# Microsoft Parallel Extensions

Parallel Extensions provide several new ways to express parallelism in your code:

* **Declarative data parallelism** - Parallel Language Integrated Query (or Parallel LINQ) is an implementation of LINQ-to-Objects that executes queries in parallel, scaling to utilize the available cores and processors of the machine. Because queries are declarative, you are able to express what you want to accomplish, rather than how you want to accomplish it.
* **Imperative data parallelism** - Parallel Extensions also contains mechanisms to express common imperative data-oriented operations such as for and foreach loops, automatically dividing the work in the loop to run on parallel hardware.
* **Imperative task parallelism** - Rather than using data to drive parallelism, Parallel Extensions enables you to express potential parallelism via expressions and statements that take the form of lightweight tasks. Parallel Extensions schedules these tasks to run on parallel hardware and provides capabilities to cancel and wait on tasks.

### PLINQ

Accessing PLINQ’s functionality is accomplished in one of two ways:

* The System.Linq.ParallelEnumerable class
* The AsParallel extension method, exposed from the System.Linq.ParallelQuery class

The [System.Linq.ParallelEnumerable](http://docs.google.com/html/b2d79b40-3154-e79e-8e6a-a504e8f47784.htm) class mirrors the **System.Linq.Enumerable** class exposed from System.Core.dll. providing the same set of static extension methods but with concurrent implementations.

The second approach takes advantage of extension methods and involves using the AsParallel PLINQ method.

IEnumerable<T> data = ...;

var q = data.Where(x => p(x)).Orderby(x => k(x)).Select(x => f(x));

foreach (var e in q) a(e);

Turns to be:

IEnumerable<T> data = ...;

var q = data.AsParallel().Where(x => p(x)).Orderby(x => k(x)).Select(x => f(x));

foreach (var e in q) a(e);

**Not all LINQ queries are good candidates for parallel execution.**

Some LINQ queries contain side-effects which render them incorrect when run under PLINQ, while other queries may be more I/O-bound than CPU-bound, decreasing the value of using multiple processors. Additionally, the queries that will parallelize best are those that query large amounts of data, perform expensive computations, or a combination of both.

PLINQ does not guarantee order preservation by default, due to items from the input being processed in parallel. If order preservation is important, the [AsOrdered](http://docs.google.com/html/e16dc5d9-2d59-244b-a105-569bd98fb58b.htm) method can be used to enable order-preservation

### **Using the Task Parallel Library**

**Using System.Threading.Parallel**

The [System.Threading.Parallel](http://docs.google.com/html/18816338-cac8-9d1e-c7ac-b52f54d425d3.htm) class is useful for solving data parallel problems, as it provides support for parallelizing loops and regions in .NET applications.

The System.Threading.Parallel class supports the parallelization of independent loops, whereby a developer can take a loop that executes sequentially and convert it to one where every iteration of the loop has the potential to run in parallel, provided enough processing cores are available. Please take note: loops whose iterations are dependent or contain side-effecting operations will be **incorrect**  when run in parallel without the addition of proper custom synchronization.

**ForEach**

Consider the following sequential C# for loop:

for (int i = 0; i < N; i++)

{

results[i] = Compute(i);

}

Using TPL as:

Parallel.For(0, N, delegate(int i) {

results[i] = Compute(i);

});

Similar support is provided for looping over **IEnumerable<T>** objects

Parallel.ForEach(data, delegate(MyClass c)

{

Compute(c);

});

Using ForEach is generally less efficient than For, because many threads must

access the same underlying enumerator. Parallel.ForEach is, however, intelligent

enough to detect and access IList<T> instances in a more efficient

manner.

*Parallelizing may have a performance penalties.*

**Parallel.Invoke**

The Invoke static method on Parallel supports the parallelization of blocks of

statements. Frequently in applications, a sequence of statements exists for

which the order of execution doesn’t matter and which are entirely independent.

In such cases, rather than executing the statements sequentially, one after the

other, we can execute them in parallel for a potential performance gain.

static void WalkTree<T>(Tree<T> tree, Action<T> func)

{

if (tree == null) return;

WalkTree(tree.Left, func);

WalkTree(tree.Right, func);

func(tree.Data);

}

To:

static void WalkTree<T>(Tree<T> tree, Action<T> func)

{

if (tree == null) return;

Parallel.Invoke(

() => WalkTree(tree.Left, func) ,

() => WalkTree(tree.Right, func) ,

() => func(tree.Data));

}

These kinds of situations frequently arise in recursive, divide-and-conquer

algorithms.

