ANDREW BROWNSWORD

driving the evolution of software languages to a concurrent future

Andrew Brownsword is Chief Architect at Electronic Arts BlackBox in Vancouver, Canada. He has been with the company since 1990 and has a BSc in Computing Science from the University of British Columbia.

andrew@brownsword.ca

Concurrent: existing, happening, or done at the same time.

CONCURRENCY HAS BECOME A HOT topic in the past few years because the concurrency available in hardware is increasing. Processor designers have reached a point where making sequential tasks take less time has become impractical or impossible—the physical limits encountered demand engineering tradeoffs. But writing software that takes advantage of concurrency is hard, and making that software perform as well on different CPU architectures is all but impossible. In this article, we will explore the reasons why this is currently true, with specific examples, and will consider how this evolution represents a changing paradigm that renders the traditional imperative programming model fragile and inefficient.

For the past 20+ years, hardware designers have been using concurrency in the form of pipelining, superscalar issue, and multi-transaction memory buses to improve the apparent performance of what appears to be sequential hardware. Since about the late 1990s most microprocessors have also included SIMD instructions, which are typically capable of 4 FLOPs (Floating Point OPerations) per instruction. More recently, some processors support multiple threads per core (from two in Intel's Hyperthreading, up to eight in Sun's Niagara architecture), and now processors with two or four cores are becoming the norm. Michael McCool's April 2008 *;login:* article provides an overview of these techniques.

Computational Efficiency

Most high-performance modern hardware these days is theoretically capable of about 10–20 FLOPS for each floating point value moved through the processor. In terms of latency, approximately 400-4800 FLOPS could theoretically be done in the time it takes to fetch a value directly from memory. These are theoretical peak computation rates, based on the four-way SIMD capabilities that most processors have now. Theoretical rates are not reached in practice, so what can we really expect to achieve and what do we see in practice?

How efficiently a processor executes is a function of the details of the processor's operation, its memory subsystem, and the software that is running. Typically all operations are scalar, because that is how most languages are defined. ALGOL set the pace, and most imperative languages since then have been embellishing the basic model. They embody the ideas of the sequential Von Neumann architecture and are notations for describing a sequence of scalar operations. These operations are usually dependent on the output of operations that came immediately before, or nearly so.

Also, typical code sequences are filled with decision points and loops, which appear as branch instructions that disrupt the efficient flow of instructions. Branch frequency and data dependencies in typical code are a frequently measured metric by hardware and compiler developers. In the mid 1990s, IBM found during the development of the PowerPC 604 that branches occurred, on average, once every 6 instructions. This makes it largely unproductive to have hardware dispatch more than about 3 or 4 instructions per clock cycle. To this day most hardware is aimed at 2- to 4-way dispatch, which is unsurprising, since software hasn't changed substantially. More recent tests on an in-order processor showed that most of the software for gaming platforms averaged about 10% of the potential instruction dispatch rate. And very little of that software was using the considerable SIMD capabilities of the hardware, leaving the realization at less than 3% of the processor's theoretical computational capability. Instead of the theoretical potential ~25 GFLOP, less than 1 GFLOP was realized. Out-of-order-execution processors will do somewhat better than this, but usually only by a factor of 2 or 3.

In recent years both the number of cores in one system and, on some cores, the number of threads executing on each core have increased beyond unity. I will ignore the distinction between hardware threads sharing a core and multiple cores for the rest of this article and will refer simply to "threads."

Memory and Threads

Multiple threads must share the *system* in which they exist. The most important shared resource is main memory (RAM). In most systems there is a single large pool of RAM. The system may also have a GPU processor, and some architectures give the GPU its own pool of RAM. Some systems partition the CPU memory by attaching parts of it directly to each processor; this is called NUMA (Non-Uniform Memory Access). How each processor accesses data at a given memory address and how long it takes to do it depend on where that memory is physically located.

