MSDN® Magazine
The Microsoft Journal for Developers

CLR INSIDE OUT

Using concurrency for scalability

Download the code for this article: CLRInsideOut2006_09.exe (151KB)

There's been a lot of buzz lately about concurrency. The reason for this is due mostly to major hardware vendors' plans to add more processor cores to both client and server machines, and also to the relatively unprepared state of today's software for such hardware. Many articles focus on how to make concurrency safe for your code, but they don't deal with how to get concurrency into your code in the first place.

Both tasks are important and can be difficult for different reasons. Randomly creating new threads and sprinkling calls to ThreadPool.QueueUserWorkItem all over your codebase isn't going to lead to good results. You'll need to take a more structured approach. First, let's take quick stock of the situation.

Over the course of the 90s, parallelism has grown to become a silent enabler of software scalability in newer installments of processor architectures. While most of us didn't even have to realize it was there, or even write code differently to take advantage of it, parallelism has nevertheless been used. Instruction-level parallelism (ILP) techniques are layered underneath existing sequential programming models, carried out at the granularity of an individual instruction stream, which employs branch prediction, speculation, and dataflow out-of-order execution. Pipelining, for example, can lead to 25-30 percent performance increases, depending on the depth of the pipeline and workload. Such techniques, when coupled with clock-speed increases, ensured that software continued to run faster with each generation of hardware while requiring minimal additional work of the software.

While chip vendors still expect Moore's Law to continue (doubling the number of transistors in processors every 18 months or so), what engineers can do with those transistors has begun to shift. Increasing the clock speed of new chips at the same rate seen in the past is simply not possible, primarily because of the heat generated. However, it is possible to use the additional transistors to put more low-latency memory and, if the software can take it, more cores on the chip. Note the qualification. Much software today assumes a single-threaded view of the world, and this needs to change if you expect to make use of those extra cores.

In some sense, a large chunk of the responsibility for making software go faster with the next generation of hardware has been handed off from hardware to software. That means in the medium-to-long term, if you want your code to get faster automatically, you'll have to start thinking about architecting and implementing things differently. This article is meant to explore these architectural issues from the bottom up, and aims to guide you through this new world. In the long run, new programming models are likely to appear that will abstract away a lot of the challenges you will encounter.

A Tale of Hardware Threads

Symmetric multiprocessor (SMP) computers capable of running Windows® have been on the market for years, although typically found only in servers and high-end workstations. These machines contain one or more boards, each board typically containing multiple sockets, with each socket holding a complete CPU. Each CPU in this regard has its own on-die caches, interrupt controllers, volatile state (registers), and a processor core, including its own execution units. The Windows scheduler maps individual software threads to individual CPUs, which in this case are entirely independent. I'll call these CPUs single-threaded in the hardware thread sense. Because each unit is relatively isolated (aside from shared memory architecture, which I'll discuss shortly), you can improve execution throughput for each new CPU added to a machine if enough software threads are offered up for execution.

Intel introduced Hyper-Threading (HT) for the Pentium 4 processor series. HT packs an extra set of interrupt controllers and volatile state onto a single physical CPU in a single socket, enabling multiple software threads to execute in parallel on distinct logical processors, although they share the same set of execution units. This approach is similar to that taken earlier by supercomputer companies such as Tera. Due to latencies associated with accessing memory, among other things, the instructions between the two logical CPUs threads can frequently interleave, leading to a parallel speedup. In this sense, HT-enabled CPUs are two-threaded because the Windows scheduler can map two runnable threads to an HT processor simultaneously. In reality, HT is suitable for some workloads, and has been cited as leading to a 15-40 percent improvement on real-world programs.

Multi-core technology, which is already readily available for client and server machines alike, replicates the per-CPU architecture on a single chip, enabling a single socket to contain multiple full CPUs. Dual-core is available now (two cores on a chip), with 4-core, 8-core, and beyond on the horizon. Unlike HT, multi-core CPUs have individual execution units, and therefore are generally capable of more substantial parallel speedups. Much like individual CPUs, each core is logically distinct aside from shared memory architecture. This means it's possible that twice as many cores doubles throughput. In this sense, the number of cores is the number of threads that can be running simultaneously. Of course, these technologies are not mutually exclusive. A 4-socket, 4-core HT computer amounts to 32 hardware threads. That's quite a bit of horsepower.

Memory Hierarchies

Memory interaction is often a substantial factor in the performance of modern software. Your typical computer contains a fairly complex memory system, consisting of multiple levels of cache between the processors and the actual DRAM banks. SMP machines traditionally have consistent hierarchical designs, although more exotic architectures are available and could become more prevalent with the increasing availability of massively parallel machines. One such exotic architecture is Non-Uniform Memory Access (NUMA), where multiple main memories are dedicated to nodes of multiple CPUs. Cross-node communication is possible, although extremely expensive. Various parts of Windows and the CLR change strategy to account for NUMA. Intelligent parallel code often has to do the same.

