Parallel Programming in .NET 4

Coding Guidelines

By Igor Ostrovsky
Parallel Computing Platform Group
Microsoft Corporation

Patterns, techniques and tips on writing reliable, maintainable, and performing multi-core programs and reusable libraries in .NET Framework 4.

Table of Contents

Primitives for Parallel Programming

One of the goals of .NET Framework 4 was to make it easier for developers to write parallel programs that target multi-core machines. To achieve that goal, .NET 4 introduces various parallel-programming primitives that abstract away some of the messy details that developers have to deal with when implementing parallel programs from scratch.

By appropriately using the primitives available in .NET, your code becomes more readable, easier to maintain, better performing, and less error-prone.

Tasks

Tasks are a new abstraction in .NET 4 to represent units of asynchronous work. Tasks were not available in earlier versions of .NET, and developers would instead use ThreadPool work items for this purpose. However, a task is a convenient abstraction that supports a number of handy features: you can wait on tasks, cancel them, and schedule tasks called "continuations" that run after a particular task completes.

DO use tasks instead of ThreadPool work items. Tasks provide a variety of useful capabilities such as waiting, cancellation, and scheduling of continuations. If you use tasks in your program, having these capabilities available will make maintaining the code easier.

For example, here is a task that asynchronously performs an expensive computation and then prints the result to the screen:

```
Task task = Task.Factory.StartNew(() =>
{
    double result = 0;
    for (int i = 0; i < 10000000; i++) result += Math.Sqrt(i);
    Console.WriteLine(result);
});
```

DO take advantage of Task capabilities instead of implementing similar functionality yourself.

To wait until a task completes, use the Wait method:

```
task.Wait();
```

To schedule a unit of work to run after the task completes, use the ContinueWith method:

```
task.ContinueWith(
            () => { Console.WriteLine("Computation completed"); });
```

If a task hasn't started running yet, you can cancel it:

```
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;

Task task1 = Task.Factory.StartNew(
      () => { ... },
      token);
```

```
tokenSource.Cancel();
```

For a more robust implementation of cancellation, the task itself can regularly poll the token by calling token.ThrowIfCancellationRequested (), so that the task will be canceled even if it has already started running by the time the token was canceled.

AVOID creating threads directly, except if you need direct control over the lifetime of the thread.

DO use Task<T> types to represent asynchronously computed values. The task body delegate returns a value that is exposed via the Result property on the task. When you access the Result property, you will get the result immediately if the task has already completed, or otherwise the call will block until the computation completes.

Here is an example that asynchronously starts three tasks, waits for them to complete, and then computes a sum of the three values:

```
Task<int> a = Task<int>.Factory.StartNew(() => { return Compute(0); });
Task<int> b = Task<int>.Factory.StartNew(() => { return Compute(1); });
Task<int> c = Task<int>.Factory.StartNew(() => { return Compute(2); });

int value = a.Result + b.Result + c.Result;
```

AVOID accessing loop iteration variables from the task body. More often than not, this will not do what you'd expect.

Here is an example of the problem:

```
for (int i = 0; i < 5; i++)
{
    Task.Factory.StartNew(() => Console.WriteLine(i));
}
Console.ReadLine();
```

Surprisingly, the output of this program is actually undefined, although in practice it is most likely that each task will print "5".

The problem is that each task prints what the value of i was when Console.WriteLine(i) got evaluated, and not what the value of i was at the time when the task was constructed. Most likely, the for-loop has completed by the time the tasks themselves execute, and so each task will print the value contained in i after the loop has finished, and that is 5.

To fix the problem, create a local variable inside the scope of the for-loop body:

```
for (int i = 0; i < 5; i++)
{
    int iLocal = i;
    Task.Factory.StartNew(() => Console.WriteLine(iLocal));
}
```

DO use a parallel loop instead of constructing many tasks in a loop. A parallel loop over N elements is typically cheaper than starting N independent tasks.

AVOID waiting on tasks while holding a lock. Waiting on a task while holding a lock can lead to a deadlock if the task itself attempts to take the same lock.

What's even worse, if your code contains a "logical" deadlock where you wait on a task while holding a lock that the task needs, the program may behave in an incorrect way instead of deadlocking. The reason behind that is that when you wait on a task, the Wait() method may decide to execute the task on the current thread. The task will execute on the current thread that already holds the lock, and so the task will successfully enter the critical section (due to lock reentrancy) even though the waiting thread is still in the critical section too.

CONSIDER wrapping asynchronous method calls with tasks. An asynchronous method call can be converted to a task by using the Task.Factory.FromAsync method:

```
IAsyncResult asyncResult =
    Dns.BeginGetHostAddresses("localhost", null, null);

// Convert the IAsyncResult to a task
Task<IPAddress[]> task =
    Task<IPAddress[]>.Factory.FromAsync(
        Dns.BeginGetHostAddresses,
        Dns.EndGetHostAddresses,
        "localhost", null);


// Task is often more convenient than an IAsyncResult.
// For example, you can schedule a continuation task:
task.ContinueWith(
    doneTask =>
    {
        Console.WriteLine("Found {0} IP addresses", doneTask.Result.Length);
    });

// We must wait until the continuation executes and prints the result
Console.ReadKey();
```

There are different variants of the FromAsync method that apply to different variants of the asynchronous pattern: IAsyncResult, Begin/End methods.

Parallel.For and Parallel.ForEach

The parallel looping constructs Parallel.For and Parallel.ForEach are conceptually similar to for and foreach loops, except that they use multiple threads to execute different iterations of the loop body. Parallel loops are often the easiest way to take advantage of multi-core machines to speed up an expensive operation over a sequence of elements.

DO use parallel loops Parallel.For and Parallel.ForEach to speed up operations where an expensive, independent operation needs to be performed for each input in a sequence.

For example, this code uses a Parallel.For loop to check which numbers in an input array are primes:

```
int[] numbers = ...;
bool[] isPrime = new bool[numbers.Length];

Parallel.For(0, numbers.Length, i => {
      isPrime[i] = numbers[i] >= 2;
      for(int factor = 2; factor * factor <= numbers[i]; factor++) {
            if (numbers[i] % factor == 0) {
                  isPrime[i] = false;
                  break;
            }
      }
});
```

Since multiple threads will be used to evaluate the loop body, this parallel loop will potentially execute faster on a multi-core machine, compared to a corresponding sequential loop.

DO make sure that the loop body delegate is thread-safe, since it will be called from multiple threads concurrently.

DO verify that the loop body delegate does not make assumptions about the order in which loop iterations will execute. For example, there is no guarantee that a thread will process its partition of input elements in the order in which they appear in the input, even though in the current version it will.