The sharing of memory is complicated by the fact that processors use on-chip caches to speed up memory access. When the cache isn't shared by all threads, the potential exists for a given memory location's actual current state to be sitting in one thread's cache when another one needs it. Most hardware implements "cache coherency," which is a mechanism that tracks where each memory location's actual state is and retrieves it from that location when requested. Having multiple threads reading the same location is dealt with by having multiple copies of the state in the various caches. Writing to the same location is problematic, however, because a single current state must be kept up to date and it is usually placed in the cache of the thread that most recently modified it. If more than one thread is continuously updating the same location at the same time, considerable intercache traffic may result.

From a developer's perspective the problem with multiple threads, at least in an ALGOL-like language, is that the program must be explicitly written to take advantage of them. Given a program written for a single-processor machine, all but one thread will sit idle unless the operating system has something for additional threads to do. Typically the OS has enough to keep a second thread at least partially busy, and some OS services are now internally moving computation to other threads (graphics, movie playing, animation, etc.) if those services are in use. If the hardware has more than two threads, however, then they are going to be doing little to improve your software's performance. As a result, if you are using 3% of one thread's potential and you have a four-core machine, for example, you are now only using <1% of the system potential. In theory you could speed up your software by a factor of over 100 or be doing 100 times as much computation on the same hardware.

Programming for Concurrency

So how do you take advantage of this ability of hardware to operate concurrently? Broadly speaking, there are two kinds of available concurrency: instruction-level and thread-level.

Instruction-level concurrency has largely become the domain of the compiler, but that doesn't mean there is nothing you can do about it. The choice of language you use and how you use it has an enormous impact. As already mentioned, most current languages embody a fundamentally scalar and sequential programming model. Some compile to an intermediate form that also embodies a simple sequential machine, and the need for efficient JIT severely limits how the compiler can optimize. Fully natively compiled languages or those with more powerful intermediate forms may have compilers that can perform aggressive optimizations (such as auto-vectorization) in specific circumstances, but these techniques tend to be fragile and provide limited results. In C and C++, the language provides fairly low-level-detail control over the emitted instructions, and the compilers often support hardware-specific language extensions to access SIMD instructions and other unique hardware features. With careful, labor-intensive techniques, highly specialized platform-specific code can be written to dramatically improve performance. This kind of code optimization requires substantial expertise, detailed knowledge of the hardware platform, and a great deal of time, and it incurs substantial maintenance costs. On modern hardware it commonly delivers 4–20x performance improvements, raising the 3% utilization into the 10–60% range. Notice that this is as good as or better than the improvement you might expect by going from a single- to a many-threaded processor. Not all problems can be optimized in this fashion, but more are amenable to it than most developers seem to think. This kind of data-parallel solution also tends to be more amenable to being divided across multiple threads, making it easier to achieve thread-level concurrency. Unfortunately, the aforementioned problems make the process of experimentation and exploration to find efficient new data-parallel solutions very difficult and expensive.

Achieving a high level of instruction-level concurrency boils down to writing code that fits into a set of constraints. The precise definition of these constraints depends on the hardware, but there are some general principles that apply. Choose data structures and access patterns that are uniform and predictable, organize data so that access to it is spatially and temporally dense, choose data structure alignments and sizes, apply precisely the same sequence of operations to many data elements, minimize data dependencies

across parallel elements, define interfaces where each interaction specifies a large amount of work to be performed, and so forth.

Unfortunately, most programming languages do nothing to aid the programmer in these pursuits, if they allow these things to be under explicit programmer control at all. Even worse, code written in these languages contains too much detail (i.e., the semantic information available to the compiler is at a very primitive level) and this limits what the compilers are capable of and what they are permitted to do by the language rules. There are a few languages and alternative notations available (Michael McCool's own RapidMind being one proprietary example; others are StreamIt, Intel's Ct, etc.), but they are far from pervasive and thus far there is no widely available standard that integrates tightly enough with the standard environments (such as C/C++) to be adopted. Tight integration and, where possible, familiar syntax are essential for practical efficiency reasons, for programmer comfort, and for gradual adoption by an industry heavily invested in existing languages. Exactly what such a language should look like is open to debate, but my opinion is that burdening an existing grammar of already excessive complexity is not the correct solution.