Cache-friendly concurrent software uses memory intelligently and efficiently, exploiting locality to reduce the total number of cycles required for a given computation. Locality comes in two major flavors. First is spatial locality: data close together in memory will be used close together in a program's operations. While bigger cache lines mean you may end up pulling more data than necessary into cache, a program with good spatial locality will take advantage of this by subsequently accessing other addresses that were already pulled into cache. The CLR garbage collector maximizes spatial locality by doing allocations contiguously, for example.

Temporal locality is the concept that memory stays in cache for a reason: if it was recently accessed, you can expect that it might be accessed again soon. Modern caches use eviction policies that optimize using pseudo-least-recently-used (LRU) techniques.

Well-written parallel software can even observe super-linear speedups occurring from keeping more data in cache and sharing less with other threads. That is, on an *n*-CPU machine, the software might run more than *n* times faster than on a single CPU machine. On the other hand, the cost of a "cache miss" is fairly expensive. This is illustrated further by the relative comparison of cache-access costs in **Figure 1**. As with all rules of thumb, take the numbers with a grain of salt and pay closer attention to the orders of difference.

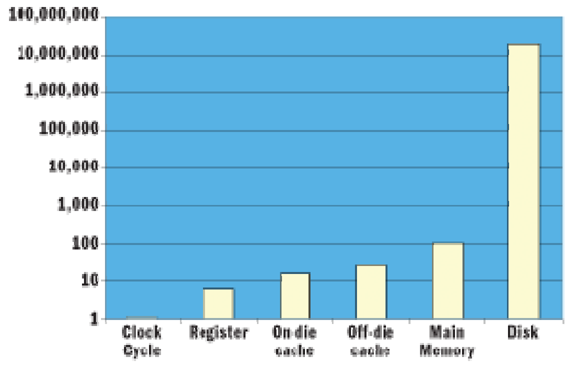


Figure 1 Logarithmic Graph of Comparative Access Costs

Parallel software in particular needs to take notice of locality. Two threads that continually update data on intersecting cache lines can cause cache line ping-pong, where the processors spend an extraordinarily high amount of time acquiring exclusive access to a cache line, which involves invalidating other processors' copies. Some cache line interactions are obvious, since there is real data sharing at the application level. Other interactions are less obvious and result from data residing close together in memory, which unfortunately can't be easily determined simply by examining the algorithm.

Similarly, thread migration—which I'll discuss in more detail later—can cause a thread to move to another processor and subsequently have to acquire and invalidate all of the cache lines it once held on the original processor. This cache migration can cost on the order of 50 times the cost of an on-die cache hit for each line access requiring migration. On NUMA machines this can be disastrous due to the cost of inter-node communication, though the migration problem can be partially avoided through clever usage of processor affinity. These are costs to be aware of when writing highly parallel code. The new GetLogicalProcessorInformation API in Windows Vista™ enables you to inquire about the machine architecture, including cache layout and NUMA information, which could be used dynamically during scheduling, for example.

Units of Work

To get your software to execute in parallel, clearly you need to somehow break up the problems encoded in your algorithms into sub-problems: smaller units of work that I will refer to as tasks. A task takes some input and produces some output, whether that output is a piece of data or an action. It can execute in isolation, although it may have subtle dependencies on state or ordering that might not be evident at first.

Functions pretty much already do this, you might say. But unlike ordinary functions, which are defined statically as you write your code, to write software that scales given an arbitrary number of CPUs, the boundaries of a task must often be discovered dynamically. Or at least the tasks must be presented to an intelligent architecture that knows whether it is profitable to execute a task in parallel. And furthermore, to make a task execute in parallel, your code has to somehow arrange for it to be called in such a manner rather than simply calling it sequentially on the current thread. On Windows, this typically means running on a separate OS thread. On the CLR, it means perhaps queueing the work to execute on the ThreadPool.

The physical unit of execution on Windows is a thread. Each process begins life with a single thread, but of course the code running in that process is free to introduce and later tear down additional threads as it sees fit. The Windows scheduler is responsible for assigning threads to hardware threads and allowing the code to execute. If there are more threads than there are hardware threads, the scheduler has to be a little more sophisticated; it picks the runnable thread of the highest priority (subject to clever anti-starvation algorithms), and lets it execute until a quantum has expired. Once the quantum expires, a context switch occurs and the same scheduling algorithm chooses the next thread to execute. The length of a quantum varies based on the OS type and configuration, but it will generally be around 20ms for client platforms and 120ms for server platforms. A thread can block as a result of performing I/O, attempting to acquire a contended lock, and so on. In this case, just as with a context switch, the scheduler will pick a new thread to execute.

