connected using an approach that can scale to "arbitrarily large" numbers of cores, we call this a many-core chip. An example of a many core chip is the 80-core terascale chip mentioned earlier. This chip used a distributed memory, MIMD architecture.

Can you build a many-core chip that is a shared memory MIMD architecture? Yes, this is possible as long as the memory architecture is scalable and the network connecting the cores is designed to scale up to large numbers. A good example of an SMP MIMD many core chip is the Larrabee chip from Intel.

There are countless details that parallel architects manage as they design many-core chips. The jargon can quickly get out of control. But for the most part, if you remember the basic terms SMP, MIMD, SIMD, many-core and multi-core, you'll be able to keep up with most conversations and sound like an expert.

Next step

So far we have covered the basic ideas of concurrency and parallelism. We've also covered the basics of parallel hardware. Next we'll look at the general issues that come up when talking about programs as they execute on parallel computers.

About the Author

Dr. Timothy Mattson

Tim Mattson is a Principle Engineer at Intel working in the Microprocessor Technology laboratory. He is also a parallel programmer. Over the last 20 years, he has used parallel computers to make chemicals react, shake up proteins, find oil, understand genes, and solve many other scientific problems. Tim's long term research goal is to make sequential software rare. This means he has many years of hard work ahead of him.

[i] http://www.intel.com/technology/mooreslaw/

This example has two parallel tasks, one task is an event-driven UI task, and the other performs an acquisition from a peripheral device. LabVIEW recognizes that it can execute the two loops independently, and in a multiprocessing or hyperthreaded environment, often simultaneously. Figure 1 - Implicit Parallelism in LabVIEW. Notice that these the above example does not include code for explicit thread management.Â LabVIEW also offers special structures that will map code to parallel hardware resources for explicit threading. For example, one such structure is the Timed Loop.Â Figure 1 below demonstrates how two Timed Loops will create two unique threads which can be balanced across two separate cores of a multicore system. Figure 2 - Explicit Parallelism in LabVIEW. Introduction to the computer hardware and software from a parallel programming point of view. Goes into Flynnâ€™s taxonomy of computer systems and what concurrency and parallelism really mean.Â This article is the first in a series called Parallel Programming Primer where we will share some of our hard earned experiences parallelizing our data processing tasks in hopes that it will help others who encounter similar situations. In this first part we introduce the basic concepts of parallel computing and the parallel programming model. This should nicely lay the foundation for the rest of the series and provides the necessary background to understand them. 1. Parallel Hardware. Review and cite PARALLEL PROGRAMMING protocol, troubleshooting and other methodology information | Contact experts in PARALLEL PROGRAMMING to get answers.Â I read some postings about performing functions like apply with more than one core, but I am not shure how to implement the approaches. Does anybody know a simple and comprehensible way of translating a classical sequential for-loop into a procedure which uses different cores simultaneously to run a few of the analyses parallel? Chapter 6: The Art of Parallel Programming. Understanding parallel efficiency. Summary. Index. [1]. Mastering Parallel Programming with R. Master the robust features of R parallel programming to accelerate your data science computations. Simon R. Chapple Eilidh Troup Thorsten Forster Terence Sloan. Birmingham - mumbai.Â   Learn about parallel programming in .NET. Use a .NET runtime, class library types, and diagnostic tools to simplify .NET development.Â To take advantage of the hardware, you can parallelize your code to distribute work across multiple processors. In the past, parallelization required low-level manipulation of threads and locks. Visual Studio and .NET enhance support for parallel programming by providing a runtime, class library types, and diagnostic tools. These features, which were introduced in .NET Framework 4, simplify parallel development. You can write efficient, fine-grained, and scalable parallel code in a natural idiom without having to work directly with threads or the thread pool.