Thread-level Concurrency

Thread-level concurrency tends to receive more attention, largely because it is easier to see and understand. It is accomplished in most languages by calling a threading interface provided by the OS, including it in a support library, or as a language feature. These vary in their details and features. The detailed semantics and capabilities of how threads work between platforms and interfaces vary significantly. How they get scheduled to run, whether priorities can be set, how the threads interact based on these priorities, whether a given software thread will stay on a given hardware thread, how long it will run without being interrupted, and so on can show up latent bugs and result in noteworthy performance differences that are hard to diagnose.

Threads running concurrently will inevitably want to interact and will therefore need to access some piece of shared state. This generally isn't a problem until one or more of the threads want to modify the shared state. Then we run into a problem called the "lack of atomicity." Most operations are actually composed of several hardware operations. Incrementing an integer variable, for example, reads the value from memory into a register, performs the add operation, then stores the value back to memory. Even with hardware that has an instruction that appears to add a value directly to memory, it in reality implements this internally as an read–modify–write sequence. When each of these operations is a pipelined instruction, which is itself broken down into many stages, it should be clear that there are a lot of steps involved in doing even this most trivial of examples. When these steps are happening in a machine where there are additional threads sharing the memory location being modified, the possibility exists of conflicting updates. Exactly what happens to the value depends on the operation, its hardware implementation, and the precise timing of each incident of conflicting change. This means that even this trivial example can do something different every time it is executed, and even if timing were somehow stable, then executing on different (even nominally compatible) hardware may impact the outcome.

Many thread interfaces provide some basic "atomic operations." These functions use capabilities provided by the processor to ensure atomicity for this kind of simple read–modify–write operation. Usually the hardware provides

a base mechanism to use instead of the atomic operation itself. This is usually either a compare-and-set operation or a reserve-and-conditional-store operation. I'm not going to describe what these two things are here, but the salient point is that using them correctly and ensuring atomicity is significantly more expensive in terms of complexity, instructions, performance, and programmer knowledge than just a simple nonatomic operation. They also don't guarantee atomicity across a series of operations.

An important point to note about primitive hardware synchronization operations is that they cause the hardware to synchronize. This seemingly trivial point has unfortunate consequences. It usually means that at least part of the hardware must stop and wait for some set of operations to complete, and often this means requests on the memory bus. This interferes with the bandwidth across the bus, which is one of the system's key performance bottlenecks. The more aggressive the processor is about concurrent and out-of-order operations, the more there is to lose by forcing it to synchronize.

The problem with the lack of atomicity goes far deeper than the trivial increment example just cited. Consider this simple piece of C code:

```
int done; // assume this is an initialized shared variable
if (done == 0)
{
    // do "stuff" here that you want to happen once
    done = 1;
}
```

If you have two threads and they try to execute this code at the same time, then both *may* perform the test and do the "stuff" before done is set to 1. You may think that by putting the done=1 before doing the "stuff" you can fix the problem, but all you will do is make it happen less often. The instant that done is read for the comparison, another thread may come along and read the same value so that it can make the same test succeed. Subtle changes in the hardware or OS can dramatically impact how often this code fails in practice. One solution to this example is to use an atomic operation that your threading interface provides, but this doesn't get you very far, because it will provide only a limited set of atomic operations and you can't restrict your programming to just using those!

Synchronization

The standard solution to this is to use synchronization primitives. Each threading interface provides some set of "synchronization primitives." These are usually fairly similar among interfaces, but the semantics can differ in subtle but important ways. Each primitive also brings with it different performance implications. Using some lightweight synchronization, even if blocking or waiting doesn't happen, might consume only a few tens or hundreds of cycles, whereas others might consume thousands (possibly many, many thousands). And this is the cost if there is no contention between threads!