As mentioned earlier, it's crucial to performance of traditional SMP systems that as much data reside in cache as possible. "Data" in this sense refers to the code being executed, the heap data being manipulated by the thread's algorithms, and the thread's stack. As threads are switched into and out of CPUs, Windows automatically employs so-called ideal processor affinity in an attempt to maximize cache efficiency. For example, a thread running on CPU 1 that gets context switched out will prefer to run again on CPU 1 in the hope that some of its data will still reside in cache. But if CPU 1 is busy and CPU 2 is not, the thread could be scheduled on CPU 2 instead, with all the negative cache effects that implies.

Know the Costs

Threads aren't free. They incur CPU and memory costs, which you should keep in mind. If your goal is to use concurrency to increase the scalability of your algorithms, presumably you'll also be spending as much (if not more) time doing traditional performance profiling work. Running a sloppy algorithm in parallel will do nothing but make your sloppy algorithm use up more system resources. Ensuring that the most important hot sections of your code are as efficient as possible in the sequential case is crucial for taking advantage of parallel scaling.

To determine what costs you can afford, there are some general rules of thumb. The cost of creating a Windows thread is approximately 200,000 cycles, whereas the cost of destroying one is about 100,000 cycles. So right there you know that if your intention is to spin up a new thread to execute 100,000 cycles' worth of work, you're paying a hefty overhead—and, if I had to guess, you won't observe any type of speedup either.

The memory overhead varies based on configuration. But most managed threads will reserve 1MB of user stack space and will commit the entire amount, which means the memory must be backed physically either in real RAM or the page file. There is also a small set of kernel stack pages required, three pages on 32-bit systems and six pages on 64-bit systems. An additional 10-20KB of virtual memory is used by other data structures, but this is dwarfed by the memory required by the stack. GUI threads are slightly more expensive still, because they must set up additional data structures such as the message queue.

Now, if you end up with too many threads of equal priority, you will have to context switch often. A context switch costs 2,000–8,000 cycles, depending on the system load and architecture, and involves saving the current volatile state, selecting the next thread to run, and restoring the next thread's volatile state. This may not sound like a lot, especially when compared to the duration of a quantum and the cost of subsequent cache misses, but it represents pure overhead that is taken away from executing application code.

Given that you'd like to minimize the cost of introducing and destroying new OS threads and the negative consequences of accidentally introducing "too much" work, you should consider using the CLR's thread pool. It hides clever thread injection and retirement algorithms beneath a simple interface, amortizing the cost of creating and destroying threads over the life of your program. And using the ThreadPool class is simple.

That said, even using the ThreadPool costs something. Invocations to QueueUserWorkItem incur a sequential cost to the caller, and the infrastructure that dispatches work from the pending work item queue also imposes an overhead to parallel work that is being executed. For coarse-grained parallelism, these costs are so small that you're likely not to notice them. But for extremely fine-grained parallelism, these costs could become a significant scalability bottleneck. You might consider building your own lightweight ThreadPool out of lock-free data structures, avoiding some of the costs incurred by a general purpose ThreadPool, like ensuring fairness between AppDomains, capturing and restoring security information, and so forth. For most uses, however, the stock ThreadPool is up for the task.

Defining Boundaries

Figuring out how to split up your work is not a negligible activity. When dealing with CPU-bound workloads, the job is focused more on avoiding performance overhead associated with concurrent execution. But most workloads are not CPU-bound; they combine various forms of I/O and synchronization among CPU work, either of which can lead to unpredictable blocking patterns. And thus, with most code, concurrent execution is more about orchestrating complex coordination patterns than it is about lower level performance.

Perhaps the simplest way to partition work is to use the server model of concurrency. In servers like SQL Server™ or ASP.NET, each incoming request is considered an independent task and consequently runs on its own thread. The host software often throttles the number of real OS threads used so as not to over-introduce concurrency. Most workloads like this are composed of totally independent tasks that access disjoint sets of data and resources, leading to highly efficient parallel speedups. For client programs, however, few workloads fit into this model cleanly. Fielding and responding to peer-to-peer communication, for example, can be done via this model, but unless a large number of work-intensive incoming requests are expected, the ceiling on potential speedups here is fairly limited.

An alternative is to carve out arbitrary sub-tasks in your code using a more logical and subjective definition of "significant task," which tends to be more conducive to client-side workloads. A complex software operation typically consists of multiple logical steps, for instance, perhaps represented in your program as independent function calls that themselves contain multiple steps, and so on. You might consider representing each function call as an independent task, at least those that are substantial enough. This is tricky in the sense that you must consider ordering dependencies, which adds substantial complexity to this idea. Most modern imperative programs are full of unstructured loops, accesses to data via opaque pointers that may not be close together in memory, and function calls, none of which document cleanly what dependencies exist. And of course there's hidden thread affinity that you might not know about. So this technique clearly requires a deep understanding of the problem your code is trying to solve, and some thought about the most efficient way to perform it in parallel, eliminating as many dependencies as possible.