For large inputs, splitting the array into segments that are processed in

parallel is generally faster. However, for smaller arrays, the overhead involved

in coordinating the work between multiple threads can lead to degraded

performance.

### Using System.Threading.Tasks.Task

[System.Threading.Tasks.Task](http://docs.google.com/html/6e745a04-8cb4-a9ba-45c1-6526f37b957c.htm) class be used directly to solve parallel problems with greater flexibility and control over the way work is partitioned

Task t1 = Task.Create(delegate { A(); });

Task t2 = Task.Create(delegate { B(); });

 Task t3 = Task.Create(delegate { C(); });

 t1.Wait(); t2.Wait(); t3.Wait();  //Or Task.WaitAll(t1, t2, t3);

Of course, as we’re dealing here with a simple sequence of statements, it can be

made even simpler using the Parallel.Invoke method already described:

Parallel.Invoke( ()=>A() , ()=>B() , ()=>C() );

Unlike the Invoke method, however, using the Task class directly

allows for control over how the Task instances are created (using various

overloads of **Task.Create**). It also allows for work to be done between the

asynchronous invocations of each of the methods:

... // do work that A relies on

Task t1 = Task.Create(delegate { A(); });

... // do work that B relies on

Task t2 = Task.Create(delegate { B(); });

... // do work that C relies on

Task t3 = Task.Create(delegate { C(); });

Task.WaitAll(t1, t2, t3);

In addition to wait support, Task also provides cancellation support (only when the task is not started).

### Using System.Threading.Tasks.Future<T>

The [System.Threading.Tasks.Future<T>](http://docs.google.com/html/b54b9478-9678-ea8d-49eb-b99359f98b49.htm) class derives from Task. A Future<T> is a Task that has a value associated with it.

a Future<T> can be created to represent a computation that returns a value that will be needed at some time in the future. When the value is needed, code calls to the Future<T>’s [Value](http://docs.google.com/html/bbe7a0d4-8642-a688-1bae-5890ddb84a39.htm) property to access the computed value; if the Value is not yet available, the caller blocks until it is. If you prefer not to wait for the value, the [IsCompleted](http://docs.google.com/html/ab168a8d-bf86-8a7a-c1a3-24794f97fbee.htm) property can be checked instead.

Single-assignment (Value) futures are also supported, which can be used to resume a blocked thread that is waiting for the value.

var data = new Future<int>[10000];

Parallel.For(0, data.Length, i =>

{

data[i] = Future.Create(() => Compute(i));

});

...

// some time later when the data is actually required

for(int i=0; i<data.Length; i++)

{

DoSomethingWithResult(data[i].Value);

}

If an individual computation has completed by the time the associated Future<int>.Value is invoked, Value will return immediately with the computed data. Otherwise, Value will **block** until the Future<int> computation completes.

Future<T> is quite useful for a variety of tasks, but a common usage is with recursive operations where a value is derived from some branch of the recursion tree

int CountNodes(Tree<int> node)

{

if (node == null) return 0;

var left = Future.Create(() => CountNodes(node.Left));

int right = CountNodes(node.Right);

return 1 + left.Value + right;

}

There is overhead to creating these Future<T> objects, to coordinating

work between the underlying threads, and so forth.

### **Using the Coordination Data Structures**

The System.Threading namespace of the .NET Framework 3.5 already contains a handful of synchronization primitives, such as events, monitors, and mutexes, which are low-level constructs useful in developing multithreaded applications. Many parallel applications, however, would benefit greatly from higher-level constructs such as thread-safe collections, more sophisticated locking primitives, data structures to facilitate work exchange, and types that control how variables are initialized. Parallel Extensions adds a plethora of such constructs to the System.Threading and System.Threading.Collections namespaces. These types include:

[System.Threading.CountdownEvent](http://docs.google.com/html/028aa201-392c-d118-d265-33a63cb7c4f5.htm)

[System.Threading.LazyInit<T>](http://docs.google.com/html/2239088b-d730-b720-17f5-d16366dc98d7.htm)

[System.Threading.ManualResetEventSlim](http://docs.google.com/html/30bdcc07-e2ab-11a2-54bc-ba43a5c4fc3e.htm)

[System.Threading.SemaphoreSlim](http://docs.google.com/html/e0527fd5-1a34-1f58-00a7-f1dd775e53f0.htm)