Most of these primitives are in the form of a lock/unlock pair of operations. In some cases it is called enter/leave instead of lock/unlock because the pair is defining a "critical section" of code that you enter with the lock (or enter) operation and that you exit with the unlock (or leave) operation. Only a single thread can be inside such a critical section at a time. Other threads attempting to enter are forced to wait until the one in the critical section leaves it (i.e., they are "mutually excluded," which is shortened to "mutex"). This ensures that the code in the critical section is executed atomi-

cally. Unfortunately, this also forces this part of the program to be effectively single-threaded, and spending too much time in critical sections reduces your performance to that of a single hardware thread (or less, since executing synchronization primitives has a cost).

```
int done;          // assume this is an initialized shared variable
enter_critical_section();
if (done == 0)
{
    // do stuff here that you want to happen once
    done = 1;
}
leave_critical_section();
```

From this you might think that synchronization is only needed when a value is going to be modified. Unfortunately this isn't so.

```
int *p;  // assume this is an initialized shared variable
if (p != NULL)
{
    if (*p == 0)
    {
        // do something wonderful
    }
}
```

Here we are testing p to be sure that it is valid before using it. Unfortunately, some other code might come along and modify it (to NULL, for example) before it is dereferenced, causing unexpected behavior, an outright crash, or an exception.

One idea to attempt a fix might be this:

```
int *p;  // assume this is an initialized shared variable
int *mycopy = p;
if (mycopy != NULL)
{
    if (*mycopy == 0)
    {
        // do something wonderful
    }
}
```

However, this can be a disaster as well. The memory referenced by p is part of the shared state, so simply making a copy of the pointer doesn't solve the problem. For example, another thread could deallocate the memory referenced by p or otherwise repurpose it. In a garbage-collected language the object will not have been deallocated, but if p now references a different object than mycopy you may be relying on (or changing) stale data. In other situations this might be a valid and efficient strategy.

There are likely to be multiple places in the code that modify a particular piece of shared state, so we need to be able to lock them all to make them mutually exclusive. To allow this, synchronization primitives are almost always objects—mutex objects, critical section objects, etc. The lock/unlock operations become methods, and all the normal issues of creation and lifetime management come into play. A synchronization object is a piece of shared state that is used to arbitrate between threads.

```
mutex m;   // assume this is initialized shared state
int *p;  // assume this is an initialized shared variable
```

```
m.lock();
if (p != NULL)
{
    if (*p == 0)
    {
      // do something wonderful
    }
}
m.unlock();
```

One obvious problem with this is that it becomes easy to forget the unlock operation or to retain the lock for long periods of time. This is particularly an issue if the lock and unlock operations are enacted indirectly through a higher-level piece of code via an interface, or if expensive function calls to other subsystems are made while the lock is held.

To mitigate these problems some threading interfaces provide lexically scoped synchronization. For example, in C#:

```
object m;
int *p;  // assume this is an initialized shared variable
lock (m)
{
    if (p != NULL)
    {
    if (*p == 0)
      {
       // do something wonderful
      }
    }
}
```

Synchronization primitives are shared resources, and in C++ it is appropriate to apply the resource acquisition through a construction paradigm, as in, for example:

```
class MutexLock
{
public:
    MutexLock (Mutex m) : mMutex(m) { mMutex.Lock(); }
    ~MutexLock () { mMutex.Unlock(); }
private:
    Mutex mMutex;
};
```

As a project grows in size and more code needs to operate concurrently, a program will come to have multiple synchronization objects. A new problem arises in code with multiple synchronization primitives: deadlock. Imagine two threads (#1 and #2), which are using two mutexes (A and B). Now consider what happens in this scenario: Thread #1 acquires mutex A, thread #2 acquires mutex B, thread #1 attempts to acquire mutex B (and blocks until #2 releases it), and finally thread #2 attempts to acquire mutex A (and blocks until #1 releases it). These two threads are now stopped, each waiting for the other to release its resource, which of course they cannot do, because they are stopped.