A common related pattern is fork/join parallelism, where a master task forks multiple child tasks (which themselves can also fork child tasks), and each master task subsequently joins with its children at some well-defined point. As an example, consider a model of task-level parallelism called fork/join futures, which encapsulates this pattern based on function calls as the unit of task. This can be illustrated with some new type Future <T> (an implementation of Future <T> is available for download from the *MSDN®Magazine* Web site):

```
int a() { /* some work */ }

int b() { /* some work */ }

int c()

{

    Future<int> fa = a();

    Future<int> fb = b();

    // do some work

    return fa.Value + fb.Value;

}
```

The meaning of this code is that the invocations of a and b can execute in parallel with the body of c, the decision for which is left to the implementation of the Future<int> engine. When c needs the results of those invocations, it accesses the Value property of the future. This has the effect of waiting for the work to complete or, if the work has not begun executing asynchronously yet, executing the function locally on the calling thread. This syntax maps closely to the existing IAsyncResult class, but has the added benefit of some more intelligence about how much concurrency to introduce into the program. While more clever implementations are easy to imagine, a straightforward translation of this code might look like this:

```
int a() { /* some work */ }

int b() { /* some work */ }

delegate int Del();

int c()

{

    Del da = a; IAsyncResult fa = da.BeginInvoke(null, null);

    Del db = b; IAsyncResult fb = db.BeginInvoke(null, null);

    // do some work

    return da.EndInvoke(fa) + db.EndInvoke(fb);
}
```

Other approaches are possible, such as using longer running sub-tasks rather than requiring that children never live longer than parents. This often requires more complex synchronization and rendezvous patterns. Fork/join is simple because the lifetime of individual workers is obvious.

The previous discussion takes a code-centric view of parallelism. Another technique is often simpler: data parallelism. This is usually appropriate for problems and data structures that are data- and compute-intensive, or whose individual operations tend to access disjoint data frequently.

One common data parallelism technique is called partitioning. Loop-based parallelism, for example, uses this approach to distribute computations over a range of elements. Say you had 16 logical CPUs, an array of 100,000 elements, and a piece of work that can execute with little-to-no dependency and tends to block 20 percent of the time. You could split the array into 20 contiguous chunks (you'll see how I calculate this number later) of 5,000 elements apiece, fork off 19 threads (reusing the current thread for one partition), and arrange for each thread to do its calculations in parallel. Parallel query processing in databases like SQL Server uses a similar approach. This technique is illustrated in **Figure 2**.

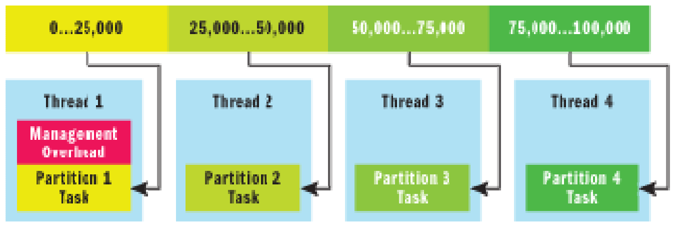


Figure 2 Partition-Based Parallelism

The example shows a 100,000-element array distributed over four threads. Notice some amount of sequential overhead is paid for the split. In cases where a merge is necessary, additional cost is often paid to merge the results, including joining with outstanding threads.

For-all loops have generally been a traditional way to express partition-based parallelism in programming languages. An example ForAll<T> API implementation can be found in **Figure 3**. Similar approaches could also be used to parallelize loops—for example, rather than taking an IList<T>, you could instead take an int from and int to set of parameters and then feed the loop iteration number into an Action<int> delegate.

This code makes one major assumption that could be disastrous: the Action<T> delegate passed in is expected to be safe to execute in parallel. This means that if it refers to shared state, it needs to use proper synchronization to eliminate concurrency bugs. If it isn't, we can expect the correctness and reliability of our program to be quite poor.

Another data-parallelism technique is pipelining, where multiple operations execute in parallel, streaming data between each other using a fast, shared buffer. This is akin to an assembly line, where each step in the process is given its chance to interact with some data and then pass it on to the next step in line. This technique requires clever synchronization code to minimize time spent at the obvious bottleneck: the point where adjacent stages in the pipeline communicate through a shared buffer.

How Many Tasks?