[System.Threading.SpinLock](http://docs.google.com/html/49ee1690-e138-0465-c0a1-93274220442d.htm)

[System.Threading.SpinWait](http://docs.google.com/html/5ae421fb-e7ca-4ce0-d867-38138083cb57.htm)

[System.Threading.WriteOnce<T>](http://docs.google.com/html/7ea4e687-3aa5-39e6-3d08-6dc15a0e5420.htm)

[System.Threading.Collections.BlockingCollection<T>](http://docs.google.com/html/8fd8f19b-be59-6022-67dd-63b34bfa9530.htm)

[System.Threading.Collections.ConcurrentQueue<T>](http://docs.google.com/html/15855316-e084-5f9a-7b47-dc713abff356.htm)

[System.Threading.Collections.ConcurrentStack<T>](http://docs.google.com/html/2826b901-8f27-3f14-6ebb-d27678535342.htm)

[**System.Threading.CountdownEvent**](http://docs.google.com/html/028aa201-392c-d118-d265-33a63cb7c4f5.htm)

[System.Threading.CountdownEvent](http://docs.google.com/html/028aa201-392c-d118-d265-33a63cb7c4f5.htm) is a synchronization primitive that allows threads to signal the event and other threads to wait for it to be set. CountdownEvent is set when a certain number of threads have signaled the event, counting down from a predetermined value.

**System.Threading.LazyInit<T>**

Lazy initialization is a commonly-used tactic for delaying data initialization until the data is actually needed. Multithreaded applications, where multiple threads may be accessing the lazily-initialized data simultaneously, need sophisticated, thread-safe constructs. To make lazy initialization simpler, Parallel Extensions provides the [System.Threading.LazyInit<T>](http://docs.google.com/html/2239088b-d730-b720-17f5-d16366dc98d7.htm) class. With [LazyInit<T>](http://docs.google.com/html/2239088b-d730-b720-17f5-d16366dc98d7.htm), the previous example is transformed into the following thread-safe code:

private LazyInit<MyData> \_data; ... public MyData Data { get { return \_data.Value; } }

Or

private LazyInit<MyData> \_data = new LazyInit<MyData>(() => CreateMyData()); ... public MyData Data { get { return \_data.Value; } }

[System.Threading.LazyInitMode](http://docs.google.com/html/5fc6170f-4175-6d6c-485c-b5d57ff7ff12.htm): AllowMultipleExecution, EnsureSingleExecution, ThreadLocal

[**System.Threading.ManualResetEventSlim**](http://docs.google.com/html/30bdcc07-e2ab-11a2-54bc-ba43a5c4fc3e.htm)

if one wants to use the Windows events, there is a high overhead: the cost of allocating the kernel object, the cost of a new finalizable object for the garbage collector to track, and the costs of the associated kernel transitions any time one wishes to wait or set the event.

The new [System.Threading.ManualResetEventSlim](http://docs.google.com/html/30bdcc07-e2ab-11a2-54bc-ba43a5c4fc3e.htm) type addresses many of these issues. It provides an interface similar to ManualResetEvent (unlike ManualResetEvent, it doesn’t provide the ability to create named events and thus can’t be used for cross-process synchronization, and it doesn’t derive from WaitHandle) and under the covers attempts to delay and eliminate all of the costs previously mentioned. ManualResetEventSlim can be used in a manner very similar to ManualResetEvent.

[**System.Threading.SemaphoreSlim**](http://docs.google.com/html/e0527fd5-1a34-1f58-00a7-f1dd775e53f0.htm)

Parallel Extensions also provides [System.Threading.SemaphoreSlim](http://docs.google.com/html/e0527fd5-1a34-1f58-00a7-f1dd775e53f0.htm), which can frequentliy eliminate many of the costs associated with Semaphore, and through a very similar manner.

**System.Threading.SpinLock (funny one).**

A spin lock is a mutual exclusion lock primitive where a thread trying to acquire the lock waits in a loop repeatedly checking until the lock becomes available (in other words, it “spins”). As the thread remains active performing a non-useful task, the use of such an advanced lock is a kind of busy waiting and consumes CPU resources without performing real work. However, spin locks can be more efficient than other kinds of locks on multi-processor machines if threads are only likely to be blocked for a very short period of time; they can be an effective low-level mechanism for avoiding unnecessary context switches and associated kernel transitions, which might otherwise be more costly than spinning

**System.Threading.SpinWait**

Spin waiting can be an effective low-level mechanism for avoiding unnecessary context switches and associated kernel transitions.