CONSIDER increasing the work done by each iteration in a parallel loop if it is very low. The body of a parallel loop is a delegate, and invoking it incurs some overhead. If the work done by the loop body is very small, the delegate invocation overhead may dominate the running time.

Here is an example of a parallel loop with an inexpensive body:

```
Parallel.For(0, arr.Length, i =>
{
    arr[i] = (int)Math.Sqrt(arr[i]);
});
```

In the parallel loop, the overhead of invoking the loop body delegate has a significant impact on the performance. To address this performance issue, we can restructure the loop so that each call of the delegate processes an entire chunk of elements.

To save you the work of breaking up the range into chunks, Partitioner class provides a solution that does this for you:

```
Parallel.ForEach(
    Partitioner.Create(0, arr.Length, 1024),
        range => {
            for (int i = range.Item1; i < range.Item2; i++)
            {
                arr[i] = (int)Math.Sqrt(arr[i]);
            }
        });
```

Now, instead of invoking the loop body delegate once for each element in the arr array, the delegate only gets invoked once for every 1,024 elements. The delegate now processes the **entire range** delimited by the Tuple<int,int>.

The computation is still parallel (provided that the arr array contains many more than 1,024 elements), but the overheads of delegate invocations has been dramatically lowered.

Parallel LINQ

Parallel LINQ (PLINQ) is to LINQ to Objects what parallel loops are to ordinary loops. PLINQ executes LINQ queries, but distributes the evaluation of the user delegates over multiple threads. To opt into PLINQ, use the AsParallel() extension method:

```
int[] arr = ...

arr.AsParallel()
      .Where(x => ExpensiveFilter(x));

foreach(var x in q) {
      Console.WriteLine(x);
}
```

DO use PLINQ to express computations with an expensive operation applied over a sequence.

BE AWARE that by default, PLINQ does not preserve ordering of elements in a query. For example, the results of this query will be printed in an unspecified order:

```
var q = Enumerable.Range(0, 100)
      .AsParallel()
      .Select(x => -x);

foreach(var x in q) Console.WriteLine(x);
```

To turn on ordering, you can either use the AsOrdered() operator to preserve the initial ordering in the input sequence, or use OrderBy() to sort the values.

Here is a version of the code sample above that preserves the ordering of the input elements, and prints the corresponding outputs in the correct order:

```
var q = Enumerable.Range(0, 100)
      .AsParallel()
      .AsOrdered()
      .Select(x => -x);

foreach(var x in q) Console.WriteLine(x);
```

DO keep PLINQ queries simple to make them easy to reason about. If possible, break up more complex queries so that the cheap but complex part is done externally to PLINQ, e.g. in LINQ to Objects. Consider this example:

```
var q = Enumerable.Range(0, 100)
```

```
      .TakeWhile(x => SomeFunction(x))
      .AsParallel()
      .Select(x => Foo(x));

foreach(var x in q) Console.WriteLine(x);
```

Notice that the AsParallel() is placed after the TakeWhile() operator rather than immediately on the data source.

BE AWARE that by default, PLINQ uses static partitioning on arrays and other collections that implement the IList<> interface. That means that the array will be statically split into as many partitions as there are cores on the machine. However, if the work distribution varies in the array, static partitioning may lead to load imbalance.

To use on-demand load-balancing partitioning, use Partitioner.Create() method, passing in the true value for the loadBalancing parameter:

```
int[] input = ...
Partitioner<int> partitioner = Partitioner.Create(input, true);
var q = partitioner.AsParallel()
      .Select(x => Foo(x))
      .ToArray();
```

Concurrent Collections

.NET 4 introduces a number of concurrent collections that are thread-safe and optimized for concurrent access from multiple threads. These collections include ConcurrentQueue, ConcurrentStack and ConcurrentDictionary that represent concurrent versions of Queue, Stack and Dictionary.

Also, .NET 4 introduces a BlockingCollection, a collection that is useful for implementing exchange buffers in producer-consumer scenarios.

CONSIDER using a ConcurrentQueue instead of a Queue with a lock. ConcurrentQueue is implemented using a low-lock algorithm that outperforms a locked Queue in some scenarios.

Here is a code sample that uses a Queue and a lock:

```
Queue<int> q = new Queue<int>();
object myLock = new object();

Task.StartNew(
      () => for(int i = 0; i < 100; i++) {
            int result = ComputeSomething(i);
            lock(myLock) {
                  q.Add(result);
            }
      });
```

You can convert the code to use a ConcurrentQueue instead:

```
ConcurrentQueue<int> q = new ConcurrentQueue<int>();
Task.StartNew(
```

```
        () => for(int i = 0; i < 100; i++) {
                q.Add(ComputeSomething(i));
        });
```

DO use ConcurrentDictionary instead of a Dictionary with a lock, in particular for dictionaries that are heavily accessed from multiple threads, especially if most of the accesses are reads. Reads of a ConcurrentDictionary ensure thread safety without taking locks.

CONSIDER using BlockingCollection to represent the communication buffer in consumer-producer scenarios. In such scenarios, one or more producer threads insert elements into a BlockingCollection, and one or more consumer threads remove elements from the BlockingCollection.

If a consumer attempts to remove an element from the BlockingCollection when the collection is empty, it will block and wait until an element gets inserted. A BlockingCollection can have a maximum size, and if a producer attempts to insert an element into a full BlockingCollection, the producer will block and wait until an element gets removed.

DO use regular collections with locks instead of concurrent collections if you need to perform compound atomic operations that are not supported by the corresponding concurrent collection.

For example, if you need to atomically remove one element and insert another element into the same queue, and no other thread should see the queue in between the two updates, you will have to use a regular queue with locking, not a ConcurrentQueue. ConcurrentQueue cannot atomically remove one element and add another one.

Coordination Primitives

New coordination primitives in .NET 4 encapsulate some of the most common parallel programming patterns.

DO use the Lazy type instead of implementing lazy initialization from scratch. Efficient and thread-safe implementation of lazy initialization is notoriously tricky to get right.

For example, you can create a lazily allocated large buffer as follows:

```
Lazy<byte[]> lazyBuffer =
    new Lazy<byte[]>(
        () => new byte[1 << 24]);
```

The buffer will be allocated on the first access to lazyBuffer.Value.

DO use ThreadLocal<> when you need to track a field or a variable that contains an independent value for each thread. For static fields, you can simply use the ThreadStatic attribute, but ThreadLocal<> also works for instance fields and local variables.