Choosing the number of tasks to create is also a tricky equation. If throughput is the sole priority, you can use some theoretical target such as the following, where BP is the percentage of time tasks will block:

```
NumThreads = NumCPUs / (1 - BP)
```

That is, you'd like the number of threads to be equal to the ratio of CPUs to the percentage of time tasks are spent doing real work. This was illustrated in the ForAll example earlier. Unfortunately, while this is a good theoretical starting point, it won't get you a precise answer. It doesn't account for HT, for example, where high memory latencies permit computations to occur in parallel, but otherwise shouldn't account for a full processor. And it assumes—quite naively—that you could actually predict the value of BP, which I assure you is difficult, especially for a component trying to schedule heterogeneous work, much like the CLR's thread pool. When in doubt, it's better to rely on the thread pool for scheduling of tasks to OS threads, and tend towards over-expressing concurrency.

There is a natural speedup curve for any algorithm. On this curve, there are two particularly interesting points to consider. First, what is the minimum number of tasks that see benefit from parallelizing the computation? For small computations, it may be that using a small number of tasks incurs too much overhead (thread creation and cache misses), but that using a larger number allows the execution to catch up to the sequential version and surpass it. Second, given an infinite number of hardware threads, what is the maximum number of tasks you can assign to a problem before beginning to see degradation in performance rather than a continued speedup? All problems reach this point of diminishing returns. As you continue to subdivide a problem, eventually you're going to reach the granularity of individual instructions.

A linear speedup would mean that the time it takes to execute a problem with *p* processors is 1/*p* the time it takes to execute a problem with one processor. Amdahl's Law tends to limit the ability to achieve such speedups. It says, quite simply, that the maximum speedup is limited by the amount of sequential execution you have remaining after introducing parallelism. More formally, this law states that, if *S* is the percentage of the problem that must remain sequential (it cannot be parallelized) and *p* is the number of CPUs being used, then the approximate speedup you can expect can be represented as:

```
1/(S + ((1 - S)/p))
```

As the number of processors grows, this expression approaches 1/*S*. Therefore, if you can only parallelize (say) 85 percent of the problem, you will only be able to achieve a speedup of 1/.15 or approximately 6.6. Any overhead associated with synchronization and introducing concurrency tends to contribute to *S*. In reality, however, there is good news: spreading work across multiple processors also carries benefits that are hard to quantify and measure, such as allowing concurrent threads to keep their (separate) caches warm.

Any algorithm managing real resources also has to take into account cross-machine utilization. Software that makes entirely local decisions to maximize parallelism—especially in a server environment such as ASP.NET—can (and will!) lead to chaos as well as an increase in contention for resources, including CPUs. A ForAll-style loop, for example, might query the processor utilization before deciding the optimal number of tasks dynamically. Instead of the algorithm used in **Figure 3**, which depends on the System.Environment.ProcessorCount property, you might consider using the GetFreeProcessors function shown in **Figure 4**.

Figure 3 Simplistic Parallel Implementation

```
static void ForAll<T>(IList<T> data, Action<T> a)
{
    ForAll<T>(data, a, 0.0f);
}

static void ForAll<T>(IList<T> data, Action<T> a, float blocking)
{
    if (blocking < 0.0f || blocking >= 1.0f)
        throw new ArgumentOutOfRangeException("blocking");

    int partitions = Math.Min(data.Count, (int)Math.Max(1.0f,
        (float)Environment.ProcessorCount / (1 - blocking)));
    int perCount = (int)Math.Ceiling((float)data.Count / partitions);

    int done = partitions - 1;
    ManualResetEvent mre = null;

    if (partitions > 1)
    {
        mre = new ManualResetEvent(false);
        // Queue a work item per-partition. This is the "fork"
        // part of the work.
        for (int i = 0; i < partitions - 1; i++)
        {
            int idx = i;
            ThreadPool.QueueUserWorkItem(delegate
            {
                for (int j = idx * perCount;
                     j < (idx + 1) * perCount;
                     j++)
                {
                    a(data[j]);
                }
                if (Interlocked.Decrement(ref done) == 0)
                {
                    mre.Set();
                }
            });
        }
    }

    // Execute one partition’s worth of operations on the
    // calling (current) thread.
    for (int i = (partitions - 1) * perCount; i < data.Count; i++)
        a(data[i]);

    // If we queued async partitions, wait for them to finish.
    // This is the "join" part of the work.
    if (mre != null) {
        mre.WaitOne();
        mre.Close();
    }
}
```

Figure 4 GetFreeProcessors

```
private static List<PerformanceCounter> utilizationCounters;

static void InitCounters()
{
    // Initialize the list to a counter-per-processor:
    utilizationCounters = new List<PerformanceCounter>();
    for (int i = 0; i < Environment.ProcessorCount; i++)
    {
        utilizationCounters.Add(
            new PerformanceCounter("Processor",
            "% Processor Time", i.ToString()));
    }
}
```

```
private static int GetFreeProcessors()
{
    int freeCount = 0;
    foreach (PerformanceCounter pc in utilizationCounters)
    {
        if (pc.NextValue() < 0.80f)
            freeCount++;
    }
    return freeCount;
}

// ForAll<T> change...
int partitions = Math.Min(data.Count, (int)Math.Max(1.0f,
    (float)GetFreeProcessors() / (1 - blocking)));
```

That algorithm isn't perfect. It's only a statistical snapshot of the machine state at the time it runs, and says nothing about what happens after it returns its result. It could be overly optimistic or pessimistic. And of course it doesn't account for the fact that one of the processors being queried is the one executing the GetFreeProcessors function itself, which would be a useful improvement. Another interesting statistical metric to look at is the System\Processor Queue Length performance counter, which tells you how many threads are in the scheduling queue waiting for a processor to become free. A large number here indicates that any new work you introduce will simply have to wait for the queue to drain (assuming all threads are of equivalent priority).