[**System.Threading.WriteOnce<T>**](http://docs.google.com/html/7ea4e687-3aa5-39e6-3d08-6dc15a0e5420.htm)

The [System.Threading.WriteOnce<T>](http://docs.google.com/html/7ea4e687-3aa5-39e6-3d08-6dc15a0e5420.htm) class provides a very simple interface for accessing the contained T value. The [WriteOnce<T>.Value](http://docs.google.com/html/bc543393-aecf-3fd9-9598-ca64b5ebc204.htm) property has get and set accessors; however, [Value](http://docs.google.com/html/bc543393-aecf-3fd9-9598-ca64b5ebc204.htm) may only be retrieved after it has been set, and it may only be set once

[**System.Threading.Collections.ConcurrentQueue<T>**](http://docs.google.com/html/15855316-e084-5f9a-7b47-dc713abff356.htm)

[System.Threading.Collections.ConcurrentQueue<T>](http://docs.google.com/html/15855316-e084-5f9a-7b47-dc713abff356.htm) is a thread-safe and scalable queue data structure. As with its System.Collections.Generics.Queue<T> counterpart, ConcurrentQueue<T> provides an [Enqueue](http://docs.google.com/html/6aa422d2-1a48-6834-05d4-67641e6caced.htm) method for adding an element to the queue. Unlike Queue<T>, however, ConcurrentQueue<T> doesn’t provide a Dequeue method for removing an item from the queue. Instead, it provides a [TryDequeue](http://docs.google.com/html/1f27ea89-9853-0da1-aef4-9ecf7c5d0788.htm) method that returns a Boolean value indicating whether an item could be dequeued and an out parameter containing the dequeued element if one could be retrieved.

[**System.Threading.Collections.ConcurrentStack<T>**](http://docs.google.com/html/2826b901-8f27-3f14-6ebb-d27678535342.htm)

As with [ConcurrentQueue<T>](http://docs.google.com/html/15855316-e084-5f9a-7b47-dc713abff356.htm) and Queue<T>, [System.Threading.Collections.ConcurrentStack<T>](http://docs.google.com/html/2826b901-8f27-3f14-6ebb-d27678535342.htm) serves as a thread-safe alternative to System.Collections.Generics.Stack<T>.

[**System.Threading.Collections.BlockingCollection<T>**](http://docs.google.com/html/8fd8f19b-be59-6022-67dd-63b34bfa9530.htm)

[BlockingCollection<T>](http://docs.google.com/html/8fd8f19b-be59-6022-67dd-63b34bfa9530.htm) acts as a wrapper around any concurrent collection that implements the [System.Threading.Collections.IConcurrentCollection<T>](http://docs.google.com/html/98a3ed48-429a-3bcd-16bc-699199ca3aeb.htm) interface, providing blocking and bounding capabilities on top of such a collection. Both the previously described [ConcurrentStack<T>](http://docs.google.com/html/2826b901-8f27-3f14-6ebb-d27678535342.htm) and [ConcurrentQueue<T>](http://docs.google.com/html/15855316-e084-5f9a-7b47-dc713abff356.htm) types implement IConcurrentCollection<T>, allowing them to be used with BlockingCollection<T>, but custom implementations of IConcurrentCollection<T> can also be used.

As an example, consider creating a blocking queue of strings:

private BlockingCollection<string> \_data = new BlockingCollection<string>();

This could also be done by explicitly providing the underlying collection to be used:

private BlockingCollection<string> \_data =

new BlockingCollection<string>(new ConcurrentQueue<string>());

### **Handling Concurrent Exceptions**

Parallel Extensions reports all concurrent exceptions by aggregating them into a [System.Threading.AggregateException](http://docs.google.com/html/4d1034db-b5f4-2f60-4827-e3f1a2add63b.htm) object. The original exceptions are accessible through the InnerExceptions property of this exception

When an exception occurs on one of the threads PLINQ uses for execution, the system first tries to stop as quickly as possible all other threads from executing additional work. This process happens transparently to the developer. However, by the time all of the threads of execution are successfully stopped, additional exceptions may have been thrown from those other threads. Once all threads have stopped processing the query, the full set of exceptions that were thrown is aggregated into a new System.Threading.AggregateException object, and that new aggregate exception object is rethrown.