This example uses ThreadLocal<> to track how many elements in a PLINQ query have been processed by each thread:

```
ThreadLocal<int> localCount = new ThreadLocal<int>(() => 0);

Enumerable.Range(0, 100).AsParallel()
    .Select(x =>
    {
        localCount.Value = localCount.Value + 1;
        Console.WriteLine(
            "{0} elements so far processed on thread {1}",
            localCount.Value,
            Thread.CurrentThread.ManagedThreadId);

        Thread.Sleep(100);
        return x;
    })
    .ToArray();
```

DO minimize the number of accesses to the Value property of a ThreadLocal<> object, especially if accessing the thread-local value is the bottleneck of the operation. Reads and writes to Value are fairly cheap, but not as cheap as accesses to local variables.

For example, the ThreadLocal<> sample above could use a local variable to reduce the number of accesses to the Value property from three to two.

CONSIDER using ManualResetEventSlim instead of ManualResetEvent and SemaphoreSlim instead of Semaphore. The "slim" versions of the two coordination structures speed up many common scenarios by spinning for a while before allocating real wait handles.

You will, however, need to use the non-slim ManualResetEvent and Semaphore types if you need to pass the corresponding WaitHandle to some API (e.g. WaitHandle.WaitAny or WaitHandle.WaitAll), or if you need the wait handles for cross-process synchronization.

Correctness

Thread Safety

Whenever working with multi-threading, it is important to understand and carefully observe the thread-safety guarantees.

DO make sure that if an object is accessed from multiple threads, all methods called on the object are thread-safe, or otherwise correctly protected by locks. If a class was only designed for single-threaded usage, using it from multiple threads may result in exceptions, incorrect results, hangs, etc.

For example, the Withdraw method on the Account class below should not be called from multiple threads, or otherwise the account balance may get corrupted:

```
class Account {
      private int _balance;

      public bool Withdraw(int amount) {
            if (amount >= _balance) {
                  _balance -= amount;
```

```
                return true;
            }
            return false;
        }

        public void Deposit(int amount) {
            _balance += amount;
        }
        ...
}
```

If the Account class were a part of a third-party library, it is important you carefully read the documentation of the class to understand its thread-safety guarantees.

DO make sure that any static methods you implement are thread-safe. By convention, static methods should be thread-safe, and methods in BCL follow this convention. Carefully verify that your static methods will also work when called from multiple threads.

DO NOT use objects on one thread after they got disposed by another thread. This mistake is easy to introduce when the responsibility for disposing an object is not clearly defined.

In the example below, one thread may call MoveNext() on an enumerator that has been disposed by another thread:

```
IEnumerator<int> e = ...;
object myLock = new object();
Action walkEnumerator = () =>
{
    while (true)
    {
        lock (myLock)
        {
            if (!e.MoveNext()) break;
        }
    }
    lock (myLock) e.Dispose();
};

Parallel.Invoke(walkEnumerator, walkEnumerator);
```

Locks

Locks are the most important tool for protecting shared state. A lock provides mutual exclusion – only one thread at a time can be executing code protected by a single lock.

DO use locks to protect shared state. Often, one lock per object provides the appropriate locking granularity.

```
Account account = new Account(1000);
object accountLock = new object();
Parallel.Invoke(
     () => {
     lock(accountLock) { account.Withdraw(500); }
},
```

```
() => {
     lock(accountLock) { account.Withdraw(500); }
});
```

In this code sample, the two parallel Withdraw calls will be done with mutual exclusion, and so the account balance is not going to get corrupted.

DO always acquire locks in the same order. If two locks can be acquired in different order, deadlock may occur.

This code contains a bug – if Transfer(A, B) and Transfer(B, A) are called concurrently, the program may deadlock:

```
static bool Transfer(Account a1, Account a2, int amount)
{
    lock (a1) lock (a2) {
        bool success = a1.Withdraw(amount);
        if (success) a2.Deposit(amount);

        return success;
    }
}
```

DO use the C# and VB lock statements instead of directly calling Monitor.Enter() and Monitor.Exit(). Lock statements result in cleaner, more readable code, and also avoid a couple of subtle issues related to exception handling and thread aborts.

```
lock (myLock) {
    Foo();
}
```

In C# 4, this gets translated into a block of code roughly as follows:

```
bool acquired = false;
try {
    Monitor.Enter(myLock, ref acquired);
    Foo();
}
finally {
    if (acquired) Monitor.Exit(myLock);
}
```

DO NOT use publicly visible objects for locking. If an object is visible to the user, they may use it for their own locking protocol, despite the fact that such usage is not recommended.

CONSIDER using a dedicated lock object instead of reusing another object for locking. This simple example uses an object in the _lock field for locking:

```
class Account
{
    private object _lock = new object();
    private int _balance = 0;

    public void Deposit(int amount) {
```

```
        lock (_lock) { _balance += amount; }
    }

    ...
}
```

When the lock object is stored in a private field, it is hidden away from code in other classes and from overridden methods. It is sufficient to look at all references to the _lock field to understand the locking protocol.

In contrast, consider this version of the class:

```
class Account
{
    private int _balance = 0;

    public void Deposit(int amount) {
        // Locking on 'this' is NOT RECOMMENDED
        lock (this) { _balance += amount; }
    }

    ...
}
```

Now, any code that has a reference to an instance of the Account class can also take the lock. Reasoning about the locking protocol is now significantly harder. And, external code can potentially break the protocol, degrading the performance or even triggering deadlocks.

The overhead associated with a small object is negligible in most situations. However, in the rare cases where it is important, you can consider reusing an existing object for locking at the cost of readability and maintainability of the code.

DO NOT hold locks any longer than you have to. No other thread can enter the critical section while one thread holds the corresponding lock, and the impact of locks on the performance increases greatly when threads hold them longer than necessary.

DO NOT call virtual methods while holding a lock. Calling into unknown code while holding a lock poses a deadlock risk because the called code may attempt to acquire other locks. Acquiring locks in an unknown order may result in a deadlock.

DO use locks instead of advanced techniques such as lock-free programming, Interlocked, SpinLock, etc. These advanced techniques are tricky to use correctly and error-prone. If you do try to apply them, make sure you fully understand the techniques and their related issues, and that you measure the performance benefit and validate that their use is worthwhile for your scenario.

Performance

Improved performance is the goal behind multi-core programming, and so obviously performance should be at the forefront of your mind whenever writing parallel programs.

Measure

As with other optimizations, when using parallel programming it is important to measure and understand the benefit you are getting. If you don't measure, you may be spending time optimizing the wrong parts of your code. Also, if you don't measure the effect of an optimization, it may in fact make your code slower, even though intuitively it seems that it should be faster.

DO measure the performance of your program before and after you parallelize it.

DO make sure to measure the right thing: the release build with no debugger attached. The debug build can be meaningless to measure, and having the debugger attached can also skew the results.

CONSIDER measuring the warm-state performance rather than the cold-state performance. Repeat the measurement multiple times and exclude the first few measurements from the score.