There are some interesting reasons to create too much concurrency rather than too little. If you're considering heterogeneous tasks, the model of letting each task execute on a thread until it completes may lead to fairness problems. Task A, which takes substantially longer to run than task B, could lead to starvation of B if additional resources are not freed up. This is especially bad if A has decided to block and your algorithm didn't take that into account.

Another reason to intentionally over-parallelize is for asynchronous I/O. Windows provides I/O completion ports for superior scalability, in which case outstanding I/O requests don't even need to utilize an OS thread. The I/O begins asynchronously and, once it is complete, Windows posts a completion packet to the underlying port. Typically an efficiently sized pool of threads is bound to the port—taken care of on the CLR by the thread pool—waiting to process completion packets once they become available. Assuming a sparse completion rate, creating a large number of I/O requests as fast as possible in parallel can lead to better scalability than if each task waited behind the other for its turn to initiate the asynchronous I/O. This applies to file, network, and memory-mapped I/O, although you should always be cognizant of the fact that there are a fixed number of shared resources on the machine; competing for them too heavily will only degrade scaling, not enhance it.

Shared State

Any time you introduce concurrency, you need to worry about protecting shared state. This is crucial. I recommend you read Vance Morrison's article in the August 2005 issue of *MSDN®Magazine* (msdn.microsoft.com/msdnmag/issues/05/08/Concurrency) about why locking is important. Correctness should always take precedence over performance, and if you're using concurrency without considering locking, your code probably isn't correct. I'm not going to reiterate what Vance has already said quite nicely, but rather I am going to focus on the performance of such techniques.

The most common techniques for synchronization are locking and low-lock operations. Locking uses primitives like the Win32® Mutex or CRITICAL_SECTION, or the CLR Monitor, ReaderWriterLock, or associated language keywords (for example, lock in C# and SyncLock in Visual Basic®) to achieve some degree of mutual exclusion. To achieve this exclusion, a call is made to an API; some internal algorithm ensures that no two pieces of code using the same lock can enter the protected region of code. So long as everybody abides by the protocol, code remains safe.

Low-lock operations can be built using interlocked primitives, which are implemented via hardware support for atomic load-compare-store instructions. They ensure a single update to memory is atomic and can be used to build highly scalable code that uses optimistic concurrency. Such code is more difficult to write, but it tends not to block. (Locks, in case you are wondering, are written using such primitives.)

But making these calls comes with a cost. **Figure 5** shows a micro-benchmark of what acquisition of various types of locks costs in CPU cycles for cases without contention.

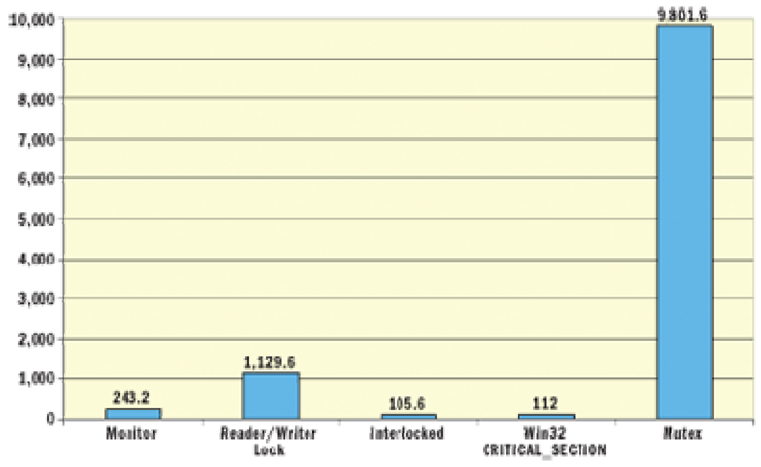


Figure 5 Comparing the Cost of Various Locks

While such measurements are interesting to understand the performance implications of locking (especially when dynamic decisions about parallel execution are made, in which case a larger amount of code "needs to be ready" than will actually be run in parallel), making certain that you synchronize at the right granularity can help to ensure costs such as this don't dominate execution of your code. There are also costs I have not mentioned, such as the interaction between interlocked operations and the memory hierarchy. Regrettably, space doesn't permit that. Nevertheless, the more interesting part is the effect on scalability. Regrettably you often need to make tradeoffs between scalability and sequential straight-line execution performance. These tradeoffs should be informed by measurements.