Deadlock is relatively simple to be aware of and to avoid in a simple piece of software. It becomes considerably more complex to avoid in larger pieces of software. One solution is to use a single mutex instead of two different ones. Some early threaded operating systems adopted this approach to pro-

tect their internal resources. The problem with this approach was, as previously described, that the program can rapidly degenerate to being effectively single-threaded. The more hardware threads you have, the more of a loss this is.

An alternative to sharing mutexes widely is to avoid shared state and to use no other systems from within critical sections. In large projects this can become difficult or impossible. Powerful and important design patterns such as delegates and iterators can lead to unexpected situations where deadlock is possible.

The term "thread safety" is often used to describe objects (or systems) that are designed to be used from multiple threads simultaneously. It is often not clear what is meant by an object being thread-safe. One approach is to simply make each of the methods in the object lock a mutex while it executes to ensure that the object's state remains consistent during the call. That does not ensure that the object's state remains consistent between calls. For example, in C++ with STL:

```
vector<int> a;              // assume this is shared
int len = a.size();         // assume this is a synchronized op
int last_value = a[len-1];// ... and so is this one
```

This code can fail even though all the operations on a are individually thread-safe. If another thread removes an element from a after the count is retrieved, then this thread will index past the end. If another thread adds an element to a, then it won't be the last value that is retrieved.

A naive solution to this problem is to provide a lock/unlock operation as part of the object's interface. The user of the object holds the lock while working with it. Unfortunately, working on objects in isolation is rare—most interesting code uses multiple objects to accomplish something interesting. If more than one of those objects uses this approach, you may now find yourself in a potential deadlock situation. One example of this arises when iterating across containers. What happens if another thread changes the container while it is being iterated? The .NET framework's containers, for example, throw an exception if this happens. The code doing the iteration must ensure that this cannot happen. The obvious solution is to lock a mutex for the duration of the iteration, but heavily used or large containers or slow per-element operations can make this prohibitively expensive. Furthermore, if the operation involves calls to other code (such as methods on the objects from the container), then deadlock can become an issue. A common example is moving objects from one container to another where both containers are using this locking strategy.

Amdahl's Law

It is useful to understand Amdahl's Law as it applies to concurrency. It actually applies to the principle of optimization in general, but here I'll just point out its implication with respect to synchronization. The time to execute a program is equal to the sum of the time required by the sequential and parallel parts of the program: T = S + P. Optimizations to the sequential part can reduce S, and optimizations to the parallel part can reduce P. The obvious optimization to the parallel part is to increase the number of processors. Twice as many processors might reduce P by 2; an infinite number might reduce it to virtually nil. If the split between S and P cannot be changed, however, then the maximum speedup possible by parallel optimizations is 1/S (i.e., P has gone to zero). If the sequential part of the program is 50% of the execution time, adding an infinite number of processors will only double

its performance! It should therefore be clear that minimizing the amount of time spent in sequential parts of the program (i.e., critical sections or holding mutex locks) is essential to performance scaling.

One important point about the preceding paragraph is whether the split between S and P can be changed. Changing this split is a powerful approach, akin to improving your algorithm instead of your implementation. It can be accomplished in three basic ways. The obvious, although difficult, one is to replace some of your sequential work with parallel equivalents. The second is simply to *not* do some of the sequential work, which is not always an option but sometimes worth considering. And the third is to do more parallel work. The parallel work is what will scale with increasingly concurrent hardware, so do more of it.