Warm-state measurement will exclude various initialization operations (JIT, threadpool initialization, memory paging, etc). Warm-state measurement is more stable and predictable, and often represents the number that is most useful for optimization.

Sometimes it can be useful to track the cold-state performance too, to get an idea of the startup costs associated with your library or program. However, cold-state results tend to be harder to understand, interpret, and act up on.

DO NOT assume that if your algorithm runs twice faster on two cores than on a single core, it will run eight times faster on eight cores.

Many algorithms do not scale linearly, either because there isn't enough parallelism in the algorithm, or because an imperfectly scalable subsystem (memory hierarchy, GC) begins to dominate the running time.

If it is important that your algorithm scales to high numbers of cores, you will need to get access to appropriate hardware and verify the scalability.

BE AWARE that there can be considerable performance and scalability differences between different hardware architectures. Notably, programs may perform differently on 32-bit and 64-bit architectures because of different pointer sizes, and also because of differences in JIT behavior.

If parallel scaling of your algorithm is important, consider testing on different architectures. Particularly, it is worthwhile to compare 32-bit and 64-bit performance.

Memory Allocations and Performance

It is generally a good idea to limit memory allocations in high-performance code. Even though the .NET garbage collector (GC) is highly optimized, it can have significant impact on performance of code that spends most of its time allocating memory.

In parallel programs, there is another reason to watch out for memory allocations. The problem is that if your program spends most of its time in GC, it will be only as scalable as the GC algorithm. And, by

default, CLR uses a single-threaded GC algorithm, and so your program will not scale on multiple cores if it spends most of its time doing GC.

AVOID unnecessarily allocating many small objects in your program. Watch out for boxing, string concatenation and other frequent memory allocations.

CONSIDER opting into server GC for parallel applications.

The default garbage collection algorithm used in .NET 4 is called "Background GC". Background GC uses a single thread that runs in parallel with your program thread. Background GC aims to minimize the pause time for your program threads, which is important for applications with a user interface.

But, Background GC does not take advantage of all cores on your machine in programs that spend most of their time in GC. To address this scenario, CLR provides an alternate GC mode: Server GC. Server GC maintains multiple heaps, one for each core on the machine. These heaps can be collected in parallel more easily.

You can opt into the Server GC by adding an app.config file to your application that contains these tags:

```
<configuration>
  <runtime>
    <gcServer enabled="true" />
  </runtime>
</configuration>
```

Caches and Performance

Caches are very important for computing performance. If your parallel program degrades the use of caches, it may well be slower than a similar sequential program. To prevent this from happening, it helps to be aware of how caches work in parallel programs.

Modern multi-core architectures have multiple levels of caches. The fastest level of caches is on the cores themselves – each core has its own very fast cache. To keep the caches consistent with each other and with the main memory, cores work together to invalidate each other's caches when necessary. If one core modifies a location in memory, all cores that have that location in cache must invalidate it.

One factor that further complicates this issue is that caches operate on the granularity of cache lines – adjacent blocks of 64 or 128 bytes on most modern architectures. When one core overwrites a location in memory, it invalidates the entire cache line (64 – 128 bytes) around the modified location in the caches of all other cores. This problem is called *false sharing*.

Here is an example that demonstrates false sharing:

```
public static void FalseSharing()
{
    int cores = Environment.ProcessorCount;
    int[] counts = new int[cores];
    Parallel.For(0, cores, i => {
        for (int j = 0; j < 10000000; j++)
```

```
        {
            counts[i] = counts[i] + 3;
        }
    });
}
```

When you replace the Parallel.For in the above code sample with an ordinary for loop, the program actually **gets faster**! Here are the measurements obtained on a quad-core Intel Core2 Quad 2.67GHz machine**:**

Sequential	90 ms
Parallel	626 ms

Even though we are hoping to see the parallel program to be four times faster than the sequential version, in fact it is seven times slower! False sharing is the issue that ruins the performance of the parallel program.

DO store values that are frequently modified in stack-allocated variables whenever possible. A thread's stack is stored in a region of memory only modified by the owner thread.

One easy way to fix the false sharing problem in the above example is to have each thread accumulate the result in its own local variable:

```
public static void FalseSharingFix1()
{
    int cores = Environment.ProcessorCount;
    int[] counts = new int[cores];

    Parallel.For(0, cores, i => {
        int localCount = 0;
        for (int j = 0; j < 10000000; j++)
        {
            localCount = localCount + 3;
        }
        counts[i] = localCount;
    });
}
```

This version of the program takes 6ms to run on our test machine. This is more than four times faster than the sequential version of the program (90ms) – beyond fixing false sharing, local variable accesses can also be faster than array accesses because the JIT compiler may cache the local variable in a register.

CONSIDER padding values that are frequently overwritten by different threads. This ensures that each value is on its own cache line. The example below pads the counter values so that each is in its own 128 byte cache line:

```
public static void FalseSharingFix2()
{
    int cores = Environment.ProcessorCount;
    const int PADDING = 128 / sizeof(int);
    int[] counts = new int[(1 + cores) * PADDING];
```

```
    Parallel.For(0, cores, i => {
        int paddedI = (i + 1) * PADDING;
        for (int j = 0; j < 10000000; j++)
        {
            counts[paddedI] = counts[paddedI] + 3;
        }
    });
}
```

The padding speeds up the parallel program from 626ms to 26ms.

If you want to set the padding precisely, you can pull that information about cache line sizes out of the SYSTEM_LOGICAL_PROCESSOR_INFORMATION structure returned by the GetLogicalProcessorInformation Windows API function. Also, there is a SysInternals tool called Coreinfo that also displays this information.

CONSIDER allocating frequently updated heap memory locations by owner threads, if they are not padded. Even though the CLR does not guarantee it, the memory allocated from one thread tends to stay together on the heap.

