# Potential Pitfalls in Data and Task Parallelism

Article • 04/14/2022

In many cases, Parallel.For and Parallel.ForEach can provide significant performance improvements over ordinary sequential loops. However, the work of parallelizing the loop introduces complexity that can lead to problems that, in sequential code, are not as common or are not encountered at all. This topic lists some practices to avoid when you write parallel loops.

## Do Not Assume That Parallel Is Always Faster

In certain cases a parallel loop might run slower than its sequential equivalent. The basic rule of thumb is that parallel loops that have few iterations and fast user delegates are unlikely to speedup much. However, because many factors are involved in performance, we recommend that you always measure actual results.

## Avoid Writing to Shared Memory Locations

In sequential code, it is not uncommon to read from or write to static variables or class fields. However, whenever multiple threads are accessing such variables concurrently, there is a big potential for race conditions. Even though you can use locks to synchronize access to the variable, the cost of synchronization can hurt performance. Therefore, we recommend that you avoid, or at least limit, access to shared state in a parallel loop as much as possible. The best way to do this is to use the overloads of Parallel.For and Parallel.ForEach that use a System.Threading.ThreadLocal<T> variable to store thread-local state during loop execution. For more information, see How to: Write a Parallel.For Loop with Thread-Local Variables and How to: Write a Parallel.ForEach Loop with Partition-Local Variables.

## Avoid Over-Parallelization

By using parallel loops, you incur the overhead costs of partitioning the source collection and synchronizing the worker threads. The benefits of parallelization are further limited by the number of processors on the computer. There is no speedup to be gained by running multiple compute-bound threads on just one processor. Therefore, you must be careful not to over-parallelize a loop.

The most common scenario in which over-parallelization can occur is in nested loops. In most cases, it is best to parallelize only the outer loop unless one or more of the following conditions apply:

- The inner loop is known to be very long.
- You are performing an expensive computation on each order. (The operation shown in the example is not expensive.)
- The target system is known to have enough processors to handle the number of threads that will be produced by parallelizing the query on `cust.Orders`.

In all cases, the best way to determine the optimum query shape is to test and measure.

## Avoid Calls to Non-Thread-Safe Methods

Writing to non-thread-safe instance methods from a parallel loop can lead to data corruption which may or may not go undetected in your program. It can also lead to exceptions. In the following example, multiple threads would be attempting to call the FileStream.WriteByte method simultaneously, which is not supported by the class.

C#

```csharp
FileStream fs = File.OpenWrite(path);
byte[] bytes = new Byte[10000000];
// ...
Parallel.For(0, bytes.Length, (i) => fs.WriteByte(bytes[i]));
```

## Limit Calls to Thread-Safe Methods

Most static methods in .NET are thread-safe and can be called from multiple threads concurrently. However, even in these cases, the synchronization involved can lead to significant slowdown in the query.

**Note**

You can test for this yourself by inserting some calls to **WriteLine** in your queries. Although this method is used in the documentation examples for demonstration purposes, do not use it in parallel loops unless necessary.

## Be Aware of Thread Affinity Issues

Some technologies, for example, COM interoperability for Single-Threaded Apartment (STA) components, Windows Forms, and Windows Presentation Foundation (WPF), impose thread affinity restrictions that require code to run on a specific thread. For example, in both Windows Forms and WPF, a control can only be accessed on the thread on which it was created. This means, for example, that you cannot update a list control from a parallel loop unless you configure the thread scheduler to schedule work only on the UI thread. For more information, see Specifying a synchronization context.

## Use Caution When Waiting in Delegates That Are Called by Parallel.Invoke

In certain circumstances, the Task Parallel Library will inline a task, which means it runs on the task on the currently executing thread. (For more information, see Task Schedulers.) This performance optimization can lead to deadlock in certain cases. For example, two tasks might run the same delegate code, which signals when an event occurs, and then waits for the other task to signal. If the second task is inlined on the same thread as the first, and the first goes into a Wait state, the second task will never be able to signal its event.

To avoid such an occurrence, you can specify a timeout on the Wait operation, or use explicit thread constructors to help ensure that one task cannot block the other.

## Do Not Assume that Iterations of ForEach, For and ForAll Always Execute in Parallel

It is important to keep in mind that individual iterations in a For, ForEach or ForAll loop may but do not have to execute in parallel. Therefore, you should avoid writing any code that depends for correctness on parallel execution of iterations or on the execution of iterations in any particular order. For example, this code is likely to deadlock:

C#

```
ManualResetEventSlim mre = new ManualResetEventSlim();
Enumerable.Range(0, Environment.ProcessorCount * 100)
    .AsParallel()
    .ForAll((j) =>
        {
            if (j == Environment.ProcessorCount)
            {
                Console.WriteLine("Set on {0} with value of {1}",
                    Thread.CurrentThread.ManagedThreadId, j);
                mre.Set();
            }
            else
            {
                Console.WriteLine("Waiting on {0} with value of {1}",
                    Thread.CurrentThread.ManagedThreadId, j);
                mre.Wait();
            }
        }); //deadlocks
```

In this example, one iteration sets an event, and all other iterations wait on the event. None of the waiting iterations can complete until the event-setting iteration has completed. However, it is possible that the waiting iterations block all threads that are used to execute the parallel loop, before the event-setting iteration has had a chance to execute. This results in a deadlock – the event-setting iteration will never execute, and the waiting iterations will never wake up.

In particular, one iteration of a parallel loop should never wait on another iteration of the loop to make progress. If the parallel loop decides to schedule the iterations sequentially but in the opposite order, a deadlock will occur.

## Avoid Executing Parallel Loops on the UI Thread

It is important to keep your application's user interface (UI) responsive. If an operation contains enough work to warrant parallelization, then it likely should not be run on the UI thread. Instead, it should offload that operation to be run on a background thread. For example, if you want to use a parallel loop to compute some data that should then be rendered into a UI control, you should consider executing the loop within a task instance rather than directly in a UI event handler. Only when the core computation has completed should you then marshal the UI update back to the UI thread.

If you do run parallel loops on the UI thread, be careful to avoid updating UI controls from within the loop. Attempting to update UI controls from within a parallel loop that is executing on the UI thread can lead to state corruption, exceptions, delayed updates, and even deadlocks, depending on how the UI update is invoked. In the following example, the parallel loop blocks the UI thread on which it's executing until all iterations are complete. However, if an iteration of the loop is running on a background thread (as For may do), the call to Invoke causes a message to be submitted to the UI thread and blocks waiting for that message to be processed. Since the UI thread is blocked running the For, the message can never be processed, and the UI thread deadlocks.

C#

```csharp
private void button1_Click(object sender, EventArgs e)
{
    Parallel.For(0, N, i =>
    {
        // do work for i
        button1.Invoke((Action)delegate { DisplayProgress(i); });
    });
}
```

The following example shows how to avoid the deadlock, by running the loop inside a task instance. The UI thread is not blocked by the loop, and the message can be processed.

C#

```csharp
private void button1_Click(object sender, EventArgs e)
{
    Task.Factory.StartNew(() =>
        Parallel.For(0, N, i =>
        {
            // do work for i
            button1.Invoke((Action)delegate { DisplayProgress(i); });
        })
         );
}
```

## See also

- Parallel Programming
- Potential Pitfalls with PLINQ
- Patterns for Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4