### Using AggregateException

Parallel Extensions always throws a single AggregateException when an unhandled exception terminates a task’s execution, even if only one exception was thrown.

The approach taken by Parallel Extensions, unfortunately, makes debugging more difficult. If an exception goes unhandled and you attach a debugger, you will break at the point where the AggregateException was thrown rather than where your exception(s) originated from to begin with.

* [**Parallel**](http://docs.google.com/html/18816338-cac8-9d1e-c7ac-b52f54d425d3.htm) **class** – Exceptions thrown from multiple iterations of a parallel loop ([Parallel.For](http://docs.google.com/html/2139c638-5357-3f19-3a78-823fb79e404c.htm), [Parallel.ForEach](http://docs.google.com/html/1962739f-0e63-676c-b165-8b3432748c4d.htm)) or from multiple statements in a parallel region ([Parallel.Invoke](http://docs.google.com/html/1962739f-0e63-676c-b165-8b3432748c4d.htm)) will be aggregated into an AggregateException.
* [**Task**](http://docs.google.com/html/6e745a04-8cb4-a9ba-45c1-6526f37b957c.htm) **class** – Exceptions thrown during the asynchronous execution of a Task will be rethrown from the Task’s [Wait](http://docs.google.com/html/00f2bae3-994c-5672-9be5-a28646fadceb.htm) method. The exceptions are also available from the Task’s [Exception](http://docs.google.com/html/3f78dc0b-d42c-9558-4a68-34fa6c0fdc61.htm) property.
* [**Future<T>**](http://docs.google.com/html/b54b9478-9678-ea8d-49eb-b99359f98b49.htm) **class** – Exceptions thrown during the asynchronous execution of a Future<T> will be rethrown both from its [Wait](http://docs.google.com/html/00f2bae3-994c-5672-9be5-a28646fadceb.htm) method (inherited from [Task](http://docs.google.com/html/6e745a04-8cb4-a9ba-45c1-6526f37b957c.htm)) and from its [Value](http://docs.google.com/html/bbe7a0d4-8642-a688-1bae-5890ddb84a39.htm) property.
* **PLINQ** – Because queries in LINQ have deferred execution, wrapping the query in a try block will not typically behave as you might expect. Rather, the exceptions are thrown where the query is enumerated

#### Where to Handle Exceptions

There are many scenarios where it is better for the application to handle exceptions within the body of the delegate itself. In general, it is best to handle exceptions as close to the source of the problem as possible. Additionally, this allows those exceptions to be handled without causing the entire loop to stop execution.

Following best practices, the developer considers it better to allow unknown exceptions to bubble up (in this case through an AggregateException), rather than handling them directly within the body of the delegate.

public static void ProcessImages(string path)

      {

          Parallel.ForEach(GetImageFiles(path), imageFilePath =>

          {

              try

              {

                  Bitmap bmp = new Bitmap(imageFilePath);

                  ProcessImage(bmp);

              }

              catch (UnauthorizedAccessException uae)

              {

                  HandleUnauthorizedAccessException(uae);

              }

              catch (FileNotFoundException fnfe)

              {

                  HandleFileNotFoundExceptions(fnfe);

              }

          });

      }

If you want to allow a loop to continue executing even in the presence of exceptions from iterations of the loop, you'll need to manually marshal the exceptions

var exceptions = new ConcurrentStack<Exception>();

      Parallel.For(0, N, i=>

      {

          try

          {

              Process(i);

          }

          catch(Exception exc) { exceptions.Push(exc); }

      }

      if (!exceptions.IsEmpty) throw new AggregateException(exceptions);

#### **Parallelism Blockers**

A very common question is: Why is an explicit programming model needed for concurrency instead of having the system automatically decide to provide parallelism?

There are several reasons why there needs to be some programmer involvement in the parallelism process:

1. Mutable data structures and impurity  (race conditions).
2. Concurrent exceptions
3. Thread affinity(The context simply means that code must run on a very specific thread in order to be correct,e.g. STA.)
4. Ordering expectations(to avoid assumptions on ordering in your programs where possible).
5. Performance - Problems with < 1.0 speedup(overhead and Amdahl's law)

#### **Performance Tips**

1. Target areas of your program where algorithms are computationally expensive
2. Consider using the server garbage collector for your parallel applications. For more information on the server garbage collector, see the documentation for [<gcServer>](http://msdn.microsoft.com/library/ms229357.aspx).
3. Parallelize outer loops, not inner loops. So, in this example:
4. Where possible, avoid enabling ordering in PLINQ (order preservation is off by default), and avoid operators that add ordering expectations, such as OrderBy.
5. Prefer having independent loop iterations and [System.Threading.Tasks.Task](http://docs.google.com/html/6e745a04-8cb4-a9ba-45c1-6526f37b957c.htm) bodies to using synchronization.
6. Be conscious about the number of [System.Threading.Tasks.Task](http://docs.google.com/html/6e745a04-8cb4-a9ba-45c1-6526f37b957c.htm) instances created explicitly.
7. Have realistic expectations.