A third possible fix for the false sharing is to have each thread allocate an array of length 1 to store the value it keeps updating. That way, each thread is mutating a location in memory that was allocated by the thread itself:

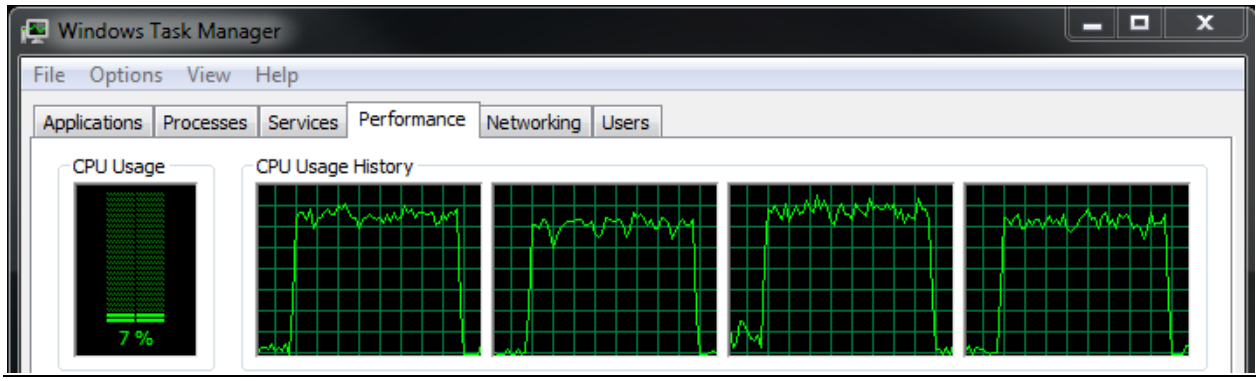
```
public static void FalseSharingFix3()
{
    int cores = Environment.ProcessorCount;
    int[][] counts = new int[cores][];

    Parallel.For(0, cores, i => {
        Counts[i] = new int[1];
        for (int j = 0; j < 10000000; j++)
        {
            counts[i][0] = counts[i][0] + 3;
        }
    });
}
```

Performance Tools

Appropriate tools can be a great help when trying to understand the performance of a parallel program.

DO look at the Performance tab in the Windows Task Manager to understand how a program utilizes cores on your machine.

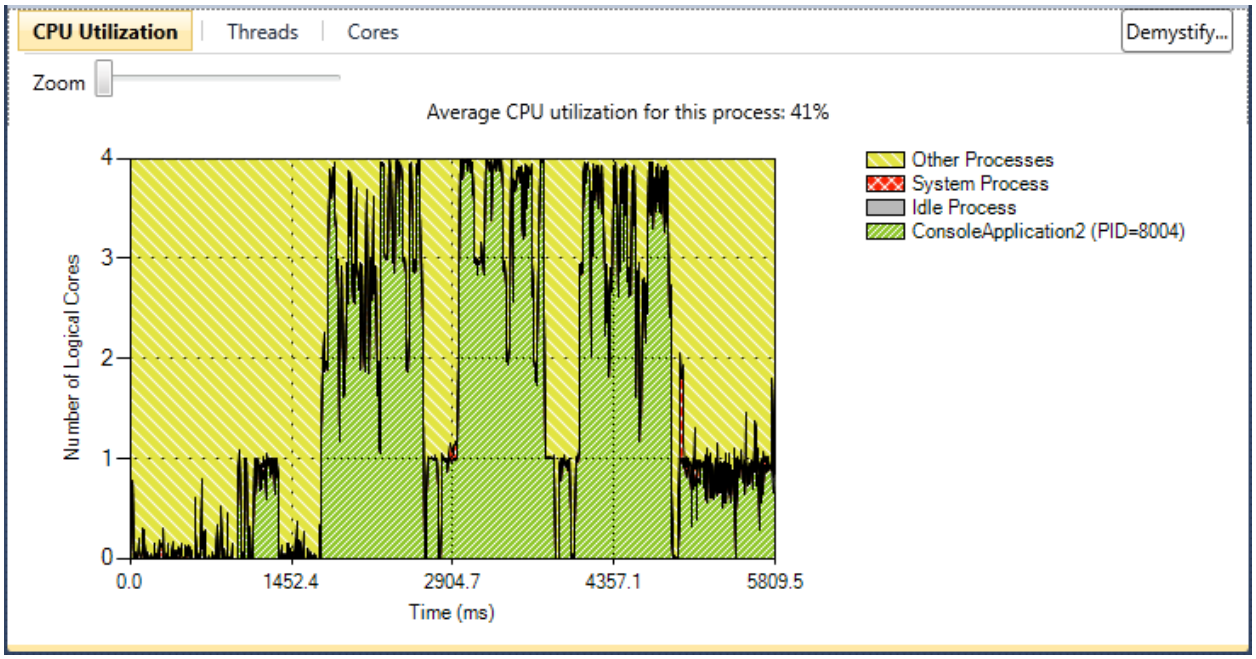
In this screenshot, the four cores have been about 80% utilized while a testing program ran:



DO use the Concurrency Visualizer tool in the Visual Studio 2010 Profiler to understand the behavior of your program. The Concurrency Visualizer provides three views into the performance of your application: CPU Utilization view, Threads view, and Cores view.

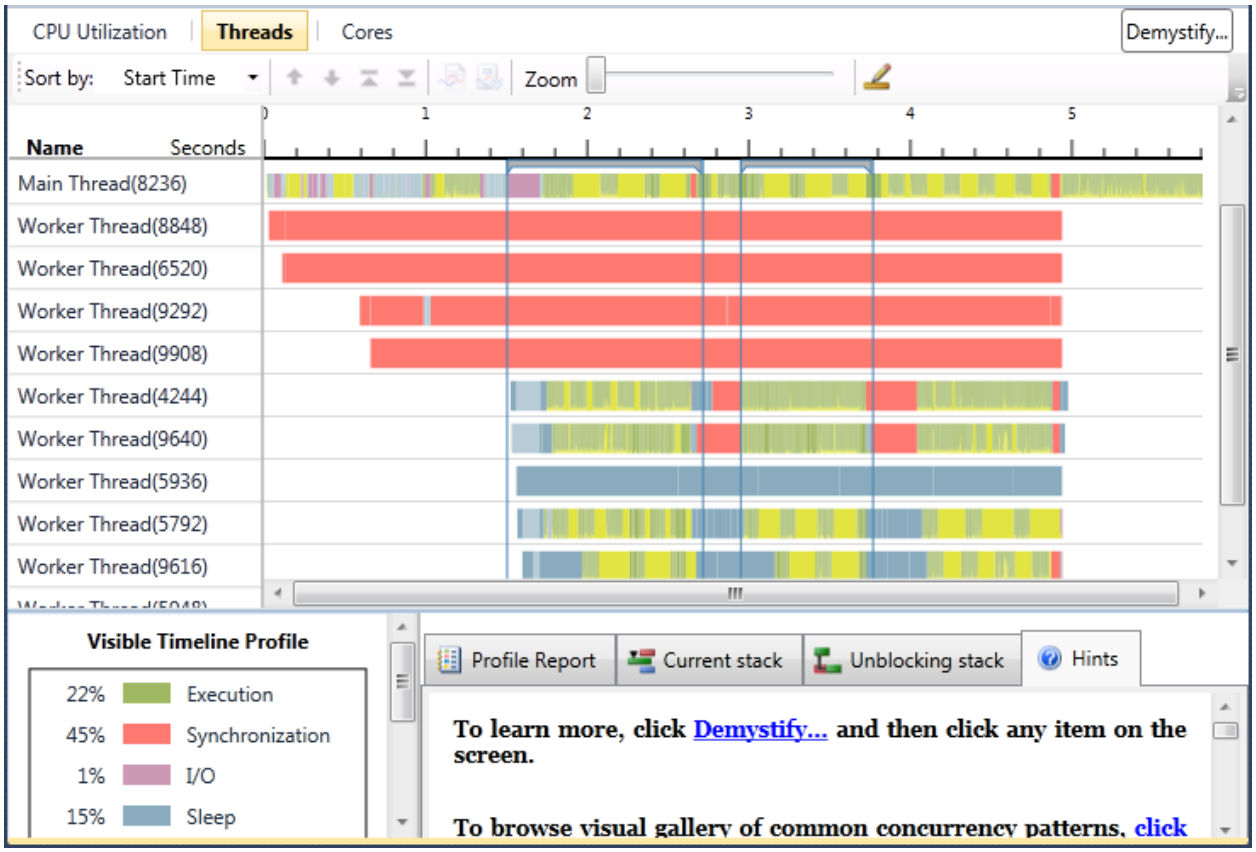
CPU Utilization view displays a graph of core utilization in your program over time. The graph can help you identify parts of your program that don't fully utilize cores on the machine, and provides a good starting point for your investigation.