Nothing guarantees that a thread remains runnable while a lock is held, and therefore if its quantum expires, a subsequent thread may run and attempt to acquire this same lock. Moreover, a higher priority thread that becomes runnable can preempt a thread running under a lock. This can cause a phenomenon known as priority inversion, and can also lead to lock convoys if the arrival rate at a contended lock is extraordinarily high. Most locks react to contended acquisition attempts via some form of lightweight spinning on multi-CPU systems, in the hope that the thread holding the lock will soon release it. If that fails, because either the owner is holding it for longer than expected or perhaps because it was swapped out as a result of a context switch, it will block. For a highly concurrent system, the more blocking, the more threads are needed to keep CPUs busy, and the lower the probability that your system is going to scale nicely.

Thus an important question to keep in the back of your mind is: how can I do the least amount of work while I'm holding a lock, in order to minimize the amount of locking required? Reader/writer locks can help with this, allowing multiple threads to read data while still ensuring mutual exclusion on writes. For most systems, the ratio of readers to writers is very high, and therefore the win to scalability can be huge. Jeffrey Richter's Concurrent Affairs column from the June 2006 issue of *MSDN Magazine* is a great starting point to learn more (see msdn.microsoft.com/msdnmag/issues/06/06/ConcurrentAffairs).

With that said, if you can avoid sharing state in the first place, you needn't synchronize access at all. A common technique to increase scalability of algorithms that manipulate hot data structures—data that most threads must access—is to avoid the use of locks altogether. This can take three interesting forms, in increasing order of difficulty to implement: immutability, isolation, and lock freedom.

Immutability means that an instance, once created, will not change—or at least won't change during a fixed and well known period of time. A CLR string, for example, is immutable and therefore doesn't require a lock around accesses to its individual characters. If state isn't in motion, you don't need to lock. This becomes difficult to achieve when there are multiple locations containing pointers to state that are supposed to be observed atomically.

Isolation eliminates any concurrent access to data by maintaining separate copies. Many thread-safe C implementations of malloc and free operations, for example, maintain a per-thread pool of available memory so that threads allocating don't contend for the pool (which is likely to be a hot spot in any C program). Similarly, the CLR's Server Garbage Collector (GC) uses a per-thread allocation context and per-CPU segment of memory to increase throughput of memory allocations. This typically requires periodic rendezvousing with a central copy of data structures, and can sometimes require costs associated with copying and ensuring interesting bits of data doesn't become stale.

Lock freedom is such a tricky technique that I will only mention it in passing. If you really understand the memory model of your target machine and are willing to write and maintain a whole lot more code, you can create clever data structures that scale nicely when accessed in parallel. More often than not, the resulting code is so difficult to test for correctness and to maintain that it isn't worth the effort. These techniques are worth investigating for those areas of your program that you've measured a scaling or performance problem associated with the use of locks.

Tools for Profiling Parallelism

Let's see how you might measure and improve the scalability of your code. Throughout this column, I've been rather fuzzy on techniques, approaches, and costs. Unfortunately there isn't one magic formula that works for all parallel problems. And similarly, there isn't an easy answer to the question of how to profile problems and/or identify better approaches to attaining a parallel speedup. It's entirely possible you will go through all of the work I outlined here (and then some—I haven't discussed debugging), and yet end up no better off than if you had stuck with a sequential algorithm. There are so-called embarrassingly parallel problems for which cookbook-like algorithms have been written and made available online and in textbooks. Unfortunately many real world programs aren't so straightforward.

Here are some tips for profiling your parallel algorithms. All of these make use of the new Visual Studio® 2005 profiler. It is built in to the ordinary Visual Studio interface (under the Tools | Performance Tools | New Performance Session menu item), and also has a command-line version located in the \Team Tools\PerformanceTools\ Visual Studio subdirectory, named VSPerfCmd.exe. (See msdn2.microsoft.com/ms182403.aspx for usage details about this tool.) This profiler creates VSP files that can piped through the VSPerfReport.exe command to create a CSV or XML file for further analysis. Here are a few items to look for.

Ensure your CPUs are busy. If you have low processor utilization, it's likely that one of two things is happening. Either you didn't use enough processors to keep your problem busy, or threads are getting backed up waiting for each other, most likely due to excessive synchronization at hot points in the code. Task Manager is typically sufficient for this, although the Processor\% Processor Time performance counter (via PerfMon.exe) can also be used.