#### **Choosing an Approach**

Data parallelism enables you to break problems into individual parallel units of work by partitioning data, like arrays, lists, loop iterations, or XML files, while task parallelism provides more control over the expression of parallelism, enabling you to divide work based on the way your program’s logic is grouped into functions and statements.

**Declarative Data Parallelism – Choose PLINQ if you…**

* don't have a strong preference of model. There are patterns for using PLINQ to express task-based parallelism.
* want to be forward looking as parallelism blockers become addressed in future releases.
* already have LINQ code that you want to parallelize.
* want a declarative syntax. You want to describe what the program should do rather than how the program should do it.
* think more in terms of the data than the tasks.

**Imperative Data Parallelism – Choose Parallel if you…**

* already have loops in your code that you want to parallelize.
* think better in terms of existing loop and region semantics.
* want to translate sequential imperative code to be parallel.

**Imperative Task Parallelism – Choose Tasks and Futures if you…**

* need more control than PLINQ and Parallel provide.
* think in terms of how you want to achieve parallelism in terms of tasks or actions.
* want a semantic similar to how threads and the threadpool work today, but you want it to be more lightweight.

#### Structured and Unstructured Parallelism

Structured parallelism is a model in which the initiation of parallel work and waiting for said work to complete happens at the same location in the program.

Unstructured parallelism begins and joins with work in separate steps, and is generally more error prone.

Data Parallelism (Task Parallel Library)

Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. In data parallel operations, the source collection is partitioned so that multiple threads can operate on different segments concurrently. In basic loops, you do not have to take locks. The TPL handles all the low-level work for you. The following code example shows a simple foreach loop and its parallel equivalent.

Parallel.ForEach(sourceCollection, item => Process(item));

public static ParallelLoopResult For(

    int fromInclusive,

    int toExclusive,

    Action<int> body

)

public static ParallelLoopResult For(

    int fromInclusive,

    int toExclusive,

    Action<int, ParallelLoopState> body

)

public static ParallelLoopResult For(

    int fromInclusive,

    int toExclusive,

    ParallelOptions parallelOptions,

    Action<int> body

)

public static ParallelLoopResult For(

    int fromInclusive,

    int toExclusive,

    ParallelOptions parallelOptions,

    Action<int, ParallelLoopState> body

)

public static ParallelLoopResult For<TLocal>(

    int fromInclusive,

    int toExclusive,

    Func<TLocal> localInit,

    Func<int, ParallelLoopState, TLocal, TLocal> body,

    Action<TLocal> localFinally

)

public static ParallelLoopResult For<TLocal>(

    int fromInclusive,

    int toExclusive,

    ParallelOptions parallelOptions,

    Func<TLocal> localInit,

    Func<int, ParallelLoopState, TLocal, TLocal> body,

    Action<TLocal> localFinally

)

    /// <summary>Estimates the value of PI using a Parallel.For.</summary>

    static double ParallelPi()

    {

        double sum = 0.0;

        double step = 1.0 / (double)num\_steps;

        object monitor = new object();

        //Not every step is allocated into a thread/core. Multiple steps can be split into a thread.

        //Thus it would be better the loop is independent to each other.

        Parallel.For(0, num\_steps, () => 0.0, (i, state, local) =>

        {

            double x = (i + 0.5) \* step;

            return local + 4.0 / (1.0 + x \* x);

        }, local => { lock (monitor) sum += local; });

        return step \* sum;

    }

**Parallel.ForEach has similar methods.**

How to: Stop or Break from a Parallel.For Loop

"break" means complete all iterations on all threads that are prior to the current iteration on the current thread, and then exit the loop. "Stop" means stop all iterations as soon as convenient

loopState.Stop();

# How to: Write a Parallel.For Loop That Has Thread-Local Variables

By using thread-local data, you can avoid the overhead of synchronizing a large number of accesses to shared state. Instead of writing to a shared resource on each iteration, you compute and store the value until all iterations for the task are complete. You can then write the final result once to the shared resource, or pass it to another method.