CPU Utilization view is effectively a more detailed version of the data shown by the Windows Task Manager:

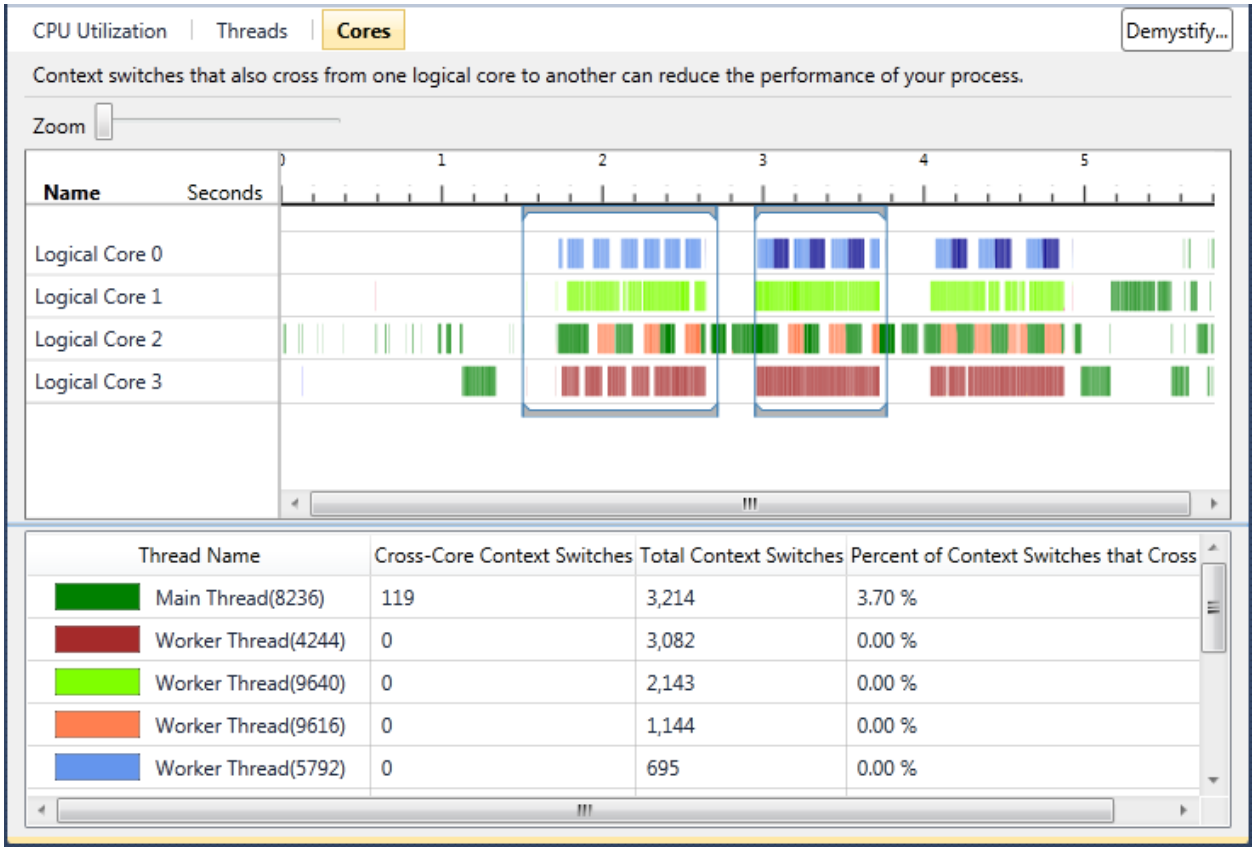


Threads view displays a timeline of what each thread did throughout the execution of your program. At each moment in time, you can see whether the thread was running, waiting on I/O, waiting on synchronization, or doing something else.

To find out why the thread was in a particular state, you can click on a particular point on the timeline and you'll see the thread's call stack. To understand why a thread was blocked, you can click on a particular waiting region and look at the call stack that eventually unblocked the wait. Here is an example of the Threads view:



Cores view displays the threads that the physical cores were executing over time. This view can help identify issues with thread migration for programs with explicitly managed thread affinity:



DO use a profiler to identify bottlenecks in your program. Visual Studio Profiler is a helpful tool for optimizing both parallel and sequential programs.

Library Development

The easiest way to write a program that scales on multi-core machines is build it on top of domain-specific libraries that employ parallelism under the covers. As machines with higher numbers of cores become more prevalent among mainstream computers, the need for parallel libraries will increase.

Linear algebra, compression, encryption and image manipulation are all examples of common computationally-expensive operations that can be parallelized. This section discusses the best practices of designing these kinds of reusable parallel libraries.

Thread Safety and Documentation

In order for users to be able to effectively and correctly use a library, the library has to be thoroughly documented. This is true for all reusable libraries, but even more so for libraries that deal with parallelism.

When using a parallel library, it is important to understand the thread safety guarantees made by each method.

DO follow good documentation practices for your library. Thoroughly document the role and usage of each type and method.

DO explain thread-safety guarantees in your documentation. If a class is "thread-safe", it generally means that its methods can be called from multiple threads concurrently. Carefully explain any tricky cases that the user is likely to run into.