Make sure your program isn't faulting heavily. Especially for data-intensive applications, you need to ensure you don't overflow physical memory on a regular basis. In such a case, a system full of threads can continually thrash the disk while it constantly has to page in and page out data. (Remember how expensive disk access is from the chart earlier?) Task Manager can give you this data (you'll need to select it as a column), as can PerfMon.exe. VSPerfCmd can also report this data via ETW events with this command:

```
VSPerfCmd.exe /events:on,Pagefault /start:SAMPLE /output:<reportFile name>
/launch:<exeFile name>
```

Then use the following command once the program completes.

```
VSPerfCmd.exe /shutdown
```

You may also want to play around with the sampling interval.

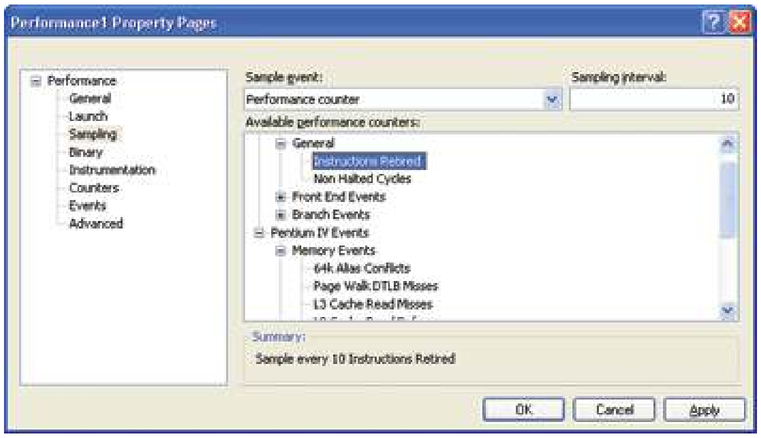
Identify where your program is spending most of its CPU time. This is especially important if this CPU time occurs while locks are held. It could also be the case that the additional code required to create threads, perform synchronization, and anything associated with those two things is dominating the execution time.

Examine the System\Context Switches/sec and System\Processor Queue Length performance counters. This can help determine whether you have too many threads, causing time wasted in context switches and thread migration. If so, try tweaking the algorithm that determines how many tasks to use.

Look for a memory hierarchy and cache problem. If none of the previous suggestions work and it seems that you should be seeing a bigger speedup, you might have a memory hierarchy and cache problem. A lot of time spent in cache misses and invalidations can dramatically limit the ability to speed up your program. Partitioning data to be more cache-line friendly and using some of the approaches mentioned above, like isolation, can help to resolve this problem. Each CPU offers a set of performance counters that can be queried in Visual Studio's profiler, covering things like instructions retired and cache misses. A low instructions-retired count indicates more time spent in high latency operations, such as cache misses, and the cache-specific counters can be used to determine where the misses are occurring and at what frequency.

While the exact counters are processor-specific, the Visual Studio interface gives you a nice menu option for using them (see **Figure 6**), and you can also query these counters through the command:

```
VSPerfCmd.exe /QueryCounters
```



Sample event: Performance counter

Sampling interval: 10

Available performance counters:
- General
  - Instructions Retired
  - Non Halted Cycles
  - Front End Events
  - Branch Events
- Pentium IV Events
  - Memory Events
    - 64k Alias Conflicts
    - Page Walk DTLB Misses
    - L3 Cache Read Misses

Summary: Sample every 10 Instructions Retired

Figure 6 Profiler Properties

Conclusion

Recommended Reading

- Writing Faster Managed Code: Know What Things Cost," Jan Gray, MSDN, June 2003
- Writing Scalable Applications for Windows NT," John Vert, MSDN, June 1995
- Concurrency: What Every Dev Must Know About Multithreaded Apps," Vance Morrison, MSDN Magazine, August 2005

Scalability via parallelism has historically been limited to server-side and high-end workstation environments. But as new hardware trends towards thread-level parallelism, namely multi-core architectures, mainstream client-side software will ultimately have to cope with and make efficient use of the resources available. This brings along with it a unique set of challenges. Clearly parallelism is not a replacement for highly efficient sequential code, but rather a technique for making already optimized sequential algorithms run even faster.

This was a very fast-paced and high-level overview. You'll probably walk away saying it's way too hard, and you wouldn't be wrong. But these techniques will help you to begin architecting code that will scale nicely on newer multi-core processors as they become more ubiquitous over time. As infrastructure like the CLR thread-pool and tools like Visual Studio evolve to better support this style of programming, you can expect many of the difficulties to become more manageable.

Send your questions and comments to clrinout@microsoft.com.

Joe Duffy works on the CLR team at Microsoft. His focus is on developing concurrency abstractions, programming models, and infrastructure for the platform. He just released a book, *.NET Framework 2.0* (Wrox/Wiley, 2006), and is in the process of writing another, *Concurrent Programming on Windows* (Addison-Wesley). He writes frequently on his blog: www.bluebytesoftware.com/blog.