// Use type parameter to make subtotal a long, not an int

Parallel.For<long>(0, nums.Length, () => 0 //Local variable initialization

, (j, loop, subtotal //thread local variable, as the return value.

) =>

{

subtotal += nums[j];

return subtotal;

},

(x) => Interlocked.Add(ref total, x) //finalize method, the thread-local as the return value.

);

**Parallel.ForEach works in a similar way.**

# How to: Cancel a Parallel.For or ForEach Loop

Use the Cancellation Framework in .NET 4.0. In a parallel loop, you supply the [CancellationToken](http://msdn.microsoft.com/en-us/library/system.threading.cancellationtoken%28VS.100%29.aspx) to the method in the [ParallelOptions](http://msdn.microsoft.com/en-us/library/system.threading.tasks.paralleloptions%28VS.100%29.aspx) parameter and then enclose the parallel call in a try-catch block.

try

{

Parallel.ForEach(nums, po, (num) =>

{

double d = Math.Sqrt(num);

Console.WriteLine("{0} on {1}", d, Thread.CurrentThread.ManagedThreadId);

po.CancellationToken.ThrowIfCancellationRequested();

});

}

catch (OperationCanceledException e)

{

Console.WriteLine(e.Message);

}

**How to: Handle Exceptions in Parallel Loops**

When you add your own exception-handling logic to parallel loops, handle the case in which similar exceptions might be thrown on multiple threads concurrently, and the case in which an exception thrown on one thread causes another exception to be thrown on another thread. You can handle both cases by wrapping all exceptions from the loop in a [System..::.AggregateException](http://msdn.microsoft.com/en-us/library/system.aggregateexception%28VS.100%29.aspx). The following example shows one possible approach.

# How to: Speed Up Small Loop Bodies

When a For loop has a small body, it might perform more slowly than the equivalent sequential loop. Slower performance is caused by the overhead involved in partitioning the data and the cost of invoking a delegate on each loop iteration. To address such scenarios, the [Partitioner](http://msdn.microsoft.com/en-us/library/system.collections.concurrent.partitioner%28VS.100%29.aspx) class provides the Create method, which enables you to provide a sequential loop for the delegate body, so that the delegate is invoked only once per partition, instead of once per iteration

// Source must be array or IList.

var source = Enumerable.Range(0, 100000).ToArray();

// Partition the entire source array.

var rangePartitioner = Partitioner.Create(0, source.Length);

double[] results = new double[source.Length];

// Loop over the partitions in parallel.

Parallel.ForEach(rangePartitioner, (range, loopState) =>

{

// Loop over each range element without a delegate invocation.

for (int i = range.Item1; i < range.Item2; i++)

{

results[i] = source[i] \* Math.PI;

}

});

# Task Parallelism (Task Parallel Library)

Tasks provide two primary benefits:

* More efficient and more scalable use of system resources.
* More programmatic control than is possible with a thread or work item.

The [Parallel..::.Invoke](http://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel.invoke%28VS.100%29.aspx) method provides a convenient way to run any number of arbitrary statements concurrently. Just pass in an Action delegate for each item of work. The easiest way to create these delegates is to use lambda expressions.

Parallel.Invoke(() => DoSomeWork(), () => DoSomeOtherWork());

For greater control over task execution, or to return a value from the task, you have to work with [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task%28VS.100%29.aspx) objects more explicitly.

var taskA = new Task(() => Console.WriteLine("Hello from taskA."));

Task.Factory.StartNew

In asynchronous programming, it is very common for one asynchronous operation, on completion, to invoke a second operation and pass data to it. Traditionally, this has been done by using callback methods. In the Task Parallel Library, the same functionality is provided by continuation tasks. A continuation task (also known just as a continuation) is a task that is invoked by another task, which is known as the antecedent, when the antecedent completes.

* pass data from the antecedent to the continuation
* specify the precise conditions under which the continuation will be invoked or not invoked
* cancel a continuation either before it starts or cooperatively while it is running
* provide hints about how the continuation should be scheduled
* invoke multiple continuations from the same antecedent
* invoke one continuation when all or any of multiple antecedents complete
* chain continuations one after another to any arbitrary length
* use a continuation to handle exceptions thrown by the antecedent