Here is an example of a simple class with documented thread-safety guarantees:

```
/// <summary>
/// A thread-safe counter that can be safely incremented and read
/// from multiple threads.
/// </summary>
class Counter
{
    /// <summary>
    /// Increments the counter. The Increment method is thread-safe,
    /// and so can be called from multiple threads concurrently.
    /// </summary>
    public void Increment() { ... }

    /// <summary>
    /// Gets the value of the counter. The Value property getter can
    /// be safely read even if the counter is concurrently modified
    /// from other threads.
    /// </summary>
    public int Value { get { ... } }
}
```

DO use examples to illustrate the correct usage of your library. Thread safety guarantees can be difficult to understand from an abstract explanation, and concrete examples can help the users understand how your library is intended to be used.

Optional Parallelism

Generally, the most efficient way to parallelize a program is to only parallelize the computation at the topmost level. For example, consider this problem:

- Compute a **sum of matrix powers** using
- **matrix exponentiation**, which uses
- **matrix multiplication**, which uses
- **big integer (> 64-bit) multiplication**.

While all computations in this stack can be parallelized, you'll likely get the best performance only by parallelizing the top-most computation. So, you'd compute different matrix powers in parallel, but use sequential algorithms for matrix exponentiation, multiplication, and big integer multiplication.

DO provide a sequential version of each algorithm in your library. There are two possible ways to incorporate this idea into your API:

1. **Expose a sequential and a parallel version of each method**. For example, you would have both Matrix.Multiply and Matrix.MultiplyParallel.

2. Or, **support an optional degreeOfParallelism argument on all methods**. The degreeOfParallelism argument tells the library how many operations should be executed concurrently.

This example implements a simple matrix multiplication algorithm with tunable degree of parallelism:

```
static int[,] MatrixMultiply(int[,] a, int[,] b, int degreeOfParallelism)
{
    // [... argument validation not shown ...]

    int aRows = a.GetUpperBound(0) + 1;
    int aCols = a.GetUpperBound(1) + 1;
    int bRows = b.GetUpperBound(0) + 1;
    int bCols = b.GetUpperBound(1) + 1;

    int[,] result = new int[bRows, aCols];
    ParallelOptions options = new ParallelOptions {
        MaxDegreeOfParallelism = degreeOfParallelism };

    Parallel.For(0, bRows, options, row =>
    {
        for (int col = 0; col < aCols; col++)
            for (int k = 0; k < aRows; k++)
                result[row, col] += a[k, col] * b[row, k];
    });

    return result;
}
```

Note that one worthwhile optimization may be to use an ordinary sequential for-loop if degreeOfParallelism is 1. Parallel.For has higher overheads than an ordinary parallel loop, and also disables some compiler optimizations such as caching values in registers.

DO optimize both the sequential and the parallel version of the algorithm. If some optimizations only apply to the sequential algorithm, use them in the sequential case.

Exceptions

When designing parallel libraries, it is important to have a consistent plan around exception handling. Since multiple operations happen in parallel, more than one of those operations may throw an exception. The solution used by .NET parallel primitives is to gather the exceptions into a collection, and throw an AggregateException instead.

For example, parallel loops, task waiting operations and PLINQ queries all throw AggregateException if the user delegate throws an exception.

AVOID throwing AggregateException out of your parallel methods whenever possible. AggregateExceptions are difficult for the user to handle because it is necessary to inspect all exceptions in the bag.

DO validate inputs early on instead of throwing an AggregateException from the parallel part of the computation.

Cancellation

Parallelism is particularly useful for long-running expensive computations. When dealing with long-running computations, it is often very useful to be able to cancel them. For example, if the user decides that they don't need the result of a running operation, they should be able to click a "Cancel" button and stop the computation.

DO expose a cancellation mechanism for parallel operations.

DO use the .NET cancellation API instead of designing your own mechanism. Using the .NET cancellation primitives makes your library easier to compose with other code that uses .NET cancellation. Also, using a standard API lowers the learning curve for the users of your library.

The .NET cancellation API is based around two primitives: a CancellationToken and CancellationTokenSource.

- **CancellationToken** represents the ability to check whether an operation has been canceled.
- **CancellationTokenSource** represents the ability to cancel a running operation.

This is how you can expose cancellation in a parallel library:

```
static void StartFoo(CancellationToken token)
{
    Task.Factory.StartNew(
        () =>
        {
            while (true)
            {
                cancel.ThrowIfCancellationRequested();

                // ... do asynchronous work
            }
        }, cancel);
}
```

And this is how the user can take advantage of the cancellation:

```
class Program
{
    CancellationTokenSource _cancelSource = null;

    void StartComputation()
    {
        _cancelSource = new CancellationTokenSource()
        StartFoo(_cancelSource.Token);
    }

    void CancelComputation()
    {
```

```
            _cancelSource.Cancel();
            _cancelSource = null;
        }
}
```

DO poll the cancellation token in expensive functions by calling token.ThrowIfCancellationRequested(). Primitives like Parallel.For and PLINQ typically do not poll the cancellation token after every call to the user delegates, in order to keep the overhead of cancellation checking low.

For example, a PLINQ query might poll the cancellation token only after every 64 elements have been processed, which can take a long time if processing each element is expensive. So, it is important that the function itself polls the cancellation token, if timely cancellation is desired.

DO prefer APIs where each method accepts a CancellationToken as a parameter instead of storing the CancellationToken in a field.

Prefer this API:

```
// Better API
class Operation
{
   public void Run(CancellationToken token) { }
}
```

Over this API:

```
// Worse API
class Operation
{
    public CancellationToken CancellationToken { get; set; }
    public void Run() { ... }
}
```

The CancellationToken should be associated with an operation (typically a method) rather than with an object or a data structure.

DO use cancellation callbacks for short bits of work to be performed when the token is cancelled:

```
CancellationTokenSource tokenSource = new CancellationTokenSource();
CancellationToken token = tokenSource.Token;

token.Register(() => Console.WriteLine("Cancellation initiated"));
tokenSource.Cancel();
```

DO NOT dispose a cancellation callback from within the callback itself:

```
// DO NOT do this
CancellationToken token;
CancellationTokenRegistration ctr = token.Register(() => { ctr.Dispose(); );
```

The call to ctr.Dispose() typically waits until the callback has finished executing. If the call to ctr.Dispose() happens inside the callback, however, waiting on the callback to finish would cause a deadlock. The current implementation works around the deadlock by skipping the wait, but it is not a good idea to rely on this.

Testing

When writing multi-threaded applications, following good testing practices is even more important than it is for single-threaded applications.

DO follow good testing practices. Thoroughly test individual components by unit tests. Use integration tests to verify that different components work correctly together. And ensure that your tests cover your entire code base, ideally by tracking code coverage metrics.

Concurrency Testing

As with testing of sequential programs, when testing parallel programs, it is important to identify portions of the code that are particularly complicated and prone to errors.