Conclusions

The point of this discussion of threaded programming is not to enumerate all the potential pitfalls of threaded programming. Nor do I claim that there aren't solutions to each of these individual problems, even if they require careful design and implementation and rigorous testing. What I am trying to convey is that there is a minefield of nasty, subtle, intractable problems and that the programming model embodied by the common programming languages does nothing to help the programmer deal with it. These languages were conceived for programming computers that no longer exist.

So what, with respect to concurrency, might a language and compiler take care of that could make life easier for the programmer? Here's a sampling (none of which I'm going to explain), just to give you an idea: structure and field alignment, accounting for cache line size and associativity, accounting for cache read/write policies, using cache prefetch instructions, accounting for automatic hardware prefetching, dealing with speculative hardware execution (particularly stores), leveraging DMA hardware, using a context switch versus spin-lock synchronization, dealing with variables co-inhabiting cache lines, loading variables into different register sets based on capabilities versus cost of inter-set moves versus usage, dealing with different register sizes, handling capability and precision differences among data types, organizing vectors of structures based on algorithmic usage and memory system in-flight transaction capabilities, vector access patterns based on algorithms and hardware load/store/permute capabilities, compare and branch versus bitwise math, branching versus selection, dealing with register load/store alignment restrictions, choosing SoA versus AoS in-memory and in-register organizations, hoisting computations from conditionals to fill pipeline stalls, software pipelining of loops, organizing and fusing loops based on register set size, selecting thread affinities, latching shared values for atomicity, object synchronization, trade-off between instruction and thread-level parallelism, and leveraging hardware messaging and synchronization capabilities. Any one of these things can result in doubling the performance of your algorithm and, although they may not combine linearly, you typically have to get them all right to get close to maximum performance. And many of them will break your code if you get them wrong. Some of them will simply make your code nonportable and fragile.

There are alternative categories of languages, and some of these offer programming models with strong advantages in concurrent programming. Functional languages (e.g., Haskell, Lisp, APL, ML, Erlang) offer some powerful advantages, but most bring with them various disadvantages and none is mainstream. Array programming languages (e.g., APL, MATLAB) are powerfully expressive of data parallel concepts but have limitations and haven't

reached their full potential in terms of optimization—and they typically eschew the object-oriented paradigms that are now deeply (and rightfully) entrenched in modern software development. Less traditional programming models and languages exist that embody powerful concurrent concepts such as message-passing objects networks (e.g., Microsoft's Robotics Studio), but they are far from standard, are not portable, and typically cannot be integrated into existing environments. Highly specialized languages and tools such as StreamIt, RapidMind, and cT also suffer from the same issues. A few very successful specialized languages exist, particularly in the domain of graphics: HLSL, GLSL, and Cg have given graphics software and hardware developers tremendous power and flexibility. They have integrated well into existing software systems, and being part of the C language family makes them readily accessible to the C/C++/Java/C# communities. The wide adoption of these shading languages gives some indication of and hope about what is possible. All of these alternatives point in directions in which we can take our programming languages and models in the future.

SUGGESTED READING

Video of Herb Sutter's talk "Machine Architecture: Things Your Programming Language Never Told You" at NWCPP on September 19, 2007: http://video.google.com/videoplay?docid=-4714369049736584770.

Slides for Sutter's talk: http://www.nwcpp.org/Meetings/2007/09.html.

Michael D. McCool, "Achieving High Performance by Targeting Multiple Hardware Mechanisms for Parallelism," *;login:*, April 2008.

Transactional memory: http://research.microsoft.com/~simonpj/papers/stm/index.htm.

Language taxonomy: http://channel9.msdn.com/Showpost.aspx?postid=326762.

Nested data parallelism: http://research.microsoft.com/~simonpj/papers/ndp/NdpSlides.pdf.

Erlang: http://www.sics.se/~joe/talks/ll2_2002.pdf.

P. Grogono and B. Shearing, "A Modular Programming Language Based on Message Passing": http://users.encs.concordia.ca/~grogono/Erasmus/E01.pdf.