DO identify parts your code base that are particularly at risk of concurrency issues. Any code that uses tricky parallel techniques (interlocked operations, volatile fields, spin locks, etc.) automatically falls into this category. You should also include other code that uses parallel primitives like tasks, parallel loops and locks.

DO focus testing effort on the parts of the code at risk of concurrency issues. Review these parts of the code multiple times, cover them with stress tests, and consider using dedicated concurrency testing tools (see the Testing Tools section below).

CONSIDER using a stress test to help you catch concurrency issues. Try running an operation in a loop for a long time, try running many instances of the operation running in parallel in one process, and try combining your operations in an artificially complex way that stresses your code as far as it can go.

Any one of these approaches may help you catch a bug that rarely reproduces in simple tests.

Testing Tools

Special testing tools can help with catching tricky concurrency bugs.

CONSIDER writing concurrency unit tests using the CHESS tool from Microsoft Research for components that are at risk of concurrency issues. CHESS is a tool for finding hard-to-reproduce concurrency bugs. To catch these bugs, CHESS runs many different interleavings of the parallel operations in each test. If an interleaving produces a wrong answer, crashes or hangs, CHESS helps you understand the issue by providing a visualization of the interleaving that caused the error.

CONSIDER tracking synchronization coverage as a part of your testing metrics. Synchronization coverage tracks how many of your lock statements have been exercised in two interesting situations: with and without blocking the thread acquiring the lock.

Interactions

User Interfaces

Virtually all modern user-interface frameworks – including Windows Forms and WPF – require that controls are only updated from one dedicated thread.

If you attempt to update a control from another thread, you may get an exception, the control may get corrupted, or the update may happen to work. It is important to be aware of this issue when developing parallel application with a user interface.

DO use an appropriate mechanism to dispatch the update to the dedicated UI thread.

This WPF example is **broken** because it modifies a TextBox from a Task:

```
static void StartComputation()
{
    Task.Factory.StartNew(
        () =>
        {
            textBox1.Text = "Starting..."; // BUG!
        });
}
```

To fix the problem, dispatch the update to the control:

```
static void StartComputation()
{
    Task.Factory.StartNew(
        () =>
        {
            textBox1.Dispatcher.Invoke(
                new Action(()=>
                {
                    textBox1.Text = "Starting...";
                }));
        });
}
```

In Windows Forms, the fix is only slightly different:

```
static void StartComputation()
{
    Task.Factory.StartNew(
        () =>
        {
            textBox1.Invoke(
                new Action(()=>
                {
                    textBox1.Text = "Starting...";
                }));
        });
}
```

An alternative solution that works with both WPF and Windows Forms is to use a task scheduler that executes its work on the main thread. Then, the work to run on the main thread is defined as a continuation task that will execute on the special task scheduler:

```
static void StartComputation()
{
    var uiTaskScheduler = TaskScheduler.FromCurrentSynchronizationContext();
    Task.Factory.StartNew(() => “Starting...”)
        .ContinueWith(
            result => {
                textBox1.Text = result;
            },
            uiTaskScheduler));
        });
}
```

For more details on this approach, see this blog post.

DO NOT take locks or wait on tasks or any other events while executing on the UI thread.

Having the UI thread wait on another thread introduces a deadlock risk. Here is an example of a problem that may occur:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    Task.Factory.StartNew(ComputeSomething);

    // Wait 100 ms to make the deadlock more likely
    Thread.Sleep(100);

    // Acquire myLock
    lock (myLock) { }
}

private void ComputeSomething()
{
    lock (myLock)
    {
        // Dispatch an update to the UI thread while holding myLock
        button1.Dispatcher.Invoke(
            new Action(() => { button1.Content = "Clicked"; }));
    }
}
```

The above example will typically deadlock:

- The UI thread is blocked waiting to acquire myLock.
- The task thread is holding myLock, waiting until the UI thread completes the dispatched update.

Since neither activity can complete, we have a deadlock.

DO NOT start parallel loops or PLINQ queries on the UI thread. If the body of the parallel loop or PLINQ query dispatches a synchronous update to the UI, a deadlock will occur, similar to the lock example above.

ASP.NET

In server scenarios such as ASP.NET, a kind of parallelism comes for free. ASP.NET schedules different requests as work items on the ThreadPool. On multi-core machines, multiple threads are used to execute work items from the ThreadPool queue, and so multiple cores get used whenever there are multiple concurrent requests.

Quite often, additional parallelism beyond the request level will not significantly improve the performance of an ASP.NET application. However, some scenarios can still benefit from parallelism within a request.

CONSIDER using parallelism in a computationally expensive ASP.NET page handler. This may help decrease the response latency in cases when there aren't enough concurrent requests to utilize all cores on the server.

DO measure the performance of your parallel page handler, and validate that the parallelism indeed is a benefit for your particular workload.

DO prefer tasks and parallel loops over PLINQ queries in ASP.NET page handlers. In .NET 4, some PLINQ queries require a fixed number of threads from the ThreadPool to execute. Running a lot of such PLINQ queries in parallel may result in temporary delays, until the ThreadPool injects enough threads to make progress.

Tasks and parallel loops degrade better than PLINQ queries if the ThreadPool is busy executing other work. Note that simple queries with operators like Select, Where, and Count would also degrade well in a presence of many concurrent queries. Only more complex operators like GroupBy, OrderBy or Join typically trigger parallel algorithms that require a fixed number of threads to execute.

Silverlight

Silverlight is a runtime for interactive web applications based on .NET that supports a subset of the .NET 4 libraries. Specifically, a number of threading and parallelism libraries are not available in Silverlight, as of Silverlight 3 and Silverlight 4 Beta.

DO use ThreadPool work items for parallelism in Silverlight applications since Task, Parallel.For/ForEach loops and PLINQ are not currently available in Silverlight.

Learn More

In addition to this document, many other resources are available for you to learn about parallel programming in .NET and keep up with the latest developments.

The Parallel Computing Developer Center (http://msdn.com/concurrency) contains a number of helpful resources:

- Technical articles (http://msdn.com/concurrency/bb895974.aspx)
- Code samples (http://code.msdn.com/ParExtSamples)
- A quick reference guide (http://msdn.com/library/dd460693(VS.100).aspx)
- And links to many other resources

Read the Parallel Programming with .NET Team Blog (http://blogs.msdn.com/pfxteam) for the latest updates on parallel programming in .NET, and Tools for Parallel Programming Blog (http://blogs.msdn.com/visualizeparallel) for news on parallel programming tools in Visual Studio.

If you have a question about parallel computing, ask in the parallel programming forums (http://social.msdn.microsoft.com/Forums/en-US/category/parallelcomputing)

And finally, various videos and screencasts on parallelism are available on Channel 9 (http://channel9.msdn.com/tags/Parallel+Computing/).